International Journal on Recent and Innovation Trends in Computing and Communication ISSN: 2321-8169
Volume: 3 Issue: 3 1302 – 1306

A Technical Road Map from System Verilog to UVM

Ms. Deepa Kaith
Student- M.Tech ECE
Amity University Haryana
Gurgaon, Haryana
deepakaith@rediffmail.com

Dr. Janakkumar B. Patel
Professor (ECE)
Amity University Haryana
Gurgaon, Haryana
janakbpatel71@gmail.com

Mr. Neeraj Gupta
Assistant Professor (ECE)
Amity University Haryana
Gurgaon, Haryana
neerajsingla007@gmail.com

Abstract**—** As the fabrication technology is advancing more logic is being placed on a silicon die which makes verification more challenging task than ever. More than 70% of the design cycle is used for verification. To improve the time to market we need a reusable verification environment that detects all functional errors and avoid re-spin. Universal verification methodology was introduced to fulfill these goals. UVM is well structured, reusable with little or no modifications, do not interfere with the device under test (DUT) and gives the speed of verification. UVM is supported by all major simulator vendors, which was not in earlier methodologies. This methodology provides a standard unified solution that compiles on all tools. This paper introduces the advantages of UVM over System Verilog, basic terminologies used in UVM and a simple functional verification environment construction using UVM.

Keywords- *Functional Verification, Universal Verification Methodology, DUT, System Verilog*

*****

I. Introduction

As the design becomes large and concurrent, it becomes difficult to verify the functionality of the design using traditional testbenches. Thus, hardware verification languages like system verilog are used for designing. More than 70% of the time is spent on verification which also consumes more resources than the design [1]. This arise the need for developing modular, reusable and robust environment for verification. Open Core Protocols (OCP) were introduced to interface address communication between the functional units of System on Chip. OCP provides independence from bus protocols without sacrificing high performance access to on-chip interconnects [6].

The Open Verification Methodology (OVM) developed as a joint initiative of Mentor Graphics and the Cadence Design System provides the first open and interoperable verification methodology in the VLSI industry. Then Mentor’s AVM, Mentor & Cadence’s OVM, Verisity’s eRM, and Synopsys’s VMM-RAL were introduced [11].

Universal Verification Methodology (UVM) created by Accellera based on OVM version 2.1.1 is methodology for functional verification. UVM 1.0 was released on May17, 2010 [6]. Its Class Library provides the building blocks needed to quickly develop well-constructed and reusable verification components and test environments. It uses system Verilog as its language an all three of the major simulation vendors (Synopsys, Cadence and Mentor) support UVM today which was not possible with earlier verification methodologies.

Verification is the process for ensuring the specifications of the design unit prior to mapping it into the chip. As the design becomes large and complex there are more chances of bugs in the design and that requires diverse verification which is done at unit level, block level, subsystem level and at IP level. Verification of a design is the most critical phase in chip design cycle. System-Verilog is a special hardware verification language that provides complex data types and constructs to enable a higher level of abstraction and modeling of complex data types [1]. Similarly a methodology is applying a language in a planned and structured way for doing verification of a design unit.

Universal Verification Methodology is an open source methodology for using System-Verilog. It is designed mainly for verification IP and testbench components so that testbenches are reusable and verification code is more portable and universal [2][3]. Each verification component follows a consistent architecture and a complete set of elements for simulating, checking and collecting functional coverage. The verification environment developed through System-Verilog may be different depending upon implementer, while that built using UVM remains the same irrespective of the vendor.

II. SYSTEM VERILOG

SystemVerilog (SV) is a special digital hardware verification language used for functional verification of the design. It is a complete object-oriented programming language that includes domain-specific features to support verification [3]. SystemVerilog provides constructs and can be used to simulate the HDL design and verify them by high level test case. SystemVerilog aims to provide a constrained random generation, temporal assertion, and functional coverage constructs. Despite the richness of the base SystemVerilog language the implementer faces many challenges while using System Verilog which limits its usefulness.

A. Supports large features, lacks reusability

The SystemVerilog language reference manual defines more than 200 reserved words, run over 1300 pages and it continues to develop. Due to richness in features all the major vendors provides a different subset of the System Verilog. It presents a chief obstacle if for any commercial or technical reason the user wants to transfer their program code from one vendor's tools to another. Consequently, software developed using one vendor's tools would be unusable with a different vendor's implementation.

B. An open reusable verification IP

The creation of a high quality infrastructure for verification of even one interrelated protocol is a difficult task. So, there is an open active market for verification intellectual property (VIP) in which third party supplier configures and maintains the reusable *verification components* (VC). The market's successful operation requires that the VCs of various vendors are interoperable in each user's design environment. For this a common language to have a consistent form and a shared set of rule regardless of the vendor is to be used.

C. An easily usable toolkit

In every project verification has a persistent set of problems that is chosen to be solved off-the-shelf. System Verilog provides a rich set of library functions with ready-to-use execution of a common constraint [5]. It is thus, logical to have verification-focused library that can be made accessible by all verification engineers.

III. DISTRIBUTING TASKS AMONG A LANGUAGE AND A LIBRARY

SystemVerilog is very large and it is irrationally difficult for a designer to become familiar with all the features, drawbacks and difficulties. If the language was smaller and the features were added in the form of library support it would be preferable. Also, it is difficult for this language to carry out every specific requirement as the features are built-in and cannot be extended by user.

A. Some features should be a part of core language

As the language is very large it seems better to remove some core features and put them in the library. But there are few domain-specific requirements which cannot be easily implemented as a library function. Apart from this, a large subset of System Verilog like switch and gate-level modeling must be continued for backward compatibility support.

B. Some features should be a part of a library

If any language is powerful enough to allow some of the functionality as an add-on library, it is unreasonable to add that functionality to the core programming language [7]. System Verilog has some built-in functions like string and array implementations that can be provided as library functions without considerable loss of usability.

The balance between SystemVerilog and the UVM to provide a designing language and a verification methodology is broadly right. Methodology like major base classes and synchronization should be provided in a library and not be built into the language as exactly in UVM. This helps in the growth of the toolkits like UVM and also to take the advantages of the upgradation in the methodology and also avoid the interference that would arise if the alterations were introduced to the main language.

IV. COMPARISON OF SV AND UVM

- In UVM communication is done using ports and exports and in system verilog mailboxes are used for the same.
- It takes less time to develop a testbench using UVM as compared to System Verilog.
- UVM has many predefined functions that can be called directly from the library. In system Verilog we have to write our own logic codes for functions like copy, print, pack etc.
- Many predefined macros are available in UVM which are not available in System Verilog. eg `uvm_error, `uvm_info etc.
- The testbench developed using UVM is interoperable and robust.

V. UNIVERSAL VERIFICATION METHODOLOGY

The testbench designed to determine the correctness of the DUT so that the design meeting the specification is confirmed. The testbench creates constrained random stimulus, and gathers functional coverage [7]. The testbench includes the following steps given below

- Generate the stimulus
- Apply stimulus to the DUT
- Gather the response
- Observe the correctness
- Measure progress against overall functional coverage.

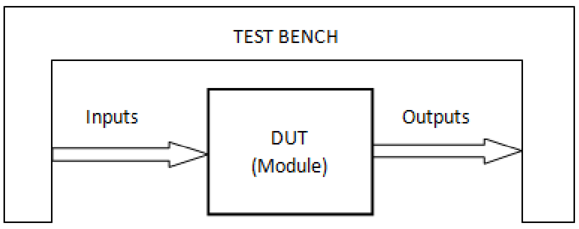

Fig.1 Basic Testbench Environment

A. Testbench architecture using UVM

A testbench in UVM can be divided into following three parts. The *Test,Top* module comes first which instantiates the DUT and interfaces for communication with main testbench components. Then comes *Testbench* containing all UVM verification components, sequencer and register model. The third is *Test scenario* which defines the input stimulus that is applied as a sequence in UVM. All components in these are *objects* of classes which are inherited from class library. The testbench contains Universal Verification Components used for interfacing is a reusable verification IP. Virtual sequencer class has all interfaces and register block handles for every IP [3]. The uvm_test class defines the *Test scenario*, the testbench for the DUT and is specified in the *Test top*. Testcase creates an Environment object and defines the required test specific functionality. Verification environment contains the declarations of the virtual interfaces. These virtual interfaces are made to point to the physical interfaces in the Testcase which are declared in the *top* module [10].

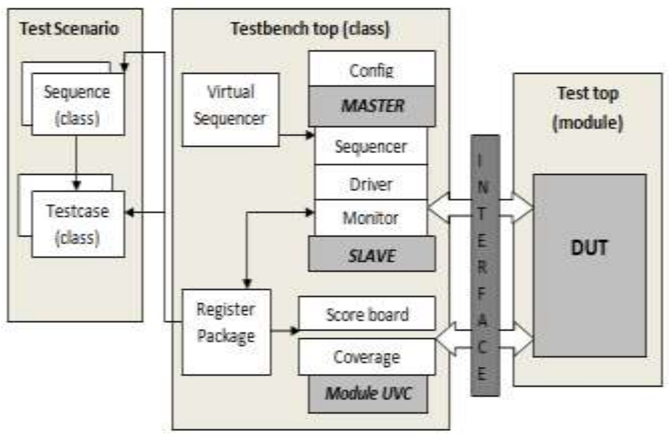

Fig.2 Architecture of Testbench using UVM

B. UVM Class Library

This Class library provides the basic building blocks needed for developing well-constructed and reusable VCs and test environments [11]. Its library provides *base* class and the facility to configure them. Base class falls into two different categories: *components* or *data*. The *uvm_component* provides a component class hierarchy used to make permanent structures of the testbench. The *uvm_sequence_item* provides data class used to design transactions.

C. UVM Component Class

All the components in a UVM for verification infrastructure are derived from the *uvm_component* class and it forms a hierarchy which includes: sequencers, drivers, monitors, scoreboards, environments, coverage collectors and tests.

Design Under Test
It is the design whose specifications need to be confirmed. This is basically the RTL description in the designing language. It tells the features and the functions of the design.
Sequencer
Sequencer is the entity on which the sequences will run. To test DUT behavior, sequence of transaction needs to be applied. Sequencer runs stimulus generation code and sends sequence items down to driver whenever driver demands by it.
Driver
Driver is used to drive the DUT signals. It receives the transaction object from the sequencer and maps the sequence items to the signal level format required by the DUT interface. It can generate read, write, address or data signal to be transferred [6]. It is the active element of the verification logic.
Monitor
A monitor is the passive element of the verification environment and is independent to an application. It scans the DUT signal to and from the interface without driving them. It assembles the pin information in the form of a packet and then transfers it to scoreboard and test verification environment for coverage information.
Agent
Agent is an abstract container. It encapsulates a driver, a monitor and a sequencer. It has two modes of operation: passive and active [1]. In active mode it drives the signal to the DUT and thus, driver and sequencer are instantiated in active mode. In passive mode it scans the DUT signals without driving them. So, monitor is instantiated in passive mode.
Scoreboard
Scoreboard is build to check the response from the DUT against the expected response. It is done by comparing them to the Reference Model. It keeps the track of how many times the response matched and how many times it failed.
Coverage collector
Coverage collector measures the verification process by registering the kind of tests and results that can occur in a Functional Coverage Model in advance. Both coverage collectors and scoreboards code is usually highly application-specific and less affected by the interface protocols and timings [11].
Environment
It is at the top of the test bench architecture that assembles the structure. It contains one or more agents, global scoreboard and other components for measurement and checking depending on design. It has parameters to be used for restructuring and reusing for various scenarios.
Test
It is the top-level of the component hierarchy in which interface instances and clock is generated. DUT instance is formed and combined with the interface instance. Tests in UVM are classes that are derived from an uvm_test class. The test class enables configuration of the testbench and verification components to determine the dynamic behavior of the processes by using sequences.

D. UVM Data class

The UVM data domain in the verification environment is represented by:

Data Items/Transaction

Data item are basically the input to the DUT. All the transfer between different verification components in UVM is done through *transaction* object. Data items are generated and applied by top level Test and by randomizing the data item object we can check corner cases and maximize the coverage on the DUT.

Sequence items

They are the primary data objects that are passed between components. Sequence items represent communication at an abstract level.

Sequences

These are gathered from sequence items and to build a real set of stimuli. Sequences create a specific pre-determined set of randomized transactions. Sequences can run other sequences and can also be layered providing transactions at various levels.

E. The UVM Class Library Hierarchy

uvm_object is the base class for all components and sequences in UVM. uvm_component class is derived from this class and all uvm components extends the uvm_component class. Transaction class is derived from uvm_object class and sequence_item and sequence extends the uvm_transaction class [2].

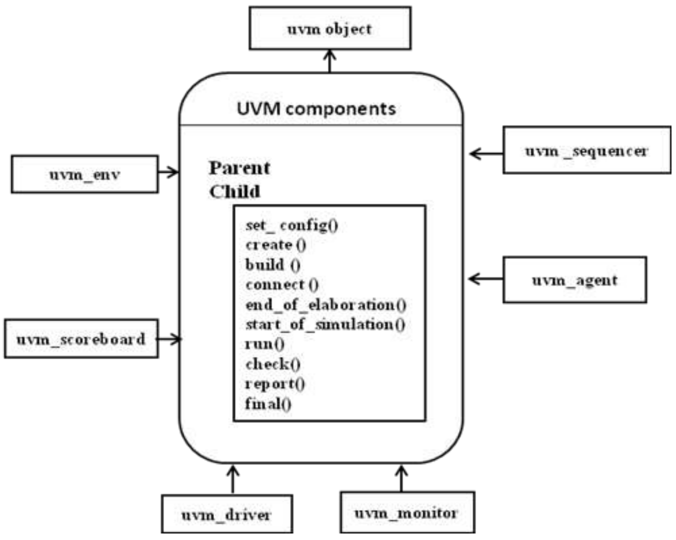

Fig.3 Partial UVM class library hierarchy

F. UVM Phases

Different from System Verilog, UVM simulation runs in predefine phases. All the components used in the verification environment need to implement the phase methods [6] and this will be called in order as defined in fig.3.

build phase: It is used to instantiate the child and parent component instance .
connect phase: It is used in child components to connect ports to exports, exports to ports and ports to ports.
end_of_elaboration phase: It is used to provide fine-tuning in the testbenches, to print the topology and for opening the files. It indicates that verification environment has been completely assembled.
start_of_simulation phase: It gives notification to DUT for simulation and indicates that verification environment is completely configured and is ready to start.
run phase: It is used to run simulation and is divided into several run phases. It is the only phase using task to define as this phase consumes more time. All other phases run in zero simulation time use function.
extract phase: It is used to extract data from different points of the verification environment. It will take all data from scoreboard and extract it.
check phase: It checks any unexpected condition in verification environment.
report phase: It provides the report of the particular performed test .
final phase: It tells that all the phases are completed and that simulation can be terminated.

VI. CONCLUSION

It is concluded that SystemVerilog lacks built-in reflection and has only limited macro and function capabilities as compared to UVM. Although there are a few SystemVerilog features that could instead have been provided as library functions and using SystemVerilog and UVM together, the balance between library and core is broadly satisfactory. It is not easy to build a robust and reusable verification environment. A proper framework and support from the base classes is needed to construct it. UVM provides a rich set of base class library and features required for efficient verification. It gives an environment that is robust, easy to understand and thus, reusable by others vendors. Using UVM it requires less time to generate a testbench as it offers higher level of abstraction. It covers almost all the possible scenarios and corner cases and thus, increases the functional coverage.

VII. REFERENCES

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

[2] Accellera Organization, "Universal Verification Methodology (UVM) 1.1 Class Reference", June 2011

[3] Young-Nam Yun, Jae-Beom Kim, Nam-Do Kim, Byeong Min, “Beyond UVM for practical Soc Verification”, IEEE- 978-1-4577-0711-7, pp 158-162, 2011.

[4] Neumann F, Sathyamurthy M. Kotynia L, “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, June 2012.

[5] Jain A., Bonanno G., Gupta H. and Goyal A., “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),Vol. 3, No. 6, pp. 13-25, Dec 2012.

[6] Bhaumik Vaidya, Nayan Pithadiya “An Introduction to Universal Verification Methodology” Journal of Information Knowledge and Research in Electronics and Communication Engineering-ISSN: 0975 – 6779, Volume – 02, Issue – 02, pp 420-424 Oct. 2013.

[7] Nimesh Prajapati, “How easier to built Basic Verification Testbench using UVM compared to SystemVerilog” International Journal of Engineering Research & Technology (IJERT), ISSN: 2278-0181, Vol. 2 Issue 11, November – 2013

[8] Alexander W. Rath, Volkan Esen and Wolfgang Ecker, “A Transaction-Oriented UVM-Based Library for Verification of Analog Behavior”, IEEE- 978-1-4799-2816-3, pp 806-811, 2014

[9] Hung-Yi Yang, “Highly Automated and Efficient Simulation Environment with UVM” IEEE -978-1-4799-2776-0, 2014

[10] T Tarun Kumar, CY Gopinath “Verification of I2C Master Core using System Verilog-UVM” International Journal of Science and Research (IJSR), ISSN - 2319-7064, Volume 3 Issue 6, June 2014.

[11] Juan Francesconi, J. Agustin Rodriguez, Pedro M. Julian “UVM Based Testbench Architecture for Unit Verification” ISBN: 978-987-1907-86-1 IEEE Catalog Number CFP1454E-CDR, 2014.