Case Study: Comparison between Conventional VHDL and UVM Test-Benches for a Slave I²S Transceiver

Mustafa Khairallah  
Boost Valley Consulting  
mustafa.khairallah@boostvalley.com

Mohamed Mahmoud  
Boost Valley Consulting  
mohamed.mahmoud@boostvalley.com

Abstract – In this survey, we prove that the Universal Verification Methodology, UVM, is not only efficient in verifying large-gate-count IP-based System-on-Chip designs, but it is also efficient in verifying small designs, in comparison with the conventional verification techniques, specifically VHDL test-benches. We have built both a UVM verification environment and a VHDL test-bench to verify the operation of an I²S slave transceiver, which is a relatively small design in terms of gate count. It utilizes 197 LUTs on Xilinx Spartan 6 FPGA. We provide a comparison between the two approaches in both the development and runtime phases.

Index Terms – Functional verification, UVM, FS, VHDL, test-bench

I. INTRODUCTION

During the past decade, the time spent by systems-on-chip developers in functional verification has risen to 60% or more on some projects. Even developers of smaller chips and FPGAs are having problems with the past verification approaches [1].

To enhance the functional verification, many proven and promising technologies have been developed, such as: simulation, emulation, object-oriented programming, OOP, constrained random stimulus, coverage based verification, formal verification …etc. It is worth mentioning that many of these techniques are based on the capabilities of the System Verilog language which combined the RTL capabilities of Verilog and the verification capabilities of open Vera language developed by Synopsys. System Verilog was adopted as an IEEE Standard in 2005 [2].

The problem with the above statement is that we have too many verification techniques [1]. This has led to several problems, among these problems:

1) Miscommunication: Different teams use different verification techniques. Consequently, the communication between teams is hard. Moreover, it is hard to get a new team member used to the techniques adopted by this team.

2) Reusability Problem: As there is no standard way to do things, it is hard to reuse parts of a project; either horizontally in other projects or vertically in the same project.

This is where a methodology appears. We need to get teams and engineers to do the same things in the same ways. A methodology provides guidance on when, where, why and how each technique should be applied for maximum efficiency [1]. It also provides building-block libraries, documentation, coding guidelines and lots of examples. A methodology lets the verification engineer focus on the verification planning and test effort rather than complex test-bench architecture creation.

The Universal Verification Methodology, UVM, was announced by Accellera, a standards organization specialized in electronic design automation and IC design and manufacturing. It is a complete methodology that includes the best practices for efficient and exhaustive verification.

In this paper, we explore the key features of UVM. We also provide a practical example to prove that UVM is not only more efficient than the conventional verification technique in large-gate-count IP-based System-on-Chip, SoC, designs, but in small designs as well [1]. We also prove that some of the features that empower VHDL in the field of RTL design; limit its capabilities when it comes to test-bench development.

To do so, we chose to build both UVM and conventional test-benches for an I²S (Inter-IC Sound) slave transceiver. The I²S standard is a standard for PCM audio communications between ICs [3]. It has a relatively small design as it deals only with audio signals. The DUT we are verifying utilizes 197 LUTs on Xilinx Spartan 6 FPGA. However, it has more than one clock domain and some complex timing requirements.

In section II, we provide some historical background on the development of functional verification methodologies. In section III, we provide a short introduction to UVM key features as described in [4]. In section IV, we provide a short introduction about the I²S standard. In section V, we describe the test-bench architecture we are implementing. In section VI, we compare between the UVM and conventional test-benches during the development stage. In section VII, we compare between the two test-benches during the runtime.

II. HISTORICAL BACKGROUND

The Universal Verification Methodology, UVM, was introduced to the user community by Accellera in 2010 [1]. However, it was not the first effort towards a unified verification methodology.

In 2000, Verisity Design, now Cadence Design Systems, Inc., introduced a collection of best known verification practices. It was targeted towards the e user community. Later, in 2002, Verisity introduced the first verification library called the e Reuse Methodology, eRM. It included packaging
guidelines, architecture requirements, messaging facilities, an object mechanism, reset solution, scoreboard example and more. It would still define the content and functionality of verification methodologies used until now [1].

In 2003, Synopsys announced its Reuse Verification Methodology library, RVM, for the Vera verification language. It didn’t include architecture guidelines and was considered as a subset of the eRM. Over time, it was converted into the System Verilog Verification Methodology Manual, VMM, supporting the evolving System Verilog Standard [1].

In 2006, Mentor introduced its Advanced Verification Methodology, AVM. It was the first open-source methodology and the first methodology to adopt the SystemC Transaction-Level Methodology standard [1].

Meanwhile, Cadence acquired Verisity in 2005 and started converting the eRM to System Verilog, introducing the Universal Reuse Methodology. Not only did it include the proven capabilities of eRM, but it also used TLM and was open source [1].

In 2008, Cadence and Mentor collaborated to release the Open Verification Methodology, OVM. The impact of OVM was great as it was the first multi-vendor methodology tested against different vendors’ simulators. This was important due to the fact that System Verilog was in the early stages and many of its constructs lacked clarity [1].

The collaboration in OVM proved to be a very good solution, which made Synopsys collaborate with Cadence and Mentor to introduce a unified methodology. In 2010, OVM 2.1.1 was chosen as the basis for the UVM standard. It is tested by all vendors and no more technical comparisons between VMM and OVM are needed. UVM is currently an Accellera standard. It represents an alignment on verification methodology across the industry, supported by the major EDA suppliers and their eco-systems.

III. UVM

The UVM is a complete methodology that codifies the best known verification practices. One of its key principles is to produce reusable verification components called Universal Verification Components, UVCs. It is targeted to verify both the small designs and large IP-based SoCs [1].

The key features of UVM are [1]:

1) Data Design

The UVM provides the ability to clearly partition your verification environment into a set of data objects and components. Moreover, it provides means for setting and getting data values hierarchically, textually printing and graphically viewing objects and automating commonplace activities, such as copying, comparing and packing items, which we will refer to later as transactions. This allows engineers to focus on what objects contain and how they work, instead of the supporting code.

2) Stimulus Generation

The UVM provides infrastructures and built-in stimulus generation that can be customized to include user-defined transactions and transaction sequences. These sequences can be randomized and controlled based on the current state of the DUT, interface or previously generated data.

3) Building and Running Reusable Test-Benches (Test/Test-bench Separation)

The UVM includes well-defined build flows for creating reusable verification environments. Moreover, it includes configuration mechanisms that allow customizing the runtime behavior without modifying the original implementation. This is beneficial when creating a test-bench for a design with different IPs, interfaces or processors.

4) Coverage Model Design and Checking Practices

The UVM includes the best-known practices for incorporating functional coverage, in addition to protocol and data checks, into a reusable UVC.

5) User Example [4]

The UVM library and user guide include a golden example, based on an understandable, yet complete, protocol called the UBus.

Moreover, UVM provides a framework to achieve coverage-driven verification, CDV. It combines automatic test generation, self-checking test-benches and coverage metrics. It eliminates the efforts and time spent in creating hundreds of tests and ensures thorough verification using up-front goal setting [4].

A UVM test-bench is composed of UVCs. Each UVC is an encapsulated, ready-to-use, configurable verification environment for an interface protocol, sub-module or a full system [4]. A UVC consists of a sequencer and a driver for stimulating the design, a monitor for monitoring the pin-level activity and scoreboard for checking. It can optionally contain a coverage collector. Consequently, UVM enables the verification process to be divided into three different levels, as shown in figure 1.

![Fig. 1 The UVM environment different development levels](image)

In addition to all these features, the fact that UVM adopts the TLM 2.0 standard eases the reuse. In addition to the UVM factory methods, it empowers the configurability of the UVM
environment, providing the ability to replace any of the UVCs or their internal components at the runtime.

Moreover, UVM enables the creation of a high-level, object-oriented model of memory maps using the UVM register layer [4].

IV. I$^2$S STANDARD

As digital audio equipment have held a huge part of the audio devices market, and in order to achieve flexibility for both the equipment and IC manufacturers, standardized communication structures are vital.

The I$^2$S standard, also known as Inter-IC Sound standard, is an electrical serial bus interface standard for communicating PCM audio data between ICs and electronic devices [3]. It separates the clock and serial data signals resulting in lower jitter than other communication protocols that recover clock from the data stream.

The I$^2$S bus carries only audio signals, while other signals such as control signals are transmitted separately. The bus consists of at least three lines:

1) Bit/System Clock Line BCLK/SLCK.
2) Word Clock Line/ Word Select/ Left Right Clock WS.
3) At least one multiplexed data line SD.

The master I$^2$S module generates the clock signals for both the transmitter and the receiver. However, in complex systems there may be several transmitters and receivers, which make it difficult to define the master. In such systems, there is usually a system master controlling digital audio data-flow between the various ICs.

The serial data is transmitted serially in two’s complement format with the MSB first. The serial data line carries two time-multiplexed data channels.

The word select line indicates the channel being transmitted:

1) WS = 0; channel 1 (left).
2) WS = 1; channel 2 (right).

V. TEST-BENCH ARCHITECTURE

As we described in section IV, the I$^2$S standard has different clock signals and clock domains. It also has several constraints on how the data is sent, received and extracted, how the width of a data word is extracted from the clocking signals and how data-width violations are handled. To reduce the complexity of such issues in our UVM test-bench, we have made use of the test/test-bench separation feature in UVM; the test environment deals with all these requirements in an organized way, drives the DUT interface with the stimulus, monitors the responses and checks for the correctness of received/transmitted data, while the test sequences generate random data packets with different widths for both the transmitter and receiver.

In figure 2, we provide a block diagram of the UVM test-bench architecture.

![Fig. 2 I$^2$S UVM Test-Bench Block Diagram](image)

The environment consists of the following UVCs:

A. Transmitter (TX) Driver

It has the following functions:

1) Word Clock Generation [3]: It generates the correct word clock required by the transmitter according to both the system clock (generated in the top level test) and the width generated by the TX sequence.
2) Driving Data to be sent by the transmitter.

B. TX Monitor

It acts as an I$^2$S slave receiver. It extracts data from the bit-stream sent by the transmitter, packs it and sends it along with the original transaction to the scoreboard.

C. RX Driver

It has the following functions:

1) Word Clock Generation: similar to the TX Driver.
2) Acts as a slave transmitter: It calculates the two’s complement of the data fields in the transactions generated by the RX sequence and coverts them into a serial bit-stream compliant with I$^2$S standard.

D. RX Monitor

It monitors the data extracted by the receiver, packs it and sends it along with the original transaction to the scoreboard.

E. TX/RX Scoreboard

The scoreboards check for the correctness of the data sent/received by the transmitter/receiver respectively, counts the number of faulty packets, reports errors and terminate the test when a certain number of test cases is applied.

It is worth mentioning that the I$^2$S is a simple standard in terms of operating and control modes (its complexity arises from the timing requirements of the clocking signals along with the serial data streams) so we do not need to perform coverage-driven verification. Consequently, we do not need a coverage collector in our architecture. It might be argued that a UVM without a coverage collector doesn’t express the full power of UVM. However, it would be unfair to compare between a conventional VHDL test-bench and a coverage-driven UVM test-bench. As we need our case study to provide a fair performance and development comparison between both techniques, we need the two test-benches to do the same things
in the same way. Yet it is worth mentioning that coverage-driven verification adds great power to the UVM.

The remaining part of the test is TX/RX Sequences. Each generates a certain number of test cases in each supported width by the DUT. It also generates packets to test the extreme cases of the data registers, such as: all-ones and all-zeros.

On the other hand, the VHDL test-bench performs mostly the same testing technique, with some adaptation due to the differences between VHDL and System Verilog. Among the differences:

1) As VHDL doesn’t support test/test-bench separation, we have to perform data generation and DUT driving in the same process.

2) There is no test termination mechanism in VHDL. Consequently, the test is ended through a forced error when the number of test cases reaches a certain value [5].

3) There is no randomization algorithm in VHDL, so we have had to build our own randomization function based on the randomization algorithm in [6].

4) Different UVCs in UVM correspond to different concurrent processes in VHDL.

VI. COMPARISON: DEVELOPMENT

While developing each of the test-benchs, several issues appeared that may affect the decision of a design/verification team hesitant about adopting UVM. We are going to list and explain these issues below:

1) Inter-Process Communication:

In VHDL, a process cannot use a signal as a computing variable [5]. This is because the signal does not get updated until the next delta cycle, which will be discussed in a while. Consequently, if we assign a signal at the beginning of a process and use that value of the signal later in the process, without any wait statements in between, the simulator will have only the old value at that point. To solve this problem, we can use variables instead of signals. However, while variables provide a good alternative for solving this problem, VHDL does not support global variables, so, to transfer a value from one process to another, it has to be assigned to a signal. To sum up the problem, we have to use variables for in-process computations while we have to use signals for inter-process communication.

These two features empower VHDL in the field of RTL design. A signal in VHDL directly maps to either a wire or a register in hardware. It is important to restrict its assignment to avoid having undesired latches. On the other hand, a variable in VHDL does not have a specific hardware meaning and it might not map to a certain register, latch or wire in the synthesized hardware [5].

However, such features introduce a considerable headache for the test-bench developer.

On the other hand, UVM does not experience such problem. As UVM adopts the TLM 2.0 [8] standard as a standard way of communication between UVCs [4], it performs inter-process communication to a higher level of abstraction that is not aware of hardware details. Also UVM defines a set of API to be used for communication between UVCs in a standard way.

2) Delta Delay in VHDL Simulation [9]:

This issue also arises from the nature of VHDL as it was designed for hardware modelling and not for verification. It is due to the fact that VHDL simulators (or, more generally, event triggered simulators) separate signal assignment from signal update. Delta delay is the difference between a signal assignment and a signal update. If no delay is introduced in the VHDL code, this delta delay corresponds to zero simulation time. However, this operation corresponds to 2 software events.

What actually happens is that the simulator calculates all the signal assignment expressions at a certain delta cycle, and after all expressions are resolved the simulator updates the values. Again, this is beneficial when dealing with hardware at the RTL level as it prevents zero delay feedbacks and infinite loops. However, in terms of simulation, it increases the processing time of test-benchs and prevents having any infinitely running processes in parallel with the rest of the test-bench.

On the other hand, as UVM is built upon an Object-Oriented Programming language which is System Verilog, and as it operates in the TLM abstraction layer instead of the RTL layer, only parts of the transactors (drivers, monitors and responders) operate on delta cycles basis, while the rest of the test-bench operates as an ordinary software program. Not only does this important feature of UVM lead to a major performance enhancement in comparison with VHDL as we will discuss in the following section, but it helps the test-bench developer to treat each UVC independently, so that some UVCs, such as the scoreboard, or UVM objects such as sequences, can operate without the need of being aware of the simulation time.

This result helps the test-bench developer to separate different functionalities in different classes. For examples, a UVM test-bench has two different classes for stimulus generation (which is an infinite loop independent of the simulation time, only controlled by the sequencer) and stimulus driving. Another example is that the scoreboard can operate on a transaction basis, not needing to be aware of any clock signal event.

VII. COMPARISON: PERFORMANCE

At this point, we need to compare between the two test-benchs in terms of performance. We define three performance metrics:

1) Runtime.

2) Compilation Time.

3) Coverage Closure Time.

However, as the case we are studying in this paper does not support coverage-based verification, we are only concerned with the first two metrics.

A. Runtime:
After developing both the VHDL and UVM test-benches, we have run both for different numbers of test-cases \( \{20, 200, 2000, 20000, 200000\} \). We measured the runtime for both the test-benches for each of the previous numbers of test-cases. The results are shown in Table 1 and figures 3 and 4.

![Runtime results for each of the VHDL and UVM test-benches (logarithmic scale for the number of test cases only)](image1)

![Runtime results for each of the VHDL and UVM test-benches (logarithmic scale for both axes)](image2)

<table>
<thead>
<tr>
<th>No. of Test Cases</th>
<th>Runtime of the VHDL Test-Bench</th>
<th>Rate of Change</th>
<th>Runtime of the VHDL Test-Bench</th>
<th>Rate of Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>5</td>
<td>0.0611</td>
<td>13</td>
<td>0.0444</td>
</tr>
<tr>
<td>200</td>
<td>16</td>
<td>0.0622</td>
<td>21</td>
<td>0.0027</td>
</tr>
<tr>
<td>2000</td>
<td>128</td>
<td>0.0662</td>
<td>88</td>
<td>0.0034</td>
</tr>
<tr>
<td>20000</td>
<td>1320</td>
<td>0.0662</td>
<td>836</td>
<td>0.0042</td>
</tr>
</tbody>
</table>

Table 1: Runtime results for each of the VHDL and UVM test-benches (Time in seconds)

From the previous graph we can conclude the following conclusions:

1) Regardless of the verification technique used, the runtime increases exponentially with a slow rate with the number of test-cases.

2) UVM seems to be slower at very small sets of test cases. This is justified as at such level the overhead of the UVM library is larger than the processing need for the test-bench itself.

3) After a certain threshold value, approximately 200 test cases in our case, when the execution time of the VHDL test-bench becomes equal to the constant overhead of the UVM library in addition to the execution time of the UVM test-bench, the UVM becomes much faster than the VHDL, and the gap between the two increases by increasing the number of test cases.

Of course such results are unique for the I2S example in our hands and cannot be generalized over all UVM/VHDL pairs of test-benches. However, due to the simplicity and small size of the I2S standard, we can assume that these results will be the same for a wide range of (almost all) DUTs.

**B. Compilation time**

Measuring the compilation time of the each of the two test-benches, it appears that the VHDL test-bench has much lower compilation time. However, this is not the only metric for comparing the compilation time. Due to the encapsulation provided by the OOP nature of System Verilog, we do not need to compile the whole test bench when we change one of the components. This is not the case with the VHDL test-bench, as we need to compile the whole test-bench every time we make a change.

**VIII. CONCLUSION**

According to the results we presented in this paper, we have proved that UVM is more efficient in comparison with the conventional verification method. The UVM test-bench is easier to develop due to several UVM features; TLM, UVM Configurations, Reusability and CDV. The simulation runtime performance is also enhanced as UVM operates at an object-oriented software level rather than the slow RTL level. In short, UVM is actually a universal methodology suitable for almost all digital design projects.

**REFERENCES**


