What do Grubhub®, Doordash®, and Verification Technology Have in Common?

By Tom Fitzpatrick, Editor and Verification Technologist

As I write this, we are well into our third month of lockdown, which I’m sure has altered the way most of you work. I say “you” because I have been blessed to work from home for the majority of my career, so other than not traveling as much as I used to, I haven’t really experienced a huge disruption in my professional life.

On the personal side, I’ve gone through the same as most of you, including having my college-age children finishing up their semester online (although it was nice having them home more than normal). Before the kids came home, my wife and I started using one of those home meal delivery services, where you order online and they send you all the ingredients to prepare the dinner yourself. Since my wife is usually the chef in our family, this has given me a great opportunity to expand my culinary skills and give her a well-deserved break. Once the kids came home, we suspended the service but by that time I’d built up a sizable repertoire of meals I am able to prepare (beyond pasta, French toast, tacos and the occasional stir-fry). All I have to do now is choose the meal and then buy the right items at the grocery store (and pay considerably more to feed my 22-year-old son!) and we’re good to go.

Think of this edition of Verification Horizons as a home-delivery service for verification technology. There may be some things you haven’t tried before, and you might think they’ll taste a little weird, but trust me that you’ll find them deliciously filling. Of course, for some of you, a few of the articles may feel a little like “comfort food,” but even then I’m sure that you’ll taste something new that will make you appreciate it all over again.

We begin with “Formal Is The ‘New Normal’ – Deploy These FV Apps In Your Next Project” from our friends at VerifWorks and CVC. If you’ve never used a formal tool before, this will give you a good overview of several Questa® Formal Apps and how best to apply them in your verification flow. I particularly like the differentiation the authors make between Specification-Driven and Implementation-Driven formal verification applications.

We continue our exploration of formal verification with “Understanding the SVA Engine Using the Fork-Join Model” by our long-term contributor Ben Cohen, a noted formal verification expert. In a follow-up to his
article from our March, 2018 issue, Ben takes us through a detailed analysis of how multi-threaded assertions actually work by modeling them as fork-joined SystemVerilog tasks. This obviously isn’t how the tools actually evaluate the assertions, but by using a mechanism we’re all familiar with, Ben clearly explains what’s going on under the hood of such tools.

In “Bridging the Portability gap for UVM SPI VIP Core reuse from IP to Sub System and SoC using Portable Stimulus,” our friends at Silicon Interfaces show how the new Portable Stimulus Standard can be used to create UVM sequences for a VIP Core that can be reused in multiple contexts. In this article, you’ll see how to model basic block-level operations in PSS for an SPI VIP, and understand how these can be leveraged at the subsystem and SoC levels.

We approach verification IP from a slightly different angle in “PCle Simulation Speed-Up Using Mentor QVIP with PLDA PCle Controller for DMA Application” from our friends at PLDA. If you’ve ever wanted to know more about PCle, this article gives a great overview of the protocol and the operational modes of Mentor’s Questa® VIP PCle component. With that understanding, the article provides a nice case study of PLDA’s experience in using the QVIP component to verify their own scalable PCle controller soft IP component.

We shift gears a bit to take a look at “Extending SoC Design Verification Methods for RISC-V Processor DV” from our friends at Imperas Software. With RISC-V cores now available from multiple vendors as well as in-house designs, including the ability to customize the instruction set, the ability to ensure that your core – the heart of your entire system – functions correctly becomes paramount. The article lays out a few verification flows that can be used to do just that. The key to it all is having a reliable and flexible reference model to serve as the golden model for the processor core, which is what Imperas does.

Another gear shift brings us to “Addressing VHDL Verification Challenges with OSVVM” from well-known VHDL advocate Jim Lewis of SynthWorks Design. The Open-Source VHDL Verification Methodology is Jim’s vehicle for bringing some of the capabilities available in SystemVerilog, like constrained-random stimulus and functional coverage, to the VHDL community. Whether you’re a VHDL fan or not, you’ll find some really interesting ideas in OSVVM for how to organize tests and testbenches and begin introducing these important capabilities to that community.

We close this issue with “Effective Verification Method of Safety Mechanism Compliant with ISO 26262” from our friends at Verification Technology. As you know, safety-critical designs have the important requirement that if something goes wrong, nobody dies. Given that, the ability to build safety mechanisms into the design to avoid that outcome is critical, as is verifying that the safety mechanisms actually function correctly to ensure that the design can recover gracefully from single-point or latent hardware failures. This article lays out in great detail how to set up your verification environment to make sure that your design will be able to handle such faults.
Normally, I would end my Editor’s Note for our DAC edition with an invitation to stop by our booth at DAC to say hi, but I can’t do that this year since the conference will be online. Instead, I’ll just invite you to check out verificationacademy.com to see all the great content we have there, including past issues of Verification Horizons. There are also many great videos including online training classes, webinars and partner presentations from last year’s DAC. I’m sure the online DAC program will be as informative as always, and this year you can check it all out while you’re cooking something new for dinner. Bon appétit!

Respectfully submitted,
Tom Fitzpatrick
Editor, Verification Horizons

CONTENTS

Page 4: Formal Is The “New Normal”
– Deploy These FV Apps In Your Next Project
by Ajeetha Kumari, Hemamalini Sundaram, and Darshan Ballari,
VeriWorks, LLC and CVC Pvt., Ltd.

Page 11: Understanding the SVA Engine
Using the Fork-Join Model
by Ben Cohen, VHDLCohen Publishing

Page 22: Bridging the Portability Gap
for UVM SPI VIP Core Reuse From IP to
Sub-System and SoC Using Portable Stimulus
by Kiran Malvi, Priyanka Gharat, Past Dean
Prof Sastry Puranapanda, Silicon Interfaces®

Page 31: PCIe Simulation Speed-Up Using
Mentor QVIP with PLDA PCIe Controller
for DMA Application
by Akshay Sarup, Mentor A Siemens Business
and Colin Gilly, PLDA

Page 40: Extending SoC Design Verification
Methods for RISC-V Processor DV
by Simon Davidmann, Lee Moore, Larry Lapides
and Kevin McDermott, Imperas Software, Ltd.

Page 48: Addressing VHDL Verification
Challenges with OSVVM
by Jim Lewis, SynthWorks Design, Inc.

Page 56: Effective Validation Method of
Safety Mechanism Compliant with ISO 26262
by Toshiyuki Hamatani, Verification Technology, Inc.
INTRODUCTION

Formal verification is now pervasive in many chip design verification projects. Key to this widespread adoption is the availability of automated “apps” that makes it easy to deploy Formal in hitherto simulation-only projects. We at VerifWorks have a long history of formal deployment at many design houses and have seen the challenges engineers face while adopting the same. We have also trained hundreds of engineers to use Formal with ABV (Assertion-Based Verification) through CVC. Having such widespread experience in deploying Formal, our team has seen Formal becoming the “new normal” with many teams adopting it in their projects.

Some of the key enablers in Formal adoption are:

- Availability of assertions/properties
- Identifying the right block/task for Formal in a simulation-dominant project

In this article, we will share our experiences in overcoming the above challenges using Questa® Formal and associated apps such as:

- Questa® AutoCheck
- Questa® Register Check
- Questa® Connectivity Check

CATEGORIZING FORMAL DEPLOYMENTS

We classify Formal into two broad categories:

- SDF - Specification Driven Formal
- IDF - Implementation Driven Formal

SDF - Specification Driven Formal

SDF - starts with specification and extracts properties, then deploys a formal verification tool such as Questa® Formal. A good example is configuration and status registers (CSRs) that are omnipresent in modern day configurable designs. There are certain classes of properties that can be extracted from these specifications. These are ideal candidates for Formal as the process is automated and repeatable.

Connectivity verification is another good example of SDF. In complex subsystems and SoCs, several IPs are integrated. Given that the IPs are verified stand-alone, their connectivity is a key aspect at subsystem and SoC level verification. Also, pin muxing is a common technique used by architects to keep the pin count reasonable in the final die/assembly. Such requirements on connectivity are typically captured and verified using Questa® ConnectCheck.

Extracting properties manually from a specification is another critical step in Formal deployment. Once identified, users code these properties in SystemVerilog Assertions (SVA) syntax. Such properties can then be verified using Questa® PropCheck.

IDF - Implementation Driven Formal

Given the vast amount of reuse in modern day systems, deploying Formal in real life often involves an existing implementation along with its specification. Typically, some parts of the specification change from one chip to another (in a derivative product) and some of these changes can be quickly verified using Formal. Many a times, original designers are no longer the owners of new, derivative designs, prompting a thorough check for any potential misuse of legacy code. This is where an automated checking tool such as Questa® AutoCheck comes very handy.
SDF - QUESTA® REGISTER CHECK

As noted earlier, configuration registers are key elements in configurable IPs and SoCs. Some of the common checks needed for many of these registers are:

- Out-of-reset-value checks
- Address map related checks
- Policy Checks such as W1C (Write-1-to-clear), etc.

Users often capture register definitions in one of the standard formats such as:

- IP-XACT
- SystemRDL

Though these are well-defined formats, teams often use custom formats based on:

- Microsoft® Excel®
- CSV
- XML
- YAML
- JSON, etc.

Given the regular structure of register specifications and the formats such as CSV/XML, ready-made apps are available to generate checkers/SVA automatically from these formats. For instance,

Questa® Formal supports CSV, IP-XACT and XML as shown below in figure 1.

Often, we find that teams use slightly modified schema in XML and/or custom extensions to IP-XACT, etc. In such cases, we at VerifWorks provide custom adaptors for non-standard CSR specification formats such as YAML, XML, etc. We make them compatible with Questa® Register Check’s desired format to improve productivity.

qverify_memmap is an executable used to generate the policy checks from XML/CSV in Questa®.

A typical run shows the following output summary:

```
# Summary:
#
# Register Name Address Checkers Count
#
# mode 0x0000 2
# times 0x0004 6
# count_pri 0x0100 2
# count_sec 0x0102 2
# count 0x0108 1
#
# Total 13
```

This summary is very useful to quickly review for correct translation of user intent to generated checkers. If there are tweaks needed to get the address space correct, for example, users can modify the input XML and quickly regenerate the checkers using the `qverify_memmap` command.

Next step in Register Check is to run the generated checkers using Formal Engines. The Questa®
Formal GUI provides an intuitive, easy-to-navigate view of the formal runs as shown above in figure 3.

As a quick summary, register verification can easily be pushed to formal verification with apps such as Register Check.

**SDF - QUESTA® CONNECTIVITY CHECK**

SoCs have several IPs connected and logic to handle pin muxing, etc. At the SoC level, one of the key challenges is to verify the connectivity of various IPs.

**Verifying connectivity/paths**

Consider a connection as shown below in figure 4:

![Figure 4: IP Connectivity](image)

To verify the above paths, we need 3 key elements:

- Stimulus (Src, Enable, Clk, Reset)
- Checkers - `a_path_chk_0 : assert property (en_0 == 0 |-> ##LAT dst_0 == $past(src_0, LAT));`
- Coverage - ensure all combinations are verified

Writing simulation tests to verify such structures is tedious and error-prone. Questa® Formal provides an efficient alternative for connectivity verification via the Connect Check app. Before we look at how we deploy Questa® Connectivity Check, let’s enumerate typical types of connections that we find in designs.

**Types of connections/paths**

We can classify different paths as below:

- Simple, direct connections:

  ![Simple Connection](image)

- A set of signals multiplexed to a common bus:

  ![Multiplexed Connection](image)

- Tie-offs - 0/1:

  ![Tie-offs](image)

- Several flip-flops may appear between the source signal point to the destination:

  ![Several Flip-flops](image)

- While several flip-flops may appear between the source signal point to the destination along with a multiplexor at the destination, each path may have a different latency.
Using Questa® Connectivity Check App

As with any Formal verification, assertions need to be fed as input to the formal engines. Questa® Connectivity Check is an app that generates the necessary assertions from a simple CSV format as input. Figure 5 above is a flow diagram from Questa® documentation to show the process.

We used this on a cloud platform IP that performs compression of the data stored on the cloud. Figure 6 above is a sample CSV file that we used as input to Questa® FV.

Once the connectivity CSV file is ready, we can run with Questa® Connectivity Check. A sample debug session is shown in figure 7.

Questa® FV provides a fully integrated GUI to show source code of the design, connections and the result of FV runs.

Simplifying connection specification

One of the challenges with CSV files is the readability of the code. Also for certain repeated connections, a CSV file tends to become too verbose and lengthy. Our consulting team at VerifWorks developed a Python app around Questa® Connectivity Check that takes a PATHS file as input and generates a Questa-friendly CSV file. A sample PATHS file is shown in figure 8 below:

![Figure 6: Sample CSV file](image)

![Figure 7: Questa® Connectivity Check sample debug session](image)

![Figure 8: Sample PATHS file](image)
With the PATHS file, the code is very readable, and users can add additional comments. It also supports iterative connectivity to capture recurring connections using a for loop like syntax. Connectivity verification becomes more productive when using a PATHS file and Questa® Connectivity Check.

**Bridging Connectivity to the Simulation World**

At times, users want to replay some of the connectivity checks in a typical simulation world using UVM. We have built a Python-based app to replay Formal connectivity with an auto-generated UVM test. We use our Go2UVM layer to simplify the stimulus creation and assist in auto-generated tests.

This flow comes in very handy when some of the connections are inconclusive in Formal engines even after long runs.

**IDF - QUESTA® AUTOCHECK**

As mentioned earlier, often teams find it necessary to deploy Formal on an existing RTL design. We deploy IDF - Implementation Driven Formal methodology in such cases.

**Who will write the assertions?**

Many engineers face this dilemma - what are some of the possible assertions/properties for my design? In our experience, this has been perhaps the single biggest reason why engineers hesitate to use Formal: its lack of properties/assertions.

Questa® AutoCheck addresses this problem nicely by automatically generating properties for common, known issues by analyzing your RTL code, as shown in figure 10 below.

Given the RTL design, Questa® AutoCheck analyzes and presents a set of extracted properties/observations in a table format as shown in figure 11 on the next page. It also provides the design hierarchy, schematic and a table/list of checks.

Questa® classifies these properties into 40+ different categories and presents each to the user. While some of the checks are like classical lint-type checks, having a formal engine to extract such checks is powerful as it reduces the noise that is prevalent in a typical RTL lint tool.

Some of the common checks our teams found very useful in Questa® AutoCheck, along with its perceived value are discussed below. It is important to appreciate that this is a subset that our consulting team found most valuable in recent projects, this list can vary from design to design.

**X_ASSIGN_REACHABLE**

One of the common coding styles is to assign X (Unknown) even at RTL stage to catch any errors as early as possible. The intention is not to exercise that piece of logic under normal working conditions and if it indeed occurs, a pessimistic X propagation would help flag it in simulations.

An orthogonal design practice in low power designs is to let tools introduce/inject X’s when a block is in
power-down mode. In a recent customer design that involved Low Power at the RTL stage, it was critical to differentiate X’s emerging due to power-down semantics from those due to explicit X assignments.

Questa® AutoCheck is designed to identify issues such as X propagation automatically and flags if an X-assignment is reachable, without having to run any simulation vectors. This can save many hours of debug in such scenarios.

**SLIST_INCOMPLETE**

Synthesis-Simulation mismatch is a well-researched topic. One of the sources for this issue is an incomplete sensitivity list in a typical always block in Verilog. Though well-known, new comers still make this mistake, especially when they don’t use the SystemVerilog enhanced always_comb block. Finding these issues during Synthesis and gate-level simulation is a painful and long process. Questa® AutoCheck enables users to find these much earlier in design cycle.

**FSM_UNREACHABLE_TRANS**

Many Lint tools perform detailed FSM analysis and flag potential issues. Sometimes tools tend to be very pessimistic and throw too many violations, making such reports too verbose. This leads to lot of time spent in analyzing such violation reports. However, a clear selection of most valuable checks from FSM reports based on design style can help prevent costly design errors. One such example in Questa® AutoCheck is FSM_UNREACHABLE_TRANS.

This is reported when a defined transition (as in RTL code below, in RED) can never occur due to the logic surrounding the FSM; in this specific example, assume that the logic around this FSM ensures that the value of i is always 1 when state == S2. Such cases will be harder to analyze by code reviews and an automated, formal mechanism to flag these is very useful.

```verilog
if (rst | c) state <= S0; else case (state)
S0: state <= S1;
S1: state <= S2;
S2: state <= ? S1: S0;
endcase
```

**Figure 11: Questa® AutoCheck extracted properties/observations**

**Reusing AutoCheck in Simulation**

Once these properties are generated for a given RTL, they are typically verified in Questa® Formal itself. This flow provides confidence for those properties that pass. However, for the failing ones, it requires user intervention to debug and analyze the failures. Some of these failing properties can be further propagated to simulation as part of the well-known assume-guarantee principle. For instance, if AutoCheck reported a possible out-of-bound access issue, to ensure that the bounds are within legal limits, it requires to be guaranteed by the preceding block driving this signal.

**Figure 12: FSM example**
We at VerifWorks have developed a library of properties that can help in this process. Think of these as templates that users can leverage and plug these auto-generated properties into a simulation with Questa®, or any simulator.

SUMMARY
As the title of the article reads, Formal is the “new normal” and with real-life deployment case studies, we have shared our experience of how Questa® Formal apps help in adopting Formal verification. We highlighted common problems that can be easily solved by Formal tools along with apps. We have also highlighted some of the custom plug-ins that we have developed to bridge Formal and Simulation. With that, we hope that more and more teams adopt this “new normal” and become more productive!
Understanding the SVA Engine Using the Fork-Join Model

by Ben Cohen, VHDLCohen Publishing

INTRODUCTION
SVA (SystemVerilog Assertions) is a powerful shorthand assertion language with many constructs; it is built as an integral part of SystemVerilog but with a specific syntax and sets of rules. Unlike a scoreboard that tends to focus on a model implementation that mimics the DUT, SVA addresses the requirements; that brings out a better understanding of the requirements, along with its weaknesses for lack of definitions. Over the years, I experienced the difficulties that engineers have in the thorough understanding of the SVA underlying model and why sometimes the assertions behave unexpectedly to the users’ intent.

This article is a follow-up to the SVA Alternative for Complex Assertions article published in the March 2018 issue of Verification Horizons. Unlike that issue that stressed the use of tasks to model complex assertions, this article explains SVA through the modeling of the underlying principles of some of its core elements using SystemVerilog procedural constructs. This modeling style emphasizes the concepts of "threads", as typically demonstrated in debug tools. The focus is in the modeling of threads for properties where both the antecedent and the consequent are sequences with range delays. Note that the simulation implementations use optimization features, thus differing from what is presented in this article; however, the underlying principles presented here are still valid.

Three different types of SVA properties are used to emphasize different important concepts of SVA:

1) Antecedent/consequent: This model demonstrates the concepts of spawned threads, leading clocking event, attempts.

2) Range delays in the consequent: This model demonstrates the testing of each element of the range until success is reached and the lack of need to limit the ranges to a matched (i.e., valid) consequent.

3) Range delays in the antecedent: This model demonstrates the testing of each element of the range and the need for a first_match operator or other technique to limit the ranges to a matched antecedent.

This inner-depth of SVA modeling provides a greater understanding of SVA and that appreciation will make you a more efficient SVA coder. It also makes you appreciate the value of SVA short-hand notation and how SVA is more concise and readable when defining design properties. As a bonus, this assertion modeling approach provides more insights when used in the verification process within SystemVerilog classes or to solve complex verification problems that are difficult to implement in SVA. Note that this modeling approach maintains the spirit (but not vendor’s implementations) of SVA. Also note that it is possible to use the task modeling approach with fewer threads, such as using the while loop instead of launching threads.

FUNDAMENTAL CONCEPTS OF A PROPERTY AND AN ASSERTION
Before getting into the modeling aspects two definitions are needed:

Property: A property is a characteristic of the design or the body of the requirements; it is a collection of logical and temporal relationships between and among subordinates. Thus, a property is a statement about a design’s intended behavior; it does not simulate. For example:

```
property p_a2bnext3cnextd; @([posedge clk]);
$rose(a) ##2 b |
##3 c ##1 d);
endproperty : p_a2bnext3cnextd
```

The leading clocking event
The antecedent
The implication operator
The consequent (a property)
**Assertion:** A statement or directive that a given property is required to hold (be true) during its execution cycles. Thus, an assertion is a simulation/verification statement to execute the concurrent property as a parallel block.

```
assert property(p_a2bnext3cnextd)
  -> ep_pass;
else -> ep_fail;
```

Since a property represents a block statement that executes as a parallel block when triggered it can be emulated using tasks that are spawned in **fork-join_none** blocks. SystemVerilog 1800'2017:9.3.2 Parallel blocks states that the **fork-join** parallel block construct enables the creation of concurrent processes from each of its parallel statements. If the task is **automatic**, it is dynamically created during the forking, it executes concurrently with other blocks, and it dies when completed. In many respects it parallels an SVA evaluation in that the assertion is also dynamically created in the attempt phase (i.e., the start), it runs as a concurrent process, and it dies when completed. Thus, a spawned automatic task creates a dynamic thread that contains a set of possible sequential statements and can spawn other secondary threads with other **fork/join_none** tasks. Each of those dynamically created processes has a life of its own, independent from other threads. The figure below demonstrates the high-level model for the assertion with an antecedent, an implication operator, and a sequence for the consequent.

1. **Property triggering:** At every clocking event, the **automatic** task that represents the model for the property is forked

2. **Task evaluation:** The code representing the antecedent is evaluated. If the antecedent is nonvacuous then one or more consequent threads are forked. In SVA, properties that include range delays and sequence-repeats are multi-threaded; that would be modeled with tasks, one task for each thread. Those separate forked tasks do represent a better representation of the SVA model, though writing those models in that fashion can get very complex. Depending upon the type of property, it is possible to write the code with fewer tasks, thus varying a bit from the "true" SVA model, but still be correct. *This article will use that simpler code approach wherever it is appropriate.*

3. **Result evaluation:** Three possible results are possible for an assertion: Vacuous, Pass, or Fail. Vacuity is determined in the evaluation of the antecedent; a vacuous result is considered true (vacuously) as opposed to a truly valid pass where the antecedent and consequent are both matches. Pass or Fail of the property can in some instances be determined during the evaluation of the consequent. However, in many multithreaded cases the result from each consequent thread

---

**Figure 1:** High-level model for an assertion
must be collected to determine the final assessment of the assertion. Thus, in this model, each consequent thread sends its results to the master thread (i.e., the antecedent thread).

**The task SVA model**

A property can be emulated with an automatic `task`. The first expression of the antecedent is evaluated with an `if` statement that represents the attempt phase, and a `return` statement to emulate an assertion vacuity when that condition is false. If the `if` condition is true (i.e., a successful attempt), then the task proceeds to emulate further sequence elements until it reaches the antecedent endpoint, but checking at each clocking event that the sequence still matches. If at any cycle in the sequence there is no match then a `return` statement emulates vacuity.

At the nonvacuous successful completion of an antecedent thread (i.e., reaching an end point of a sequence) the task proceeds to emulate the consequent. Since the consequent is a property, it is also emulated using another task that is spawned using a `fork/join_none`. If an antecedent is multithreaded with a range or a repeat construct, each thread of that antecedent may trigger a separate consequent thread; this is demonstrated further down in this article. The consequent thread reports its success or failure to the calling task so that it may decide to short-circuit the evaluation in case of failure, or to consider all the evaluations before considering a success. The generic structure of the task with pseudo-code is shown in the two code samples below left.

**The assert model**

// Emulation of the assert statement

```vhls
always @(posedge clk) begin // emulate the assertion firing
    fork
        property_name(); // other tasks emulating properties
    join_none
end
```

In the model of the `assert` statement shown above the `task` `property_name()` is triggered in parallel to other possible tasks using the clocking event `posedge clk`.

**Simple Assertion**

Consider the following simple assertion:

```
ap_a2bnext3cnextd : assert
property(@(posedge clk)
$rose(a) ##2 b|->  ##3 c ##1 d);
```

The property of that assertion states that if the attempt of `$rose(a)` is true then `b` is evaluated two cycles later. If the end point `b` is true then the consequent is immediately evaluated as a separate thread. When the results of the consequent thread is received, provide the report. The task modeling is shown on the following page.
The consequent task implements the sequence thread ($##3 c ##1 d$). It states that for the property, represented by a sequence, after 3 cycles the variable $c$ must be true. This must then be followed after 1 cycle by $d$ to also be true. The consequent task is shown below.

```
// $rose(a) ##2 b |-> ##3 c ##1 d$
task automatic t_a2bnext3cnextd(); // $rose(a) ##2 b |-> ##3 c ##1 d$

bit failing, done; // Every forked consequent return its status here

// attempt
if($rose(a, @(posedge clk))) begin : rose_a
    -> e_instart;
    repeat(2) @(posedge clk); // ##2 b
    if(b) begin : success_antecedent
        fork // Spawing a new thread for consequent
        // $##c ##1 d$
        t_consequent(failing, done); // test consequent for that thread
    end
    join_none
end

// EVALUATE RESULTS OF CONSEQUENT
if(!failing) begin
    pass
    -> e_pass; // for debug
    end
else begin
    fail
    -> e_fail;
end

end :
else return;
```

Using the task modeling structure defined above, the assertion can be expressed as:

```
always @(posedge clk) begin // emulate the assertion firing
    fork
        t_a2bnext3cnextd(); // Assert for the above property emulated with a task
    join_none
    end
```

**Variation with a range in the consequent**

Consider a variation to the previous model

```
ap_a2bnext3cnextd_range : assert property(@(posedge clk)
    $rose(a) ##2 b |-> ##[1:3] c ##1 d); // range in consequent
```

This property uses the same antecedent as the previous assertion. However, in the consequent instead of ($##3 c ##1 d$) we have a range in the $c$, specifically ($##[1:3] c ##1 d$). The $##[1:3] c ##1 d$ is equivalent to ($##[1 c ##1 d]$ or ($##[2 c ##1 d]$ or ($##[3 c ##1 d]$), and that sequence (with the sequence ORing) represents three separate threads launched when the antecedent succeeds. For the assertion to succeed, the antecedent must match and any thread in the consequent must match. For an assertion failure, all three threads of the consequent must fail.

The modeling is identical to the previous model with the exception of the consequent task $t_consequent$ that forks three separate tasks (the threads) with each one passing its thread results. Those results are then collected in the $t_consequent$ task local variables. When all the forked threads are completed with the `wait((done1, done2, done3) == 3'b111)` the analysis of the results is then reported back to the antecedent thread for final reporting. Note that this modeling...
approach uses a style to emphasize the notion of threads and does not necessarily represent a simulation implementation.

The modeling changes include the following two tasks:

```verilog
task automatic t_range(int iteration, inout bit fail, done_one);
repeat(iteration) @(posedge clk); // ##[iteration] c
if(c) begin : c_pass
  repeat(1) @(posedge clk); // ##1 d
  if(d)
  begin
    fail=1'b0; // PASS
done_one=1'b1;
  -> e_done_one; // debug
  end
: d_pass
else begin
  fail=1'b1; // PASS
done_one=1'b1;
  -> e_done_one; // debug
end
: d_fail
end
: c_pass
else begin
  fail=1'b1; // PASS
done_one=1'b1;
  -> e_done_one; // debug
end
: c_error
endtask
```

```verilog
task automatic t_consequent (inout bit failing, done); // ##[1:3] c
bit done1, done2, done3, fail1, fail2, fail3;
-> e_start_consequent; // debug
fork
t_range(1, fail1, done1); // ##1 c
  if(fail1)
    begin : a_failure // all iterations failed
      failing=1'b1; // flag for reporting back calling task
done=1'b1; // tell calling test of done
    end
: a_failure
else begin : a_pass
  failing=1'b0; // flag for reporting back calling task
done=1'b1;
end
: a_pass
end
: t_consequent
```

Model for $(\text{rose}(a) \#2 b|\rightarrow \#[1:3] c \#1 d)$,
File: ap_a2bnext3cnextd_range.sv

The simulation of that code is shown on the following page in figure 3.
Demonstrated SVA Concepts

1. Leading clocking event: SVA requires that every assertion must have a leading clocking event to start the evaluation of the property; in the examples, it is expressed by the @(posedge clk) as the first component of the SVA assertion statement. That “start” of the assertion is called “the attempt”; in the above assertions this is the evaluation of the $rose(a). Thus, one can say that at every leading clocking event there is an attempt that starts a thread of evaluations over zero or more cycles. In the task modeling, the leading clocking event is the @(posedge clk) in the block that initiates the task. The attempt is the evaluation of the first element of the property, which can contain the evaluation of other variables over one or more cycles. Since no other clocking event is specified, all clocking delays (e.g., ##2) use the same clocking event as the one used in the attempt phase.

2. Threads: A new thread is started to evaluate the property (defined as a task). This is emulated with the forked task t_a2bnext3cnextd(). Within that thread, if the antecedent succeeds, the consequent thread is initiated, emulated with t_consequent forked task. If the consequent has a range, then within that last forked thread it forks one or more new thread. This is demonstrated in the example with the forking of the following tasks:

   t_range(1, fail1, done1); t_range(2, fail2, done2); t_range(3, fail3, done3);

Arguments are included to identify the range number needed for each of the forked threads, and for the thread to report back its results using the inout direction, specifically:

   task automatic t_range(int iteration, inout bit fail, done_one);

Important: Why is understanding the concept of threads and thread creation important? It’s because each thread represents a concurrent process that is dynamically created and this can explain why an assertion may behave unexpectedly as a result of multiple matches even after a first match. The table on the following page shows examples of code that solve some multithread issues.
Range delays in the antecedent
Consider the following property

\[ \$rose(a) \#[1:5] b \rightarrow \#3 c; \]

It states that at each end point of the sequence \((\$rose(a) \#[1:5] b)\) the variable “c” will be true three cycles later.

The issue that arises with many users is that the antecedent is multithreaded, and that can causes unanticipated errors. SVA requires that each of the threads of that antecedent with a range or a repeat statement, must be tested with its appropriate consequent.\(^2\)

Specifically, the property

\[ \$rose(a) \#[1:5] b \rightarrow \#3 c; \] is equivalent to

\[ (\$rose(a) \#[1] b) \lor (\$rose(a) \#[2] b) \lor \ldots (\$rose(a) \#[5] b) \rightarrow \#3 c; \]

This ORing in the antecedent creates multiple threads, something like:

\[
\begin{align*}
\$rose(a) \#[1] b \rightarrow \#3 c \quad \text{and} \quad \text{separate thread} \\
\$rose(a) \#[2] b \rightarrow \#3 c \quad \text{and} \quad \text{separate thread} \\
\$rose(a) \#[5] b \rightarrow \#3 c \quad \text{and} \quad \text{last thread}
\end{align*}
\]

Note: Any non-matched thread of the antecedent (when \(b=0\)) creates a vacuously true property result for that thread, meaning that though TRUE, it is of no significance. However, any failure of one of those property threads (Antecedent through consequent) causes the assertion to fail. That would occur if one of the threads of the antecedent is a match and its corresponding consequent is not a match (when \(c==0\)). For the property to success, there can be no property failure (vacuity is considered a vacuous true), and at least one thread of the antecedent matches along with a match of its corresponding consequent. Below, and on the following page, is a SystemVerilog model; admittedly it is a bit complicated!
The simulation of this code above is shown on the following page in figure 4.

In most cases, a user is interested in the evaluation of the first match of the antecedent with its consequent, instead of all threads of a multi-threaded antecedent. This is particularly important if the range delay is infinite (e.g., #[$1:5]$) because the assertion can never succeed since ALL threads of antecedent must be tested for success and the number of threads is infinite; it can fail though. To accomplish this in SVA one can use the first_match function to exclude all other matches after the first match of the antecedent sequence. The SVA property is expressed as:

\[ \text{ap_FMa1to5b_then_3c} : \text{assert property @} (\text{posedge clk}) \text{ first_match}(@($\text{rose(a)} #[$1:5]$ b) \Rightarrow \text{##3 c}); \]

The modeling of such a property is identical to the previous one with two exceptions:

1. the consequent task informs the calling task (the antecedent) that it ended with a pass or a failure. Without the first_match operator, it was the last thread of the antecedent (last_iteration) that informed its consequent thread when it was done with its execution if there were no failure in the thread.
2. The spawning of consequent threads stops (with a break of the loop) when a first_match is reached.

```verilog
// The attempt thread
task automatic t_a1to5b_then_3c();
#($\text{rose(a)} #[$1:5]$ b) // Every forked consequent
\text{bit failing, done, last_iteration};
\text{// for debug}
\text{// Every forked consequent
\text{// returns its status here}
\text{// for debug}
\text{if($\text{rose(a), @(posedge clk)) \begin{array}{c}
begin : \text{sig_a \text{// attempt}} \\
\rightarrow e_{\text{instance}}; \text{for debug}
end \end{array}}
\text{for (int i=1; i <= 5; i++) begin : the_1to5 // Loop to cover
\text{// all threads}
\text{last_iteration= (i==5); // determine the last iteration
\text{// for send to consequent
\text{@(posedge clk); // The ##1 for the ##[1:5]
\text{e_{inloop};
\text{if(b) begin : got_b // antecedent end point for
\text{// that thread
\text{// but SVA requires all threads of antecedent to be tested
\text{// Spawing a new thread for each
\text{// nonvacuous element of the range
\text{t_3c(last_iteration, failing, done); // test consequent
\text{// for that thread
\text{join_none
done: got_b
end: the_1to5
\rightarrow e_{1to5Send;
\text{wait(done); \rightarrow e_{done;
\text{// done==1 if consequent fails or the last thread of consequent
\text{if(!failing) begin : pass
\rightarrow e_{pass}; // for debug
end : pass
else begin : fail // report the failure
\rightarrow e_{fail;
end : fail
\text{end: sig_a
\text{else return; // vacuous assertion, ($\text{rose(a)}==0, exit task
\text{endtask
\text{// The attempt thread
\text{task automatic t_a1to5b_then_3c();
#($\text{rose(a)} #[$1:5]$ b) // Every forked consequent
\text{bit failing, done, last_iteration};
\text{// for debug}
\text{// Every forked consequent
\text{// returns its status here}
\text{// for debug}
\text{if($\text{rose(a), @(posedge clk)) \begin{array}{c}
begin : \text{sig_a \text{// attempt}} \\
\rightarrow e_{\text{instance}}; \text{for debug}
end \end{array}}
\text{for (int i=1; i <= 5; i++) begin : the_1to5 // Loop to cover
\text{// all threads}
\text{last_iteration= (i==5); // determine the last iteration
\text{// for send to consequent
\text{@(posedge clk); // The ##1 for the ##[1:5]
\text{e_{inloop};
\text{if(b) begin : got_b // antecedent end point for
\text{// that thread
\text{// but SVA requires all threads of antecedent to be tested
\text{// Spawing a new thread for each
\text{// nonvacuous element of the range
\text{t_3c(last_iteration, failing, done); // test consequent
\text{// for that thread
\text{join_none
done: got_b
end: the_1to5
\rightarrow e_{1to5Send;
\text{wait(done); \rightarrow e_{done;
\text{// done==1 if consequent fails or the last thread of consequent
\text{if(!failing) begin : pass
\rightarrow e_{pass}; // for debug
end : pass
else begin : fail // report the failure
\rightarrow e_{fail;
end : fail
\text{end: sig_a
\text{else return; // vacuous assertion, ($\text{rose(a)}==0, exit task
\text{endtask
\text{// The assert
\text{always @(posedge clk) begin // emulate the assertion firing
\text{fork
\text{t_a1to5b_then_3c(); // Assertion for the above property
\text{// emulated with a task
\text{// ... other tasks representing properties
\text{join_none
done
\text{end
\text{// The attempt thread
\text{task automatic t_a1to5b_then_3c();
#($\text{rose(a)} #[$1:5]$ b) // Every forked consequent
\text{bit failing, done, last_iteration};
\text{// for debug}
\text{// Every forked consequent
\text{// returns its status here}
\text{// for debug}
\text{if($\text{rose(a), @(posedge clk)) \begin{array}{c}
begin : \text{sig_a \text{// attempt}} \\
\rightarrow e_{\text{instance}}; \text{for debug}
end \end{array}}
\text{for (int i=1; i <= 5; i++) begin : the_1to5 // Loop to cover
\text{// all threads}
\text{last_iteration= (i==5); // determine the last iteration
\text{// for send to consequent
\text{@(posedge clk); // The ##1 for the ##[1:5]
\text{e_{inloop};
\text{if(b) begin : got_b // antecedent end point for
\text{// that thread
\text{// but SVA requires all threads of antecedent to be tested
\text{// Spawing a new thread for each
\text{// nonvacuous element of the range
\text{t_3c(last_iteration, failing, done); // test consequent
\text{// for that thread
\text{join_none
done: got_b
end: the_1to5
\rightarrow e_{1to5Send;
\text{wait(done); \rightarrow e_{done;
\text{// done==1 if consequent fails or the last thread of consequent
\text{if(!failing) begin : pass
\rightarrow e_{pass}; // for debug
end : pass
else begin : fail // report the failure
\rightarrow e_{fail;
end : fail
\text{end: sig_a
\text{else return; // vacuous assertion, ($\text{rose(a)}==0, exit task
\text{endtask
\text{// The assert
\text{always @(posedge clk) begin // emulate the assertion firing
\text{fork
\text{t_a1to5b_then_3c(); // Assertion for the above property
\text{// emulated with a task
\text{// ... other tasks representing properties
\text{join_none
done
\text{end
\text{// The attempt thread
\text{task automatic t_a1to5b_then_3c();
#($\text{rose(a)} #[$1:5]$ b) // Every forked consequent
\text{bit failing, done, last_itera}
The simulation for the code to the near left is shown on the following page in figure 5.

**Demonstrated Concepts - Multi-threading and need for unique match in antecedents**

The sequence `##[1:n]sequence` creates multiple threads. This model demonstrates that for assertions with range delays in antecedents there is a need for a unique match (refer to the _1to5 for Loop_). This is because SVA requires that each of the threads of a multithreaded antecedent be tested with its appropriate consequent, and for success, each of those threads must succeed.

Note that this multithreaded concept is not exclusive to delay ranges, but also apply to repeat ranges. For example:

```verilog
ap_repeat_1to5b_then_3c: assert property(@(posedge clk)
  ($rose(a) ##[1:5] b[*1:5]) |-> ##3 c); // A first_match() is needed in the antecedent
```

In many cases, the `first_match()` function can be avoided with the use of the `goto` operator ([->1]). For example, instead of:

```verilog
first_match($rose(a) ##[1:$] b) |-> ##3 c;
```

Use:

```verilog
$rose(a) ##1 b[->1]) |-> ##3 c;
// Use the goto operator
```

Equivalent to:

```verilog
$rose(a) ##1 !b[*0:$] ##1 b) |-> ##3 c;
```

**CONCLUSIONS**

SVA simplifies the coding of concurrent assertions; attempting to emulating them with SystemVerilog tasks can be complex, tedious, and error prone. However, emulating the assertions with tasks provides a deeper understanding of key concepts and guidelines including the notion of leading clocking events, vacuity, spawning of threads,
constraining each antecedent to a unique sequence (instead of multiple or infinite ranges). The use of tasks can be used for the verification of complex assertions higher level blocks such as unit tests. Another benefit of tasks is that they can be used in SystemVerilog classes where the use of concurrent SVA is illegal.

Figure 5: Model Simulation a_range_one.sv a_range_oneFM.png

REFERENCES
- http://SystemVerilog.us/vf/ap_a2bnex3cnextd.sv
- http://SystemVerilog.us/vf/a2b3c1d_simple.png
- http://SystemVerilog.us/vf/ap_a2bnex3cnextd_range.sv
- http://SystemVerilog.us/vf/ap_a2bnex3cnextd_range.png
- http://SystemVerilog.us/vf/a_range_one.sv
- http://SystemVerilog.us/vf/a_range_one.png
- http://SystemVerilog.us/vf/a_range_oneFM.sv
- http://SystemVerilog.us/vf/a_range_oneFM.png
END NOTES


1. SystemVerilog has the restriction that an actual argument passed as reference (with the ref) cannot be used within fork-join_any or fork_join_none blocks. See https://verificationacademy.com/forums/systemverilog/actual-argument-passed-reference-cannot-be-used-within-fork-joinany-or-forkjoinnone


Acknowledgement: I thank my co-author Srinivasan Venkataramanan from VerifWorks for his valuable comments and expertise.
INTRODUCTION
Over the past decades number of gates on IC’s and complexity of designs have increased rapidly which has caused various challenges in verifying circuits. Today’s IP, FPGA, and SoC engineers face biggest challenge in creating sufficient tests to verify and validate the design. In the current verification trends, Verification is a big issue for SOC level verification engineers in terms of verification product reuse, time, cost savings, etc. As we move from IP to sub-system and from sub-system to SoC the whole verification environment needs to be re-written which makes it difficult to re-use the test-intent across the various platforms for verification. Portable Standard Stimulus is the new gateway for overcoming such difficulties.

The main idea of the article is that some of the UVM Test Intents/Test cases are difficult to verify at SOC Level and not fit for SOC Level verification so leverage Portable Stimulus to verify and reuse the Complex SOC level designs. Portable stimulus approach is a scenario level graphical flow process describes all the SOC components functionalities and all the data transactions described through scenario graphical functional flow approach to reuse the verification components and to generate test intent stimulus at higher level of abstraction and support for various programming languages. PSS is used to specify test intent that can be targeted to a variety of verification platforms which leverages the verification process through reusability from block to SoC level using three Axes of Portability technique. Why Portable Stimulus? Because of single test intent representation across of various integration levels (IP to Sub-system, Subsystem to SOC) under variety of different platforms, different configurations and verification environment.

This article intends to showcase the application to reuse the UVM SPI VIP with master and slave configuration vertically (IP to Sub-system to SoC) where the single specification of the intent of VIP is described in DSL (Domain Specific Language) and generating UVM sequence using leading EDA tools. This generated UVM sequence is integrated into the targeted platform and design is verified with this sequence with more elaborated scenarios. This article also demonstrates the graphical description of the PSS test intent written in DSL.

With various trends in Semiconductor industry and advancement in verification technologies, the world welcomes autonomous cars, drones and recycling, Microelectronics and Semiconductor industry recognizes the order of the day is “reuse”, “automation” and “artificial intelligence”. The biggest challenge experienced by today’s IP, FPGA, ASIC and SoC engineers is to attain better control and achievement of test intent goals, reuse or portability of the test cases across various platforms.

Using UVM standard, methodology for IP verification, it is difficult to control dependencies between sequences, limiting the effectiveness of SoC-level
testbenches. In addition, the UVM does not address embedded processors within the SoC.

PSS is the new approach to resolve such hardships by providing a mechanism to specify single test intent that can be targeted to a variety of verification platforms.

**Fig1.b** explains the three axes of portability due to which reusability is achieved. The three axes of portability are

a) **Vertical Reuse**: Reuse of the test intent from one system to other system i.e. from IP to Sub-System to SoC level.

b) **Horizontal Reuse**: Reuse across different revision of same design.

c) **Technique Reuse**: Reuse across different verification platforms i.e. from Simulation to Emulation to FPGA. Reusability accelerates the test creation and also leverages the verification procedure.

This article focuses mostly on the vertical reuse of the test intent from IP-block to Sub-System and study of reusability from Sub-system to SoC level. The example taken to demonstrate vertical reusability is a single master and slave SPI Core IP configuration. A UVM layered testbench is wrapped around the design to verify and validate proper functioning of SPI Core IP. In order to enhance the verification process different scenarios are written in DSL language which captures possible areas of design which otherwise may not be possible with existing procedural languages at the various levels. In our article the test specification for SPI Core IP described in UVM sequences are enhanced by specifying the behavior of the design in DSL. This allows us a handle to write to the sequences of the sequencer. UVM sequences are generated and inserted into the existing testbench and verified against the newly generated UVM sequences.

**PORTABILITY FOR SPI UVM**

**Block-level**

Our SPI is a standard synchronous serial interface used for serial communication.

The above IP establish communication between SPI master and SPI slave.

To verify the design our test intent is specified in DSL by defining various scenarios described using actions, flow-objects and components. SPI master component has two actions one for writing i.e. “write_a” and other for reading i.e. “read_a”. In SPI slave “rx_wr_data_a” and “tx_rd_data_a” actions are defined.
SPI master outputs the buffer flow object which is “m_data_buff_b” declared inside “write_xfer_a” which is consumed by “rx_wr_data_a” inside slave component during write operation. While during read operation SPI slave outputs the buffer object “data_buff_b” declared in “tx_rd_data_a” action which is consumed by “read_xfer_a” embedded inside SPI master. All these components are encapsulated under top component “pss_top” where actions declared under SPI master slave are instantiated and traversed inside activity to generate higher abstraction level of scenarios.

DSL Code for Design

Following is the DSL code for specifying test intents at block-level for data transfer between SPI master and slave.

```plaintext
//Clock_rate
eenum clock_rate {
    FOSC_2,
    FOSC_4,
    FOSC_8,
    FOSC_16,
    FOSC_32,
    FOSC_64,
};

//Mode_type
eenum mode_type {
    MODE0 = 2'b00,
    MODE1 = 2'b01,
    MODE2 = 2'b10,
    MODE3 = 2'b11
};

package pkg_A {
    //STATUS_REGISTER
    //CONTROL_REGISTER
    struct status_register {
        rand bit spif; //Interrupt flag
        rand bit wcol; //Write collision
        rand bit spi2x; //Double clock rate
    }
}

package pkg_B {
    import pkg_A::*;
    buffer data_buff_b {
        rand bit ss;
        rand clock_rate s_clk;
        rand control_register ctrl_reg;
        rand status_register status_reg;
    }

    buffer m_data_buff_b : data_buff_b {
        //Constraints to select the clock rate depending on the value
        //of the spr1,spr2
        constraint clk_rate {
            if (status_reg.spi2x == 1'b0) {
                ctrl_reg.spr1 == 1'b0 &&
                ctrl_reg.spr0 == 1'b0
                -> s_clk == FOSC_2,
                ctrl_reg.spr1 == 1'b0 &&
                ctrl_reg.spr0 == 1'b1
                -> s_clk == FOSC_8,
                ctrl_reg.spr1 == 1'b1 &&
                ctrl_reg.spr0 == 1'b0
                -> s_clk == FOSC_16,
                ctrl_reg.spr1 == 1'b1 &&
                ctrl_reg.spr0 == 1'b1
                -> s_clk == FOSC_32,
            }
        }
    }
```
```verilog
ctrl_reg.spr0 == 1'b0
-> s_clk == FOSC_32;
ctrl_reg.spr1 == 1'b1 &&
ctrl_reg.spr0 == 1'b1
-> s_clk == FOSC_64;
}
else {
ctrl_reg.spr1 == 1'b0 &&
ctrl_reg.spr0 == 1'b0
-> s_clk == FOSC_4;
ctrl_reg.spr1 == 1'b0 &&
ctrl_reg.spr0 == 1'b1
-> s_clk == FOSC_16;
ctrl_reg.spr1 == 1'b1 &&
ctrl_reg.spr0 == 1'b0
-> s_clk == FOSC_64;
ctrl_reg.spr1 == 1'b1 &&
ctrl_reg.spr0 == 1'b1
-> s_clk == FOSC_128;
}
}

//===================================
//SPI_MASTER_COMPONENT
//===================================

component spi_master_c{
import pkg_B::*;
//Action to perform the write operation on the SPI SLAVE
action write_a{
output m_data_buff_b wr_prod_o;
rand bit [7:0]write_data;
rand mode_type mode_t;
//Covergroup to check the coverage for the stimulus
//during the write operation
covergroup {
WRITE_DATA : coverpoint write_data {
  bins low = [50..75];
  bins high = [76..100];
}
S_CLK : coverpoint wr_prod_o.s_clk;
SS : coverpoint wr_prod_o.ss;
}spi_cov;
}
}

//===================================
//SPI_SLAVE_COMPONENT
//===================================

cOMPONENT SPI_SLAVE_COMPONENT
//===================================

cOMPONENT SPI_SLAVE_COMPONENT
//===================================

cOMPONENT SPI_SLAVE_COMPONENT
//===================================

cOMPONENT SPI_SLAVE_COMPONENT
//===================================

cOMPONENT SPI_SLAVE_COMPONENT
//===================================
```
Generated UVM Sequence for Test Specifications given in DSL

The UVM generated sequence for SPI Master-Slave configuration with the test intent specified in DSL configuration is described as follows:

```verilog
class spi_sequence extends spi_base_sequence;
// Register sequence with UVM factory
`uvm_object_utils(spi_sequence)
string actions[0:5] = '{
  "root_a.inFact_parallel_204182259_start",
  "root_a.inFact_parallel_204182259_branch_start",
  "root_a.wr.er_c_write_a",
  "root_a.rx_wr.e_c_rx_wr_data_a",
  "root_a.tx_rd.e_c_tx_rd_data_a",
  "root_a.rd.er_c_read_a"
};
function new(string name="spi_sequence");
  super.new(name);
endfunction : new

  task body();
    string infactpss_trace;
    bit infactpss_trace_enabled = 0;
    if($value$plusargs("inFact.trace=%s", infactpss_trace))
      begin
        if(infactpss_trace == "on")
          infactpss_trace_enabled = 1;
        end
      fork
        begin
          // Exec body root_a.wr.er_c_write_a
          if(infactpss_trace_enabled == 1)
            begin
              infact_trace_entry(2);
          end
        end
  end
```
begin
    spi_seq_item req = spi_seq_item::type_id::create("req");
    start_item(req);
    assert(req.randomize() with {
        wr_data = 64'd 77;
        ctrl_reg[7] = -64'd 0;
        ctrl_reg[6] = 64'd 1;
        ctrl_reg[5] = -64'd 0;
        ctrl_reg[4] = 64'd 1;
        ctrl_reg[3] = -64'd 0;
        ctrl_reg[2] = 64'd 1;
        ctrl_reg[1] = -64'd 0;
        ctrl_reg[0] = 64'd 1;
        ss = -64'd 0;
        sclk = 64'd 2;
        status_reg[7] = -64'd 0;
        status_reg[6] = -64'd 0;
        status_reg[0] = 64'd 0;
    });
    `uvm_info("Data_Values", $sformatf("wr_data = %0h, ss = %0b", wr_data, ss), UVM_LOW);
    finish_item(req);
end
if (infactpss_trace_enabled==1)
begin
    infact_trace_exit(2);
end
// End exec body root_a.wr.er_c_write_a
end
begin
    // Exec body root_a.tx_rd.e_c_tx_rd_data_a
    if (infactpss_trace_enabled==1)
    begin
        infact_trace_entry(4);
    end
    begin
        spi_seq_item req = spi_seq_item::type_id::create("req");
        start_item(req);
        assert(req.randomize() with {
            ctrl_reg[7] = 64'd 1;
            ctrl_reg[6] = 64'd 1;
            ctrl_reg[5] = -64'd 0;
            ctrl_reg[4] = 64'd 1;
            ctrl_reg[3] = -64'd 0;
            ctrl_reg[2] = 64'd 1;
            ctrl_reg[1] = -64'd 0;
            ctrl_reg[0] = 64'd 1;
            ss = -64'd 0;
            sclk = 64'd 2;
            status_reg[7] = -64'd 0;
            status_reg[6] = 64'd 1;
            status_reg[0] = 64'd 0;
        });
    end
end
join
fork
begin
    // Exec body root_a.wr.er_c_write_a
    if (infactpss_trace_enabled==1)
    begin
        infact_trace_exit(2);
    end
begin
    // Exec body root_a.tx_rd.e_c_tx_rd_data_a
    if (infactpss_trace_enabled==1)
    begin
        infact_trace_entry(4);
    end
    begin
        spi_seq_item req = spi_seq_item::type_id::create("req");
        start_item(req);
        assert(req.randomize() with {
            ctrl_reg[7] = 64'd 1;
            ctrl_reg[6] = 64'd 1;
            ctrl_reg[5] = -64'd 0;
            ctrl_reg[4] = 64'd 1;
            ctrl_reg[3] = 64'd 0;
            ctrl_reg[2] = -64'd 0;
            ctrl_reg[1] = -64'd 0;
            ctrl_reg[0] = 64'd 0;
            ss = 64'd 0;
            sclk = 64'd 2;
            status_reg[7] = -64'd 0;
            status_reg[6] = 64'd 0;
            status_reg[0] = 64'd 0;
        });
        `uvm_info("Data_Values", $sformatf("wr_data = %0h, ss = %0b", wr_data, ss), UVM_LOW);
        finish_item(req);
    end
end
if (infactpss_trace_enabled==1)
begin
    infact_trace_exit(4);
end
// End exec body root_a.tx_rd.e_c_tx_rd_data_a
end
equip: end
assert(req.randomize() with{
    ctrl_reg[7] = -64'd0;
    ctrl_reg[6] = 64'd 1;
    ctrl_reg[5] = 64'd 1;
    ctrl_reg[4] = -64'd0;
    ctrl_reg[3] = 64'd 1;
    ctrl_reg[2] = 64'd 1;
    ctrl_reg[1] = -64'd0;
    ctrl_reg[0] = 64'd 1;
    ss = -64'd0;
    sclk = 64'd 2;
    status_reg[7] = -64'd0;
    status_reg[6] = 64'd 1;
    status_reg[0] = -64'd0;
});
finish_item(req);
end
if (infactpss_trace_enabled==1)
    begin
        infact_trace_exit(4);
        end
    // End exec body root_a.tx_rd.e_c_tx_rd_data_a
end
join
fork
    begin
        // Exec body root_a.wr.e_r_c_write_a
        if (infactpss_trace_enabled==1)
            begin
                infact_trace_entry(2);
            end
        begin
            spi_seq_item req = spi_seq_item::type_id::create("req");
            assert(req.randomize() with {
                wr_data = 64'd 81;
                ctrl_reg[7] = -64'd0;
                ctrl_reg[6] = 64'd 1;
                ctrl_reg[5] = 64'd 1;
                ctrl_reg[4] = -64'd0;
                ctrl_reg[3] = 64'd 1;
                ctrl_reg[2] = 64'd 1;
                ctrl_reg[1] = -64'd0;
                ctrl_reg[0] = 64'd 1;
                ss = -64'd0;
                sclk = 64'd 2;
                status_reg[7] = 64'd 1;
                status_reg[6] = -64'd0;
                status_reg[0] = -64'd0;
            });
            finish_item(req);
        end
        if (infactpss_trace_enabled==1)
            begin
                infact_trace_exit(2);
            end
        // End exec body root_a.wr.e_r_c_write_a
        end
    begin
        // Exec body root_a.tx_rd.e_c_tx_rd_data_a
        if (infactpss_trace_enabled==1)
            begin
                infact_trace_entry(4);
            end
        begin
            spi_seq_item req = spi_seq_item::type_id::create("req");
            start_item(req);
            assert(req.randomize() with {
                ctrl_reg[7] = -64'd0;
                ctrl_reg[6] = -64'd0;
                ctrl_reg[5] = 64'd 1;
                ctrl_reg[4] = -64'd0;
                ctrl_reg[3] = 64'd 1;
                ctrl_reg[2] = 64'd 1;
                ctrl_reg[1] = -64'd0;
                ctrl_reg[0] = 64'd 1;
                ss = -64'd0;
                sclk = 64'd 2;
                status_reg[7] = -64'd0;
                status_reg[6] = 64'd 1;
                status_reg[0] = -64'd0;
            });
            finish_item(req);
        end
        if (infactpss_trace_enabled==1)
            begin
                infact_trace_exit(4);
            end
        // End exec body root_a.tx_rd.e_c_tx_rd_data_a
        end
    joinork
    begin
        // Exec body root_a.wr.e_r_c_write_a
        if (infactpss_trace_enabled==1)
            begin
                infact_trace_entry(2);
            end
        begin
            spi_seq_item req = spi_seq_item::type_id::create("req");
            start_item(req);
            assert(req.randomize() with {
                wr_data = 64'd 65;
                ctrl_reg[7] = -64'd0;
                ctrl_reg[6] = 64'd 1;
                ctrl_reg[5] = -64'd0;
                ctrl_reg[4] = 64'd 1;
                ctrl_reg[3] = -64'd0;
                ctrl_reg[2] = 64'd 1;
                ctrl_reg[1] = -64'd0;
                ctrl_reg[0] = 64'd 1;
                ss = -64'd0;
                sclk = 64'd 2;
                status_reg[7] = -64'd0;
                status_reg[6] = 64'd 1;
                status_reg[0] = -64'd0;
            });
            finish_item(req);
        end
        if (infactpss_trace_enabled==1)
            begin
                infact_trace_exit(2);
            end
        // End exec body root_a.wr.e_r_c_write_a
    end
The SPI master slave IP test intent reuse at subsystem level is achieved by performing few modifications in the original DSL source code in order to incorporate the interaction of design with different blocks at Sub-System level. Fortunately, PSS allows us to extend existing actions, components, and data structures to add new constraints without actually modifying the original code. In our example we have wishbone on the other side of SPI master which is used for communication with the processor to access the data. New components and actions with flow objects are added to establish the communication between SPI master and slave to achieve our goal in verifying the test intent.

PSS provides with some of distinctive features which enables user to generate scenarios with a greater flexibility. Resources, state, exec-blocks, pool-binding, actions, components, activity, etc., from the basis of our test intent specified PSS model.
SoC Level

When we get to SoC level, the subsystem that we’ve verified will be combined with a processor subsystem. One big change, from a verification perspective, is that our tests will now run as C tests on the embedded processor in the design. In addition to this big change, there will be some other smaller changes such as the memory map for the full system. The same PSS benefits of flexibility and configurability that we saw at the block and subsystem levels make it easy to customize our PSS content for use in a SoC environment.

CONCLUSIONS

The early findings indicate that the Semiconductors/Microelectronics/VLSI Design industry will gain through Portable Stimulus Specification for Vertical/Horizontal and Technique Reuse but the gains would be measuring a few times if the existing UVM VIP can be reused for this purpose. The expectations that the paper study will result in positive conclusions for portability with a display of an SPI UVM IP enveloped with a PSS layer.
INTRODUCTION

PCI Express® (PCIe®) is a dominant technology for hardware applications requiring high-speed connectivity between networking, storage, FPGA, and GPGPU boards to servers and desktop systems. It is a robust technology that has evolved over decades to keep up with advancements in throughput and speed for I/O connectivity for computing requirements.

For memory-intensive and high-performance computing, Direct Memory Access (DMA) is an indispensable application. The trend over the years has been to move the DMA controller into devices using a point-to-point bus architecture to reduce latency and increase memory access throughput. A typical DMA operation in PCIe is the transfer of data from the system memory—that the host has access to—to end point devices.

This article discusses how verification engineers can use Mentor’s Questa® Verification IP (QVIP) to improve productivity during the functional verification of PCIe designs with DMA engines.

PCIe is built upon a layered architecture consisting of a transaction layer for payload transfers, a data link layer for link management, and a physical layer for initialization and training of a reliable PCIe link between two devices. In terms of PCIe verification, each layer has its own challenges and complexities. For verification of DMA engines, the verification effort is concentrated on the data transfer aspect of PCIe, which resides in the transaction layer. This article mentions techniques to speed up the PCIe link training and initialization processes as well as PCIe device enumeration in order to reduce the initial simulation runtime required to set up tests targeted towards verification of DMA functionality.

The following sections address the processes used to speed up initial PCIe set up and reference a use case where QVIP is used with the PCIe controller for DMA applications from PLDA, a developer of semiconductor intellectual property specializing in high-speed interconnects.

1. Initial PCIe link training and speed negotiation process: PCIe link up is a prerequisite phase in all tests exercising data transfers across PCIe link. Reduction in simulation runtime in this phase boosts verification productivity by having a shorter runtime for tests focused on verification of design features.

2. PCIe device enumeration and configuration space set up process: PCIe device discovery process (aka, device enumeration) is performed by the root port to discover end point device capabilities and set up the device and DMA engine after link up. A shortened set up time after link up also assists with reduced simulation runtime.

3. QVIP use case with the PLDA PCIe controller for DMA applications: PLDA chose Mentor QVIP to test standard compliance for PCIe and AMBA/AXI in their XpressRICH-AXI product line. In Mentor QVIP, PLDA found a flexible and reliable tool for building its proprietary test suite on a highly-scalable testbench.

INITIAL PCIE LINK TRAINING AND OPERATING SPEED NEGOTIATION

The essential step in a functional test using PCIe is to perform PCIe link training and initialization before data transfer can commence between the two PCIe devices. This step is an integral part of every test that utilizes PCIe for data transfer. Optimizing PCIe link up will result in a reduction in the simulation runtime to reach the L0 state (the fully-operational link state for data transfer). Figure 1 on the following page shows the states in the Link Training Status State Machine (LTSSM) that gets executed by the two devices when negotiating a PCIe link. The four main states through which the LTSSM traverses in order to establish a
reliable PCIe link are Detect, Polling, Configuration, and Recovery (Figure 1). The PCIe link traverses these four main states (and various sub-states defined within them) starting from Detect and following the path shown by the highlighted arrows to reach the L0 state.

During traversal of the LTSSM, the two PCIe devices exchange training sequences to negotiate a number of link parameters; including elements such as lane polarity, link/lane numbers, equalization, data rate of operation, and so on. Each of the main states define a set of counters for transmitting and receiving these training sequences in order to calibrate the transmitter and receiver of the two devices while advertising their link capabilities to form a reliable link based on mutual negotiation. These LTSSM transitions are mandated with a fixed number of training sequences ordered sets that need to be transmitted and received in these states.

Alongside these counters, there are timeouts defined per state and sub-state to avoid LTSSM deadlock issues that may occur due to transmitter/receiver errors and reset the LTSSM to the detect state.

The default values of these counters and timeouts need to be scaled down such that a reliable PCIe can be formed within an optimal simulation time with the desired link operation speed.

**PCIE GEN5 Introduces An Optional Link Equalization Bypass Mode**

Aside from reducing the counter values and scaling down the timeout, PCIe GEN5 introduces an optional link equalization bypass mode for faster link up. To train PCIe link at 32 GT/s, a conventional speed change process comprises: initially training the link to L0 at 2.5 GT/s and then initiating speed change followed by link equalization at the intermediate speeds: 8 GT/s, 16 GT/s, and finally 32 GT/s. Since equalization at data rates equal to or greater than 8 GT/s is an essential process for higher link reliability and lower bit error rates, the time spent performing speed change and equalization at each speed consumes approximately 100ms of simulation runtime.

With the equalization bypass mode, the PCIe link in L0 at 2.5 GT/s directly transitions the link speed to 32 GT/s. This process eliminates stepping through the intermediate data rates of 8 GT/s and 16 GT/s.

There are two variants of the optional equalization flow introduced in PCie GEN5.

**Equalization Bypass to 32 GT/s**

- PCIe GEN5 introduces the ability to bypass equalization at the highest data rate, thereby removing the requirement of performing equalization at every data rate ≥ 8 GT/s. Negotiating this option during LTSSM means that there is no equalization needed in 8 GT/s or 16 GT/s. If the link works in 32 GT/s and then has reliability issues, then the link will need to perform equalization at 8 GT/s and then 16 GT/s.
- Within the equalization process there are effectively three main sub-phases – Phase1, Phase2, and Phase3 – for a downstream port and an additional phase – Phase0 – for an upstream port. If the downstream port is satisfied by the Phase1 equalization outcome, then it can skip Phase2 and Phase3 and complete the equalization process. This further reduces the simulation time spent in equalization while ensuring link reliability.
No Equalization Needed

- The option to not perform equalization at all is another simulation time saver. In this case the equalization parameters for transmitters and receivers are applied from a previously negotiated equalization process. These parameters are stored in persistent storage when equalization is once performed. These values are then applied directly during link reset such that equalization is not needed at all and link reliability is not compromised because no equalization was performed during LTSSM.

When verification engineers develop a testbench environment specifically for verifying DMA features, it is crucial that they configure the LTSSM parameters of the DUT and the configuration settings of the verification IP used in the testbench in-sync so that both devices can successfully transition the LTSSM states in step with each other and achieve PCIe link up in a reduced amount of simulation time.

Fine tuning these configuration parameters for both devices can become quite cumbersome and is an error-prone task, especially if these parameters are not known to the testbench developer. In this case, having a DUT and verification IP that provide a highly configurable design component becomes an absolute necessity to achieve the desired optimization.

PCIe QVIP provides a well-documented standard set of APIs to access LTSSM related configuration variables which include training sequences OS counters, timeouts, and the ability to configure LTSSM state and sub-state specific timeout configurations. With the ability to configure a varied set of LTSSM parameters in QVIP, it is imperative to keep the use-model as simplistic as possible. For ease of use, the default settings of these configurations are chosen such that the QVIP achieves an optimized LTSSM transition for PCIe link up. Having a highly configurable QVIP and optimized default setup greatly improves the usability of the Verification IP in a testbench.

PCIE DEVICE DISCOVERY AND CONFIGURATION SPACE SET UP PROCESS

This section elaborates on the PCIe device discovery process and configuration space set up performed by the host software from the root port in order to configure the end point device and DMA engine after the PCIe link is established. This step of device configuration is called the enumeration process. This step requires a series of Type0 configuration read and write TLPs issued by the root port in order to assign a bus number and device id to uniquely identify the end point device and configure the configuration space registers for all physical functions present in it.

In a functional verification environment, QVIP is configured as a root port connected to the PCIe DUT. The enumeration process is a lengthy sequence of configuration reads and writes that the QVIP performs. With QVIP built-in features and capabilities, this process can be reduced significantly by reducing the number of configuration reads and writes after link up, resulting in shorter simulation runtime set up.

Let us now inspect the enumeration process in more detail. For a PCIe topology where a root port is connected to an end point device, the enumeration process begins with sending a configuration read request — with bus number = 1, device number = 0, and function number = 0 — to read the vendor ID of the end point device. On receiving a successful completion for this, the configuration read confirms that there is a device connected to the root port.

The next step is to discover the type of device connected, by reading the header type field in the configuration space registers (shown in Figure 2 on the following page). For an end point device the header layout field of the header type register should read Type 0 Configuration Space Header. This confirms that the device connected to the root port is an end point device. The multi-function device field of this register indicates that the device may contain multiple physical functions.
From this point on, the enumeration process accesses the 4KB configuration space registers for each physical function of an end point device to determine the PCIe capabilities supported by a physical function as well as the memory resource requirements of the physical function by the host.

The steps below describe QVIP’s enumeration flow for end point device discovery.

1. Read all the base address registers (BAR), starting at offset 10h, to determine the memory space region and the memory size requirements for this physical function.
2. Read the capability pointer at offset 34h. PCIe uses a link-list structure to access the standard device capabilities and extended capabilities registers supported by a physical function. Here is a list of essential capabilities defined in the configuration space registers of a physical function (note: this is not a complete list of capabilities structures):

   - **PCIe Base Capabilities**
     - PCI Power Management Capability Structure
     - PCI Express Capability Structure
     - MSI/MSI-X Capability Structures

   - **Extended Capabilities**
     - SR-IOV
     - Advanced Error Reporting
     - Power Budgeting
     - Latency Tolerance Reporting
     - L1 PM Sub-states
     - Enhanced Allocation
     - Physical_Layer_16GT/s
     - Lane Margining Receiver
     - Physical_Layer_32GT/s
     - Alternate Protocol

3. After reading the complete capabilities list present in the configuration space registers by traversing the nodes of the linked list, the enumeration sequence follows a similar approach for physical functions supporting SR-IOV. The SR-IOV capability defines a set of lightweight...
PCIe functions, called virtual functions, that share one or more physical resources with the physical function. The enumeration sequence then follows a similar process of discovering the virtual functions supported by the physical function.

4. Finally, the enumeration sequence now starts configuring the device by issuing a series of configuration writes to set up the device, based on the settings provided by the user in the QVIP agent configuration and the capabilities it discovered. This series of configuration writes are targeted specifically to set up the following:

- Initialization of BAR addresses for all the physical functions and virtual functions based on its memory requirements.
- Initialization of different device capabilities like power-management, max-payload size, maximum read request size, and read completion boundary.
- Enable bus-mastering capabilities of the device to initiate transactions on the PCIe bus.
- Initialization of MSI/MSI-X addresses for the devices.

While executing the enumeration sequence, the QVIP is also maintaining a data structure per the physical function it contains in order to utilize this information for user-specific test scenario creation after the enumeration is complete. This feature enables QVIP to easily execute extensive verification scenarios based on design capabilities, by providing the test writer with APIs to query design capabilities and provide address offsets for updating the configuration space registers in the device.

The number of configuration transactions executed in the enumeration sequence has a multiplication factor dependent on the number of physical functions and virtual functions per device. This setup phase needs to be performed for every test that uses the PCIe link for verifying DMA functionality. As a result, simulation runtime increases before any actual user-specific test scenario is executed. Reducing the simulation runtime by lowering the number of configuration transactions, significantly improves the set up time needed for a device.

QVIP provides two verification capabilities for enumeration sequences to reduce runtime dramatically.

**Fast Enumeration**

In fast enumeration mode, the QVIP is configured through a backdoor mechanism while the DUT is configured through configuration writes only. The advantage here is that the configuration reads for the configuration space registers do not take place, instead the QVIP does configuration writes to configure and set up the device. In this mode, runtime is reduced by half or even more (since configuration writes are fewer in number than the configuration reads performed during the enumeration sequence).

In this mode, configuration reads are not performed by QVIP. Still, the device capabilities information and memory resource requirements to perform configuration writes are needed. This crucial information is provided to QVIP using built-in utilities to accurately capture the required settings in a testbench usable format. The following are the steps needed to extract this information and feed it back into QVIP.

1. Run a test case, with default full enumeration mode setting:

   ```
   <agent cfg handle>.agent_descriptor.auto_bring_up.enum_mode = PCIE_FULL_ENUM;
   ```

2. Enable the following commands to capture the configuration space register settings of the device:

   ```
   <agent cfg handle>.agent_descriptor.auto_bring_up.bus_enum_setting.print_fast_bus_enum_setting = 1'b1;
   ```

3. Run the test and then open the simulation log to find the setting captured by QVIP, which will be used in the configuration phase of the test:
The output between the banner’s FAST_BUS_ENUM CONFIGURATION in the simulation log is directly copied into the testbench for configuring the QVIP through the backdoor. Once the above settings are applied, the test case configuration is complete and ready to run in fast enumeration mode.

Providing QVIP with the above settings ensures that no further configuration read is necessary for accessing device capabilities. QVIP will now only perform the necessary configuration writes needed to set up the device for normal operational mode.

**Backdoor Enumeration**

In backdoor enumeration mode, configuration reads and writes are not performed at all. Configuration space registers for QVIP and the device are configured through a backdoor mechanism. The enumeration sequence in this mode is not performed.

This feature is dependent on the DUT to be able to update the configuration space register settings through a backdoor mechanism before the link is trained. PCIe design IP built with this capability can take advantage of this feature in QVIP and reduce the initial simulation runtime even further, as compared to fast enumeration.

The steps to extract the configuration space settings are similar to the fast enumeration mode with minor updates in the configuration option assigned to the PCIe QVIP agent.

1. Run a test case with the default full enumeration mode setting:

   ```
   <agent cfg handle>.agent_descriptor.auto Bring_up
   .enum_mode = PCIE_FULL_ENUM;
   ```

2. Enable the following commands to capture the configuration space register settings of the device:

   ```
   <agent cfg handle>.agent_descriptor.auto Bring_up
   .bus_enum_setting
   .print_bk_door_enum_setting = 1'b1;
   ```

   For this mode, output between the banner’s BACKDOOR CONFIGURATION in the simulation log is directly copied into the testbench for configuring the QVIP through the backdoor. Once the above
settings are applied, the test case configuration is complete and ready to run in backdoor enumeration mode.

Providing QVIP the above settings ensures that no configuration read is necessary for accessing the device capabilities, and QVIP assumes that since the user is running the test with the backdoor enumeration option, the configuration write is also not necessary.

When running QVIP in this mode, after the link is established the user can start initiating test scenarios with the assurance that QVIP and the device have completed the enumeration process.

To summarize, initial PCIe link training and the enumeration process is an essential part of every test for verification of DMA engines using PCIe. With the techniques described in the preceding sections, the simulation runtime required to establish an operational PCIe link at 32 GT/s is drastically reduced. Furthermore, the new equalization bypass mode in the PCIe GEN5 specification also helps reduce link training time. These optimizations ensure that the PCIe design set up is not compromised and does not impact the verification capability of the testbench. On average, taking advantage of the most optimized settings in a QVIP assisted testbench, the simulation runtime to establish a PCIe link is reduced by twenty percent. In one such typical case, link training time was reduced from 61 microseconds to 13 microseconds. This reduction in simulation runtime boosts the productivity of the verification engineer developing tests and reduces the overall turnaround time for debug and analysis.

QVIP USE CASE WITH THE PLDA PCIE CONTROLLER FOR DMA APPLICATIONS

For 20 years, PLDA has been a pioneer of PCIe technologies. PLDA created the XpressRICH-AXI product line a few years back, and they now propose a fifth generation, running at PCIe 5.0 speed (32 GT/s).

XpressRICH-AXI is a configurable and scalable PCIe controller Soft IP, as illustrated in figure three on the following page. It is designed for ASIC and FPGA implementation, which is compliant with the PCI Express 5.0, 4.0, and 3.1/3.0 specifications. The IP can be configured to support endpoint, root port, and dual-mode topologies, allowing for a variety of use models, and it exposes a configurable, flexible AMBA AXI interconnect interface to the user. Users may optionally enable the built-in DMA engine, or connect an external DMA engine, such as PLDA’s vDMA-AXI DMA, depending on their application requirements.
The extreme configurability and scalability of PLDA’s XpressRICH-AXI sub-system IP raised their verification challenges. PLDA decided to use Mentor QVIP to test standard compliance for PCIe and AMBA/AXI. With Mentor QVIP, PLDA discovered a flexible and reliable tool for building its proprietary test suite on a highly-scalable testbench.

For the XpressRICH-AXI verification, PLDA uses the following Mentor QVIP features:

- Up to 64bit PIPE width (different widths at different link speeds)
- Up to 16 PCIe lanes
- Up to PCIe 5.0 speed
- PLDA MAC is ARI device => PLDA supports 32 physical function number
- SR-IOV
- Power management
- External clock and reset

For DMA applications, XpressRICH-AXI implements up to eight highly-configurable scatter-gather DMA engines. These engines can transfer data between PCIe and AMBA AXI4 interfaces. This functionality is itself a challenge, as PCIe transaction layer features need to be managed, with certain information carried in AXI4 packets. The DMA engine must take into account the PCIe Max Payload and Max Read Request sizes, as well as the BME bit in end point device mode. The DMA must also handle the number of outstanding requests via tag management. Some TLP parameters may need conversion. For DMA verification, PLDA uses the following QVIP features:

- SR-IOV – DMA engine resources are sharable between multiple physical functions/virtual functions
- MSI/MSI-X – at the end of DMA or at error occurrence (per physical function/virtual function)
- Checking MSI data and Requester ID
- PASID prefix support
- Carrying PASID information
- ECRC generation/check
- Error Injection scenarios
- Errors occurred while fetching SG descriptor or data: completion with UR/CA status; completion timeout; poisoned completion; completion with ECRC error
• Reporting of error events in AER
• Checking AER registers
• Check the error messages sent to host by an end point device
• Testing the affect of FLR on DMA traffic (ongoing)
• Fast bus enumeration sequence for multiple implemented functions
• Scaled FC feature is mandatory for devices supporting GEN4 or above:
  • Using scaled credits to size RX buffer
  • Checking UFC DLLPs with QVIP
• 10-bit tag feature is mandatory for devices supporting GEN4 or above:
  • Checking the TLP tags with QVIP

Example scenario of error detection and logging:

1. DMA fetches a descriptor from PCIe domain by issuing an MRD TLP
2. QVIP returns a completion which is poisoned (end point bit set)
3. PCIe controller detects the error and reports to DMA, logs the error in AER, sends error message if end point
4. DMA sends MSI/MSI-X to report that error has occurred (using specific MSI vector)
5. Continued operation of DMA is either permitted or not (depending on DMA transfer parameters)

CONCLUSION

XpressRICH-AXI combines a PCI Express 5.0 controller with complete AXI interconnect and DMAs. This complex architecture requires a highly-scalable and configurable testbench for verification. The high flexibility of Questa VIP was key to creating custom testbenches from scratch that can dynamically adapt to the different IP topologies and configurations, mixing PCIe interfaces with multiple AXI interfaces. PLDA and Mentor have collaborated throughout the different PCIe generations, enabling advanced leadership on the latest technologies together.
INTRODUCTION
As SoC developers adopt RISC-V and the design freedoms that an Open ISA (Instruction Set Architecture) offers, DV teams will need to address the new verification challenges of RISC-V based SoCs. The established SoC verifications tasks and methods are well proven, yet depend on the industry wide assumption of ‘known good processor IP’ based on the quality expectations associated with IP providers such as Arm or MIPS Technologies. However, the new DV challenges are not purely focused on the processor IP, since an Open ISA allows much greater design freedom whose impact extends well into the SoC itself.

With RISC-V there are possibly four new verification challenges to address for SoC projects:

1. Verification of the RISC-V processor IP (including different source options)
2. Verification of the Processing Element (PE) containing the RISC-V core(s) (especially relevant in SoCs with a fabric designed for AI processing)
3. Connection of the processor itself or the PE to the network on chip (NoC)
4. Multiple PEs communicating through the NoC to each other

For the processor core verification, three different scenarios exist: the processor IP (RTL) can be purchased from one of the many processor IP vendors in the RISC-V community, the processor IP can be downloaded from one of the open source repositories or the SoC developer can build the processor RTL from scratch. In the first two situations, there will have been a significant amount of verification performed on the processor IP; however, it is almost a certainty that the processor IP will have less verification and less maturity than a core from a traditional processor IP vendor. Also, if custom instructions are added to the core, no matter the source of the original RTL, the core needs to be thoroughly verified for both the new features and to confirm the original base core quality has not been compromised.

In many AI (Artificial Intelligence) architectures, the design is structured in a hierarchy with a processing element consisting of one or more CPUs, plus AI accelerators or co-processor(s), plus some additional logic to connect to the SoC AI fabric. This is a critical feature to support the desired applications, and offers convenient abstraction levels to align the verification methods. Next, while the processor IP and PE have been verified, and the NoC has been verified (assuming that an existing NoC IP is used), the interaction of the RISC-V processor, PE and the NoC is unique to the design and requires verification.

Last, while the verification of a single PE is needed, verifying multiple PEs working with each other through the NoC is also needed. This is especially true in the case of designs based on RISC-V, since the Open ISA flexibility allows for optimization of each of the cores, so all the various combinations of PEs will need verification as well.

These verification challenges can be addressed within the general framework of the UVM verification methodology and tools, however, some innovation is needed, along with collaboration between processor IP vendors, EDA vendors, other tool developers and the RISC-V SoC developers. For example, several test generation and instruction stream generation tools have been developed to address RISC-V specific requirements, and new directed test suites have been developed for specific RISC-V extensions such as the vector instructions. New reference models are needed for the RISC-V processors and PEs. New metrics are needed, especially for the processor and PE verification areas, perhaps such as instruction coverage. Flows with these tools need to be robust to handle the variety of processor IP scenarios elaborated above. Plus, a robust flow is needed to ensure that a solitary “bad actor” does not insert a backdoor into the processor or SoC.
Other areas that are being looked at to address RISC-V SoC verification include using hybrid emulation-virtual platform systems for hardware-software co-verification, using Portable Stimulus (PSS) for multiple PE and full chip verification, and using the nature of the AI algorithms to constrain the SoC state space.

In this article, the verification challenges for RISC-V SoCs are discussed and an overview given of potential solutions. Specific verification flows including new test and instruction stream generators, and reference models and metrics, are presented in detail including the results of using these flows on real processor IP and SoC designs. Mentor Questa® is fundamental to the RISC-V processor verification, with the RTL of the processor DUT (Device Under Test) and Imperas’ RISC-V golden reference model encapsulated in the SystemVerilog UVM testbench for lock-step comparison and testing.

COMPLIANCE IS NOT VERIFICATION

With RISC-V, as an open ISA specification, any implementation will need to be tested against the latest RISC-V compliance suite. The objective of the compliance process is to ensure that implementations are correctly following the specifications, with the expectation that compliant devices will exhibit sufficient compatibility to leverage the emerging ecosystem for tools and software. Put more simply, compliance is confirming that the designers have understood the specifications. Since the ISA specification does not include details of microarchitecture, differences in device performance and application focus are expected and of course permitted. Since the compliance tests use expected functionality as the basis of the test suite, this incurs an overlap with some aspects of Design Verification (DV). However, the compliance suite is not exhaustive for all functionality and is focused purely with the structural specification aspects of the ISA, i.e. compliance is a subset of DV. The RISC-V Compliance Suite is developed within the RISC-V International Working Group on Compliance (“Compliance WG”), and the latest test suites are available from the RISC-V compliance GitHub repository.

CUSTOM INSTRUCTIONS AND REFERENCE MODELS

A reference model is a key to processor-related DV tasks. This is usually an instruction accurate (IA) model of the processor, often called an Instruction Set Simulator (ISS). The Compliance WG GitHub repository includes the riscvOVPsim ISS. The riscvOVPsim simulator implements the full and complete functionality of the RISC-V Unprivileged (formally known as User) and Privilege...
specifications. The simulator is command-line configurable to enable/disable all current optional and processor specific options in the RISC-V specification. The simulator is developed, licensed and maintained by Imperas Software Ltd., and is fully compliant to the Open Virtual Platforms (OVP) open standard APIs. Most recently, support for the vector and bit manipulation instructions were added to the OVP RISC-V processor models. When custom instructions are added to the RISC-V processor, those instructions need to be added to the reference model. Imperas has previously developed a methodology for profiling and analysis for custom instructions, and the outline for this flow is shown in figure 1 on the previous page. The resulting model, which includes both the standard RISC-V instructions and the custom instructions, can then be used as a reference model for DV of the processor RTL.

PROCESSOR VERIFICATION

There are three techniques currently being used for RISC-V processor verification: directed tests, constrained random test generation and test generation and execution.

Directed Tests

Directed test suites are an established technique, however, what has not been previously done is the measurement of instruction coverage of these test suites. Also, with the vector extensions to the RISC-V ISA, the difficulty involved in building a comprehensive test suite is increased exponentially. For example the RISC-V vector engines have 90 different possible configurations, and nearly 500 instructions. This is obviously a complex problem.

With this article, the authors report new instruction coverage metrics, with the coverage tool included in the ISS. An example of the coverage results for the RV32I compliance test suite is shown in figure 2.

The authors are also developing a directed test suite (“Vector Test Suite”) for the RISC-V vector instructions. Data on this test suite will be available later in 2020.

Constrained Random Test Generation

Constrained random test generation for SoC DV is also an established technique. However, for processor DV, this needs to be an Instruction Stream Generator (ISG). Google has developed and made open source an ISG for RISC-V.

Figure 3 shows the basic flow for RISC-V processor DV using the ISG. This flow was originally developed to use a trace or signature compare methodology, but is now being evolved to support a step-and-compare methodology using the ISS encapsulated in SystemVerilog, as shown in figure 4.

The encapsulation of the RISC-V reference model within SystemVerilog allows direct interaction with the testbench environment. When an issue is uncovered, a direct debug and analysis can be initiated. In an automated test environment, action can be taken to minimize the lost efficiency in continuing beyond the point of failure in a typical log compare approach. In addition, the testbench can be extended within SystemVerilog to use stimulus objects to expected response objects as both an aid for debug and a more exhaustive DV test plan. With the help and assistance of
the experts at Mentor under the Vanguard program, the SystemVerilog extensions have been set-up to support the close and efficient coupling with the Imperas OVPsim simulator.

Figure 5 on the following page shows the step-and-compare flow. This flow has been implemented for the testing of the Ibex core that was originally developed by ETH Zurich under the name “Zero-riscy” and was recently adopted by LowRISC as Ibex$. Ibex implements the RISC-V RV32IMC instructions, which is the 32-bit RISC-V processor with integer (I), multiplier/divider (M) and compressed (C) instructions.

Table 1 on the following page shows the different categories of bugs found in the Ibex processor using this approach, while figure 6, also on the following page, shows an example of the types of bugs found.
An alternative, and complementary approach to test generation for processor DV, which also can be applied to SoC DV, is the generation of tests as an executable which can be run on the RTL. In this approach, tests are randomly generated, then run on the processor reference model. The results from running the tests on the processor reference model are then combined with the random tests and used as reference test results. This flow is shown in figure 7.

Test Generation and Execution

Table 1. Categories of bugs found using the ISG-based DV flow

<table>
<thead>
<tr>
<th>Bug Category</th>
<th>% of Bugs Found</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug Mode</td>
<td>31.3%</td>
</tr>
<tr>
<td>Illegal/Hint Instructions</td>
<td>25.0%</td>
</tr>
<tr>
<td>Interrupt</td>
<td>18.8%</td>
</tr>
<tr>
<td>Memory Access Fault</td>
<td>12.5%</td>
</tr>
<tr>
<td>Pipeline Issue</td>
<td>6.3%</td>
</tr>
<tr>
<td>Others</td>
<td>6.3%</td>
</tr>
</tbody>
</table>
PE, MULTIPLE PE AND NOC-PE VERIFICATION

The key pieces of SoC DV include verification of single PEs, verification of multiple PEs and verification of the interface between the PE and the Network on Chip (NoC) as shown in figure 8.

Verification of Processing Elements (PE)

In many RISC-V based SoCs targeted at AI applications, the architecture includes Processing Elements (PEs) which have more than one RISC-V processor, plus an AI accelerator, plus some custom logic. The custom logic is typically comprised of custom instructions added to the RISC-V processors, plus additional logic for controlling the communications between processors. For these PEs, the individual processors (including custom instructions) must be verified as discussed above. However, with the integration of multiple processors into the PE, there can be different interactions that cannot be tested at the individual processor level.

For this level of integration, the PE can also be modeled using the same instruction accurate techniques that were used to model individual processors. The OVP APIs are used to build a model of the PE, and the same SystemVerilog encapsulation techniques are used to encapsulate the model of the PE, again enabling step-and-compare verification of the PE RTL.

A key piece here is the test generation. Certainly, constrained random test generation could work, however, this might spend too many cycles “re-verifying” the individual processors and not focusing
on the new, unique interactions at the PE level of integration. Another possibility is to run actual software meant to execute on the real PEs. This should bring out these additional interactions.

In these AI architectures, often one PE interacts regularly with multiple neighbor PEs. It is unclear how best to verify these PE-PE interactions. Two ideas being explored now are 1) to combine the instruction accurate models with RTL simulation; and 2) to combine the instruction accurate models with hardware emulation. In the first scenario, one might have one PE represented in RTL, and the remainder of the PEs as IA models. This could enable the IA models to run the actual software, generating more interesting “stimuli” for testing the RTL PE. In the second scenario, something similar to the first is contemplated. However, in this situation the RTL blocks would be implemented in the hardware emulator. Such a hybrid IA simulation-emulation environment is shown in figure 9.

**Processor/PE–NoC Verification**

Verification of the interface between a processor or processor subsystem and a NoC is a well-established process. This element of RISC-V verification is raised because there has been only a limited number of RISC-V based SoCs built using the various NoCs, so DV engineers should realize that this is not the fully mature NoC interface that one receives when other processor architectures are used.

**CONCLUSIONS**

RISC-V is generating some significant attention in many market segments and applications. The freedom of the Open ISA and custom extensions together with a framework of ecosystem support provides system designers and SoC architects new options and flexibilities for optimised processor implementations. The SoC DV teams will need to accommodate processor verification in addition to addressing the flexibilities that will affect the complete SoC verification task. To maintain project schedules and tapeout deadlines verification methodologies will need to adapt and evolve to accommodate the coming wave of new SoC design complexity.

**ACKNOWLEDGMENTS**

The authors wish to thank Richard Ho and Tao Liu of Google LLC for their help with the Google ISG, Valtrix for their support of STING, and the support the Imperas team received under the Mentor Questa® Vanguard program.

**ABOUT IMPERAS**

Imperas is revolutionizing the development of embedded software and systems and is the leading provider of RISC-V processor models and virtual prototype solutions. Imperas, along with Open Virtual Platforms (OVP), promotes open source model availability for a spectrum of processors, IP
vendors, CPU architectures, system IP and reference platform models of processors and systems ranging from simple single core bare metal platforms to full heterogeneous multi-core systems booting SMP Linux. All models are available from Imperas at www.imperas.com and the Open Virtual Platforms (OVP) website at www.ovpworld.org.

END NOTES

1. RISC-V International ISA specification at https://riscv.org/specifications/
2. RISC-V International Compliance GitHub repository: https://github.com/riscv/riscv-compliance
3. OVP (Open Virtual Platforms) http://www.ovpworld.org
4. Google ISG GitHub repository at https://github.com/google/riscv-dv
5. Ibex by LowRISC https://github.com/lowRISC/ibex
Addressing VHDL Verification Challenges with OSVVM

by Jim Lewis, SynthWorks Design, Inc.

INTRODUCTION

Most people don’t think of VHDL as a verification language. However, with the Open Source VHDL Verification Methodology (OSVVM) utility and verification component libraries it is. Using OSVVM we can create readable, powerful, and concise VHDL verification environments (testbenches) whose capabilities are similar to other verification languages, such as SystemVerilog and UVM.

This article covers the basics of using OSVVM’s transaction-based test approach to write directed tests, write constrained random tests, use OSVVM’s generic scoreboard, add functional coverage, add protocol and parameter checks, add message filtering, and add test wide reporting.

WHY VHDL? WHY OSVVM?

According the 2018 Wilson Research Group Functional Verification Study:

- 62% of FPGA designs worldwide use VHDL
- 17% of FPGA verification projects worldwide use OSVVM (or 38% of VHDL FPGA verification projects)
- For Europe, 30% of FPGA verification projects use OSVVM while only 20% use UVM

This makes OSVVM the #1 VHDL FPGA verification methodology worldwide and the #1 FPGA verification methodology in Europe.

BENEFITS OF OSVVM

For the VHDL community, OSVVM is a clear win. We can write tests in the same language we already know and re-use components, tests, and testbenches from other projects. More importantly OSVVM’s transaction-based approach simplifies creating readable and reviewable tests (an important metric in the safety critical community). In addition, OSVVM uses the same component/model based approach used by RTL design. Hence, not only can RTL designers read tests and verification components, they can write them. While having independent design and verification teams is important, it is also important to be able to deploy engineers to either a design or verification role on a project by project basis.

WHAT ARE TRANSACTIONS?

A transaction is an abstract representation of an interface operation (such as UART transmit) or directive (such as get transaction count). In OSVVM, a transaction is initiated with a procedure call. In the OSVVM verification component approach, the procedure places the transaction information into a record and passes it to the verification component. The component in turn executes the transaction and provides stimulus to the device under test (DUT).

Figure 1 shows two calls to a send procedure and the corresponding waveforms produced by the UartTx verification component.

```
UartTcpProc : process
begin
WaitForBarrier(StartTest) ;
Send(UartRxRec, X"4A") ;
Send(UartRxRec, X"4B") ;
end
```

Serial Data

| Send X"4A" | Send X"4B" |

Figure 1: Two Calls to Send transaction and the resulting waveform

Each verification component in the OSVVM library implements a set of model independent transactions. The table at the top of the following page gives a brief summary.
THE OSVVM TESTBENCH FRAMEWORK

The OSVVM testbench framework looks identical to other frameworks, including SystemVerilog. It includes verification components (AxiMaster, UartRx, and UartTx) and TestCtrl (the test sequencer) as shown in figure 2. The top level of the testbench connects the components together (using the same methods as in RTL design) and is often called a test harness. Connections between the verification components and TestCtrl use VHDL records as an interface. Connections between the verification components and the DUT are the DUT interfaces. Tests are written by calling transactions in TestCtrl. Separate tests are separate architectures of TestCtrl.

The rest of this article focuses on writing tests in TestCtrl.

<table>
<thead>
<tr>
<th>Bus Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write(TransactionRec, 'AAAA', 'DDDD');</td>
</tr>
<tr>
<td>Read(TransactionRec, 'AAAA', DataOut) ;</td>
</tr>
<tr>
<td>ReadCheck(TransactionRec, 'AAAA', 'DDDD');</td>
</tr>
<tr>
<td>Streaming or Serial Transactions (AxiStream, UART, …)</td>
</tr>
<tr>
<td>Send(TransactionRec, 'DDDD');</td>
</tr>
<tr>
<td>Get(TransactionRec, DataOut);</td>
</tr>
<tr>
<td>Check(TransactionRec, 'DDDD');</td>
</tr>
</tbody>
</table>

Common Directive Transactions

GetTransactionCount(TransactionRec, Count);

Table 1: OSVVM Standard Transactions

TESTCTRL, THE OSVVM TEST SEQUENCER

The TestCtrl architecture consists of a control process plus one process per independent interface, see the code block in figure 3 below. The control process is used for test initialization and finalization. Each test process creates interface waveform sequences by calling the transaction procedures (Write, Send, …).

Each architecture of TestCtrl creates a separate test in the test suite. Hence, a single test is visible in a single file, improving readability.

Since the processes are independent of each other, synchronization is required to create coordinated events on the different interfaces. This is accomplished by using synchronization primitives, such as WaitForBarrier (from TbUtilPkg in the OSVVM library).

architecture UartTx1 of TestCtrl is

begin
  ControlProc : process
  begin
    . . .
  end process;

  . . .
  WaitForBarrier(TestDone, 5 ms);
  ReportAlerts;
  std.env.stop;
  end process;

  CpuTestProc : process
  begin
    . . .
    WaitForBarrier(TestInit);
    Write( . . . );
  end process;

end architecture UartTx1 of TestCtrl;
TEST INITIALIZATION
The ControlProc both initializes a test and finalizes a test. Test initialization is shown in figure 4. SetAlertLogName sets the test name. Each verification component calls GetAlertLogID to allocate an ID that allows it to accumulate errors separately within the AlertLog data structure. Accessing the IDs here allows the message filtering of a verification component to be controlled by the test. WaitForBarrier stops ControlProc until the test is complete.

A SIMPLE DIRECTED TEST
A simple test can be created by transmitting (send) a value on one interface and receiving (Get) and checking (AffirmIfEqual) it on another interface. This is shown in figure 5.

USING RANDOMIZATION
Constrained random randomly selects test values, modes, operations, and sequences of transactions. In general, randomization works well when there are a large variety of similar items to test.

The OSVVM package, RandomPkg, provides a library of randomization utilities. A subset of these is shown in figure 7 on the following page.
An OSVVM constrained random test consists of randomization plus code patterns plus transaction calls. For example, the code in figure 8 generates a UART test with normal transactions 70% of the time, parity errors 10% of the time, stop errors 10% of the time, parity and stop errors 5% of the time, and break errors 5% of the time.

Hence, creating constrained random tests in OSVVM is simply a matter of learning the patterns. All of the pattern is written directly in the code, and hence, visible to review.

Constrained random introduces two issues to our testing. First, how do we self-check the test?

Previously we recreated the transmit pattern on the receive side. Due to the complexity, this would be tedious and error prone. In the next section, we solve this problem by using OSVVM’s generic scoreboards.

Second, how do we prove the test actually did something useful? We solve this problem by using OSVVM’s functional coverage.

OSVVM’S SCOREBOARDS

A scoreboard facilitates checking data when there is latency in the system. A scoreboard receives the expected value from the stimulus generation process and checks the value when it is received by the check process, as shown in figure 9.

The OSVVM scoreboard supports small data transformations, out of order execution, and dropped values. It uses package generics to allow the expected type and actual type to differ. The “match” function that determines if the expected and actual values match is also a package generic. The FIFO-like data structure of the scoreboard is created internal to a protected type.

The use model for OSVVM’s scoreboard is shown in figure 10 on the following page. The scoreboard instance is created using a shared variable declaration. On the transmit side (TxProc), the expected value is pushed into the scoreboard (SB.Push), and then a transaction is transmitted (Send). On the receive side (RxProc), the transaction is received (Get), and then the received value is checked in the scoreboard (SB.Check). This greatly simplifies RxProc since it no longer reproduces what the transmit side did. Scoreboards can also be used to simplify checking in directed tests.

```
-- Random Range: randomly pick a value within a range
Data_slv8 := RV.RandSLvl(Min => 0, Max => 15, 8);

-- Random Set: randomly pick a value within a set
Data1 := RV.RandInt( (1,2,3,5,7,11) ) ;

-- Weighted distribution: randomly pick a value between
-- 0 and N-1
-- where N is number of values in the argument
-- the likelihood of each value = value / (sum of all values)
Data2 := RV.DistInt( (70, 10, 10, 5, 5) );
```

Figure 7: Subset OSVVM’s Random library

An OSVVM constrained random test consists of randomization plus code patterns plus transaction calls. For example, the code in figure 8 generates a UART test with normal transactions 70% of the time, parity errors 10% of the time, stop errors 10% of the time, parity and stop errors 5% of the time, and break errors 5% of the time.

```
TxProc : process
  variable RV : RandomPType;
  ...
  for I in 1 to 10000 loop
    case RV.DistInt( (70, 10, 10, 5, 5) ) is
      when 0 =>  -- Nominal case 70%
        ErrorMode := UARTTB_NO_ERROR;
        TxD := RV.RandSlv(0, 255, Data'length);
      when 1 =>  -- Parity Error 10%
        ErrorMode := UARTTB_PARITY_ERROR;
        TxD := RV.RandSlv(0, 255, Data'length);
      when . . .  -- (2, 3, and 4)
        end case;
      Send(UartTxRec, Data, ErrorMode);
    end loop;
```

Figure 8: An OSVVM Constrained Random Test

Hence, creating constrained random tests in OSVVM is simply a matter of learning the patterns. All of the pattern is written directly in the code, and hence, visible to review.

```
Figure 9: Scoreboard Block Diagram
```

```
The OSVVM scoreboard supports small data transformations, out of order execution, and dropped values. It uses package generics to allow the expected type and actual type to differ. The “match” function that determines if the expected and actual values match is also a package generic. The FIFO-like data structure of the scoreboard is created internal to a protected type.

The use model for OSVVM’s scoreboard is shown in figure 10 on the following page. The scoreboard instance is created using a shared variable declaration. On the transmit side (TxProc), the expected value is pushed into the scoreboard (SB.Push), and then a transaction is transmitted (Send). On the receive side (RxProc), the transaction is received (Get), and then the received value is checked in the scoreboard (SB.Check). This greatly simplifies RxProc since it no longer reproduces what the transmit side did. Scoreboards can also be used to simplify checking in directed tests.
```
**ADDING FUNCTIONAL COVERAGE**

Functional coverage is code that tracks items in the test plan. As such it tracks requirements, features, and boundary conditions.

If a test uses constrained random, functional coverage is needed to determine if the test did something useful. Going further as design complexity increases, functional coverage is recommended to assure that a directed test actually did everything that was intended.

There are two categories of functional coverage: item (aka Point) coverage and cross coverage. Item coverage tracks relationships within a single object. For a UART, were transfers with no errors, parity errors, stop bit errors, parity and stop bit errors, and break errors seen?

Cross coverage tracks relationships between multiple objects. For a simple ALU, has each set of registers for input 1 been used with each set of registers for input 2?

Why not just use code coverage that is provided with a simulator? Code coverage only tracks code execution. Hence, code coverage cannot track the examples above since the information is not in the code. On the other hand, if a design's code coverage does not reach 100% then there are untested items and testing is not done. Hence, both code coverage and functional coverage are needed to determine when testing is done.

Functional coverage in OSVVM is implemented as a data structure within a protected type.

Figure 11 continues with RxProc from the constrained random test and adds functional coverage. First an instance of the coverage object (RxCov) is created using a shared variable. Next "RxCov.AddBins (GenBin(N))" is called to construct the functional coverage model. The value "N" corresponds to the integer representation of the UART status bits for Break, Stop, Parity, and Data Available. The calls to AddBins all complete at time 0, before any stimulus is generated or checked. Next, after the received stimulus has been retrieved (using Get), RxCov.ICover(RxErrorMode) is called to record the coverage. At the end of the test, RxCov.WriteBin prints the coverage results.

Figure 10: OSVVM Scoreboard Use Model

Figure 11: UART RxProc with functional coverage added
ADDING PROTOCOL AND PARAMETER CHECKERS

OSVVM alerts are used to check for invalid conditions on an interface or library subprogram. Alerts both report and count errors. Alerts have the levels FAILURE, ERROR (default), and WARNING. By default, FAILURE level alerts cause a simulation to stop. By default, ERROR and WARNING do not cause a simulation to stop. When a test completes, all errors reported by Alert (and AffirmIf) can be reported using ReportAlerts.

Figure 12 shows a protocol checker used in a memory model to detect if a write enable (iWE) and read enable (iOE) occur simultaneously while the memory is addressed (iCE). Parameter checkers are similar to protocol checkers and check for invalid parameters to library programs.

Alerts can be enabled (default) or disabled via a call to SetAlertEnable. The stopping behavior of Alert levels can be changed with SetAlertStopCount. Figure 13 shows the usage of both of these.

Figure 12: Memory Model Protocol Checker

SimultaneousAccessCheck: process
begin
wait on iCE, iWE, iOE;
AlertIf(SramAlertID, (iCE and iWE and iOE) = '1',
"nCE, nWE, and nOE are all active");
end process SimultaneousAccessCheck;

Figure 13: Usage of SetAlertEnable and SetAlertStopCount

-- Turn off Warnings for a verification component
SetAlertEnable(UartRxAlertLogID, WARNING, FALSE);

-- If get 20 ERRORs stop the test
SetAlertStopCount(ERROR, 20);

TEST FINALIZATION

Test finalization is error checking and reporting that is done in ControlProc after test completion. This is shown in figure 16. Finalization starts when "WaitForBarrier(TestDone, 5 ms)" resumes. This happens either when all of the test processes have called their corresponding WaitForBarrier (normal completion) or 5 ms passes. The 5 ms is a test timeout (watch dog) that activates if one of the test processes did not complete properly. The sequence of calls to AlertIf check for proper test finish conditions. ReportAlerts prints test results (see Test Wide Reporting).

ControlProc : process
begin
\ldots
WaitForBarrier(TestDone, 5 ms);
AlertIf(TBID, NOW >= 5 ms, "Test timed out");
AlertIf(TBID, not SB.Empty, "Scoreboard not empty");
AlertIf(TBID, GetAffirmCount < 1, "Checked < 1 items");
ReportAlerts;
wait;
end process ControlProc;

Figure 16: Test Finalization
TEST WIDE REPORTING
The AlertLog data structure tracks FAILURE, ERROR, and WARNING for the entire test as well as for each AlertLogID (see GetAlertLogID). ReportAlerts prints a test completion message using this information. If GetAlertLogID was not called during the test, ReportAlerts prints either the simple PASSED or FAILED message shown in figure 17.

If GetAlertLogID was called during the test, ReportAlerts will include errors and passed for each AlertLogID as shown in figure 18.

INCLUDING OSVVM LIBRARY
OSVVM provides context declarations (VHDL-2008) to allow the utility library and each verification component to be referenced with a single context reference, rather than multiple use clauses. This is shown in Figure 19.

GETTING AND RUNNING OSVVM
OSVVM is available on GitHub at https://github.com/OSVVM. Retrieve it using git as shown in figure 20.

The Axi4Lite, AxiStream, and UART verification components come with OSVVM style testbenches. Figure 21 shows how to compile and run the tests for the AxiStream verification component. The tests for the UART and Axi4Lite verification components are run in the same manner.

SUMMARY
OSVVM goes well beyond the basics shown in this article. To learn more, see the documentation on the GitHub site, or take SynthWorks' Advanced VHDL Testbenches and Verification class.
REFERENCES
The metrics to measure the effectiveness of Safety Mechanisms include code coverage rate, SPFM (Single-point failure metric) and LFM (Latent failure metric). Especially in SPFM and LFM, if the specified value is not reached on the Fault Injection Simulation (using Gate Level) at the end of verification, it will cause iterations, which will cause a significant increase in time and cost compared to consumer LSIs.

A method for efficiently performing a logic simulation of the Safety Mechanism will be described.

**QUANTITATIVE GOALS OF LSI VERIFICATION FOR ISO 26262**

In an ISO 26262 compliant LSI, it is required to achieve the quantitative target value of hardware metrics for each ASIL (Automotive Safety Integrity Level). The hardware metrics include PMHF (Probabilistic Metric for Random Hardware Failures) and hardware architecture metrics. The combination of SPFM and LFM is called the hardware architecture metric. In terms of design quality, the target value of code coverage is 100% for circuits that include the Safety Mechanism. PMHF still depends on the failure rate of the LSI.

The quantitative target values of SPF and LPF for each ASIL are as follows.

<table>
<thead>
<tr>
<th>Single-point fault metric</th>
<th>ASIL B</th>
<th>ASIL C</th>
<th>ASIL D</th>
</tr>
</thead>
<tbody>
<tr>
<td>≥90%</td>
<td></td>
<td>≥97%</td>
<td>≥99%</td>
</tr>
</tbody>
</table>

Table 1: Possible source for the derivation of the target “single-point fault metric” value

ASIL A: Does not target “single-point fault metric” value

SPFM and LFM in an ASIC/ASSP are calculated using Fault Insertion Simulator for Functional Safety using Gate-Level; however, if the target value is not achieved, the Safety Mechanism must be modified/revised. When carrying out revisions, it is necessary to comply with the change management regulations of the standard. Compared with ordinary consumer products ASIC/ASSP, this can take an enormous amount of time, which significantly extends the development period and increases costs for safety-critical designs.

With RTL verification of the logic circuit, including the Safety Mechanism by improving the verification completeness, the risk of failure occurring during Fault Insert Simulator for Functional Safety can be greatly reduced.

**Definitions:**

- **Single-point fault**: A hardware fault in an element that leads directly to the violation of a safety goal and no fault in that element is covered by any safety mechanism
- **Latent fault**: A multiple-point fault whose presence is not detected by a safety mechanism nor perceived by the driver within the multiple-point fault detection time interval
- **Random hardware failure**: A failure that can occur unpredictably during the lifetime of a hardware element and that follows a probability distribution
- **Element**: A system, components (hardware or software), hardware parts, or software units

<table>
<thead>
<tr>
<th>Latent fault metric</th>
<th>ASIL B</th>
<th>ASIL C</th>
<th>ASIL D</th>
</tr>
</thead>
<tbody>
<tr>
<td>≥60%</td>
<td></td>
<td>≥80%</td>
<td>≥90%</td>
</tr>
</tbody>
</table>

Table 2: Possible source for the derivation of the target “latent-fault metric” value

ASIL A: Does not target “latent-fault metric” value
CAUSES AND COUNTER MEASURES FOR DETERIORATION OF HARDWARE ARCHITECTURE

If a safety mechanism cannot detect when a fault occurs, the hardware architecture metric is degraded. A fault occurs due to transistor destruction, breakage of aluminum wiring, or the like. To verify the safety mechanism, we model a fault outbreak and use the model in simulation to ensure that the safety mechanism works properly. There are two types of models to create: Single Point Fault Model and Latent Fault Model. To check the operation of two types of Fault Models, we must check the Functional Coverage of the output of the Fault Models. In the verification for improving the hardware architecture metric, it is necessary to confirm the functional completeness of the safety mechanism by functional coverage using the created model. The configuration image is shown in figure 1. Verify and debug using Questa® Advanced Simulator using the environment pictured below.

CREATING A VERIFICATION ENVIRONMENT AND EXECUTING VERIFICATION

To improve the hardware architecture metric, the steps for creating the verification environment and the verification execution, including the goals to be achieved, are described in the following lists.

Steps for creating the verification environment:

1. Creating a base verification environment
   Create a verification environment that does not consider the verification of functional safety as the base. The Safety Mechanism is built in the DUT (Device Under Test).

2. Creating a Single Point Fault model
   Create an element failure functional model (EFFM) for the intended failure based on the results of FMEA (Failure Mode and Effect Analysis). A Single Point Fault is a Fault that occurs only once in the element. The EFFM needs to create a model that works for all FMEA items. Acquire functional coverage of the output of the behavior model in order to confirm that all the intended behaviors have been verified.

3. Creating a Latent Fault model
   Conduct a safety analysis on the safety mechanism. A safety-mechanism failure function model (SFFM) is created for the intended failure of the Safety Mechanism from the safety analysis results. A Latent Fault is a fault that occurs in the Safety mechanism. A Latent Fault occurs in the Safety Mechanism during a Single Point Fault. However, only one occurs. This must be integrated with the Single Point Fault model. The SFFM should be modeled for all intended behavior for safety analysis. Acquire functional coverage of the output of the behavior model in order to confirm that all the intended behaviors have been verified.

4. Creating a verification environment
   Incorporate the EFFM and the SFFM into the verification environment. When incorporating, the functional coverage of the EFFM and the SFFM is output to Scoreboard. Output the Functional Coverage of Safety Mechanism input/output to the Scoreboard.
Steps for executing verification and goals:

5. Verification when functional safety failure does not occur
   No malfunction occurs in normal operation without the Fault Model. The safety mechanism does not operate unless functional safety failures occur. If the output of a Safety Mechanism is activated with EFFM or SFFM inactive, there is a problem in the verification environment or the Safety Mechanism. Functional coverage is 100% in the absence of functional safety failures. Also, it is exactly the same as the expected value of the DUT. Deactivate the EFFM and the SFFM and execute verification.

6. Verification goal when functional safety failure does not occur – the goals are:
   • It exactly matches the expected value of the DUT
   • Functional coverage is 100% under the condition that no functional safety failures occur
   • The output of the Safety Mechanism must not be active
   • The Functional Coverage of the input of Safety Mechanism is less than 100%
     If the above conditions are not met, debug the DUT and verification environment.

7. Verification when a single point fault occurs
   This is an SPFM test. For each test where a functional safety failure does not occur, EFFM causes one fault in the element. This test does not cause a Latent Fault. Activate the EFFM and deactivate the SFFM and execute verification.

8. Verification goal when a single point fault occurs – the goals are:
   • It exactly matches the expected value of the DUT
   • The output of the Safety Mechanism must be active
   • The Functional Coverage of the input of Safety Mechanism is 100%
   • The coverage that is a function of the EFFM is 100%
     If the above conditions are not met, debug the DUT and verification environment.

9. Verification when a Latent Fault occurs
   This is an LFM test. Latent Fault occurs in Safety Mechanism after Single Point Fault occurs for one test in which functional safety failure does not occur. Latent Fault requires a combination of Single Point Fault tests. Only one Latent Fault occurs in the Safety Mechanism. Activate the EFFM and the SFFM and execute verification.

10. Verification goal when a Latent Fault occurs – the goals are:
    • It exactly matches the expected value of the DUT
    • The output of the Safety Mechanism must be active
    • The Functional Coverage of the input of Safety Mechanism is 100%
    • The coverage that is a function of EFFM is 100%
    • The function coverage of the SFFM is 100%
    • The cross coverage of EFFM and SFFM is 100%
     If the above conditions are not met, debug the DUT and verification environment.

11. Code Coverage goal:
    • Code Coverage must be 100%, with approved exclusions
     If the above conditions are not met, debug the DUT and verification environment.

SUMMARY
The following is a summary of the four items for logical verification of an ISO 26262 compliant LSI.

Verification environment
An ISO 26262 compliant LSI must meet the goal of the hardware architecture metrics (SPFM, LFM). In order to verify the hardware architecture metric, a hardware architecture metric verification environment must be created. The contents added to the construction of the verification environment for consumer products LSI are described below.

1. Creating a single point fault model (EFFM)
2. Creating a latent fault model (SFFM)
3. Incorporation of single point fault model and latent fault model into the verification environment

Verification flow
An ISO 26262 compliant LSI must meet the goal of the hardware architecture metrics and metrics must
be verified. The processes added to the consumer product LSI verification flow are described below.

1. Single point fault simulation
2. Latent fault simulation

**Single point fault/Single point fault model and the Latent fault/Latent fault model**
The features of the Single point fault / Single point fault model and the Latent fault / Latent fault model are described below.

1. Single point fault / Single point fault model
The detection rate of Single point fault is Single point fault metric (SPFM). The single point fault model is for testing the single point fault metric; and, the generation behavior of the single point fault is modeled.

**Feature**
- The cause of the single point fault is transistor breakdown or aluminum wire disconnection
- It is necessary to create a single point fault model that models transistor breakdown and aluminum wiring disconnection
- Single point fault is a fault that occurs only within the element
- Only one single point fault occurs in one test
- The safety mechanism is a module that finds a single point fault
- In FMEA, all the high-risk features need to be modeled

2. Latent fault / Latent fault Model.
The detection rate of Latent fault is Latent fault metric (LFM). The latent fault model is a model for testing the latent fault metric; and, the generation behavior of the latent fault is modeled.

**Feature**
- The cause of the latent fault is transistor breakdown or aluminum wire disconnection
- It is necessary to create a latent fault model that models transistor breakdown and aluminum wiring disconnection
- Latent fault is a fault that occurs only within the safety mechanism
- Latent fault is a fault that occurs when a single point fault occurs
- Only one latent fault occurs in one test
- Even if a latent fault occurs, the Safety Mechanism must detect a single point fault
- It is necessary to perform a safety analysis of the Safety Mechanism and model the analysis items based on the analysis results

**Verification goal**
Each verification and verification goal is summarized below.

1. Verification goal when functional safety failure does not occur.
   - It exactly matches the expected value of the DUT
   - Functional coverage is 100% under the condition that no functional safety failure occurs
   - Safety Mechanism output is always non-active
   - Safety Mechanism does not have 100% functional coverage

2. Verification goal when a single point fault occurs.
   - It exactly matches the expected value of the DUT
   - The output of the Safety Mechanism must be active
   - The Functional Coverage of Safety Mechanism is 100%
   - The Functional Coverage of Single Point Fault Model is 100%

3. Verification goal when a Latent Fault occurs.
   - It exactly matches the expected value of the DUT
   - The output of the Safety Mechanism must be active
   - The Functional Coverage of Safety Mechanism is 100%
   - The Functional Coverage of Single Point Fault Model is 100%
   - The Functional Coverage of Latent Fault Model is 100%
   - The cross coverage of Single Point Model and Latent Model is 100%

   - Code Coverage must be 100%

Perform each of the above verifications and debug using Questa® Advanced Simulator so as to achieve the above stated goals.
VERIFICATION ACADEMY
The Most Comprehensive Resource for Verification Training

33 Video Courses Available Covering
• Functional Safety
• UVM Framework
• UVM Debug
• Portable Stimulus Basics
• SystemVerilog OOP
• Formal Verification
• Metrics in SoC Verification
• Verification Planning
• Introductory, Basic, and Advanced UVM
• Assertion-Based Verification
• FPGA Verification
• Testbench Acceleration
• Power Aware Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 12,800 questions asked
Verification Patterns Library

www.verificationacademy.com