

WEBINAR

Embedding Intelligence Everywhere with SiFive 7 Series Core IP

▶ Thursday, April 16



SEARCH IP	NEWS	INDUSTRY ARTICLES	BLOGS	VIDEOS	SLIDES	EVENTS	<input type="text" value="Search Industry Articles"/>
-----------	------	-------------------	-------	--------	--------	--------	---

Guidelines for Successful SoC Verification in OVM/UVM

By Moataz El-Metwally, Mentor Graphics
Cairo Egypt

Abstract :

With the increasing adoption of OVM/UVM, there is a growing demand for guidelines and best practices to ensure successful SoC verification. It is true that the verification problems did not change but the way the problems are approached and the structuring of the solutions, i.e. verification environments, depends much on the methodology. There are two key categories for SoC verification guidelines: process, enabled by tools, and methodology. The process guidelines are about what you need to do and in what order, while the methodology guidelines are about how to do it. This paper will first describe the basic tenets of OVM/UVM, and then it tries to summarize key guidelines to maximize the benefits of using state of the art verification methodology such as OVM/UVM.

The BASIC TENETS of OVM/UVM

1. Functionality encapsulation

OVM [1] promotes composition and reuse by encapsulating functionality in a basic block called `ovm_component`. This basic block contains a run task, i.e a functional block that can consume time that acts as an execution thread responsible for implementing functionality as simulation progress.

2. Transaction-Level Modeling (TLM)

OVM/UVM uses TLM standard to describe communication between verification components in an OVM/UVM environment. Because OVM/UVM standardizes the way components are connected, components are interchangeable as long as they provide and require the same interfaces. One of the main advantages of using TLM is in abstracting the pin and timing details. A transaction, the unit of information exchange between TLM components, encapsulates the abstract view of stimulus that can be expanded by a lower-level component. One the pitfalls that can undermine the value of TLM is adding excessive timing details by generating transaction and delivering them on each clock cycle.

3. Using sequences for stimulus generation

The transactions need to be generated by an entity in the verification environment. Relying on a component to generate the transactions is limiting because it will require changing the component each time a different sequence of transactions is required. Instead OVM/UVM allows for flexibility by introducing `ovm_sequence`. `ovm_sequence` is a wrapper object around a function called `body()`. It is very close to an OOP pattern called "functor" that wraps a function in an object to allow it to be passed as a parameter but SystemVerilog does not support operator overloading [1]. `ovm_sequence` when started, register itself with an `ovm_sequencer` which is an `ovm_component` that acts as the holder of different sequences and can connect to other `ovm_components`. The `ovm_sequence` and `ovm_sequencer` duo provides the flexibility of running different streams of transactions without having to change the component instantiation.

4. Configurability



SEARCH SILICON IP

16,000 IP Cores from 450 Vendors

RELATED ARTICLES

- ▶ Guidelines for complex SoC verification
- ▶ Creating SoC Integration Tests with Portable Stimulus and UVM Register Models
- ▶ Design patterns in SystemVerilog OOP for UVM verification
- ▶ Transactor vs CPU in SoC Verification
- ▶ SoC Functional verification flow

[See Mentor Graphics Latest Articles >>](#)

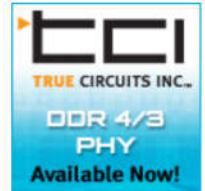
NEW ARTICLES

- ▶ The Thriving Silicon IP Business
- ▶ SRAM PUF: A Closer Look at the Most Reliable and Most Secure PUF
- ▶ NeoPUF, A Reliable and Non-traceable Quantum Tunneling PUF
- ▶ Understanding Physical Unclonable Function (PUF)
- ▶ Shift Power Reduction Methods and Effectiveness for Testability in ASIC

[See New Articles >>](#)

MOST POPULAR

1. Dynamic Memory Allocation and Fragmentation in C and C++
2. System Verilog Macro: A Powerful Feature for Design Verification Projects
3. Method for Booting ARM Based Multi-Core SoCs
4. Using SystemVerilog Assertions in RTL Code



Configurability, an enabler to productivity and reuse, is a key element in OVM/UVM. In OVM/UVM, user can change the behavior of an already instantiated component by three means: configuration API, Factory overrides and callbacks.

5. [How to achieve low latency audio/video streaming over IP network?](#)

[See the Top 20 >>](#)

5. Layering

Layering is a powerful concept in which every level takes care of the details at specific layers. OVM layering can be applied to components, which can be called hierarchy and composition, and to configuration and to stimulus. Typically there is a correspondence between layering of components and objects. Layering stimulus, on the other hand, can reduce the complexity of stimulus generation.

 E-mail This Article

 Printer-Friendly Page

6. Emphasis on reuse (vertical and horizontal)

All the tenets mentioned above lead to another important goal which is reuse. Extensibility, configurability and layering facilitate reuse. Horizontal reuse refers to reusing Verification IPs (VIPs) across projects and vertical reuse describes the ability to use block-level VIPs in cluster and chip level verification environments.

PROCESS GUIDELINES

1. Ordering of development tasks

The natural process for developing OVM/UVM verification environment is bottom-up. Blocks are first verified in block-level environments, and then the integration of the blocks into SoC is verified in chip-level testbench. Some refers to this methodology as IP-centric methodology because the blocks are considered IPs [4]. The focus of block-level verification is to verify the blocks thoroughly, while the chip-level is focused on verifying the integration of the blocks and the application scenarios. A bottom-up verification approach has several benefits:

- Localization of bugs: finding bugs easily
- Easier to test all the block modes at the block-level
- Confidence in the block-level allowing them to be reused in several projects.

In this section we describe the recommended ordering for development of verification environment elements. Such ordering must be in mind when developing executable verification plans.

Table 1: Components Development Order

Interfaces
Agents
Transaction
Configuration
Agent Skeleton
Transactors
Basic Sequences
Block level Subsystem
Configuration
Virtual Sequencer
Initial Sequences/Tests
Scoreboards & Protocol Checkers
Coverage Model
Constrained Random Sequences/Tests
Chip Level
Integration of Subsystem environments
Chip-Level Sequences/Tests

It is worth noting the following:

- Once transaction fields are defined and implemented, the agent skeleton can be automatically generated.
- Transactors refer to drivers and monitors
- The reason for having the scoreboards & protocol checkers early on is to make sure that what was developed is functioning
- Coverage model needs to be before the constrained random tests to guide the test development and eliminate redundancy. This is a corner stone of Coverage Driven verification. The coverage model not only

guides the test writing effort but rather gives is a metric for verification progress and closure.

- Each block/subsystem/cluster verification environment and tests act as a VIP for this block.

2. Use code and template generators

Whether you are relying on script or elaborate OVM template generators, these generators are keys to increase the productivity of verification engineers, reduce errors and increase code conformity. Code generators are also used to generate register models from specification thus automating the creation of these models

3. Qualify your VIPs

Qualify your VIP during development and before releasing them. First, several tools can conduct static checking on your VIP components for common errors and conformance to coding styles. They can also provide statistics about the size of your code, checks and covergroups.

Second, typically a simulator can provide statistics about memory consumption and performance bottlenecks of your VIP. Although SystemVerilog has automatic garbage collection, you can still have memory leaks because you keep a reference to dynamically allocated objects somewhere and forget about them.

Third, your VIPs should be robust to user mistakes whether in connections or proper use. You need to have sanity checks that can flag early a user error.

Finally, peer review is still beneficial to point-out issues that are missed in the other steps.

4. Incremental integration

As described in the introduction, OVM/UVM facilitates composition and layering. Several components/agents can form an environment and two or more environments can form a higher level environment. Incremental integration is important to reduce debugging time.

5. Better regression management and result analysis

The usual scripts that compile and run testcases come short when running complex OVM/UVM SoC verification environment. Typical requirements on run management is to keep track of seeds, log files of different tests, execution time, flexibility of running different groups of tests and running on local machine or grid. Once a regression is run we end up with data that needs to be processed to come out for useful information such as which tests passed/failed, common failure messages, which tests were more efficient and which seeds produced better coverage.

6. Communication and change management

Communication between verification engineers and specification owners should be captured in an issue tracking tool to avoid losing the information along the project. Also verification engineers need mechanism to share what they learn between each other, Wikis serve as good vehicles to knowledge sharing.

Change management is the other crucial element. By change management we are not only referring to code version management but the way the changes in RTL and block-level environments are handled in cluster or chip level environments.

METHODOLOGY GUIDELINES

1. CPU modeling

SoCs typically have one or more software programmable component such as microcontroller, microprocessor or DSP. Processor Driven Verification refers to using either a functional model of the processor or RTL model to verify the functionality of the SoCs. This approach is useful to verify the firmware interactions and certain application scenarios. However, for thorough

verification of subsystems/cluster this approach can be costly in terms of effort, complexity, and simulation time. This paper proposes two level approach: for the verification of subsystems use a pin-accurate and protocol accurate Bus Functional Model (BFM), this will enable rapid development of the verification environment and tests and at the same time gives flexibility to the verification engineer in creating the environment and test. The BFM usually comes as VIP for the specific bus standard that the processor connects to. While the VIP usually models the standard interface faithfully, the processor might have extra side-band signals and interrupt. There are two approaches to this: the VIP can model in a generic way the side-band and interrupt controller behavior through the use of configuration, transactions and sequences. The other approach is to model the functionalities in different agents for side-band signals and interrupts. This increases the burden on the development and requires synchronization between different agents.

For the verification of firmware interaction, such as boot-loading or critical application scenarios, the RTL model or a full functional model can be used guarantee that firmware is validated versus the hardware it is going to run on and that the hardware.

2. Environment Reuse

Environments should be self-contained having only knowledge about its components and global elements and can communicate only through configuration mechanism, TLM connections or global events such as reset event. Following these rules, an environment at the block-level can be reused at the chip-level making the chip-level environment the integration of block-level environments.

3. Sequence Reuse

It is important to write sequences with eye on reusing them. In OVM/UVM, there are two types of sequences: sequence which sends transactions and sequences that starts sequences on sequencers. The latter is called a virtual sequence. Below is further classification of the sequences based on the functionality:

- Basic agent sequence: this sequence allows the user to control the fields of a transaction that sent by the basic sequence from outside. The basic agent sequence acts as an interface or API to randomize or set the fields of the transactions sent by a higher layer which is usually the virtual sequence.
- Register read/write sequences: these are sequences that try to write and read address mapped registers in the DUT. Two important rules need to be considered: they should have API that is independent of the bus protocol and rely on use the name of the register rather than address. A register package can be used to lookup the register address by name. For Example: OVM register package built-in sequences [5] supports this kind of abstraction. It is also expected that the UVM register package will support these rules. Abiding by these rules make these sequences reusable and maintainable because there is no need to update the sequence each time a register address changes.
- DUT configuration sequences: some verification engineer try to provide sequences that abstracts the different configurations of the DUT into enum fields to ease the burden on the test writer. This way the test writer does not need to know about which register to write and with what value. These sequences are still reusable at the chip-level.
- Virtual sequences on accessible interfaces at chip-level: These sequences are reusable from block-level to chip-level; some of them can be used to verify the integration into full-chip.
- Virtual sequences on internal interfaces that are not visible at the chip-level: Special attention should be paid for sequences generating stimulus on interfaces that are no longer visible at the chip-level.

Although goals are different between block and chip level testing, some virtual sequences from block-level can be reused at chip-level as integration tests. Interfaces that become internal at the chip-level can be usually stimulated through some external interface. In order to make the last type of virtual sequences reusable at chip-level, it is better to plan ahead to abstract the data from the protocol. For example in Figure 1 of SoC diagram peripherals 1 through N are on peripheral bus which might be using a

different protocol than the system bus. There are two approaches to make the sequences reusable:

Use functional abstraction by defining functions in the virtual sequence that can be overridden like:

```
write(register_name, value);
read(register_name, value);
```

Or rely on a layering technique like `ovm_layering[3]`. In this approach, a layering agent sits on top of a lower level agent and it forwards high-level transactions that can be translated by the low-level agent according to the bus standard. The high-level agent can be connected to a different low-level agent without any change to the high-level sequences.

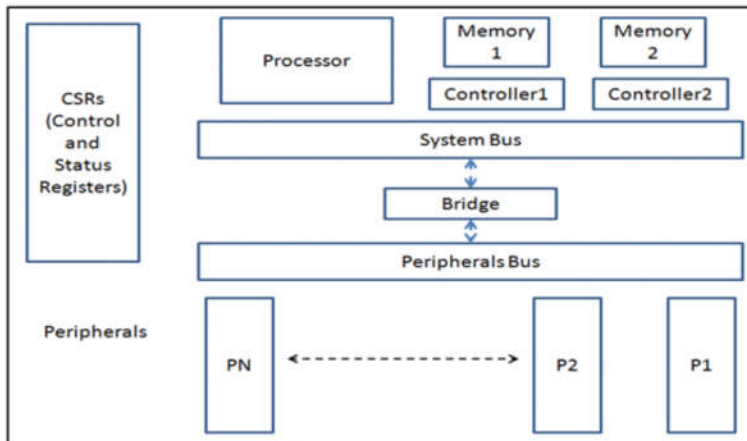


Figure 1: Typical SoC Block Diagram

4. Scoreboards

A critical component of self-checking testbenches is the scoreboard that is responsible for checking data integrity from input to output. A scoreboard is a TLM component, care should be taken not activate on a cycle by cycle basis but rather at the transaction level. In OVM/UVM, the scoreboard is usually connected to at least 2 analysis ports one from the monitors on the input(s) side and the other on the output(s) Figure 2 depicts these connections. A Scoreboard operation can be summarized in the following equations:

```
Expected = TF(Input Transaction);
Compare(Actual, Expected);
TF : Transfer function representing the DUT functionality from inputs to outputs
```

Sometimes the operation is described as predictor-comparator. Where the predictor computes the next output (transfer function) and the comparator checks the actual versus predicted (compare function). Usually the transfer function is not static but can change depending on the configuration of the devices. In SoC, most peripherals have memory-mapped registers that are used for configuration and status. These devices are usually called memory-mapped peripherals and they pose two challenges:

- DUT transfer function and data-flow might change based on the configuration
- Status bits should be verified

The common solution to the first one is to have a handle of the memory-map model and connect an analysis port from the configuration bus monitor to the scoreboard. On reception of new transaction on this analysis port, the scoreboard updates the peripheral's registerfile model and then uses it to update the transfer function accordingly. This approach has one disadvantage; each peripheral scoreboard has to implement the same functionality and needs to connect to the configuration bus monitor. A better approach is that the registerfile updates occur in a central component on the

bus. To eliminate the need for the connections to the bus monitor, the register package can have an analysis port on each registerfile model. Each Scoreboard can connect to this registerfile model internally without the need for external connections. One of the requirements on the UVM register package is to have update notification method [6].

The second challenge is status bit verification. Status bits are usually modeled in the register model and register model can act as a predictor of the value of status bits. This requires that the scoreboard predicts changes to status bits, update the register models and on register reads the value read from the DUT is compared versus the register model.

There are other aspects to consider when implementing the scoreboards:

- Data flow analysis: data flow can change based on configuration, or data flow can come from several inputs towards the output.
- Scoreboard connection technique: Scoreboards can be connected to monitors using one of two ways: through `ovmimps` in the scoreboard or through `ovm_exports` and `tlm_analysis_fifos`: the latter requires a thread on each `tlm_analysis_fifo` to get transactions while the former executes in the context of the caller.
- Threaded or thread-less: the scoreboard can have 0 or more threads depending on a number of factors such as the connection method, the complexity of synchronization and experience of the developer. As a general rule, verification engineers should avoid spawning unnecessary threads in the scoreboard.

At the SoC level, there are two approaches to organize scoreboards with End-to-End and Multi-step [2]. Figure 3 depicts the difference between the two. The multi-step approach has several advantages over the end-to-end:

- By product of the block-level to chip-level reuse.
- The checking task is simpler since it is divided over several components each concerned with specific block.
- Easy to localize bugs at block-level since the violating block scoreboard will flag the error

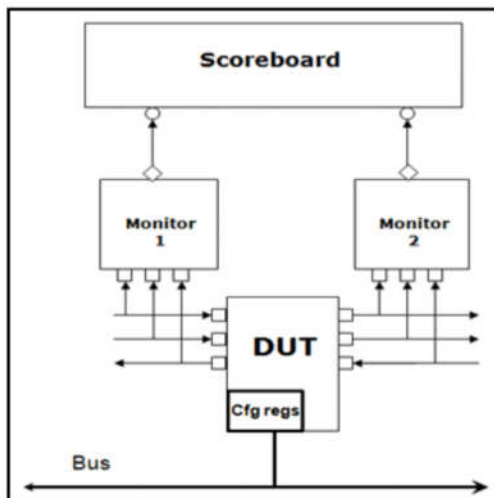


Figure 2: Scoreboard Connection in OVM

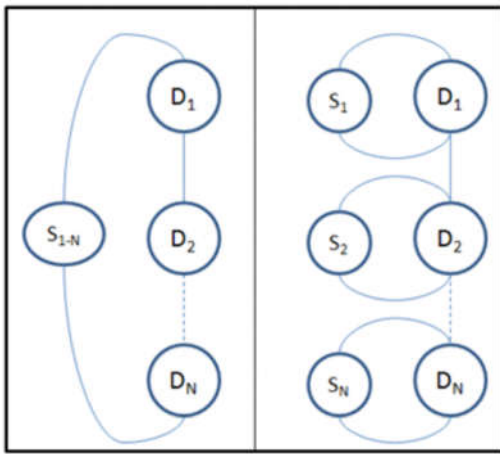


Figure 3: End-to-End vs. Multi-Step Scoreboard

CONCLUSION

OVM/UVM is a powerful verification methodology. To maximize the value achieved by adopting OVM/UVM there is a need for guidelines. These guidelines are not only for the methodology deployment but also for the verification process. This paper tried to summarize some of the pitfalls and tradeoffs and provide guidelines for successful SoC verification. The set of guidelines in this paper can help you plan ahead your SoC verification environment, avoid pitfalls and increase productivity.

REFERENCES

1. Mark Glasser, Open Verification Methodology Cookbook, Springer 2009
2. Sasan Iman and Sunita Jushi, The e-Hardware Verification Language, Springer 2004.
3. Rich Edelman et al., You Are In a Maze of Twisty Little Sequences, All Alike – or Layering Sequences for Stimulus Abstraction, DVCON 2010.
4. Victor Besyakov et al., Constrained Random Test Environment for SoC Verification using VERA, 2002.
5. OVM Register Package, www.ovmworld.org, May 18, 2010, ovm_register-2.0.
6. Accellera VIP TSC, UVM Register Modeling Requirements, www.accellera.org/activities/vip/

REQUEST OF INFORMATION

Contact Mentor Graphics

Fill out this form for contacting a *Mentor Graphics* representative.

Your Name:	<input type="text"/>
Your E-mail address:	<input type="text"/>
Your Company address:	<input type="text"/>
Your Phone Number:	<input type="text"/>
Write your message:	<input type="text"/>
	<input type="button" value="send"/>

Partner with us

Visit our new Partnership Portal for more information.

[**Partner with us**](#)**List your Products**

Suppliers, list your IPs for free.

[**List your Products**](#)**Design-Reuse.com**

[Contact Us](#)
[About us](#)
[D&R Partner Program](#)
[Advertise with Us](#)
[Privacy Policy](#)

© 2020 Design And Reuse

All Rights Reserved.

No portion of this site may be copied, retransmitted, reposted, duplicated or otherwise used without the express written permission of Design And Reuse.