International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 05 Issue: 12 | Dec - 2021 ISSN: 2582-3930

Design Verification of Configurable SPI

[1]Aayushi Dey, [2]Bhavini Kumawat
[1]Research Scholar, Department of Electrical and Electronics Engineering, Oriental University, Indore, MP, India
[2]Assistant Professor, Department of Electrical and Electronics Engineering, Oriental University, Indore, MP, India

---

Abstract— Synchronous serial interfaces are frequently utilized to provide cost-effective board-level interfaces between various devices such microcontrollers, DACs, and ADCs. Components compatible with SPI and Microwire/plus are available from a variety of IC vendors. With some added functionality, the SPI Master core is compatible with both protocols as master. The core works as an APB compliant slave device on the hosts' side. Serial interface, clock generator, and APB interface are the three sections of the SPI master core. Using the APB compatible interface, the SPI core includes five 32-bit registers. Slave select lines, serial clock lines, and input and output data lines are all part of the serial interface. All transfers are full duplex with a customizable bit rate (64 bits). There are eight slave select lines, but only one is active at any given moment. We use System Verilog to create the SPI, and we use QuestaSim to verify our design.

Keywords—*UVM (Universal Verification Methodology); SPI (Serial Peripheral Interface); DUT (Design Under Test); APB; Coverage.*

1. INTRODUCTION

Almost every system nowadays incorporates some form of intelligent control, most commonly a Microcontroller Core. LCD drivers, remote I/0 ports, RAM, EEPROM, and data converters are examples of general-purpose circuits. Communication interfaces and/or computation-intensive tasks require application-oriented circuits. As a result, communication between these components is critical. In this regard, reuse of intellectual property (IP) macro-cells is becoming the center of gravity for design productivity and the key to producing functional chips. All integrated components must be connected to one another, and each SoC must be linked in a fashion that allows for quick and error-free communication. Communication between SoCs is critical for achieving high performance; the most common approach for interconnecting SoCs is a serial bus, which offers significant cost savings. One of the advantages of SystemVerilog is the functional coverage model [1]. System Verilog allows doing functional coverage-driven verification. To do that we need to code the technical documentation into a set of cover points and creates the coverage model. The coverage model will determine which features have been tested by the environment and which have not. These help in understanding the verification holes and, cover them to assure in a completely verified product. Other than the Coverage model SystemVerilog has the object-oriented programming (OOP) concepts integrated inside. That allows separating the verification environment into smaller parts. Those smaller parts are the base of the verification methodology. In general, the components are generators, for generating certain packets, drivers to send those packets, receivers to receive, monitors to monitor the interfaces, and a scoreboard to check the correctness of the operation. There are other components, but these are the widely used ones [2]. This paper concentrates on developing a verification environment for configurable SPI interfaces. In upcoming chapters, the SPI interface overview is given to have a general idea of the interface. Also, the test environment architectures are described, and everything is concluded with the functional coverage results.

APB bus protocol, I2C bus protocol, ARM bus protocol, and others are currently extensively used protocols that allow hardware devices to connect by assigning rules and matching time for the purpose of transmitting data. SPI is a serial interface protocol that, when compared to [3] other protocols, offers the advantages of high transmission speed, ease of use, and few pins. At the very least, the four interfaces are required by the standard SPI protocol. For data transmission, devices that use the SPI protocol are usually separated into master and slave devices. When the data exchange is completed, the master-device generates the chip choose signal and the clock signal. As a result, the master device must be equipped with when controlling numerous slave devices, the master device must have multiple chips select interfaces for slave devices. This will no longer comply with the SPI protocol. The standard SPI

communication is a single-master communication, which means that there is only one master device for all communications [4]. As a result, both factors limit devices that use the SPI standard. This design (adopts the parameterization approach, automatically recognizes the master/slave devices, and uses TSM (Time Sharing Multiplex) technology to control the same slave device at the same time) is used to target the faults. The design fully complies with the four standard SPI protocol interfaces.

2. Overview of SPI

Motorola developed the Serial Peripheral Interface (SPI) module in the mid-1980s, which permits synchronous, serial, and full duplex communication between a microcontroller and peripheral devices. The structural link between the master and slave cores is depicted in Figure 1. The SPI bus is typically used to send and receive data between microcontrollers and other tiny peripherals such as shift registers, sensors, SD cards, and other similar devices [5]. When compared to other protocols, the SPI protocol has the advantages of a relatively fast transmission speed, ease of usage, and a limited number of signal pins. For transmitting and receiving data, the protocol typically separates devices into master and slave. A master device generates distinct clock and data signal lines, as well as a chip-select line that selects the slave device for which communication is required. If there are several slave devices, the master device will need multiple chips select interfaces to control them.

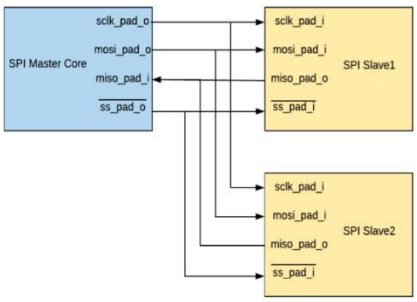

Figure 1 Master and Slave Connection

- Data Transmission - The SPI bus interface consists of four logic signals lines namely MOSI, MISO, Serial Clock (SCLK) and Slave Select (SS).

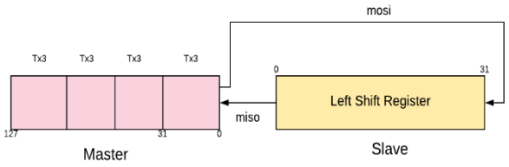

Figure 2 SPI Shift Register

- MOSI - The MOSI is a transversal signal line that can be used as both an output and an input signal line in a master and slave device. It oversees data transfer from master to slave in one direction.
- MISO – The MISO is a unidirectional signal line that can be used as an input in a master device and as an output in a slave device. It oversees data transmission from slave to master in one direction. The MISO line will be in a high impedance state if a specific slave is not selected.
- Slave Select (SS) - The slave select signal is used as a chip-select line to select the slave device. It is an active low signal and must stay low for the duration of the transaction.
- Serial Clock (SCLK) - The serial clock line is used to synchronize data transfer between both output MOSI and input MISO signal lines. Based on the number of bytes of transactions between the Master and Slave devices, required number of bit clock cycles are generated by the master device and received as input on a slave device [6].

2.1 Hardware Architecture

The designed SPI is compatible with the SPI protocol and bus principle. At the host side, the design is equivalent to the slave devices of AMBA bus specification. The overall structure of the AMBA complaint SPI Master core device can be divided into three functional units: Clock generator, Serial Interface and AMBA Interface [7].

2.2 Design of Clock Generation module (spi_clk_gen)

The clk_gen oversees generating the clock signal from the external system clock wb_clk_i, according to the clock register's varied frequency factors, and producing the output signal s_clk_o. The clk_gen module can produce reliable serial clock transmission with odd or even frequency division in the register to assure timing reliability [8]. By dividing the wb_clk_i, the core creates the s_clk_o; altering the value of the divider allows for arbitrary clock output frequency. fsclk = fwbclk/(DIVIDER + 1) x 2 is the expression for s_clk_o and wb_clk_i.

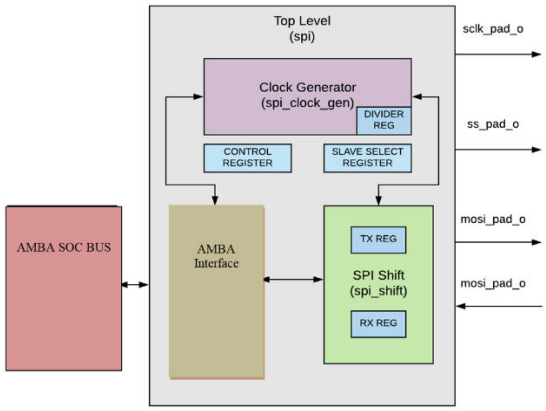

Figure 3 Clock Generation Module

2.3 Serial data transfer module design (spi_shift)

The data transfer core module is made up of serial data transfer modules. It's in charge of converting parallel input data into serial output data for MOSI transmission and MISO serial data into parallel out. The flip flops in the Receive and Transmit registers are the same [9]. If no write access to the transmit register was done between the transfers, the data received from the input data line in one data transfer will be communicated on the output line in the next data transfer. The benefit is that it consumes less electricity because it uses less hardware resources. SPI on the host side receives input data and broadcasts output data in real time as the master device.

2.4 Top-level module (spi)

The top-level module's job is to ensure that the basic framework of high-speed reusable SPI bus sub-components functions properly. As a result, the SPI module's top-level controls the clock generator and serial data transmission modules' operational phase.

3. Verification Architecture

The verification process is like the design creation process. A designer examines a block's hardware specification, understands the human language description, and writes the appropriate logic in a machine-readable format, commonly RTL code written in Verilog or VHDL. To do so, the user must first grasp the input format, transformation function, and output format. This interpretation is always ambiguous, either due to ambiguities in the original material, missing details, or contradictory descriptions. In comparison to Verilog, the System Verilog language offers three significant advantages. Design verification is the most significant part of the product development process, accounting for up to 80% of the overall time spent on the project. The goal is to make that the design meets all the system's needs and specifications [10].

Logic simulation/emulation and circuit simulation are approaches to design

- Verification in which detailed functionality and timing of the design are checked using simulation or emulation
- Functional verification, in which functional models describing the functionality of the design are developed to check against the behavioral specification of the design without detailed timing simulation, and
- Formal verification, in which the functionality is checked using formal methods. Property checking (or model checking), in which the design's properties are checked against some assumed "properties" specified in the functional or behavioral model (e.g., a finite-state machine should not enter a certain state), and equivalence checking, in which the functionality is checked against a "golden" model, are also part of formal verification. Although equivalence checking can be used to evaluate synthesis outputs at lower levels of the EDA cycle, property checking is required for the initial design capture.

4. UVM Introduction

As digital systems grow in complexity, verification methodologies get progressively more essential. While in the early beginnings, digital designs were verified by looking at waveforms and performing manual checks, the complexity we have today don't allow for that kind of verification anymore and, as a result, designers have been trying to find the best way to automate this process. The System Verilog language came to aid many verification engineers. The language featured some mechanisms, like classes, cover groups and constraints, that eased some aspects of verifying a digital design and then, verification methodologies started to appear [11]. UVM is one of the methodologies that were created from the need to automate verification. The Universal Verification Methodology is a collection of API and proven verification guidelines written for System Verilog that help an engineer to create an efficient verification environment. It's an open-source standard maintained by Accellera and can be freely acquired in their website. By mandating a universal convention in verification techniques, engineers started to develop generic verification components that were portable from one project to another, this promoted the cooperation and the sharing of techniques among the user base. It also encouraged the development of verification components generic enough to be easily

extended and improved without modifying the original code. All these aspects contributed for a reduced effort in developing new verification environments, as designers can just reuse test benches from previous projects and easily modify the components to their needs [12].

We create a road map for achieving the goal in the test plan, which is a live document. Introduction, assumptions, a list of test cases, a list of features to be tested, approach, deliverables, resources, risks and timing, and entrance and exit criteria are all included in the test plan. The test strategy aids the verification engineer in comprehending how the verification should be carried out. A test plan can be in the form of a spreadsheet, a paper, or even a plain text file. Sometimes, the test plan is just in the engineer's head, which is risky because the process cannot be accurately measured and controlled. The test plan also includes descriptions of the Test Bench architecture as well as each component's functionality.

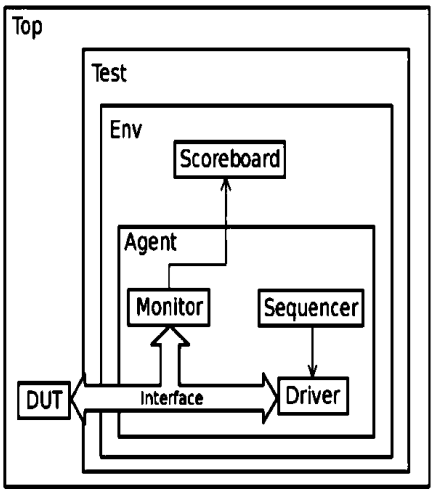

Figure 4 Representation of UVM Environment

4.1 Defining the Verification Environment

Before understanding UVM, we need to understand verification. Right now, we have a DUT, and we will have to interact with it to test its functionality, so we need to stimulate it. To achieve this, we will need a block that generates sequences of bits to be transmitted to the DUT, this block is going to be named sequencer. Usually, sequencers are unaware of the communication bus, they are responsible for generating generic sequences of data and they pass that data to another block that takes care of the communication with the DUT. This block will be the driver. While the driver maintains activity with the DUT by feeding it data generated from the sequencers, it doesn't do any validation of the responses to the stimuli. We need another block that listens to the communication between the driver and the DUT and evaluates the responses from the DUT. This block is the monitor. Monitors sample the inputs and the outputs of the DUT, they try to make a prediction of the expected result and send the prediction and result of the DUT to another block, the scoreboard, to be compared and evaluated. All these blocks constitute a typical system used for verification and it's the same structure used for UVM test benches. Usually, sequencers, drivers and monitors compose an agent. An agent and a scoreboard Compose an environment. All these blocks are controlled by a greater block denominated of test. The test block controls all the blocks and sub blocks of the test bench. This means that just by changing a few lines of code, we could add, remove, and override blocks in our test bench and build different environments without rewriting the whole test. To illustrate the advantage of this feature, let's imagine a situation where we are testing a another DUT that uses SPI for communication. If, by any chance, we want to test a similar DUT but with I2C instead, we would just need to add a monitor and a driver for I2C and override the existing SPI blocks, the sequencer and the scoreboard could reuse just fine [13].

4.2 Components of UVM Test Bench

A UVM Test bench is composed of reusable universal verification components (UVCs). A UVM-UVC is and encapsulated, ready to use and configurable verification environment intended for an interface protocol, a design sub-module or even for software verification. Each UVC follows a consistent architecture and contains a complete set of elements for sending stimulus, as well as checking and collecting coverage information for a specific protocol or design. The interface UVC is applied to the Device under test (DUT) to verify implementation of the design protocol logic or as a means of program the DUT. Module UVCs contain internal verification logic for a subsystem or a module and enable the subsystem verification in a larger system. UVM-UVCs speedup the process of creating efficient testbench for the DUT, and are structured to work with any hardware description language (HDL) and a high-level verification language (HVL) including Verilog, VHDL, e, System Verilog, and System C. The UVCs can be reused for multiple verification environments. The verification environment also contains a multi-channel sequence mechanism i.e., a virtual sequencer that synchronizes the timing and the data between the different interfaces and allows fine control of the test environment for a particular test. The main components and detailed explanation about each universal verification component is as follows:

4.3 Data Items

Data items represent stimulus transactions that are input to the DUT. Examples of data items are networking packets, bus transactions and instructions. The fields and attributes of

a data item are derived from data item's specification. In a typical test, many data items are generated and sent to the DUT. By randomizing data items using System Verilog constraints, it helps to create many meaningful tests and maximum coverage. As Driver deals with signal activities at bit level, it does not make sense to keep this level of abstraction in DUT. So, concept of transaction was created. A transaction is a class object usually extended from uvm_transaction or uvm_sequence_item classes, which includes information needed to model the communication between two or more components. Transactions are the smallest data transfers that can be executed in a verification model. They can include variables, constraints and even methods for operating themselves. Due to their higher abstraction level, they are not aware of communication protocol between components so they can be reused and extended for different kind of tests if correctly programmed [14]. The transaction could include two variables; the address of the device and the data to be transmitted to that device. The transaction would randomize these two variables and verification environment would make sure that the variables would assume all possible and valid values to cover all combinations. To drive stimulus to DUT, a driver component converts transactions into pin wiggles. Sequences are ordered collection of transactions; they shape transactions to our needs and generates as many as need. Sequence is extended from uvm_sequence and their main job is generating multiple transactions. After generating transactions, sequencer takes them to driver.

4.4 Top Block

There are mainly two components that connects DUT with Test bench in UVM.

- Top block of Test bench,
- A Virtual Interface

The top block will create instances of the DUT and of the Test bench and the virtual interface will act as a bridge between them. Interface is a module that holds all signals of DUT. The monitor, the driver and the DUT are all going to be connected to this module. The top block is responsible for:

- Connecting DUT to test class using interface,
- Generating clock for DUT,
- Registering the interface in the UVM factory. This is necessary to pass this interface to all other classes that will be instantiated in the test bench,
- Running Test

4.5 Sequencer

A sequencer is an advanced stimulus generator that controls the items provided to the driver for execution. By default, a sequencer behaves similarly to a simple stimulus generator and returns a random data item upon request from the driver. This default behavior allows you to add constraints to the data item class to control distribution of randomized values.

4.6 Driver

A driver is an active entity which emulates logic that drives the DUT.A typical driver repeatedly pulls data items generated by a sequencer and drives it to the DUT by sampling and driving the DUT signals.

4.7 Monitor

The monitor is a self-contained passive entity that observes the communication of the DUT with the testbench by converting pin wiggles into transactions. Monitor observes the outputs of the design and incase of not respecting protocol rules, the monitor must return an error. The monitor is a passive component, it does not drive any signals into the DUT its purpose is to extract signal information and translate it into meaningful information to be evaluated by other components. A verification environment is not limited to just one monitor, it can have multiple of them. Monitors collects transactions from virtual interface and use the analysis ports to send those transactions to the score board.

4.8 Agent

Sequencers, drivers and monitors can be reused independently, but this requires the environment integrator to learn the names, roles, configuration, and hookup of each of these entities. To reduce the amount of work and knowledge required by test writer, Agent is used. Agent is basically a container. Some agents are proactive and initiate transactions to the DUT, while other agents react to transaction requests. Agents should be configurable so that they can be either active or passive. In active mode it drives the signal to the DUT. So, driver and sequencer are instantiated in active mode. In passive mode it just samples the DUT signals does not drive them. So only monitor is instantiated in passive mode.

4.9 Scoreboard

Scoreboard is a crucial element in a self-checking environment, it verifies the proper operation of a design at functional level. This component is the most difficult one to write, it varies from design to design and from designer to designer.

4.10 Environment

Environment is at the top of the test bench architecture; it will contain one or more agents depend on design. The environment contains configuration properties that enable you to customize the topology and behavior to make it reusable. For example, active agents can be changed into

passive agents when the verification environments are reused for system verification. The environment class (uvm_env) is designed to provide a flexible, reusable, and extendable verification component. The main function of the environment class is to model behavior by generating constrained-random traffic, monitoring DUT responses, checking the validity of the protocol activity and collecting coverage.

4.11 Interface

Interface is the bridge between the design-under-test and the verification environment. The interface encapsulates all the pin-level connections that are made to the DUT. An interface is a bundle of nets or variables.

5. Conclusion and Results

My design is subjected to automated test-case generation and application. The SPI Protocol's functionality was tested. Using UVM, creating Verification IP for any design (DUT) becomes a breeze. The Universal Verification Methodology checks the design in the most efficient way possible. The basic functioning and operation of SPI is described, as well as the descriptions of registers, signals, and pins. It explains how to set up a serial communication environment between the master and the slave device of choice [15]. SPI functional verification describes the verification platform that uses System Verilog to test the design under test (DUT), which is SPI. We designed the verification environment to test the functionality and operation of configurable intellectual property SPI in compliance with the design requirements. The VIP built for the SPI Protocol was reusable, and it could be used to effectively verify designs. It is possible to get 100% functional and assertion coverage with this verification environment.

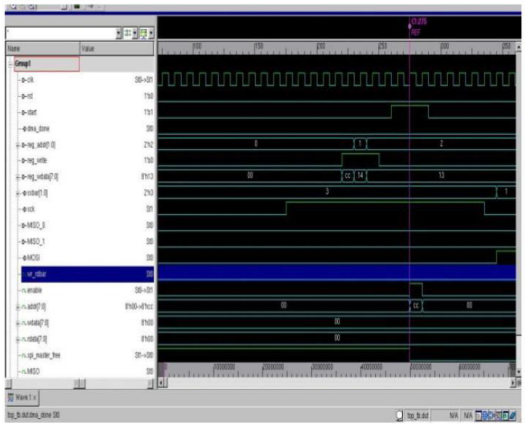

Figure 5 DMA issues read operation to SPI Master

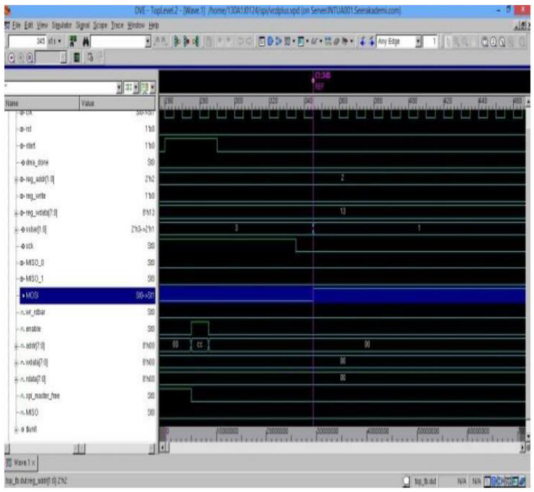

Figure 6 SPI Master issues read transactions to slave-1

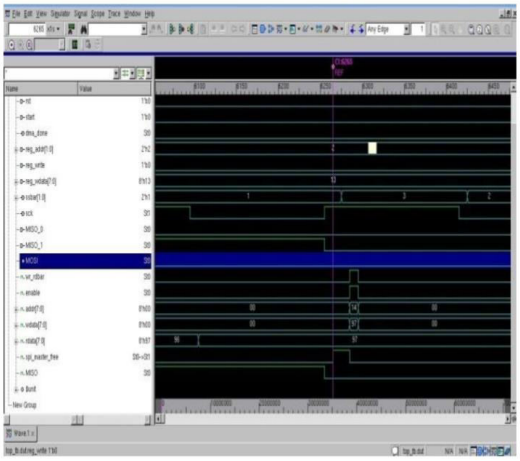

Figure 7 Slave-1 gives read data to Master

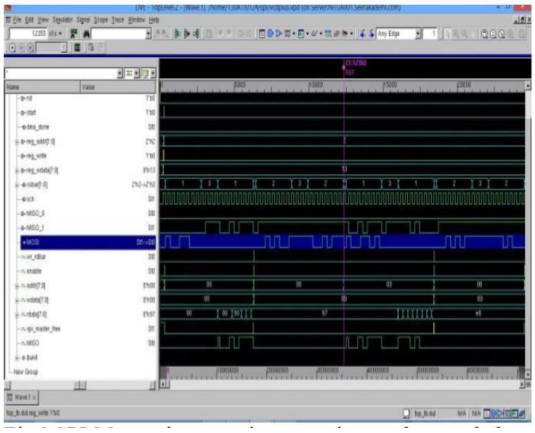

Figure 8 Master issues write operation

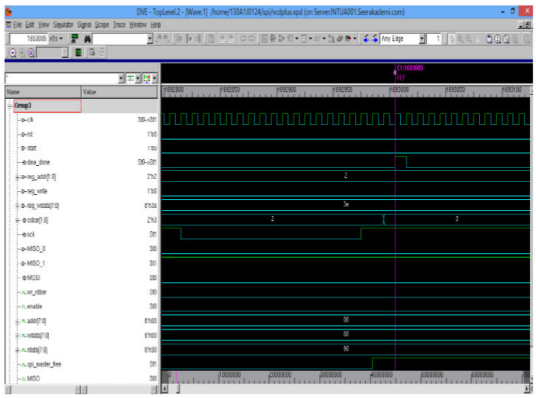

Figure 9 Operation Completed

REFERENCES

[1] Tianxiang Liu ”IP Design of Universal Multiple Devices SPI Interface” IEEE.978-1-61284-632-3, 2011.

[2] Freescale Semiconductor, Freescale SPI Block Guide V04.01, October 14).2008.

[3]AccelleraOrganization,“UniversalVerificationMethodology (UVM) 1.1 Class Reference”, June 2011.

[4] IEEE Computer Society “IEEE Standard Verilog® Hardware Description Language”, IEEE Std 1364™- 2005, 07 April 2006.

[5] http://www.testbench.in.

[6] IP Design of Universal Multiple Devices SPI Interface Tianxiang Liu1, Yunfeng Wang1 * Department of Electronic Engineering, Xiamen University,2011 IEEE.

[7] www.Colorlesscube.com.

[8] http://www.systemverilog.in/systemverilog.php

[9] M.Sandya1, K.Rajasekhar, “Design and Verification of Serial Peripheral Interface”, International Journal of Engineering Trends and Technology- Volume3Issue4- 2012

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

[11] Alexander W. Rath, Volkan Esen and Wolfgang Ecker, 2014.ATransaction-OrientedUVM-BasedLibraryfor Verification of Analog Behavior, IEEE- 978-1-4799- 2816-3, pp 806-811.

[12]http://www.low-powerdesign.com/article_CadenceUVM.

[13] An Accellera Organization, June 2011. Universal Verification Methodology (UVM) 1.1 Class Reference.

[14] http://www.totalphase.com/support/articles.

[15] Clifford E. Cummings, Tom Fitzpatrick, “OVM & UVM Techniques for Terminating Tests” DVCon 2011.