Bulletin of the *Transilvania* University of Braşov • Vol. 10 (59) No. 2 - 2017
Series I: Engineering Sciences

SPECMAN-UVM BASED TESTBENCH

F. IONIȚĂ[1] M. CARP[2]

Abstract: *The scope of the document is to present the advantages of using the Universal Verification Methodology (UVM) in creating a verification environment for a given block of the chip over others methodologies. The Hardware Verification Language (HVL) in which testbench written is called Specman. The paper will cover the basic building blocks of a Universal Verification Component (UVC) and the way they are used into creating a flexible testbench and validating a given Device Under Test (DUT).*

Key words: *UVM, UVC, Specman.*

1. Introduction

There are many families of integrated circuits on the market capable of offering a great deal of computing power. This is because the integrated circuit industry reached a point where it is able to "pour" in a few square millimetres of silicon, billions of transistors. Reaching such transistor density gives a lot of "freedom" to the designers, now being able to implement complex hardware algorithms, boosting overall computing power and speeding up the response time of the chip.

Before entering production, a newly designed Integrated Circuit (IC) must be tested in many ways to ensure that it is operating as desired. The verification of the IC begins from its early design stages where the source files of the design itself and a specially developed testbench are fed into a simulator. This offers the possibility of finding bugs in the design long before it is sent into production, minimizing the cost of the project.

Further, in this article, the testbench used for verification will be described, alongside with the UVM techniques used to implement it in a reusable, flexible and understandable way.

2. Related Work

Regardless of the techniques used in verifying an IC (formal verification and/or functional verification) the need of a standardized way to create a testbench was needed in order to improve code reusability and to ramp up the new engineer that was assigned to the task as fast as possible.

Some of the first methodologies used to solve this issue is VMM (Verification Methodology Manual) proposed by Synopsis in 2003. VMM moved to an object-oriented programming way of writing the verification environment. Its advantages are that

[1] Master degree student at Electronics and Computers department, specialization Electronic Systems and Embedded Communications of *Transilvania* University of Braşov.
[2] Electronics and Computers Dept., *Transilvania* University of Braşov.

encapsulation, inheritance and polymorphism concepts were available to writing reusable components [1], [5].

OVM (Open Verification Methodology) was an effort made by Mentor and Cadence to make their methodologies open source. This methodology was born by merging each side concepts, they being AVM (Advanced Verification Methodology) by Mentor and eRM (e Reuse Methodology) by Cadence [1].

UVM is the methodology release after Synopsis joined the "Mentor-Cadence alliance", the first draft being published in 2011 by Acellera. UVM is the methodology which is popular today and offers a lot of flexibility offering the capability of mixing languages such as System Verilog, e, System C. This offers the possibility to exploit each language strengths, all in the same testbench [6].

3. Specman and UVM

Specman is an Electronic Design Automation (EDA) tool, provided by Cadence, that offers the capability of developing advanced verification environments. The language used to write the testbench is called "e Hardware Verification Language" and is an AOP (Aspect Oriented Programming) language [4]. AOP allows e users to add fields, constraints, events and methods to structs and units; to override existing implementations of methods or to append/prepend them; to override events; to override and extend coverage; and to extend enumerated types.

Specman with UVM is a powerful combination when it comes to writing reusable components. The basic building blocks of e-UVM based environments are eUVC (Universal Verification Component) which are delivered/developed as packages. A package is a well-defined structure of directories in which the UVC is placed. The package contains not just source code but also documentation and examples.

A directory structure proposed by Cadence for an e based UVC can be seen in Figure 1.

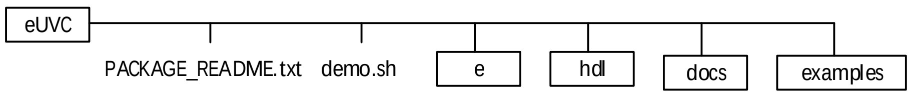

Fig. 1. *Proposed UVM directory structure* [3]

The PACKAGE_README.txt file is mandatory. In this file, important headers must be placed such as Title, Name and Author of the UVC. All headers must begin with an "*". Other headers can be added to give the reader a better view of the UVC such as: Version, Description, Release Notes, To demo etc.

The demo.sh represents the script used to run a simulation with the UVC standalone. Details at how the script is used should be found the PACKAGE_README.txt file.

The rectangles represent other folders which can store e source code, RTL (Register Transfer Level) source code, documents related to the UVC design and examples on how to connect the UVC.

Regardless of the language used in writing the testbench, the structure of an UVC is the same. An UVC is composed of:

- signal map - in this file the ports which connect the RTL with the verification environment are defined;

• configuration - configurations made to the UVC are made such as: active/passive UVC, flavours etc.;

• BFM (Bus Functional Model) - is the block responsible of knowing the way of communicating with the RTL. If the UVC is configured as active, it receives an item from the sequence driver and injects it into the RTL. For passive configuration, this block is disabled;

• Sequence Driver - is a block that act as a mediator between the sequence and the BFM. The generated items are passed from the sequencer to the sequencer driver and the sequence driver acts upon them one by one, usually passing them further to the BFM;

• Sequence - is a structure that represents a stream of items signifying a high-level scenario of stimuli. The items are generated one after another, according to specific rules;

• Item - a structure that represents ma main input to the DUT (packet, transaction, instruction etc.);

• Monitor - block responsible of capturing the transaction made between the DUT and the UVC. It is a standalone block and regardless of the type of UVC kind (active or passive), it will permanently monitor the bus;

• Checkers/Coverage - blocks responsible of checking and collecting coverage. Usually those blocks are an extension of the monitor block;

• TLM (transaction level modelling) ports - are ports defined to pass information to other UVC. An example is passing a collected item to the Scoreboard or to the Predictor.

All of the above-mentioned blocks are instantiated and connected in the top-level design of the UVC. An example of such a structure can be seen in Figure 2.

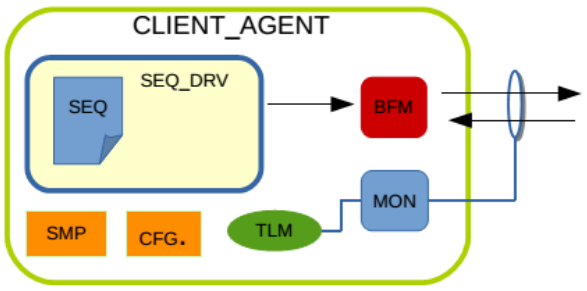

Fig. 2. *Block diagram of an agent*

Figure 3 represents an UVC for a client block that makes a request (arrow from left to right) as long as the backpressure signal is not activated (arrow from right to left).

Multiple UVC may have to be included into an environment to verify a certain DUT. In order to have a centralized control of all the sequences in the verification environment a virtual sequence is used.

4. DUT Description and Proposed UVM Verification Environment

The DUT for which an UVM verification environment was created is called Reception Desk, further noted in this paper as RD. It is a networking device that transfers service requests between a number of 12 Clients units and a Service Resources unit. The RD selects between the input requests using a configurable arbitration mechanism. The MGM (Management) is responsible for selecting the arbitration mode and the SR (Service

Resources) for additional request checking. The RD reports data traffic information to a Management unit and data traffic statistics to a STAT (Statistics) unit.

Features:

- 12 Clients which can request 2 kinds of services through 3 lanes;
- Possibility to send only one request at a time to SR;
- Capacity to queue 4 requests per lane;
- Internal configurable register to select the arbitration mechanism;
- External statistics are updated after each event on RD-SR interface. The RD has a special output signal to synchronize STAT block when reading statistics ports;
- Each error is reported to MGM block.

The proposed UVM based verification environment for the Reception Desk is presented in Figure 3.

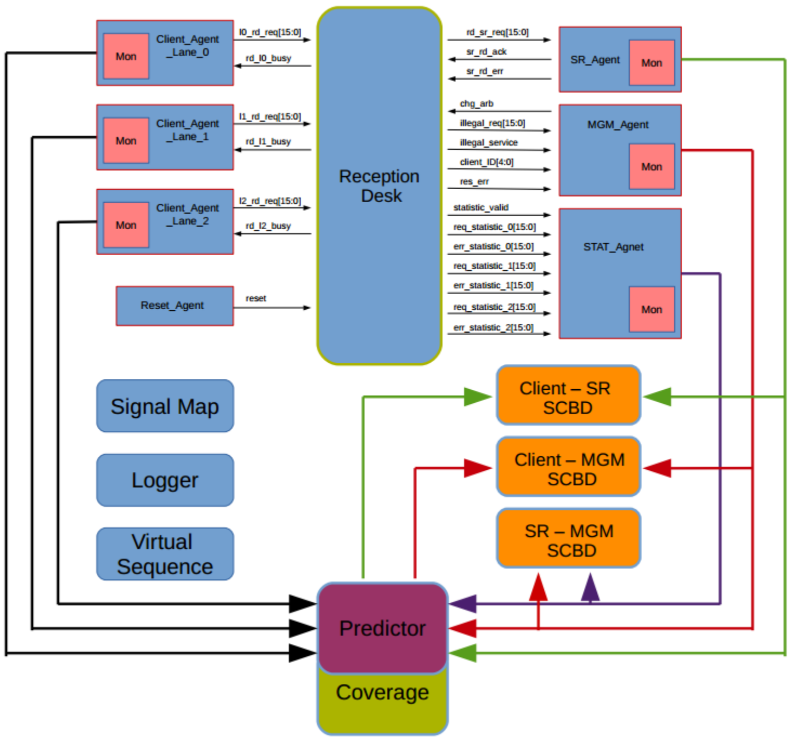

Fig. 3. *Proposed verification environment for the reception desk*

The agents can be configured to be active or passive; meaning that they have the sequence driver and the BFM defined, if they are active, and only the monitor defined if they are passive. Another configuration that can be made to them is their kind, that being master or slave. A master agent is the one that initiates the transaction on the interface. The slave agent waits for a trigger/request to be made on the interface and depending on the interface protocol it may have to replay to it or not.

The verification environment is composed of five kinds of agents as follows:

1. Reset Agent - its main purpose is to generate the reset to the DUT and set the environment test phase to hard reset. The agent used is configured to be an active-master agent.

2. Client Agent - is an active-master agent, which makes a request per clock cycle. Here the reusability of the agent can be seen because for each lane an instance of the agent is made.

3. SR Agent - is an active-slave agent, which analyses the request made by the reception desk and responds randomly with an acknowledge and error (the response kind and time is configurable within the sequence).

4. MGM agent is split in two. The first agent is used to change the arbitration mode of RD by generating a pulse. This agent is configured as active-master. The second agent is a passive agent which only samples the signals that the RD sends to the Management Unit.

5. STAT Agent - is a passive agent which only samples the information that RD sends to the statistics unit.

The predictor block models the Reception Desk behaviour. It receives data through TLM ports from the Client Agents and computes the expected results such as status of the FIFO and winner lane of the arbiter. Its output is send to the scoreboard for comparing it with the DUT's output.

There are three scoreboards used in the design. As it can be seen in Figure 3 each agent from the right has its output connected to an input of the scoreboard, through TLM ports. The scoreboards make a one on one match, and if a mismatch is detected an error message is printed with detailed information about the scoreboard and the item that failed matching.

The logger is also UVM specific and its job is to provide an easy but powerful mechanism to display information during the simulation. The logger has 5 verbosity levels: NONE (messages that can't be disabled), LOW, MEDIUM, HIGH and FULL (all the details).

The virtual sequence, as stated before, provides an easy way to control the agents' sequences. The tests are defined by extending this sequence and placing in its body() method the flow of the test. UVM provides a mechanism to make the test more understandable by using the test-phases [3]. The predefined test-phases available in e are: env_setup, hard_reset, reset, init_dut, init_link, main_test, finish_test, post_test. In the created verification environment only the hard_reset, main_test and finish_test phases were used, the other ones being excluded due to the fact that the DUT doesn't require any special configuration. In the hard_reset phase the reset_agent brings the DUT to a known state. After the reset is deactivated the next test phase will be main_test. Apart from the reset agent, all other agents will perform transactions with the DUT. After all the statements in the body() method of the main_test phase finish, the finish_test phase begins. The role of this phase is to smoothly end the test and by this the client agents stop performing transactions, letting the DUT free to finish computing rest of its internal stored packets. After a certain time after entering this phase the scoreboards are checked to see if there is any residual data left in them. This is a very important stage of the test because if the scoreboards are not empty at this point either the verification environment needs adjustments or the DUT has a bug because skips packets.

5. Conclusions

Using the Universal Verification methodology alongside with "e" in creating a verification environment proved to be efficient due to its unique way to "plug and play" agents in the testbench. The UVCs can be stored in a repository and reused each time the design has a component for which a VIP (Verification Intellectual Property) was already written. Even if the UVC does not cover fully the entire DUT features the UVC can be extended and events/methods can be added or redefined.

Alongside with those powerful provided concepts UVM also helps reducing the time needed to ramp up a fresh engineer entering a project due to its standard way of defining the verification environment. The methodology is also language independent and by so, it lets the code writer the freedom to exploit each language's strength.

References

1. https://www.aldec.com/en/solutions/functional_verification/uvm_ovm_vmm. Accessed: 18.05.2017.
2. *** Cadence *Specman e Language Reference*. March 2011.
3. *** Cadence *Universal Verification Methodology* (*UVM*) *e User Guide*. March 2011.
4. *** *e/eRM to SystemVerilog/UVM*. http://www.specman-verification.com/user_files/DVCon_2012/e_eRM_to_SV_UVM_Mind_the_Gap_But_Dont_Miss_the_Train.pdf. Accessed: 19.05.2017.
5. *** *Introduction to Design Verification with VMM - a Quickstart Guide*. https://www.vmmcentral.org/pdfs/intro_to_dv_with_vmm.pdf. Accessed: 18.05.2017.
6. *** *Universal Verification Methodology* (*UVM*) 1.2. User's Guide. http://www.accellera.org/images//downloads/standards/uvm/uvm_users_guide_1.2.pdf. Accessed: 21.05.2017.