MultiCraft

International Journal of Engineering, Science and Technology
Vol. 6, No. 4, 2014, pp. 31-44

INTERNATIONAL JOURNAL OF ENGINEERING, SCIENCE AND TECHNOLOGY

www.ijest-ng.com
www.ajol.info/index.php/ijest

Functional verification of I2C core using SystemVerilog

Rakhi Nangia[1], Neeraj Kr. Shukla[2*]

[1,2]*VLSI Group, Department of Electrical, Electronics & Communication Engineering, ITM University, Gurgaon, (Haryana), INDIA*
[*]*Corresponding author: e-mail: neerajkumarshukla@gmail.com*

Abstract

The verification phase carries an important role in design cycle of a System on Chip (SoC). A verification environment may be prepared using SystemVerilog without using any particular methodology but that will be different for every variation of the design. There are various verification methodologies out of which Universal Verification Methodology (UVM) is the state-of-the-art and widely preferred by the verification industry worldwide, as the verification environment created using UVM is reusable, efficient and well structured. In this work we have compared the SystemVerilog and UVM verification environments. The Inter Integrated Circuit (I2C) Master Core is the Design Under Test (DUT). The environments created using SystemVerilog and UVM, completely wrap the DUT. The assertion coverage found is 100% from both approaches and functional coverage is found as 99.21% and 96.42% from SV environment and UVM environment respectively. Therefore, the overall coverage found is 99.60% and 98.21 from developed SV and UVM environment.

Keywords: Environment, I2C, SoC, SystemVerilog, Testbench, Verification, UVM.

DOI: http://dx.doi.org/10.4314/ijest.v6i4.4

1. Introduction

As large and complex is the design, more are chances of bugs in the design and that requires extensive and diverse verification. If design is quite big, verification is done at various levels like at unit level, block level, subsystem level and at IP level. Verification of a design is the most critical phase in chip design cycle and takes nearly 70-80% of the total design cycle. Any IP verification requires in-depth coverage based constrained random verification. If the design is already verified before and only minor changes are done then few directed test cases may be used to verify the design (Glasser, 2011; Yun et al., 2011). Verification may be done by a hardware design engineer or by an experienced verification engineer, so to work together without confusion it is necessary to make a defined directory structure and file names. Different verification languages and verification methodologies are there in VLSI industry. SystemVerilog is a design and verification language that provides complex data types and constructs required to build a verification environment and is most commonly used nowadays for verification purposes. Similarly a methodology is applying a language in a planned and structured way for doing verification of a design. UVM is the latest verification methodology being used in VLSI industry for verification. The Environments developed through SystemVerilog may be different depending upon implementer, while verification environment built using UVM remains the same for different vendors, i.e. it can be reused for different vendor IPs. We can say that use of a particular verification methodology is advantageous in terms of better communication among engineers and reusability of verification environment.

The paper is structured as follows: Section I covers the introduction and Section II discusses the literature review of the methodologies and work reported in this domain. Section III has a small description of the DUT that is I2C Master Core. Section IV talks about SystemVerilog Environment built for the DUT under consideration and its simulation outputs. Section V discusses about UVM, developed UVM testbench architecture and its simulation outputs. Section VI shows the comparison between two environments i.e. how UVM environment is different than SystemVerilog environment. Section VII discusses how one environment is better than other and finally section VIII closes with conclusion of the work.

2. Literature Review

SystemVerilog built over Verilog 2001 provides a higher level of abstraction to design and verification. SystemVerilog provides a complete verification environment, with constrained random test case generation, assertion and coverage driven Verification (Mulani et al., 2010). The Constrained Random Verification (CRV) improves the coverage dramatically in lesser time than directed test cases (RM, 2004; Spears, 2006; Mulani et al., 2010).

The Open Verification Methodology (OVM) was developed as a joint initiative of Mentor Graphics and the Cadence Design Systems that provides the first open and interoperable, SystemVerilog verification methodology in the VLSI industry. It is supported by multiple EDA vendors. It provides comprehensive verification due to its rich base classes and OOP features. OVM provides base classes library that allows users to create modular and reusable verification environment. It has been built over Advanced Verification Methodology (AVM) from Mentor Graphics and Cadence Universal Reuse Methodology (URM) (Kumar et al., 2012; Edelman et al., 2014). OVM based verification is better than normal SV environment because of a well structured testbench and reusability (Kumar et al., 2012).

UVM created by Accellera based on Open Verification Methodology (OVM) version 2.1.1 is a verification methodology for functional verification (Glasser, 2011; Yun et al., 2011). It is the latest methodology being used in current trends in verification. It is well suited for complex designs as UVM environment has reusable verification components (UVC) and base class libraries and is considered as a methodology of constrained random, coverage driven verification (CDV) (UVM, 2011b). CDV combines test cases generation, self-checking testbenches, and coverage metrics to reduce the time that is needed to verify a design [(Jain et al., 2012; Neumann et al., 2012).

3. I2C Master Core

The I2C bus was developed in the early 1980's by Philips Semiconductors. I2C is a two-wire, bi-directional serial bus that provides a simple and efficient method of data exchange between devices. It is most suitable for short distance communication between devices. It is a multi-master bus with collision detection and arbitration facilities to prevent data corruption in case of more than one master tries to access the bus simultaneously. The Device that provides the clock signal is considered to be master at that time. The DUT taken is wishbone compatible and has two interfaces, first the Wishbone classic interface and second I2C interface. The internal registers are configured using wishbone interface. Wishbone is an standard interface that is used in many designs so that it is easy to communicate between two devices means if the two designs are wishbone compatible then the signals are common and so communication is easier (WISHBONE, 2002). Wishbone signals used in this DUT are as shown in Table1. The I2C interface uses a serial data line (SDA) and a serial clock line (SCL) for data transfers. Data is transferred between a Master and a Slave on the SDA line in synchronization with SCL line on a byte-by-byte basis. Each data byte is 8 bits long (Herveille, 2003).

Table1: Wishbone Signals of DUT

Signal Name	Signal Description
wb_clk_i	Input master clock
Wb_rst_i	Input synchronous reset
arst_i	Input asynchronous reset
wb_adr_i (Jain et al., 2012)	Input lower address bits
wb_dat_i (Fitzpatrick, 2014)	Input data towards the core
wb_dat_o (Fitzpatrick, 2014)	Output data from the core
wb_we_i	Input write enable input
wb_stb_i	Input strobe signal/core select input
wb_cyc_i	Input valid bus cycle input
wb_ack_o	Output bus cycle acknowledge output

A standard communication consists of four main parts:
i. START signal generation ii. Slave address transfer
iii. Data transfer iv. STOP signal generation

A master initiates a transfer by sending a START signal. A START signal is defined as a high-to-low transition of SDA while SCL is high. The first byte of data transferred by the master immediately after the START signal is the seven bit slave address plus one read-write bit that tells the direction of data transfer. Slave acknowledges to master by pulling the SDA line low at the 9^{th} SCL clock cycle. On successful slave addressing, the data transfer can precede byte-by-byte in the direction specified by the read-write bit. Each transferred byte is followed by an acknowledge bit on the 9th SCL clock cycle. The master can terminate the communication by generating a STOP signal. A STOP signal is defined as a low-to-high transition of SDA when SCL is at logical high.

4. SystemVerilog Functional Verification Environment

A. Hierarchy of developed environment

The SystemVerilog code generated has the following components. The hierarchy of the code is as shown in Figure.1

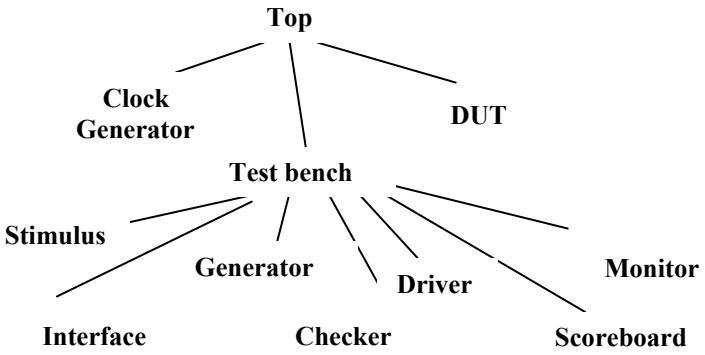

Figure 1: Hierarchy of Developed SystemVerilog Environment

The purpose of a Test bench is to check the correctness of the design under test (DUT). For this following steps have to be followed (Shetty, 2011), i. Generate stimulus. ii. Apply stimulus to the DUT. iii. Capture the response. iv. Check for correctness. v. Measure coverage.

i. Stimulus

Stimulus is the name given to various fields those may be randomized with required constraints. Random Stimulus makes the packet. The generator generates the random stimulus from random stimulus packet class and sends this stimulus to Driver using mailbox or by other means like callbacks (Spears, 2006; Fitzpatrick, 2014). In this environment mailbox has been used.

```
Class stimulus;
        rand logic [7:0] wb_dat_i;
        rand logic [2:0] wb_adr_i;
        ……..
                assert (tr.randomize);
                mbx.put (tr);     // putting transactions into the mailbox
                mbx.get (tr);     // getting the transactions from the mailbox
```

ii. Driver

Driver first unpacks the packet and translates the operations produced by the generator into the actual inputs for the DUT (VF, 2014). The Driver sends these signals using Virtual Interface to reset and configure DUT as shown in Figure.2. Driver also sends this stimulus to the Scoreboard using Mailbox or Callback.

```
Class drigen;
stimulus tr;                    //handle of stimulus class
mailbox mbx;                    //handle of mailbox
@ (posedge pif.wb_clk_i);       //stimulating the DUT at signal level at positive edge of clock
pif.wb_adr_i=tr.wb_adr_i;       //wishbone interface address input
pif.wb_dat_i=tr.wb_dat_i;       //wishbone interface data input
pif.wb_we_i=tr.wb_we_i;         //wishbone interface write enable input
pif.wb_stb_i=tr.wb_stb_i;       //wishbone interface strobe input
pif.wb_cyc_i=tr.wb_cyc_i;       //wishbone interface valid cycle input
```

Here, pif is the instance of interface. Similarly other signals are also there.

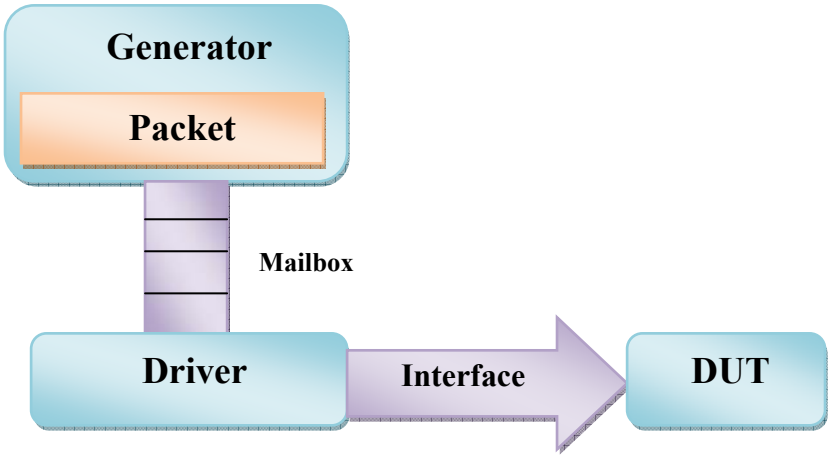

Figure.2 Mailbox, Packet, Interface

iii.Monitor

Monitor will keep track of scl and sda lines and also displays various messages according to the operations being performed like whether it is read or write operation. Similarly it also shows start, stop and transfer of data operations. In the monitor code Task 'run' is calling start, stop, data tasks.Figure.3 shows scl and sda lines for start and stop condition.

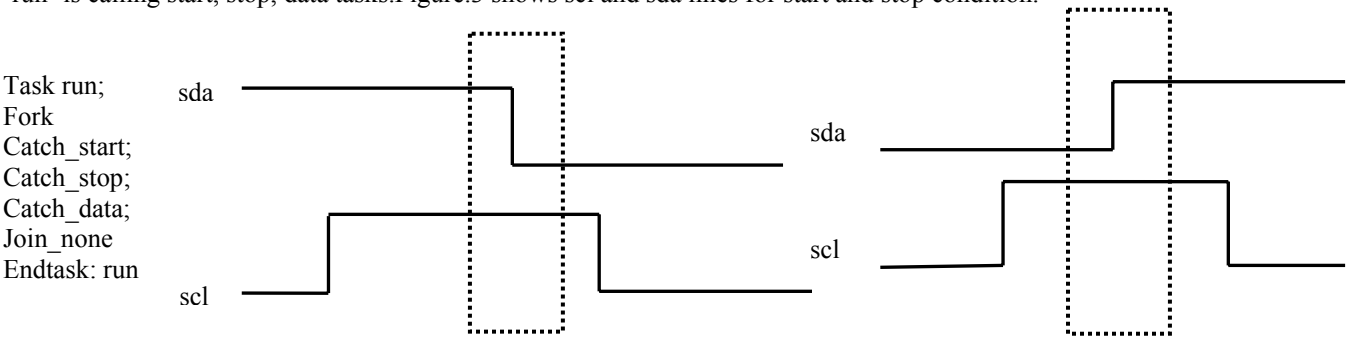

Figure.3 Start and Stop Condition

iv. Checker and Coverage

Checker checks whether the data is same as expected or not. Covergroups and coverpoints are inserted in the code to find the coverage.

```
Covergroup cg1;
Coverpoint tr.wb_adr_i;
Coverpoint tr.wb_dat_i;
```

B.Simulation Outputs

Some simulation outputs, messages obtained from monitor and coverage outputs are shown below.

i.Waveforms obtained

Figure.4 shows values of various configuration registers and core enable signal.

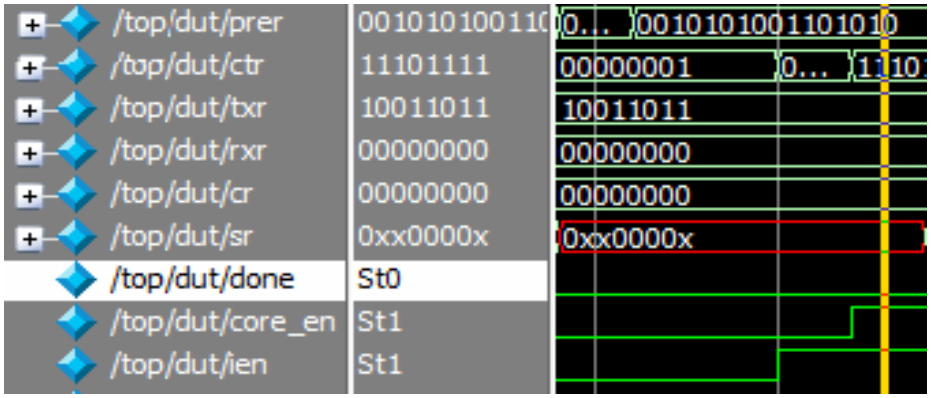

Figure.4 Simulations showing Enabled I2C core and Various Configuration Registers

ii.Monitor messages

Monitor gives message according to the operation being performed. Few are shown below.

@2490: The bit_operation is start, data: x
#@4790: The bit_operation is xfr, data: 1
#@7480: The bit_operation is stop, data: x

iii.Coverage output

Functional and total coverage obtained are shown in Figure.5 and in Figure.6 respectively.

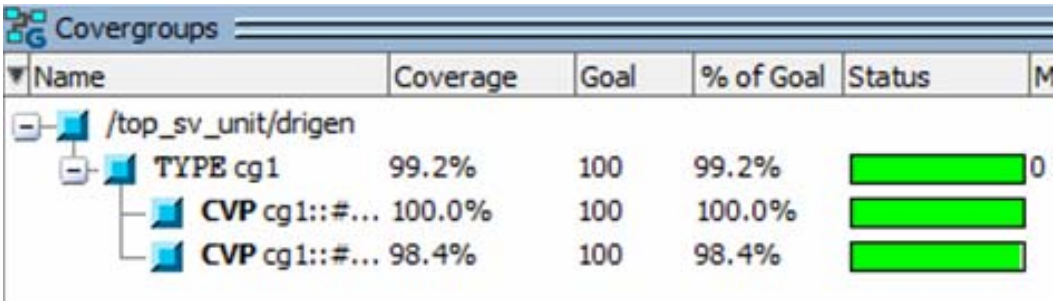

Figure.5 Functional Coverage Output

Coverage Summary by Type:

Weighted Average:				99.60%
Coverage Type	**Bins**	**Hits**	**Misses**	**Coverage (%)**
Covergroup	72	71	1	99.21%
Assertion Attempted	1	1	0	100.00%
Assertion Failures	1	0	-	0.00%
Assertion Successes	1	1	0	100.00%

Figure.6 Total Coverage Output

5. UVM Based Functional Verification Environment

A. Universal Verification Methodology

To adapt testbenches to increasing design complexities UVM evolved to develop a testbench, so that a testbench can be developed in the same time as the design itself. Thus we can say that because of many reasons like complex designs, time to market, better communication among companies and also within the company, verification experts have defined some common classes and functions which are definitely required to build test benches and have formed one library and this base class libraries with defined functions etc. is called UVM.

UVM is a verification methodology for functional verification (Aynsley and Doulos, 2010; Jain et al., 2012). UVM was created by Accellera based on Open Verification Methodology (OVM) version 2.1.1. UVM mainly uses simulations to do functional verification of the digital hardware. UVM environment has reusable verification components (UVC), and is considered as a methodology of constrained random, coverage-driven, verification (CDV) (UVM, 2011b). CDV combines test cases generation, self-checking testbenches, and coverage metrics to reduce the time that is needed to verify a design (UVM, 2011a). A verification component is instantiated and configured as per requirement. The UVM Class Library provides all the building blocks we need to quickly develop well-constructed, reusable, verification components and test environments as shown in Figure.7. The UVM library consists of base classes, many utilities, and macros (Glasser, 2011; UVM, 2011a). Components may be encapsulated and instantiated hierarchically and are controlled through an extendable set of phases to initialize, run, and complete each test. Building the environment using UVM has many advantages like reusability, lesser time to build testbenches, simple debugging of code and the most important is that test bench architecture and run phases are independent of vendors.

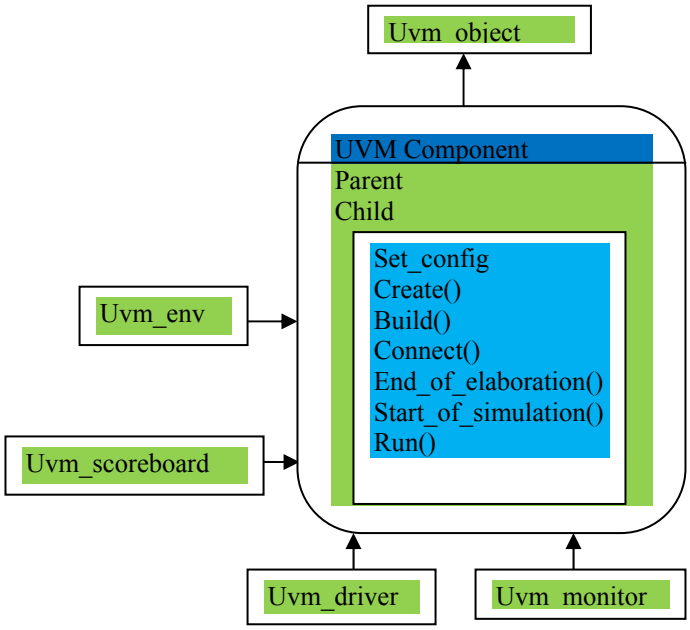

Figure.7 UVM class library hierarchy (partially) (UVM, 2011a)

B.UVM Testbench Architecture

An UVM based test bench has three main parts (Yun et al., 2011)

- Top module which instantiate the DUT, test bench and the interfaces to communicate between test bench components and the DUT.
- Testbench contains all UVM verification components UVCs. UVCs are reusable components those can be extended to the requirement. Test bench also contains register models and sequencers.
- Test Scenario part consists of objects of UVM classes which are inherited from its base library.

Hierarchy of UVM environment that has been built is as shown in Figure.8. The testbench has a number of class objects that are connected hierarchically.

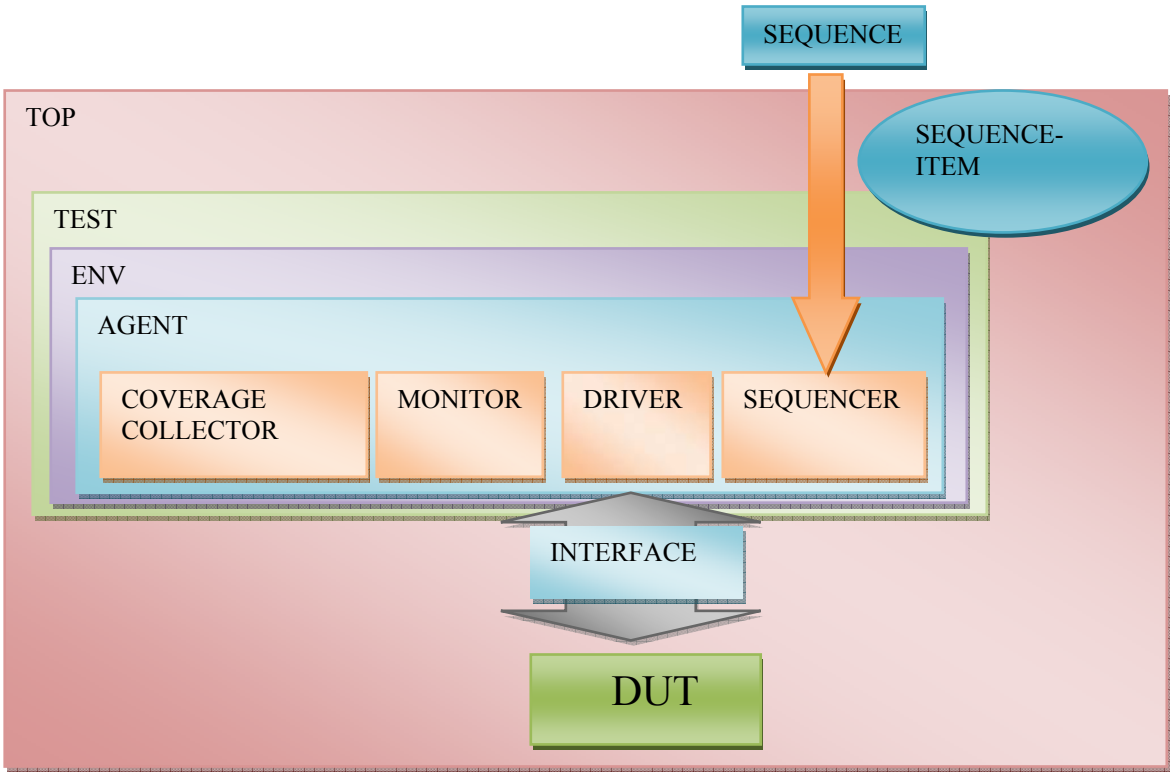

Figure.8 Hierarchy of developed UVM environment

i. Test, Environment

Top module connects the Test and DUT .The interface is passed in the top module to connect test to the DUT. Top module also has clock generator which is responsible for running the desired test. Test is the top level class in UVM. Test class initiates the construction process by building the next level down in hierarchy and initiates the stimulus by starting the main phase. Test class function is to instantiate the environment and then configures it for particular test at a time. Environment class derived from uvm_env class contains all the structural verification components of the testbench. Environment class may have one or more agents.The environment instantiates the testbench components like agents.

ii. Agent

Agent is the class present in the environment. It is responsible for communicating with DUT through an interface. Each interface of DUT communicates with its individual agent component. Agents can be active agent that emulates devices and drive transactions according to test directives or passive agent that monitors the DUT response.

iii. Sequence, Sequence Item, Sequencer

Sequence class is created by inheriting from the base class uvm_sequence and lies outside the top and works on sequence-item and sends these sequence items to the sequencer using two methods start_item and finish_item.

```
req = packet:: type_id::create ("req");
        start_item (req);
            if (! (req.randomize ()))
                `uvm_error ("error”,” randomization");
                finish_item (req);
```

Sequence item defines which pin level activity will be generated by agent or reports which pin level activity has been observed by the agent. Sequencer is a library component that synchronizes between one or more sequences and the driver. It routes sequence-item from a sequence where they are generated to/from a driver i.e. sequencer works as a sequence controller having built-in arbiter and buffer. REQ is the transaction type sent from a sequence to the driver and RSP is the transaction type sent from the driver to the sequence. The driver and sequencer are connected through a bi-directional Transaction Level Model (TLM) port/export connection as shown below.

```
driverr.seq_item_port.connect (sequencer.seq_item_export);
```

iv. Driver

Driver class converts data inside a series of sequence-items into pin level transaction. It takes input from sequencer and passes it to DUT through an interface.

v. *Monitor, Coverage Collector*

The monitor class observes the pin level activity and converts its observation into sequence items which are sent to components such as coverage collector that use them to calculate the functional coverage of the DUT. Coverage class contains one or more covergroups to gather functional coverage information. It is usually specific to a DUT.

C.Simulation Outputs

Various simulation outputs are obtained after stimulating the DUT, out of which some inputs and outputs are shown below

i.Waveforms obtained

Few outputs of the waveforms obtained after running simulation are shown in Figure. 9

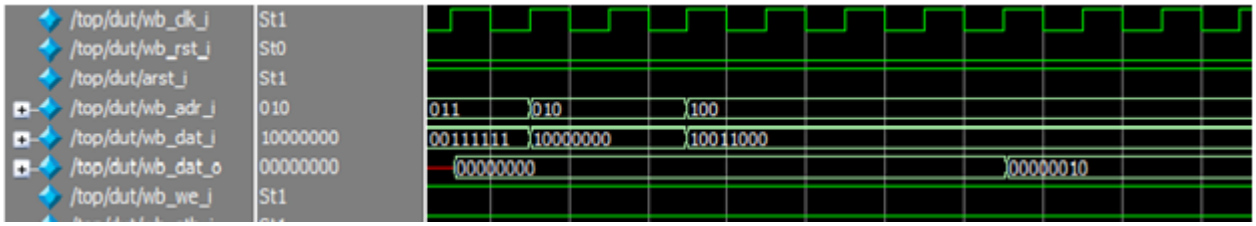

Figure.9 Simulations showing address and data inputs to the DUT for first few cycles

ii. Monitor messages

UVM_INFO monitor.sv (29) @ 135000 uvm_test_top.my_env_h.agent4inputs.my_monitor [START] pkt.wb_dat_i: 94 my_interface.wb_dat_i: ' 94 # pckt_1.wb_dat_i 94 = pkt.wb_dat_i 94

iii. Coverage outputs

Functional and total coverage obtained through UVM environment are shown in Figure.10 and in Figure.11 respectively.

Covergroups

Name	Coverage	Goal	% of Goal	Status	Merge_instances	Get_inst_coverage	Comment	% over Goal
/test_sv_unit/i2c_c...								
TYPE cg	96.4%	100	96.4%		0			96.4%
CVP cg::#c...	92.8%	100	92.8%					92.8%
CVP cg::#c...	100.0%	100	100.0%					100.0%

Figure.10 Functional coverage output

Coverage Summary by Structure:

Design Scope	Coverage (%)
uvm_pkg	100.00%
uvm_callbacks	100.00%
uvm_phase	100.00%
uvm_component	100.00%
test_sv_unit	96.42%
i2c_coverage	96.42%

Coverage Summary by Type:

Weighted Average:				98.21%
Coverage Type	**Bins**	**Hits**	**Misses**	**Coverage (%)**
Covergroup	15	14	1	96.42%
Assertion Attempted	22	22	0	100.00%
Assertion Failures	22	0	-	0.00%
Assertion Successes	22	22	0	100.00%

Figure.11 Total coverage output

6. Comparison: SystemVerilog and UVM based Environment

Developing the verification environment using two approaches that is SystemVerilog verification environment and the UVM based verification environment, we are comparing few testbench features. However the features somewhat depends on the implementer i.e. one can make any of the two environment ‘the best’. But in a general sense if the implementation is done in the same way, we can compare some features.

A. Simulation statistics

'Simstat' command gives performance-related statistics about active simulations. The statistics measure the simulation kernel process for a single invocation of vsim (QS, 2014).

simstats [memory | working | time | cpu | context | faults]

The simulation statistics obtained is tabulated as shown in Table2. Memory option returns the amount of virtual memory that the operating system has allocated for vsimk. CPU option gives simulation time for the test. For these two information we can also use $finish (2) option in our testbench. The simulation statistics obtained can be shown graphically as in Figure 12.

Table2: Simulation statistics of SV and UVM environment

Parameter	SV Env	UVM Env
Elaboration Time	20.2731	14.4538
Elaboration CPU Time	2.9643	2.6208
CPU Time	0.7213	0.639604
Memory usage	44 MB	25 MB

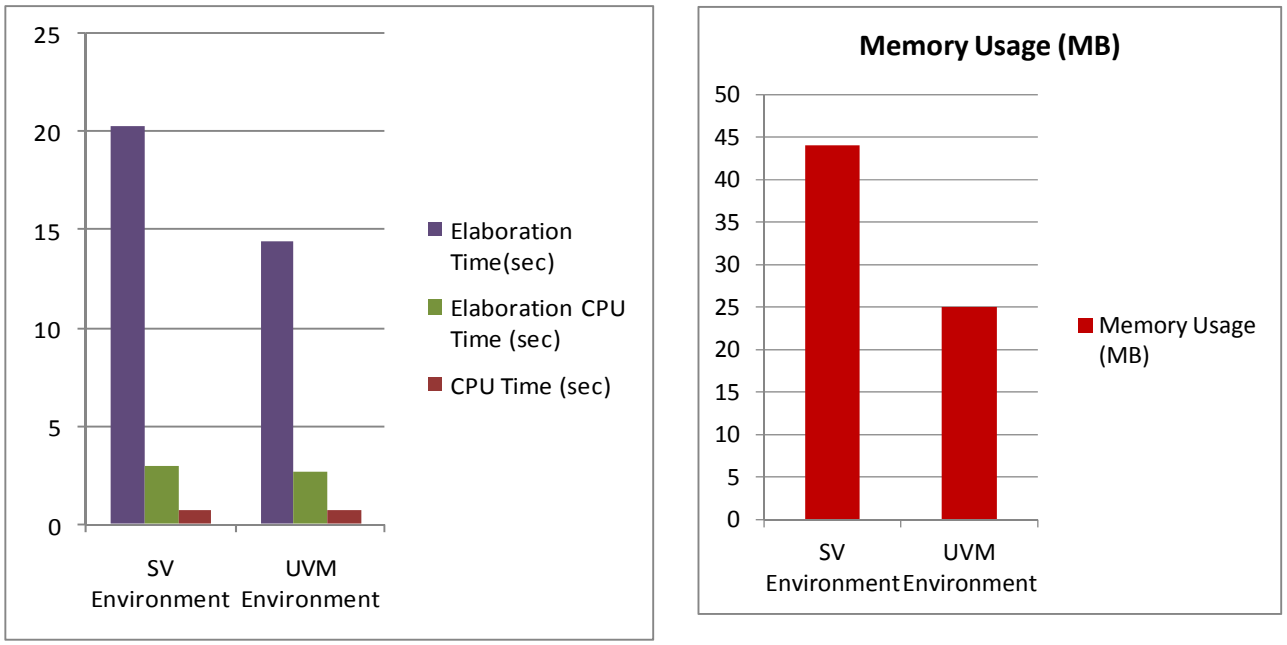

Figure.12 Simulation statistics obtained from SV and UVM environment

B.Debugging

A testbench may produce different messages. The severity level of each message may be different. If reporting is done with severity level, category and verbosity level, it is much easier to debug the errors. Verbosity is a positive integer whose lower value indicates higher importance. Thus reporting with these many properties allows managing the messages accordingly. These features are supported by UVM only and not by SystemVerilog. Each message can trigger actions based on their severity, ID and verbosity, like it can only be displayed to the transcript window or make log in a file or can call a hook function. Message would be printed only if the verbosity level is lower than or equal to the UVM report setting. Few messages used in our testbench are shown below.

```
if (! (req.randomize ()))
`uvm_error ("error", "randomization");
if (! uvm_config_db # (virtual i2c_interface):: get (this,"","dut_vi",my_interface))
`uvm_fatal ("No virtual interface", "Virtual interface must be set");

`uvm_info ("data", $sformatf ("req.inst=%d", req.wb_adr_i), UVM_HIGH);
```

This message will only be printed if the uvm report verbosity is set to uvm_high (300) or higher.

Uvm_verbosity is set by the user. It can be set for all components. In comparison to the above features, we have only $display option in case of SystemVerilog. So messages with any severity level can only be printed but simulation can't be stopped

depending upon the type of message. This makes debugging much harder in case of SystemVerilog. Below Figure.13 shows UVM Vs SV testbench message upon getting an interface error. In case of UVM simulation stopped upon getting an error, while SV testbench is displaying the message only.

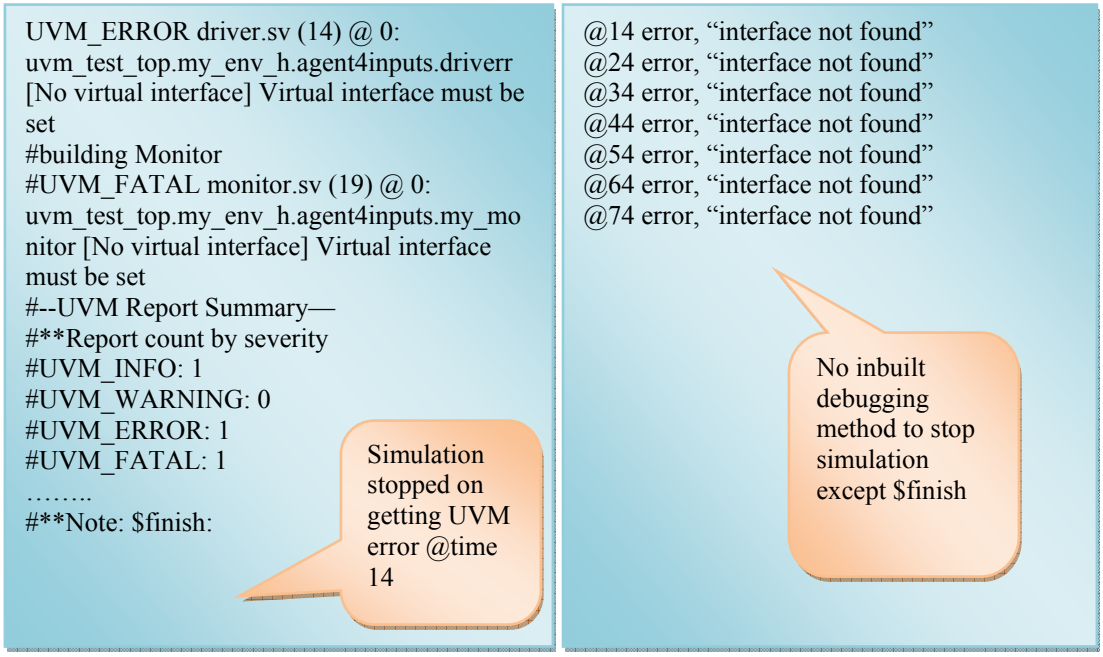

Figure.13 UVM Vs SV Testbench message upon getting an interface error

C.Reusability

Built-in UVM component base classes are extended to build the UVM testbench. In UVM there are built-in base classes and methods that can be used and extended depending upon requirement. Base class and methods reuse concept makes it very easy to build the new extended class. Extended class has to only call the methods of base class. Reuse of groups of verification components either in different projects or at a higher level of integration in the same project makes UVM testbench architecture very modular (UVM, 2014). Below is a part of our UVM environment showing reuse concept that is not present in SystemVerilog environment.

```
class my_driver extends uvm_driver # (packet);
`uvm_component_utils (my_driver)
virtual i2c_interface my_interface;
        task run_phase (uvm_phase phase)
         ……
          begin
         seq_item_port.get_next_item(req)
```

Reusing driver class

Reusing method

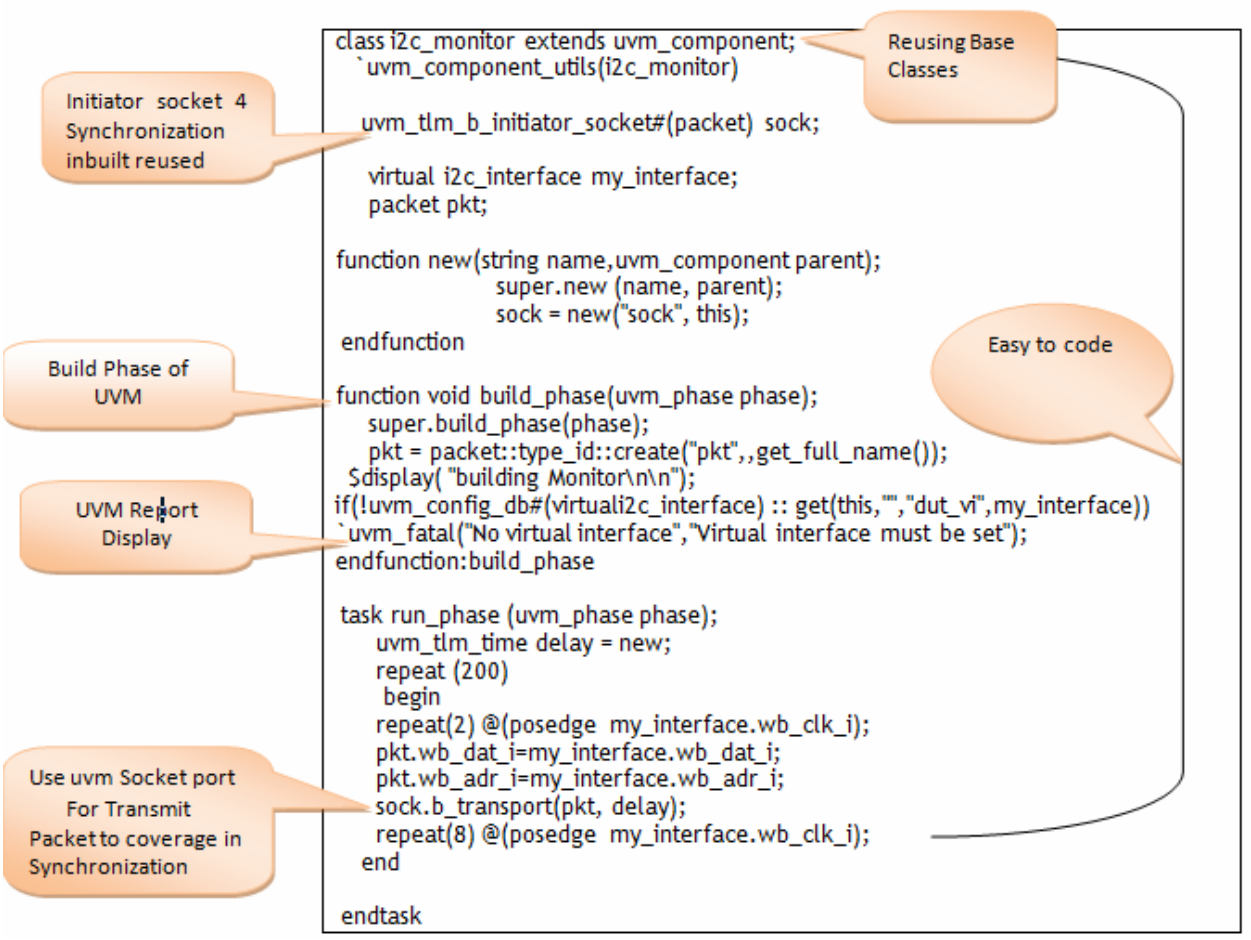

Figure.14 UVM testbench part showing reuse, report, synchronization concepts

D.Synchronization

To communicate between two verification components there should be some synchronization scheme, so as not to lose any data. UVM environment has Transaction Level Modeling (TLM) ports for these purpose.TLM connections are parameterized to the type of transaction to synchronize the communication between verification components. Figure.14 shows how easy is to send a packet in synchronization from monitor to coverage through a UVM socket port.TLM port connections are made in UVM connect phase. In opposition to this there is no such facility in SystemVerilog. A user has to make a way to synchronize the communication between components, like use of SystemVerilog events and mailboxes as depicted from Figure.15.

E.Testbench Structure and run flow

UVM has three main types of classes, uvm_components, uvm_objects and uvm_transactions. UVM testbench is built from classes derived from uvm_components base classes. Object Oriented Programming (OOP) technique and library of base classes facilitate the creation of structured testbench i.e. it is very much fixed that which component contains which other components (UVM, 2014). We also have option to print the topology of testbench using following command.

uvm_top.print_topology ();

The output obtained through this command is shown below, from which It is very clear that uvm testbench has a fixed topology. Standardization of testbench structure makes the Verification Intellectual Properties (VIPs) more reusable and vendor independent. There is no such in built option in SystemVerilog to print the testbench topology until we intentionally display it. In addition to this UVM also has a consistent testbench execution flow because of uvm phases. Three main phases are build phase, run phase and clean up phase (UVM, 2014). All of these have sub-phases. These phases make verification components to develop in isolation and execute the main steps in an order that take place during simulation.

```
# UVM_INFO @ 63990: reporter [UVMTOP] UVM testbench topology:
# Name                    Type
# uvm_test_top            my_test
```

```
# my_env_h              wb_env
# agent4inputs          driving_agent
# driverr               my_driver
#   rsp_port            uvm_analysis_port
#   sqr_pull_port       uvm_seq_item_pull_port
# my_coverage           i2c_coverage
#   sock                uvm_tlm_b_target_socket
#my_monitor             i2c_monitor
#   sock                uvm_tlm_b_initiator_socket
#sequencer              uvm_sequencer
#   rsp_export          uvm_analysis_export
#   seq_item_export     uvm_seq_item_pull_imp
#   arbitration_queue    array
#   lock_queue           array
#   num_last_reqs        integral
#   num_last_rsps        integral
```

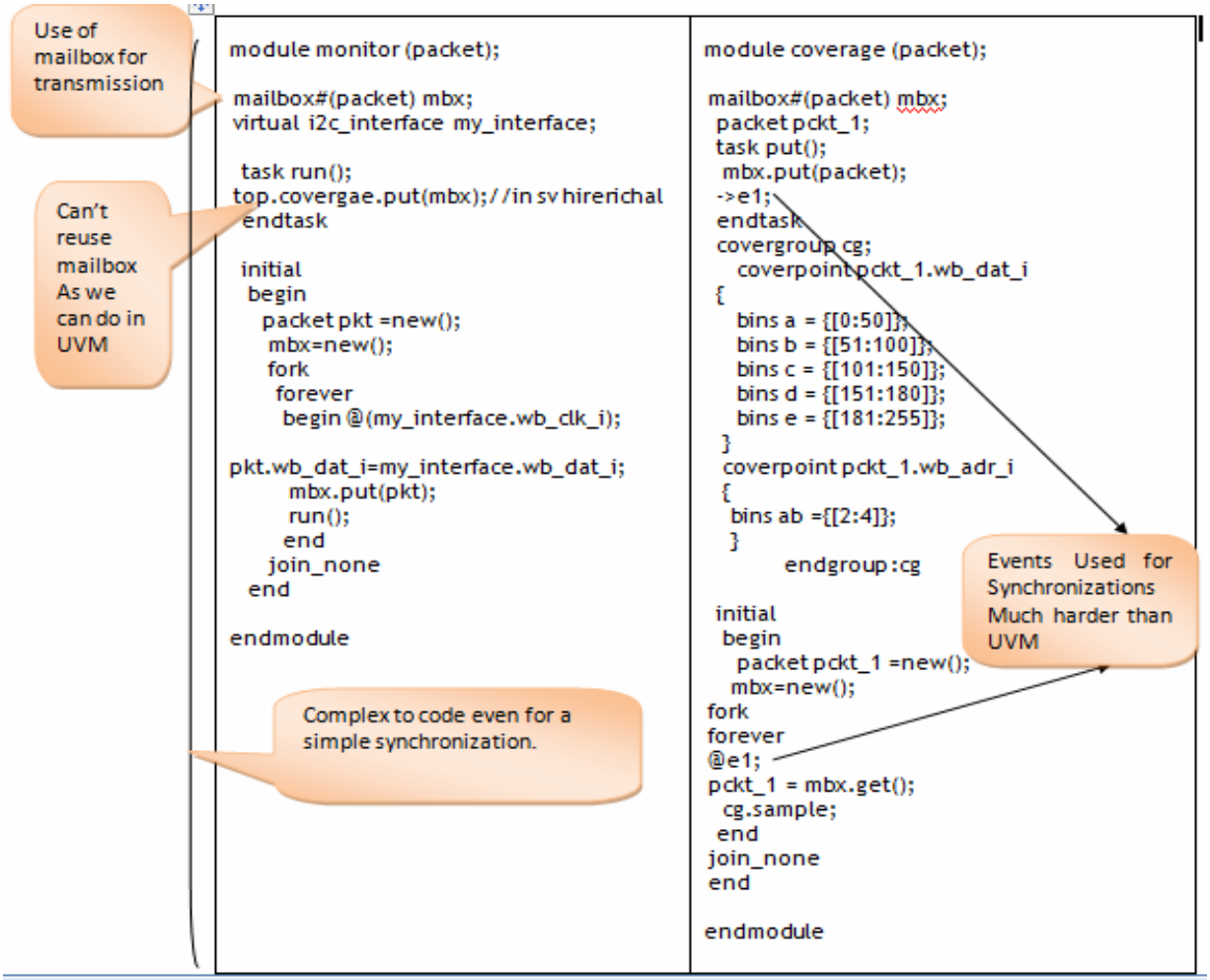

Figure.15 SystemVerilog environment showing use of mailbox and events for synchronization

7. Results And Discussion

From the above two figures we can see that in a UVM testbench we have base classes of verification components and many methods those can be reused. In UVM we also have constructs to make synchronization an easy task as compared to SystemVerilog mailboxes and events. UVM also makes debugging very simple as compared to SystemVerilog, because of many options available for displaying the messages depending upon their severity. From the above comparison we can reach to a conclusion that UVM based environment is better in many terms than SystemVerilog environment. Few of them are as

- Better Performance in terms of CPU time and memory usage.
- Much easier debugging
- Reusability of base classes, methods, environment
- Easier to synchronize the communication between verification components
- Very much structured testbench
- Fixed run flow because of uvm phases
- Easy to prepare the code
- Lesser time to build the testbench

8. Conclusions

The SystemVerilog verification environment developed along with the complete flow of verification has been discussed. The various classes for driver monitor, stimulus, environment etc. and modules or programs made have been compiled and simulated and the outputs observed are shown. The environment created completely wraps the DUT and functional and assertion based coverage have been found. Assertion coverage found is 100% and functional coverage is found as 99.21%. The overall coverage is found to be 99.60%. The SystemVerilog environment developed has been extended to UVM by calling the base class library and various verification components. The UVM based environment developed has been discussed with a little discussion of UVM. The various classes of UVM testbench like test, environment, agent, sequence, sequencer, driver, monitor, coverage collector have been compiled and simulated. The outputs obtained are shown. Assertion coverage found is 100% and functional coverage is found as 96.42%. The overall coverage is found to be 98.21%.

The two environments developed are compared for some testbench features like performance, debugging, reusability, and synchronization. From comparison it has been found that UVM based environment is better in many terms. It is easy to develop, once we know the concepts of UVM. It is also better in terms of performance, reusability, synchronization, and debugging, structure and run flow.

.

Acknowledgement

The authors are grateful to their organization for its help and support.

References

Glasser M., 2011. UVM: The Next Generation in Verification Methodology, Methodology Architect, February 4, Courtesy of Mentor Graphics Corporation.
URL:http://s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_65287.pdf

Yun Y.-N., Kim J.-B, Kim N.-D., Min B., 2011, Beyond UVM for practical SoC verification, *SoC Design Conference* (*ISOCC*), 2011 *International*, 17-18 Nov., pp. 158-162.

Jain A., Bonanno G., Gupta H. and Goyal A., 2012. Generic system verilog universal verification methodology based reusable verification environment for efficient verification of image signal processing IPs/SoCs, *International Journal of VLSI design & Communication Systems* (*VLSICS*), Dec.,Vol. 3, No. 6, pp. 13-25.

Aynsley J., Doulos, 2010. UVM Verification Primer, June.
URL:http://www.doulos.com/knowhow/sysverilog/uvm/tutorial_0/

Universal Verification Methodology (UVM) 1.1 Users Guide, June 2011a.

Universal Verification Methodology (UVM) 1.1 Class Reference, June 2011b.

Neumann F., Sathyamurthy M., et.al. 2012. UVM-based verification of smart-sensor systems, *International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design* (*SMACD*), pp. 21-24.

Fitzpatrick T., 2014. Realizing advanced functional verification with Questa, courtesy of Mentor Graphics Corporation, URL: http://s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_27149.pdf, Accessed 30th April.

Spears C., 2006. SystemVerilog for Verification, A Guide for Learning the Testbench Language Features," Second Edition, Springer.

VF, 2014. URL: http://testbench.in/TB_07_VERIFICATION_FLOW.html, Accessed 30th April.

Mulani P., Patoliya J., Patel H., Chauhan D., 2010. Verification of I2C DUT using SystemVerilog, *International Journal of Advanced Engineering Technology*, Oct.-Dec., Vol. 1, No. 3, pp 130-134.

WISHBONE 2002, WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Opencores.org, revision B.3, Sep 7.

RM, 2004, SystemVerilog 3.1a Language Reference Manual Accellera's Extensions to Verilog®, Accellera Organization, Inc.

Kumar B.S., Chandra L.R. et. al, 2012. Design and functional verification of I2C master core using OVM, *International Journal of Soft Computing and Engineering,* ISSN: 2231-2307, May, Vol-2, No. 2, pp. 528-533.

Edelman R., Crone A., et.al. 2014. Improving Efficiency, Productivity, and Coverage Using SystemVerilog OVM Registers, courtesy of Mentor Graphics Corporation
URL: http://s3.mentor.com/public_documents/whitepaper/resources/mentorpaper_49112.pdf, Accessed 30th April.
Herveille R., 2003. I2C-Master Core Specification, Rev. 0.9, July 3.
Shetty A., 2011. SystemVerilog Testbench Tutorial," Nano-Electronics & Computing Research Center, School of Engineering, San Francisco State University, San Francisco, CA, Fall.
QS, 2014. Questa Simulator User's manual software version 10.0d, Mentor Graphics Corporation, Accessed 30th April.
UVM, 2014. uvm-cookbook-complete-verification-academy.pdf, courtesy of Mentor Graphics Corporation.
URL:http://verificationacademy.com, Accessed 30th April.

Biographical notes

Rakhi Nangia, a student of Master of Technology (M.Tech) in VLSI Design at ITM University, Gurgaon, (Haryana) India. She has completed B.Tech in Electronics and Communication Engineering from Aligarh Muslim University in 2003. She has worked in LRDE, DRDO, Bangalore in 2003 and then in VLSI R&D industries Logic Eastern and Virage Logic International (now Synopsys) from 2006 to 2009. She has also worked as an Assistant Professor in Manav Rachna University, Faridabad and in other Engineering Colleges from 2009 to 2012. Her interest areas are Digital System Design, ASIC Design, VLSI Testing and Verification.

Neeraj Kr. Shukla (IETE, IE, IACSIT, IAENG, CSI, ISTE, VSI-India), an Associate Professor in the Department of Electrical, Electronics & Communication Engineering, and Project Manager – VLSI Design at ITM University, Gurgaon, (Haryana) India. He received his PhD from UK Technical University, Dehradun in Low-Power SRAM Design and M.Tech. (Electronics Engineering) and B.Tech. (Electronics & Telecommunication Engineering) Degrees from the J.K. Institute of Applied Physics & Technology, University of Allahabad, Allahabad (Uttar Pradesh) India in the year of 1998 and 2000, respectively. He has more than 50 Publications in the Journals and Conferences of National and International repute. His main research interests are in Low-Power Digital VLSI Design and its Multimedia Applications, Digital Hardware Design, Open Source EDA, Scripting and their role in VLSI Design, and RTL Design.

Received November 2013
Accepted February 2014
Final acceptance in revised form April 2014