UVM Based Testbench Architecture for Coverage Driven Functional Verification of SPI Protocol

Vineeth B[1], Dr. B. Bala Tripura Sundari[2]
Department of Electronics and Communication Engineering
Amrita School of Engineering, Coimbatore
Amrita Vishwa Vidyapeetham, India
vinetb93@gmail.com[1], b_bala@cb.amrita.edu[2]

***Abstract*—The scale and complexity of integrated circuit designs are ever expanding which makes the verification process increasingly difficult and progressively time consuming. This dictates the need for testbench architectures with generic verification components that are reusable and easily extendable across designs. The UVM can realize such testbench architectures with coverage driven verification environments useful for constrained random testing. This paper focus on the UVM based verification of SPI protocol according to the verification plan prepared after an exhaustive analysis of SPI functional specifications. The UVM testbench generates random vectors, drives the SPI module and compares the captured response with expected response using scoreboard mechanism to verify the SPI functionality. The testbench also validates the characteristic features of SPI by running appropriate test cases and reports coverage cumulatively at the end of each test. The simulation results are analyzed to evaluate the effectiveness of the proposed testbench.**

***Index Terms*—Universal Verification Methodology, Functional Verification, Functional Coverage, Serial Peripheral Interface**

I. Introduction And Background

Today's integrated circuit designs based on reusable IP cores allows for the integration of increasing number of components per chip while augmenting its complexity. The verification of such designs consumes about 70-80% of the product development cycle, requires a lot of effort and cannot be accomplished by the conventional directed testing methodology [1]. With the incorporation of OOPs concepts into Verilog and the introduction of System Verilog, coverage driven constrained random testing emerged as the new choice of verification methodology adopted for simulation based functional verification. However, the development of constrained random testbench for large designs still proved difficult and had limited flexibility and reusability [2]. The verification procedure still lacked a strict set of rules and guidelines to standardize the process and make it more efficient and robust.

The first verification methodology to ever come out which emphasized on component reuse was the e Reuse Methodology (eRM) developed by Verisity Design. eRM introduced standards that facilitated re-usability, interoperability and consistency of the verification components used for the development of a structured testbench [3]. It laid the foundation for future methodologies developed by major EDA vendors such as Universal Reuse Methodology (URM) by Cadence Design Systems, Advanced Verification Methodology (AVM) by Mentor Graphics and Reference Verification Methodology (RVM) by Synopsys. The joined effort from Cadence Design Systems and Mentor Graphics to merge the concepts of URM and AVM resulted in the Open Verification Methodology (OVM) which brought out the best of both worlds. Later on, Synopsys also joined the venture and combined the RVM and OVM methodologies to develop Universal Verification Methodology (UVM) which the Accellera committee approved to establish as the new standard to be adopted for the functional verification of integrated circuit designs.

The UVM provides rich base class libraries which facilitate the development of robust and reusable layered testbench architectures [4] by defining a standard set of verification and analysis components using open source codes. The UVM factory mechanism further simplifies the process by allowing the substitution of existing components with derived components of the same type without altering the testbench structure. This enables the overriding of components in different environments or tests without modifying the original code [5]. The UVM supports Transaction Level Model (TLM) based communication interfaces for the interconnection of different verification components at the transaction level. The UVM automates verification process to some extent through the introduction of new constructs such as sequences and by adding various data automation features such as copy, compare etc. The UVM based verification approach enables the independent development of the test cases separate from the actual testbench hierarchy which makes the stimulus reusable across different projects.

The paper has been organized into different sections. The section II gives a brief introduction to the UVM base class libraries, describes the functionality of different verification components used in the proposed testbench architecture and summarizes how the testbench is executed in the UVM phases. The section III elaborates the verification plan [6] after thoroughly exploring the various functional specifications of SPI protocol [7]. The section also outlines the functional coverage plan and describes the different test cases to be executed to achieve the complete functional verification of SPI. The section IV presents the simulation results and discussion which evaluates the relevance of the proposed work. Finally, the section V concludes the work.

978-1-5386-5314-2/18/$31.00 ©2018 IEEE

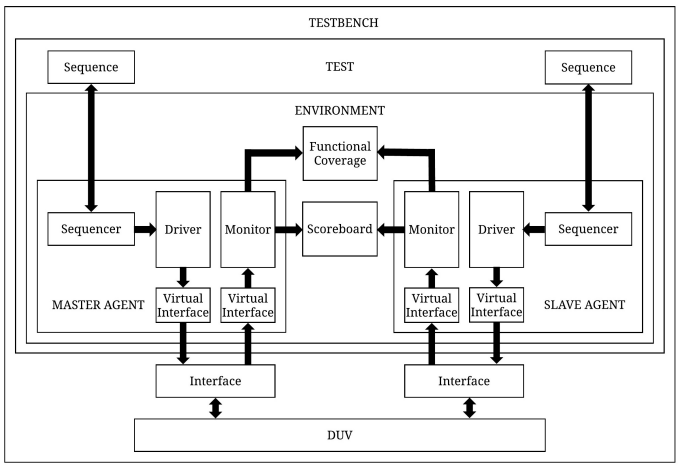

Fig. 1. UVM Testbench Architecture.

II. The UVM Based Testbench Architecture

The UVM testbench is developed using the verification and analysis components derived from the UVM base class libraries [8] and is shown in Fig. 1. The testbench generates random vectors and drives the DUV. The stimulus applied as well as DUV response is monitored for checking purposes to verify the DUV functionality. The testbench also uses the coverage mechanism to report functional coverage as a measure to evaluate the progress of verification process.

A. The UVM Package

The UVM package contains the base class libraries from which various verification components can be derived using which the hierarchical UVM testbench architecture can be developed. The three types of classes are:

1) *uvm_object*: It is the root class for all the data and hierarchical classes of UVM which defines methods for common operations such as create, copy, compare and print for all the objects.
2) *uvm_component*: It is the root class for all the major verification components of UVM. It inherits all the features of uvm_object and provides interface to hierarchy, phasing, reporting and the UVM factory.
3) *uvm_transaction*: It is the root class for all the transactions used for stimulus generation and analysis in UVM. It inherits all the features of uvm_object and provides interface to timing and recording.

B. The UVM Testbench Hierarchy

The UVM testbench hierarchy is made up of a number of verification and analysis components [5]. These components along with their functionality are described below:

1) *Sequence Item*: The data properties such as control, payload, configuration and analysis information of the random stimuli to be applied to the DUV are defined by the sequence item. The input data properties are declared as random variables for constrained random testing.
2) *Sequence*: The sequence items are generated and randomized by the sequence before transmitting them to the driver. Sequences can be test specific.
3) *Sequencer*: The sequencer runs the sequence i.e. it channels the sequence items from the sequence where they are generated to the driver.
4) *Driver*: The driver converts the data fields of sequence items into pin level transactions to drive the DUV using a virtual interface.
5) *Monitor*: The monitor tracks the activity at pin level of DUV, converts them into transfer level transactions and send them to the scoreboard and coverage collector using the TLM analysis ports.
6) *Agent*: The sequencer, driver and monitor are encapsulated into a single entity called agent. Agents can either be active or passive depending on whether it instantiates all the components and stimulate the DUV or only the monitor to track the DUV responses respectively.
7) *Scoreboard*: The transactions coming from two agents are compared using scoreboard to ascertain whether the DUV response matches with the expected behaviour. It decides whether a test case has passed or failed.
8) *Coverage*: The coverage monitor keeps track of all hits occurring on the DUV signals that are specified as coverpoints in the coverage model during simulation.
9) *Environment*: All the UVM verification components are grouped together inside a single container called environment. The verification components can be configured in any arbitrary way by defining an environment. A UVM testbench can have multiple such environments.
10) *Test*: The test instantiates the environment and the sequence required for verifying the different characteristic features of DUV. The complete functional verification and appropriate coverage convergence is obtained by running a proper number of test cases.

C. The Standard UVM Phases

The UVM simulation is carried out in an orderly fashion with a distinct set of phases to achieve a consistent testbench execution flow. The UVM phasing mechanism is supported by the set of virtual methods and corresponding callbacks predefined in the uvm_component class [5]. A UVM testbench is executed in the following phases:

1) *Build Phase*: This phase is implemented by the build phase methods which execute in zero time before the simulation starts. During this phase the testbench hierarchy is constructed in a top down manner and TLM connections between testbench components are established in a bottom up manner.
2) *Run-time Phase*: This phase corresponds to the time consuming part of simulation where the stimuli generated by the testbench is applied to the DUV. This is the only phase implemented using tasks and is used by all the analysis components. The phase also provides methods for displaying the testbench topology or configuration information before simulation begins.

VARIABLE	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL
CP_SPIE	2	0	2	100.00	100
CP_SPTIE	2	0	2	100.00	100
CP_CPOL	2	0	2	100.00	100
CP_CPHA	2	0	2	100.00	100
CP_LSBFE	2	0	2	100.00	100
CP_SPPR	8	0	8	100.00	100
CP_SPR	8	0	8	100.00	100
CP_SPIDR	256	0	256	100.00	100
CP_SPIF	2	0	2	100.00	100
CP_SPTEF	2	0	2	100.00	100
CP_MODF	2	0	2	100.00	100

Fig. 2. Functional Coverage Report.

```
UVM_INFO @ 2538138500: uvm_test_top.env.u_scoreboard [run] Master Transmit: 116, Slave Transmit: 10
UVM_INFO @ 2538138500: uvm_test_top.env.u_scoreboard [run] Master Receive: 10, Slave Receive: 116
UVM_INFO @ 2538138500: uvm_test_top.env.u_scoreboard [cmp] Test: PASS!
UVM_INFO @ 2538714500: uvm_test_top.env.u_scoreboard [run] Master Transmit: 207, Slave Transmit: 151
UVM_INFO @ 2538714500: uvm_test_top.env.u_scoreboard [run] Master Receive: 151, Slave Receive: 207
UVM_INFO @ 2538714500: uvm_test_top.env.u_scoreboard [cmp] Test: PASS!
UVM_INFO @ 2540606500: uvm_test_top.env.u_scoreboard [run] Master Transmit: 117, Slave Transmit: 98
UVM_INFO @ 2540606500: uvm_test_top.env.u_scoreboard [run] Master Receive: 98, Slave Receive: 117
UVM_INFO @ 2540606500: uvm_test_top.env.u_scoreboard [cmp] Test: PASS!
```

Fig. 3. Scoreboard Results.

CROSS	EXPECTED	UNCOVERED	COVERED	PERCENT	GOAL
CPOLxCPHA	4	0	4	100.00	100
CPOLxCPHAxLSBFE	8	0	8	100.00	100
SPPRxSPR	64	0	64	100.00	100
CPOLxCPHAxSPIDR	1024	0	1024	100.00	100
CPOLxCPHAxSPPRxSPR	256	0	256	100.00	100
CPOLxCPHAxSPPRxSPRxSPIDR	65536	0	65536	100.00	100

Fig. 4. Cross Coverage Report.

3) *Clean-up Phase*: The scoreboard and coverage results obtained after executing the run phase are collected in the clean-up phase. These results are used by analysis components to verify the behaviour of DUV.

III. The Verification Plan

The verification plan is essential to any verification effort as it describes the functional requirements of the design and identifies the different features to be validated. It also establishes the coverage goals and defines the test cases to be executed to achieve the complete verification of design.

A. The SPI Functional Specifications

The SPI is a source-synchronous serial communication protocol that allows a serial bit-stream to be shifted in and out of a device at a programmable bit-rate. The distinctive features of SPI are master and slave mode of operation, double buffered data register, serial synchronous clock with programmable polarity and phase, programmable bit-rate, mode fault error detection and interrupt capability. These features are realized by the functional registers of SPI. The SPI control register controls the mode of SPI operation. The serial clock polarity and phase is controlled by the CPOL and CPHA bits respectively. The bit to be transferred first, whether LSB or MSB of data, is controlled by the LSBFE bit. The bit-transfer rate of SPI can be controlled by configuring the baud rate register. The baud rate is specified by the baud rate preselection bits (SPPR) and the baud rate selection bits (SPR) as per the equation [7]:

$$BaudRateDivisor = (SPPR + 1)2^{(\mathrm{SPR}+1)} \qquad (1)$$

The SPI status register shows the status of data transfer using interrupts. The SPI interrupt flag (SPIF) indicates whether a data byte is transferred to the SPI data register (SPIDR). A low on SPIF shows that current transfer is not yet completed. The SPI transmit enable flag (SPTEF) indicates whether the SPI data register is empty or not. A high on SPTEF shows that SPI data register is empty and a new transfer can be initiated. These interrupts are enabled by the SPIE and SPTIE bits respectively in the control register. The mode fault flag (MODF) detects the mode fault error where the slave select line of the master device becomes low during transfers.

The SPI protocol specifies four bus signals: slave select (SS), serial clock (SCK), master-out-slave-in (MOSI) and master-in-slave-out (MISO) [7]. The SPI data transfers are initiated by the master device by asserting the SS line low. After a specific delay, the master starts sending the synchronous serial clock at the programmed bit-rate to the slave. The data from the SPIDR are shifted out serially to the MOSI and MISO lines and are sampled synchronous with SCK. The data sampling and shifting occurs at alternate edges of SCK irrespective of the mode of transfer to makes sure that data is stable and consistent at the time of sampling. For example, in case of CPOL=0 and CPHA=0 configuration the data sampling occurs at the positive edge of SCK while the data shifting occurs at the negative edge of SCK. After eight cycles of SCK the data in SPIDR of master device is exchanged with the data in SPIDR of the slave device. After the data transfer is complete the SS line is asserted high. There is a minimum idling time between transfers before the SS line can be asserted low again for another transfer.

B. The Functional Coverage Plan

The coverage plan includes the control signals, baud select signals, interrupts and the data register of SPI as the cover-points in the coverage model. At large, appropriate stimuli is generated by constrained random test cases which covers the different characteristic features of SPI. The CPOL, CPHA and LSBFE bits together contributes to eight possible data transfer modes of SPI. In each of these modes data transfer can take place at any one of the thirty-six possible baud rates dictated by the SPPR and SPR bits as per (1). The cross overlapping of these configuration parameters can result in 288 different data transfer modes for SPI. In order to ascertain that SPI operation in the different configurable modes at any programmable baud rates is consistent and reliable, relevant cross coverage models are provided. Cross coverage keeps track of the hits that occurred simultaneously on two or more cover points and thereby uncovers all the bins corresponding to various modes of SPI data transfer.

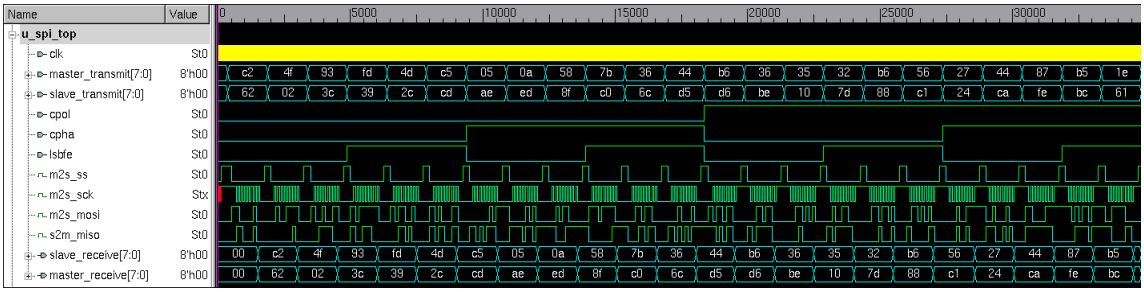

Fig. 5. UVM Testbench Based Simulation of SPI.

TABLE I
CUMULATIVE COVERAGE COLLECTION BY CONSTRAINED RANDOM TEST CASES

SI. NO.	Test Cases	Constraints	Coverage	Sim. Time
1	constrained-random	CPOLL–CPHAL–LSBFEL	65.44%	96.11s
2	constrained-random	CPOLL–CPHAL–LSBFEH	69.12%	94.70s
3	constrained-random	CPOLL–CPHAH–LSBFEL	78.68%	94.90s
4	constrained-random	CPOLL–CPHAH–LSBFEH	79.41%	91.94s
5	constrained-random	CPOLH–CPHAL–LSBFEL	88.97%	94.19s
6	constrained-random	CPOLH–CPHAL–LSBFEH	89.71%	91.09s
7	constrained-random	CPOLH–CPHAH–LSBFEL	96.32%	94.60s
8	constrained-random	CPOLH–CPHAH–LSBFEH	97.02%	91.94s
9	random	NONE	100%	30.24s

C. The Test Structure

The UVM tests are modelled such that it covers all the functional specifications of the SPI. There are eight constrained random test cases and some random stimuli which collectively contributes to the total functional coverage. The constrained random tests target the different modes of data transfer supported by SPI and check whether the response coincides with the expected behaviour. The random stimuli are applied to cover the test scenarios which are not covered during constrained random testing and complete the coverage plan. The constrained random tests characterize the transfer mode in terms of the deterministic main parameters: clock polarity high (CPOLH) or low (CPOLL), clock phase high (CPHAH) or low (CPHAL) and LSB first enable high (LSBFEH) or low (LSBFEL). Each of these modes are tested at all possible baud rates realized by the SPPR and SPR bits.

IV. SIMULATION AND RESULTS

Synopsys VCS tool is used for the UVM testbench based simulation of SPI module and coverage analysis. The testbench applies random stimuli to the SPI top module which instantiates the SPI master and slave modules internally and interconnects them by the SPI bus signals. The on-screen displayed scoreboard results are shown in Fig. 3. The functional coverage and cross coverage of different SPI signals as reported by the VCS is shown in Fig. 2 and Fig. 4 respectively. Synopsys DVE is used for waveform analysis and the resulting SPI simulation waveform is shown in Fig. 5. The Table I specifies the different test cases executed along with the cumulative coverage reported at the end of each test. It also specifies the simulation time elapsed for running each test case.

V. CONCLUSION

In this paper, a UVM based testbench architecture is proposed for the functional verification of SPI protocol. The testbench accomplished complete functional verification of SPI and achieved 100% functional coverage through systematic execution of relevant test cases in less than fifteen minutes. The simulation waveform demonstrates the successful exchange of data between SPI master and SPI slave modules through the SPI bus signals. This fact is backed by the scoreboard results. The functional coverage report ensures that different SPI signals are properly exercised and the cross coverage report proved consistent SPI operation across different transmission formats at all the possible baud rates. Thus, the proposed testbench can be effectively used for the verification of SPI as well as SoCs featuring SPI protocol.

REFERENCES

[1] Anil Deshpande, “Verification of IP-core based SoCs,” 9th International Symposium on Quality Electronic Design, pp. 433–436, 2008.
[2] Chris Spear, SystemVerilog for Verification (2nd Edition): A Guide to Learning the Testbench Language Features, Springer, 2008
[3] Verisity Design Inc., e Reuse Methodology (eRM) Developer Manual Version 4.3.5, pp. 1–5, 2004.
[4] N Dohare, S Agrawal, “APB based AHB interconnect testbench architecture using uvm_config_db,” International Journal of Control Theory and Applications, vol. 9, pp. 4377-4392, 2016.
[5] Accellera, Universal Verification Methodology (UVM) 1.2 User’s Guide, October, 2015.
[6] Zhili Zhou, Zheng Xie, Xinan Wang and Teng Wang, “Development of verification environment for SPI master interface using system verilog,” IEEE 11th International Conference on Signal Processing (ICSP), pp. 2188–2192, October, 2012.
[7] Motorola Inc., SPI Block Guide V03.06, March, 2003.
[8] Pavithran T M, Ramesh Bhakthavatchalu, “UVM based testbench architecture for logic sub-system verification,” IEEE International Conference on Technological Advancements in Power and Energy, in press.