A comprehensive approach to scalable framework for both vertical and horizontal reuse in UVM verification

Roman Wang
Sr. Design Verification Engineer, Advanced Micro Devices Inc. Shanghai, China

Abstract—The complexity of today’s multi-core SoCs can be addressed only if the main challenges in verification are handled successfully. With increasing adoption of UVM, each IP could be well verified in a block level UVM testbench and environment. SoC functional verification involves integrated IP blocks, and there will be more challenges such as integration checking, multiple complex scenarios and use case driven verification in different SoC design stage. The key points are to improve the verification quality at SoC level in early phases of design integration. To do this, the IP level's testbench, including sequences must be highly reused properly during SoC level complex tests, especially the use case driven verification. To reduce cycle time and effort, both vertical (from IP to sub-system to SoC level) and horizontal (from project to project) reuse need to be ensured. In this paper, we will show a comprehensive approach from an architecture point of view to create a scalable framework to address those key challenges with very little effort across multiple aspects of today’s SoC functional verification process. UVM Shared Physical Memory Dynamic Memory Allocation Manager "SDMAM" techniques are also used for this framework. This paper assume you are adopting Cadence VIP, UVM methodology, and AMBA protocol for both fabric and main bus.

1. INTRODUCTION

One of our biggest challenges is building a flexible and scalable testbench framework which is essential for a robust verification environment and reduces the reuse effort from block to SoC level. Setting up a constrained random test environment, however, can seem like a difficult task, especially when you consider that environments need to be flexible, scalable, and reusable. The framework for constrained random verification requires more planning and structure, but the benefits in the end are well worth the investment.

In this paper, we show a well-architected framework based on UVM.

1.1 Paper scope

The focus of this paper is the reusable architecture. We do not discuss the actual implementation of process because the
implementation details would be project-specific. We believe that it is first necessary to understand how to architect a scalable framework before delving into the details of how to implement the solutions.

1.2 Paper organization

In Section 2, we describe the challenges in using UVM in today’s ARM-based SoCs. Section 3, the bulk of the paper, describes important considerations when architecting a scalable framework from IP to sub-system to SoC; although this paper does not focus on implementation details, we do in Section 4 discuss important implementation considerations. Section 5 provides some concluding thoughts on what to expect after adopting framework proposals.

2. Challenges in Using UVM in IP/Sub-system/Soc levels

SoC design requires extensive and diverse verifications due to factors such as large design size, variety of verification views, and cooperation among the engineers with different experiences. To handle the problems of SoC verification effectively, it’s common to divide the SoC design into smaller pieces according to design hierarchy. In general, there are two verification levels before full-chip verification: IP level verification and sub-system level verification. The testbench for IP or sub-system level verification should be reused in its upper level or SoC verification.

2.1 UVM in IP level

Each IP block is generally developed by its own team focusing on a specific requirement. When the blocks are integrated, the SoC verification team needs to debug and analyze the system and IP without the knowledge of IP specification. Each IP will be a black-block to SoC team, so all features of each IP need be fully verified at the IP level and to the greatest extent possible, delivered the bug-free IP to SoC team. The IP level testbench should be architected for reuse in the sub-system testbench.

2.2 UVM in sub-system level

This level of verification verifies interconnects of each IP block, functionality of fabric, and system bus utilization. Shared memory allocation (which uses UVM SDMA [1] technique) may need to be examined to determine whether the SoC functions as specified. Figure 1 gives an example of this case, depicting multiple complex IP blocks containing external memory sub-system.
2.3 UVM in SoC level

The greater number and increasing complexity of IP blocks being integrated into SoSc have brought many new challenges in understanding what SoC functionality has been verified. Some of the more challenging questions include:

- How to reuse the IP level testbench at SoC level with very little effort?
- How to verify all IP blocks are connected properly?
- How to verify “system-level” features that include interactions between multiple IP blocks are working correctly?
- How to finish the project within limited time and reduce bug-escapes?
- How to synchronize the IP environment with that of the core at SoC?
- How to simulate test scenarios at SoC level using the IP environment?
- Were the complex programming requirements for a particular IP block verified?
- Were the various IP block power management features properly verified?

3. How to address these challenges

A common UVM-based testbench could be divided into three major parts. The first is the top module which simply instantiates the DUT and interfaces for communication with components in main testbench. The second is the testbench which is the main part and includes all testbench components such as UVM verification component (UVC), register model, and virtual sequencer. The third is the test scenario,
which defines the input stimulus based on UVM virtual sequences. All components in the testbench and test scenario parts are objects of classes normally inherited from the UVM base class library.

In SoCs, there are many masters and slaves. They need to be well verified in UVM IP level. Figure 2 gives an example of master IP testbench case, depicting an AHB DMA master programmed via the UVM RGM package[2] (using UVM SDMAM for random memory region allocation) through APB UVC, IP IRQ is handle by CORE UVC. Cadence AHB VIP is acting a memory controller with memory model to respond to the AHB DMA master.

Figure 2: Master IP Testbench

Figure 3 gives an example of a slave IP testbench, depicting a memory salve programmed via the UVM RGM package through Cadence AXI UVC; IP IRQ is handle by CORE UVC. The SDMAM is working on AXI UVC’s sequence not on UVM RGM. This is because the UVM RGM sequences are used to program the slave DUT, and AXI UVC sequences use SDMAM to manage the random regions for memory access.
If your IP protocol contains multiple layers, you may decide to architect multiple separate unit testbenches for each protocol layer as in Figure 4. For example, inside each testbench, the unit level teams may independently develop UVCs for the protocol stack layer: a transaction UVC, data link UVC and physical layer UVC. The teams may develop their entire stimulus at each layer without taking into account data coming from the higher layers. This will result in stimulus that can't be reused easily among the testbenches, and that is difficult to reuse at the system level.

The “UVM stack” concept could make it easier to reuse each of the upper layer’s UVC and their associated stimulus libraries be reused in subsequent testbenches. For example, the transaction's UVC's sequences developed in Environment1 could be reused in Environment 2 and Environment3. In addition, the transaction UVC developed in Environment1 is reused in Environment2 and Environment3. A stacked UVC is a standard UVC that includes the flexibility to optionally be connected to other UVCs. An SV “adapter package” is used to setup an interconnection strategy for UVCs. To build a complete UVC stack testbench, refer to the UVM stack user guide [3].
If your team has VMM VIP on hand, you have to integrate the VMM VIP in the UVM environment (Figure 5). The VMM/UVM interoperability kit is a requirement for the implementation, obtain it by completing the request from http://www.vmmcentral.org/cgi-bin/interopkit/req1.cgi.

**UVM Environment**

To make IP UVC reusable on the sub-system and SoC levels, we introduce XUVCs which contain the CORE XUVC, XM UVC and XS UVC. When building sub-system and SoC level UVC verification testbenches, the CORE (e.g. ARM core) should be the stub. All traffic is generated by CORE XUVC (Figure 6). The CORE XUVC’s config could decide which VIP should be used based on Specific SoC requirement.

If your SoC need AXI main bus and AHB debug port from (ARM) CORE, then you could enable the active AHB UVC and AXI UVC. Figure 7 describes the Master XUVC and Figure 8 describes the Slave XUVC. If Master/Slave XUVC is configured to stand-in mode, the DUT will use the real RTL and the active AMBA UVC will drive the system AMBA bus. If
Master/Slave XUVC is configured to stand-by mode, the module level UVM testbench will be reused to this layer, and the DUT will be replaced with the stub which need the weak driving for all output signals to avoid the multi-drive case. In the sub-system, only the RGM register layer UVC is active; others should be passive. In the SoC, all UVCs should be passive, the CORE UVC will take charge of register programming. The stand-by mode will run quickly, because only the AMBA OVC drives the bus not DUT RTL.

Figure 6: CORE UVM XUVC

Figure 7: Master UVM XUVC

Figure 8: Slave UVM XUVC

Figure 9 describes one sub-system example. To verify the system integration quickly at an early design stage, all masters’ XM UVCs are configured to stand-in mode; only memory controller DUT is configured to stand-by mode and others are configured to stand-by. It needs the real memory devices working on sub-system verification. All masters’ register sequences could use the SDMAM to make the random the regions write
and read accessing. PW SVF scoreboard [4] is used for multi-stream checking with in-order or out-of-order rules on the sub-system system bus.

Figure 9: Sub-system Testbench

Figure 10: UVM SoC Testbench
Figure 10 describes one SoC example. To verify the system level scenarios, all masters’ and slaves’ XM UVCs are configured to standby mode, and the RGM register sequences are reused at the SoC level and controlled by CORE XUVC which is taking charge of the register programming and interrupt handling. SDMAM helps every master get the random regions for specific requirements, and then it could make all random regions absolutely exclusive in shared memory space. The PAD strap OVC will be connected to PADs on top of the SoC layer for sending data to or receiving data from SoC. The PW SVF scoreboard is used for multi-stream checking with in-order or out-of-order rules on the SoC system bus. One difference for master XUVC between the IP level and sub-system/SoC level, the Denali memory model is connected to the DUT directly in standalone, but connected to the PADs via iomux in sub-system. So it need to create one register sequence to program the iomux RTL.

In standalone, you just need to connect virtual interface on the DUT, it’s on the testbench layer. But in sub-system or SoC level, many designs’ RTL are involved. The former DUT hierarchy is changed in big netlists. So the UVM harness concept is a better approach. Like a car stereo harness, a UVM harness [5] is a collection of wires (grouped together in interfaces) with at least two ends or connectors to it. One end connects to module(s) and the other end connects to the UVM environment. System Verilog allows you to bind (or add) some of your own items to modules from a separate file allowing you to amend the definition of the module. When you bind an interface to a module, the compiler acts as if the interface is instantiated inside the module. This also means that you can access signals and ports as if the context of your code is also inside the module. With a harness, the connections made in a block-level testbench could be reused in multi-block testbenches. Also, a collection of harnesses can be grouped together into virtual (or system) harnesses that connect system-level environments to yield SoC environments.

```verilog
bind dut_inst interface intf
 (.clk(clk), .reset_n(reset_n), .address(addr), .data(data), .wr(write), .req(request), .ack(ack));
```

4. CONCLUSION

This paper explored how to build a scalable framework to minimize the effort required to reuse IP from block to SoC levels. The set of guidelines in this paper can help you architect your UVM testbench for every verification stage and increase productivity, and flexibility.
References


Acknowledgments
Any trademarks or other proprietary names mentioned in this paper are acknowledged as the property of their respective owners.