

# Autonomous Agent for Universal Verification Methodology Testbench of Hard Memory Controller

R. Logeish Raj and Rosmiwati Mohd-Mokhtar

**Abstract** Pre-silicon verification process is important in an application having integrated chip design cycle. It is considered one of the biggest bottle-neck in modern day design projects. This paper intends to describe a testbench architecture that will improve verification efficiency and productivity for a Hard Memory Controller's Sideband verification from the perspective of the test writer. The testbench architecture described by Universal Verification Methodology is reused, adapted and improved to allow higher level of automation within the testbench. The implemented autonomous agent is analyzed and compared against the regular agent for its efficiency in terms of lines of code need to be written by the test writer. The result obtained shows that the autonomous agent implemented in the architecture reduces the test writer's burden by at least 60 % and up to 78 %.

**Keywords** UVM • HMC • DDR4 • Sideband agent • Testbench

## 1 Introduction

Functional verification is the art of making sure an Application-Specific Integrated Circuit (ASIC) design or system on chip (SOC) is functioning according to the designer's expectations and fulfilling the design requirements. In the past when ASIC design was still in its infancy, verification was not a prominent idea. As ASIC design grew and SOC's become a household name, verification started to get very important, as it allows design bugs to be caught very early in design cycle and produce quality integrated circuits (IC's). SystemVerilog [1] is the language

---

R. Logeish Raj (✉)

Altera Corporations Sdn. Bhd, FTZ Bayan Lepas, 11900 Pulau, Pinang, Malaysia  
e-mail: logeish@gmail.com

R. Logeish Raj • R. Mohd-Mokhtar (✉)

School of Electrical and Electronic Engineering, Universiti Sains Malaysia,  
Engineering Campus, 14300 Nibong Tebal, Pulau Pinang, Malaysia  
e-mail: rosniwati@ieee.org

© Springer Science+Business Media Singapore 2017

H. Ibrahim et al. (eds.), *9th International Conference on Robotic, Vision, Signal Processing and Power Applications*, Lecture Notes in Electrical Engineering 398,  
DOI 10.1007/978-981-10-1721-6\_2

designed to aid functional verification while Universal Verification Methodology (UVM) [2] is the state of the art verification methodology that is based on SystemVerilog. UVM contains guidelines, additional base class libraries, toolkits, macros and industry best practices to build verification environments in a structured way. Thus, allowing widespread adoption of UVM for verification.

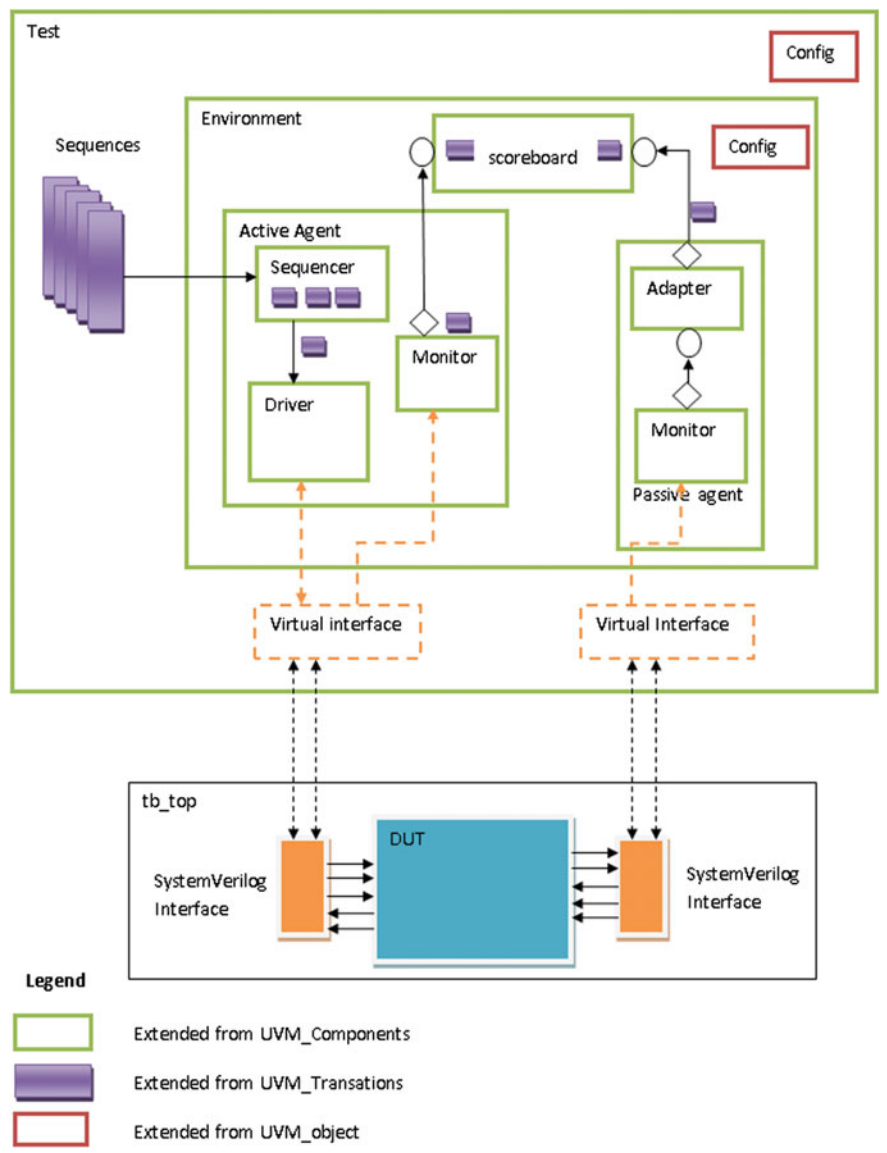
While UVM testbench architecture has many advantages, it gives lesser attention to automation within the verification environment. The UVM testbench architecture gives a lot of control to the users (or test writers) of the verification environment. Test writers have high degree of freedom to inject stimulus into the design under test (DUT) as they please. Although this is good for a very advanced test writer who wants to do manual testing, it may not be attractive to the normal test writer. The testbench should be automated to drive and check the DUT without much human intervention.

The purpose of this research is to improve the basic architecture described by UVM to allow higher level of automation in the testbench. An automated UVM agent with DDR protocol awareness will be developed to drive stimulus into the DUT (Hard memory controller). A point to note is that this automated UVM agent is not only capable of driving stimulus into the DUT (based on test writer's inputs), it is also able to ensure the user does not violate the DDR (Double Data Rate) protocol on the memory interface.

## 2 UVM Testbench Theory of Operation

The UVM describes high level testbench architecture to verify a DUT. The architecture is made of multiple individual components that are designed to perform specific tasks. Figure 1 shows the testbench architecture described by UVM. This architecture will be reused and improved to perform automated agent approach with sideband DDR protocol awareness and to reduce the test writer's burden. The Test is a standard UVM class that will encapsulate the verification environment and will instantiate sequences to be parsed to the environment. The Environment is a top level verification component that wraps around and connects the other verification components. The Agent typically contains, a driver, a sequencer, a monitor and configuration objects. Most of the architectural changes to build an automated agent approach will take place within this level.

The Agent can be configured to be active or passive through the Environment's configuration object. While the driver, sequencer and monitor can be reused independently, UVM's guidelines recommend using an agent as a container for these components. The Sequencer is an advanced stimulus generator that will generate streams of transactions to be consumed by the driver. The Driver is an active component that consumes the transactions from the sequencer and translates the transaction into actual pin wiggle to the DUT. This is the UVM component that actively interacts with the DUT and it merely acts as a translator between object oriented programming (OOP) to actual pin activity.



**Fig. 1** Basic UVM Testbench architecture

The purpose of this research is to build an autonomous agent to verify the Hard Memory Controller (HMC) [3]. Figure 2 indicates a simplified illustration of the HMC. This research is limited to verify the Sideband module through the MMR (Memory Mapped Register) and the IO (Input Output) interface. The protocol on memory interface must obey the JEDEC DDR4 specification [4].

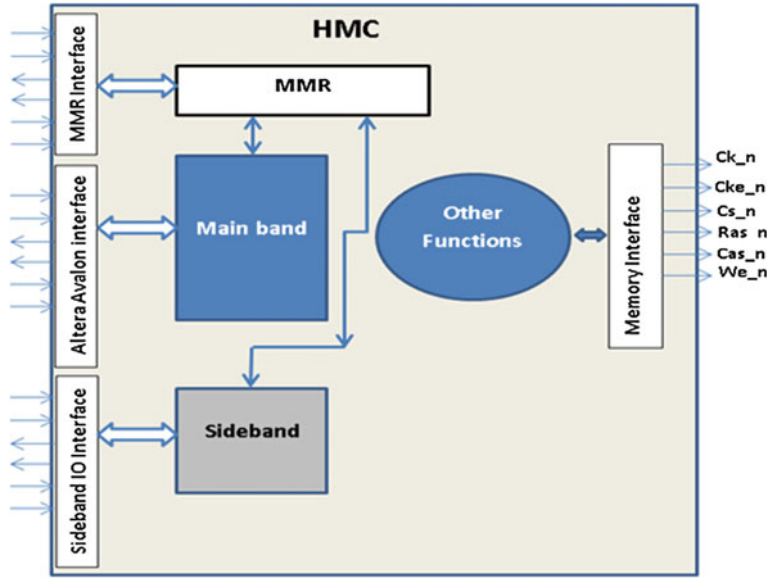


Fig. 2 Simplified illustration of the HMC

Table 1 JEDEC memory interface protocol requirement

JEDEC protocol requirement	
Refresh insertion	Periodic refresh—DRAM refreshed every 7.8 μs
	Insert refresh in between back to back self refresh

Table 2 HMC requirement on user interface

DUT requirement	
Refresh insertion	Insert at initialization
	Insert after MPS Exit
Short calibration insertion	Insert at self refresh exit
Long calibration insertion	Insert after MPS exit
ACK checking	ACK to be de-asserted before issuing next command

Tables 1 and 2 describes the JEDEC protocol requirement and the HMC requirements respectively. Using the regular UVM approach, the test writer must be aware of these requirements when sending stimulus to the HMC. However with the autonomous agent approach, these requirements awareness will be built into the Agent itself, specifically within the Driver. This will reduce the test writer’s burden.

### 3 Automated Agent Implementation

The Autonomous Sideband agent is implemented to address the requirements in Tables 1 and 2. The major architectural changes will be within the Driver and the method used to drive the pins of the HMC. In this research, a Bus Functional Module (BFM) will be coupled with the Driver to interface with the HMC. This is so that the requirement in Table 1 can be met easily. The Driver is a class based component that has limited capability in time domain. Having a module that is more aware of time domain eases implementation. Therefore the autonomous agent approach will be divided between the Driver and the BFM. Figure 3 shows the high level architecture of the Sideband Agent coupled with the BFM. The Driver and the BFM is coupled together based on the works of [5].

Figure 4 shows the updated Driver. The changes include a new UVM verification component called the protocol block, and an improved run phase. The protocol block inserts extra sideband transaction based on the incoming sideband transaction from the sideband sequencer. The run phase has an updated code structure, where a traffic manager and a SystemVerilog task called “get & drive” controls the flow of transaction in the driver, since both the protocol block and the BFM can now issue transactions.

When a sideband transaction is received from the sequencer, the driver routes this transaction into the protocol block. The protocol block will decode this

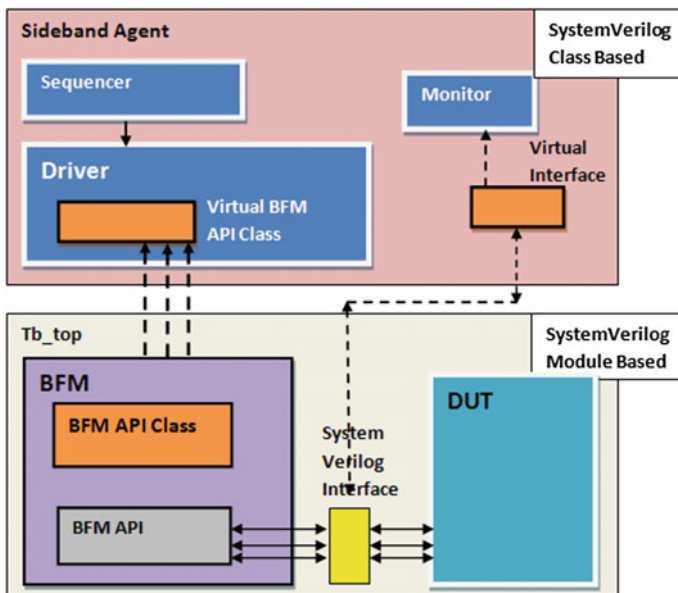
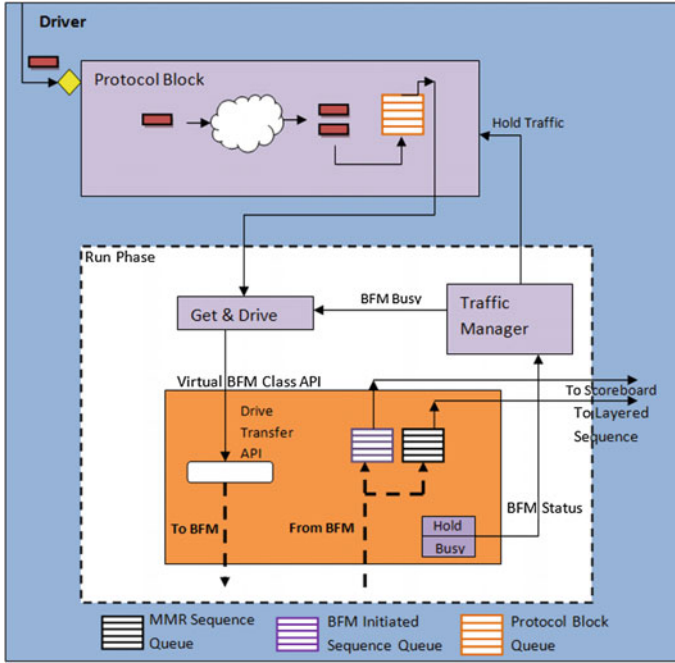


Fig. 3 High level architecture of sideband agent



**Fig. 4** Autonomous sideband driver

transaction and build new transactions. These new transactions and the original transaction issued by the test writer will be added into the transaction queue in the correct sequence as dictated by the protocol and the HMC requirement. The contents of this queue is later get and driven into the BFM through the virtual BFM API class.

The traffic manager ensures that the BFM's drive transfer API is not accessed as long as the BFM is busy completing its previous transaction. The BFM's architecture has a limitation where it cannot process multiple commands in parallel. The traffic manager also completely halts traffic in the protocol block at the request of the BFM. The BFM does this to satisfy the second protocol requirement, where a refresh command must be sent before refresh timeout occurs in the DRAM which is every 7.8  $\mu\text{s}$ .

In Fig. 5, The BFM takes care of the protocol requirement in Table 1. A Refresh must be sent to the DRAM before 7.8  $\mu\text{s}$ . Beyond this time, if not refreshed, the DRAM will lose its memory contents. This behaviour is very hard to mimic from the test or the class based components. It is easier to mimic this behaviour from the BFM, thus this protocol requirement is supported through the BFM.

From Fig. 5, the refresh counter keeps track of time and signals the refresh manager before the 7.8  $\mu\text{s}$  interval. The refresh manager will access the Drive

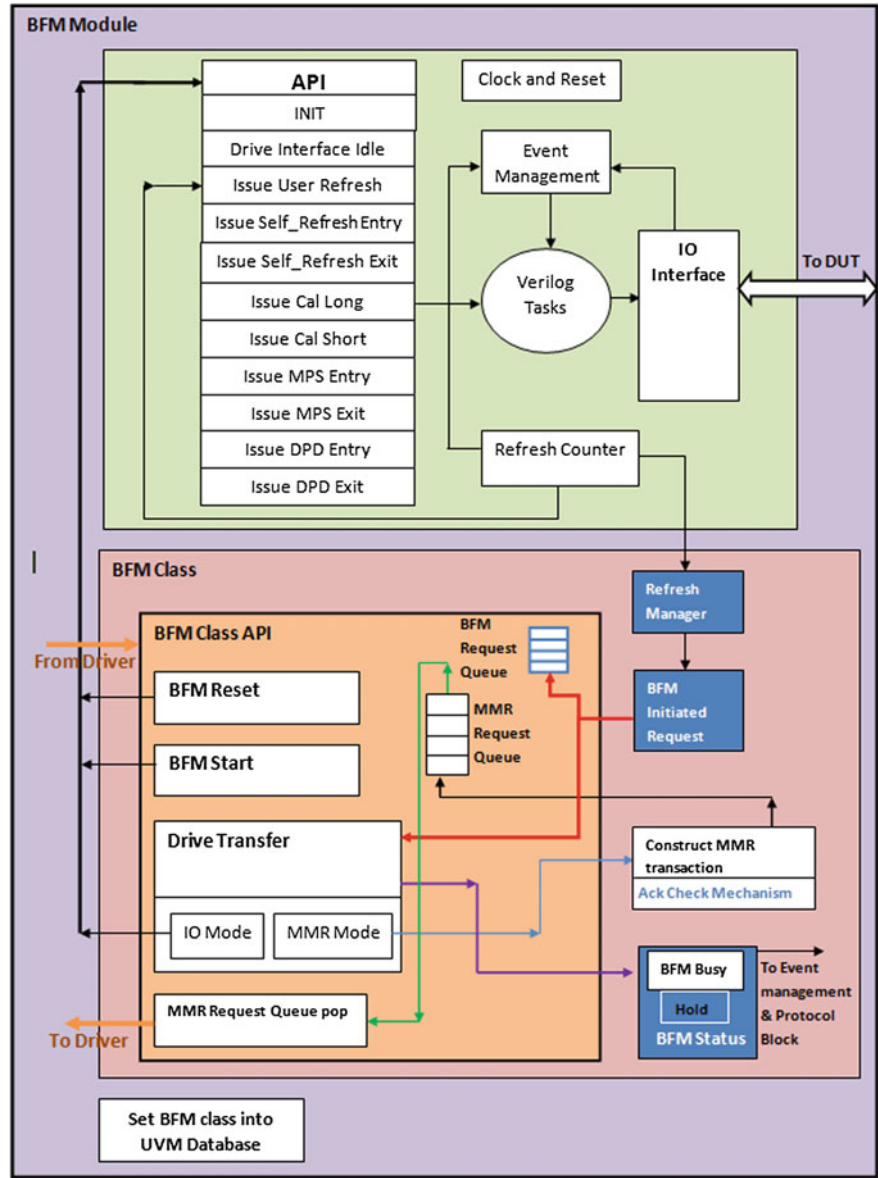


Fig. 5 Autonomous sideband BFM

Transfer function which will signal the BFM’s status block. The BFM Status block is built to pass BFM’s status to the Driver’s traffic manager so that the BFM can automatically issue a refresh and thus other transactions from the protocol block to be halted until refresh operation completes.

## 4 Result and Analysis

In order to demonstrate the result from autonomous agent approach, two tests are conducted in this paper: (1) Short calibration test and (2) Self refresh test. Only two test cases are demonstrated, because the HMC's Sideband Commands can be grouped based on two major classifications. One is a pulse based command and second is a period based command. Case 1 is a pulse based command where a request to the HMC will result in the memory device entering and exiting a state by itself. Whereas, Case 2 is a period based command where the device will remain in the asserted state until an exit command is sent through HMC by the user/test writer.

The analysis was done in terms of lines of code needed to use the autonomous agent as opposed to the regular agent approach by a test writer upon giving similar result. Figure 6 shows the outcome from Case 1 test (Case 2 test is omitted due to page limitation). A comparison as shown in Table 3 is created to summarize the observation. The Lines of Code (LOC) signifies the complexity needed to be

Regular Agent Approach
<pre>// Create the sequences test_seq_1 = sb_zqcal_short_seq::type_id::create("test_seq_1", this); phase.raise_objection(this); 'uvm_info(get_type_name(), "Start running Sideband Simple Transactions", UVM_MEDIUM) if(!test_seq_1.randomize() with {txn_number == 1;}) 'uvm_fatal("RANDERR", "Randomization Error"); //////// Step 1 // READ ACK m_ral[0].reg_sideband8.read(status, read_data, UVM_FRONTDOOR, .parent()); m_env.m_hmc_env[0].m_mem_env.m_mem_if_env.vif_spy.wait_for_signal("clk", "posedge", 1); if (read_data[0] == 1'b1) 'uvm_error(get_type_name(), \$formatf("ERROR::MMR_ACK_READ = %0h\n", read_data)); //////// STEP 2 // REQ assert WRITE repeat (1) test_seq_1.start(m_env.m_hmc_env[0].m_mem_env.m_mem_sbif_env.m_master_agent.m_sequencer); #1ns //////// STEP 3 // POLL For ACK == 1 for (int i=0; i&lt;64; i++) begin m_ral[0].reg_sideband8.read(status, read_data, UVM_FRONTDOOR, .parent()); m_env.m_hmc_env[0].m_mem_env.m_mem_if_env.vif_spy.wait_for_signal("clk", "posedge", 1); if (read_data[0] == 1'b1) break; end //////// Step 4 // Deassert Req m_ral[0].reg_sideband3.write(status, 32'h0, UVM_FRONTDOOR, .parent()); #1ns //////// Step 5 // READ ACK m_ral[0].reg_sideband8.read(status, read_data, UVM_FRONTDOOR, .parent()); m_env.m_hmc_env[0].m_mem_env.m_mem_if_env.vif_spy.wait_for_signal("clk", "posedge", 10); if (read_data[0] == 1'b1) 'uvm_error(get_type_name(), \$formatf("ERROR::MMR_ACK_READ = %0h\n", read_data));</pre>
Autonomous Agent Approach
<pre>// Create the sequences test_seq_1 = sb_zqcal_short_seq::type_id::create("test_seq_1", this); phase.raise_objection(this); 'uvm_info(get_type_name(), "Start running Sideband Simple Transactions", UVM_MEDIUM) if(!test_seq_1.randomize() with {txn_number == 1;}) 'uvm_fatal("RANDERR", "Randomization Error"); // Just 1 Step repeat (1) test_seq_1.start(m_env.m_hmc_env[0].m_mem_env.m_mem_sbif_env.m_master_agent.m_sequencer); 'uvm_info(get_type_name(), "Complete running Sideband Simple Transactions", UVM_MEDIUM) phase.drop_objection(this);</pre>

Fig. 6 Short calibration test



**Table 3** Test comparison for regular and autonomous agent

Test case	Lines of code (LOC) in test		% of code reduced
	Regular agent	Autonomous agent	
Case 1 (1 short calibration command)	23	9	60
Case 2 (1 self refresh cycle)	46	10	78

handled by the test writer. With the autonomous agent approach, the test in case one sees a reduction in LOC by 60 %. For case two where both protocol requirement and DUT requirements must be considered, the test sees a reduction in LOC by 78 %. Therefore, the test complexity and burden of the test writer can be considerably reduced using the autonomous agent approach.

## 5 Conclusion

The Autonomous agent made the sideband UVM environment highly abstracted and automated while being less dependent on test writer inputs. The comparison in Table 3 clearly shows that the autonomous agent approach reduces the burden of the test writer. The autonomous agent approach not only reduces the lines of code in the tests, it also makes the testbench very user friendly. The autonomous agent deals with the lower level protocol requirements and DUT requirements. Thus reducing test level complexity. Most importantly, the agent ensures transactions to the HMC satisfy memory protocol requirements and special DUT requirements listed in Tables 1 and 2.

**Acknowledgments** The authors would like to thank Altera Corporations Sdn. Bhd and Universiti Sains Malaysia in supporting the carried out research investigation.

## References

1. IEEE Standard 1800 (2012) IEEE standard for SystemVerilog—hardware design and verification language. IEEE Standards Association, New York, NJ
2. Accellera.org (2011) Universal Verification Methodology (UVM) v 1.1 User Guide
3. Altera emi\_rm, 2015 Functional Description-Arria 10 EMIF, Altera Corp. [https://www.altera.com/en\\_US/pdfs/literature/hb/external-memory/emi\\_ip.pdf](https://www.altera.com/en_US/pdfs/literature/hb/external-memory/emi_ip.pdf)
4. JEDEC Standard JESD79–4A (2014) DDR4 SDRAM Specification, JEDEC Solid State
5. Bhutada S (2011) Polymorphic interfaces: an alternative for SystemVerilog interfaces, Mentor Graphics Verification Horizons

<http://www.springer.com/978-981-10-1719-3>

9th International Conference on Robotic, Vision, Signal  
Processing and Power Applications

Empowering Research and Innovation

Ibrahim, H.; Iqbal, S.; Teoh, S.S.; Mustaffa, M.T. (Eds.)

2017, XXI, 861 p. 448 illus., 322 illus. in color.,

Hardcover

ISBN: 978-981-10-1719-3