SoC Verification using OVM :
Leveraging OVM Constructs to Perform Processor Centric Verification

Cedric Macadangdang, Paul Yue
Cedric.R.Macadangdang@raytheon.com Paul.Yue@raytheon.com

Raytheon Space and Airborne Systems
2000 E El Segundo Blvd., El Segundo, CA 90245

ABSTRACT

This paper provides an in-depth presentation of our experiences with verifying an SoC design including replacing a processor model with an OVM driver to drive all peripherals on the chip, utilizing reserved addresses in the memory map, building custom and reusable APIs to send high-level commands, monitoring protocols and packet transmissions, and scoreboarding. In addition, this paper will cover methods of co-simulation, where the hardware and software aspect of the design can be verified together using realistic stimulus to drive all the components in the system towards complete functional verification.

Keywords: SoC, OVM, verification, processor-centric, assertions, bus functional model.

1.0 INTRODUCTION

As advances in technology allow transistors to decrease in size, engineers now have the liberty to design an entire complex system where custom designed hardware elements and interfaces coexist with an embedded processor on a single piece of silicon – a System-on-Chip. This will make verification a daunting task**.** However, with new advances in verification methodologies and languages, mainly the Open Verification Methodology (OVM) and SystemVerilog respectively, testbenches can be designed to be more robust and flexible – a need for verification of today's designs.

With the advent of the OVM, testbenches can now have a defined structure, with standardized components. Furthermore, together with SystemVerilog, OVM introduces a higher level of abstraction using transaction level modeling, allowing verification engineers more freedom to develop robust testbenches. However, off-the-shelf, OVM is more suitable towards verification of component level and module level designs without a processor. Verification where a processor is involved is not so much plug-and-play – a few tweaks need to be made for OVM and an SoC to play nicely. Introducing a processor into the design adds a software aspect to the verification task. One needs to make sure that any software program that can run on the processor can be supported by the rest of the design without breaking the system. This requires having C code and APIs to coexist with the traditional OVM testbench environment. So how would one leverage the benefits of OVM to perform verification on a processor centric design?

Given a simple SoC design, one has to ensure that all the components on the system can easily communicate with each other - send commands, successfully perform reads and writes, transfer data - following proper protocols. One may suggest constructing drivers for each of the components on the board and driving them simultaneously. Depending on the size of the design, that may become a pain-staking chore. Instead, why not replace the role of the driver with the processor? This is a good idea for several reasons: (1) it is already physically connected to the other devices on the chip and can speak the language (adheres to bus protocols) and (2) it utilizes custom APIs to send high-level commands to the processor which eliminates the need for a sequencer. There are also methods where we can go one step further and replace the processor with bus functional model that mimics the processor's behavior. This will reduce the simulation time that would otherwise be required for having a full processor model in the simulation environment.

1.1 WHY OVM?

Traditional functional verification methods for SoC designs usually lack a systematic approach. Verification of one design can have a completely different approach to another and differing

testbench structures limit reuse. Placing OVM and using SystemVerilog into the equation will enable several things – OVM's modular constructs will increase reusability and vertical integration into higher level testbenches, increasing productivity, and the OVM libraries and tools make it easier to generate constrained random stimulus that will drive towards coverage closure, and lends itself towards automation. In addition, all testbenches following OVM will have similar layouts making it easy for other verification engineers to use. This reduces the learning curve of the testbench setup and the "transfer-of-knowledge" can happen a lot faster.

1.2 SCOPE

It is important to note that this verification methodology is designed to verify the SoC design centered about the processor and does not include verifying the processor itself. We are under assumption that the processor has been fully verified and we are only concerned with verifying from the processor's interface, moving outwards. To us, the internals of the processor is simply a black box. We will also be assuming that each sub-block in the SoC design has been verified by the designer. Our responsibility is to perform top-level verification - to verify that all the sub-blocks on the SoC interface correctly to the system bus and can be properly integrated with the rest of the system and that the top-level design conforms to its requirements.

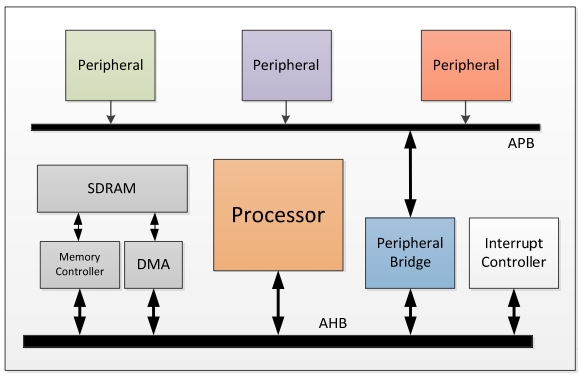

Figure 1 Basic SoC Design

2.0 REPLACING THE PROCESSOR MODEL

Most SoC designs are centered about a processor that governs the whole system. During simulation, a bus functional model (BFM) of the processor is instantiated in the design. Each unique processor has its own BFM that models everything that processor is capable of doing its own unique way. The disadvantage to using these processor BFMs is that it increases simulation time.

Replacing a processor BFM will require two steps. The first would be to create an OVM driver capable of transferring and receiving data to and from the system bus. The second would be to create an interface between the driver and the verification engineer to properly model the sequence of events (device configuration, line reads and writes, interrupts, cache hits/miss, data transmission between peripherals, etc.) that occur during certain processor tasks given a set of processor instructions.

2.1 OVM Driver

Since we only care about what goes on at the processor's interface and not what goes on inside of it, we can simply create an OVM driver that just mimics the processor's interface behavior. At this level, we will not need to concern ourselves with how a processor does its job internally. This enables us to just focus on the interface activity, making sure the driver to bus interface adheres to the protocol and can drive instructions (address + data) correctly. This will enable us to remove the complex processor BFM from the simulation environment and replace it with this simpler OVM driver that drives address and

data onto the system bus to all the attached peripherals and devices.

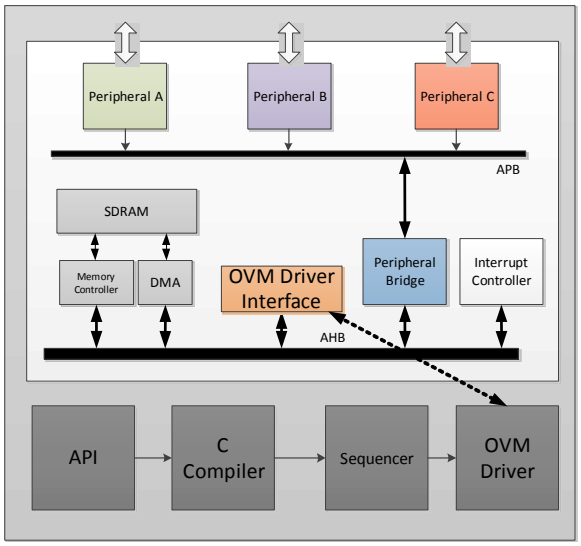

Figure 2 SoC under an OVM testbench environment. Processor BFM replaced with an OVM driver

2.2 OVM Sequencer and API

The next challenge is to model the behavior of the processor by creating a sequence of instructions. This involves using custom API and a C-compiler that can translate high level processor functions down to transaction level model (TLM) packets that the OVM sequencer can feed to the OVM driver.

Once this path is accomplished, the verifier can easily write sequences of instructions for the OVM driver to perform. As long as the driver was constructed accurately to model the processor's behavior and the C-compiler can translate high level abstractions to specific processor commands, the OVM driver substitution it will serve as a faster model during simulation compared to the BFM.

Once this is complete, it is up to the verification engineer to come up with well-thought and interesting test scenarios to exercise the SoC through.

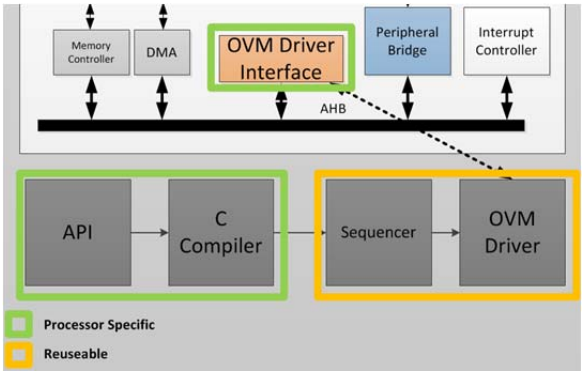

Figure 3 Each time a verification component is reused, the block becomes more functionally robust, and will increase productivity by reducing design costs.

3.0 MONITORS & SCOREBOARDING

The interfaces at the top level of the SoC can be monitored through the use of OVM monitors. For example, if one of the peripherals is a SPI interface, we can have the processor, or in our case, the OVM driver send out data to the SPI peripheral and inspect that it comes out the other end by monitoring the SPI interface. This will require constructing a monitor that adheres to the SPI protocol.

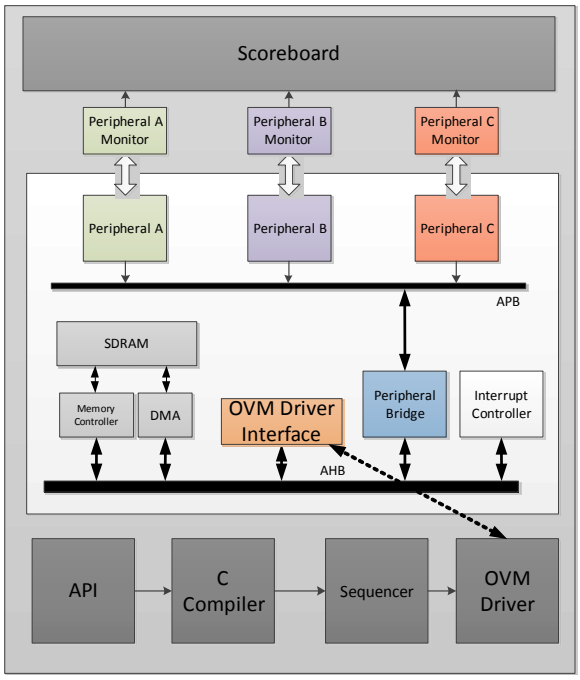

Figure 4 Top level OVM test environment with driver, monitors, and scoreboard

Each top level interface will each have its own monitor to capture data that comes out of it. In the case where the peripheral might request data back, a responder would be needed in addition to the monitor to drive data back into the SoC when required.

Both the driver and the interface monitors will be connected to the scoreboard. Each time the driver drives data onto the SoC's system bus, it will also send that data to the scoreboard. Similarly, anytime the monitor receives data at its assigned interface, it will capture it and send it to the scoreboard as well. The scoreboard is designed to perform a comparison between the data sent by the driver to the data received by one of the monitors.

Since each peripheral or device is mapped to a specified address range, we can utilize the driver to send data to specific address locations to target specific peripherals. The scoreboard can have knowledge of this address mapping scheme so that when the driver reports to the scoreboard that it sent data 0xBEEF, to address 0x04, the scoreboard can determine which interface 0xBEEF should come out of.

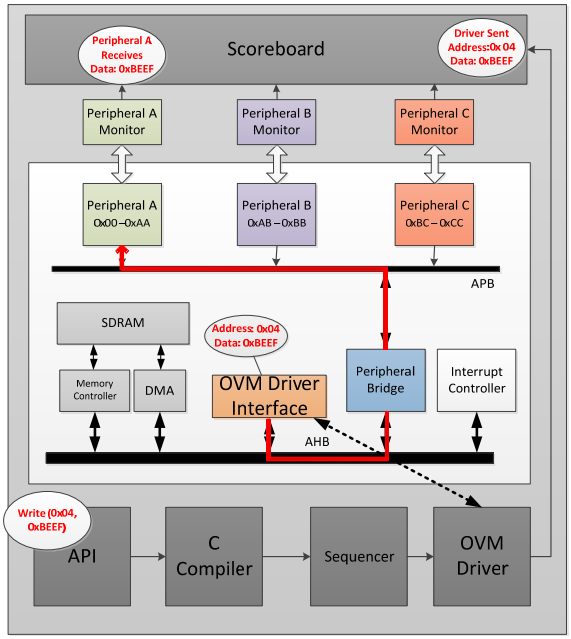

Figure 5 Example test path: Write instruction, write(0x04, 0xBEEF). Red line shows exercised data path from the ovm driver through the AHB bus, through the peripheral bridge, through the APB bus and finally to one of the SoC's interface via Peripheral A. Peripheral A's monitor captures the data and sends it to the scoreboard where it is checked against the data sent out by the driver.

All the peripherals, devices, and interfaces can be exercised in this manner. Each peripheral or corresponding interface can be exercised individually with utilizing the constrained random feature provided by OVM by constraining the address space to a certain range.

4.0 ASSERTIONS

In an SoC design, there can be many sub-blocks and interfaces. Verifying these internal interfaces simultaneously with verifying the top level interfaces, depending on the design, can become difficult to manage. We can mitigate this difficulty to a certain degree by incorporating assertions into the design. While the monitors and scoreboard components maintain eyes on the top-level interfaces of the SoC, assertions keep watch on the inside.

SystemVerilog provides us with the "bind" directive to allow verification engineers to bind assertions into

the design without modifying design code, providing them with "white-box" testing abilities. Now as the top level simulations run, these assertions will be constantly asserted and any violation that fails the assertion will be noted.

Because assertions are not limited to the interfaces at the top level, bugs can be detected closer to the source. The more assertions there are, the easier it will be to trace down the source of a bug.

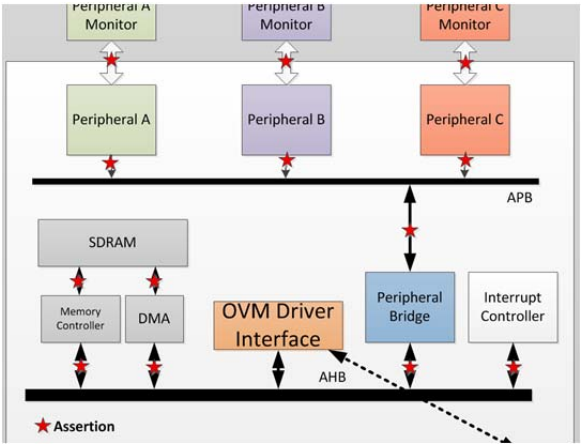

Figure 6 Assertions bound to the various interfaces inside of the SoC – keeps watch for any protocol violations

5.0 CONCLUSION

Incorporating OVM constructs and SystemVerilog into traditional SoC verification environments brings about many benefits.

OVM's modular and systematic approach enables reuse and together with SystemVerilog offers users the tools that provide a software-like automated approach towards achieving functional coverage. Reusing verification components and building off of them only make them better. There is no need to re-invent the wheel.

Substituting the processor BFM with an OVM driver and test cases that mimic the processor BFM's behavior at the interface level removes the complexities that hinder simulation time.

Analysis components such as the top-level scoreboard and interface monitors allows verification engineers to spend more time creating meaningful test cases by eliminating the time spent on eye-balling waveforms to verify correctness of the SoC.

Assertions provide verification engineers with visibility into the design enabling them to detect bugs closer to their source.

All of these methods, utilizing OVM constructs, combined together running in concert under one testbench provides a versatile and agile approach towards verification of a SoC design.

6.0 REFERENCES

IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language, IEEE Std 1800TM-2005, IEEE Computer Society, 2005.

P. Wilcox, Professional Verification: A Guide to Advanced Functional Verification, Norwell, MA: Kluwer Academic Publishers, 2004.

Mentor Graphics. UVM/OVM Online Methodology Cookbook [Online]. Available: http://verificationacademy.com/uvm-ovm

Duolos. Getting Started with OVM. [Online]. Available: http://www.doulos.com/knowhow/sysverilog/ovm/tutorial_2/

Duolog. OVM Golden Reference Guide [Online]. Available: http://ovmworld.s3.amazonaws.com/contributions/OVM%202.0%20Golden%20Reference%20Guide_0.pdf