A Unified UVM Methodology For MPSoC Hardware/Software Functional Verification

Sherif Hosny

ST Microelectronics

{*sherif.hosny@st.com*}

***Abstract*—Over the past few years the complexity of Multi-Processor System on Chip (MPSoC) designs increased drastically. This made product verification very challenging and illusive. In order to cope with design complexity, Universal Verification Methodology (UVM) associated with System Verilog Assertions (SVA) are used extensively to build up robust verification environments revealing design issues. This work introduces a new methodology verifying SoC design blocks in two modes: Stubbing mode, where all blocks serving the Design Under Test (DUT) are implemented as UVM active and passive agents; Physical hardware mode, where all blocks are physically running along with the firmware driver. A complete SoC system contains: processor, controller, and encryption engine is studied while implementing the proposed verification approach. Functionality check and coverage collection are performed through UVM scoreboard and subscriber respectively. The proposed approach provides the capability of verifying both hardware and firmware simultaneously in the simulation phase.**

***Keywords*—MPSoC (Multi-Processor System on Chip), UVM (Universal Verification Methodology), DUT (Design Under Test), and Firmware**

I. Introduction

As we live in era of technology, the size of SoC (System on Chip) designs has increased rapidly over the past few years leading to higher design complexity. According to Moore's law, the number of transistors per silicon chip gets doubled every year. However, as Moore's law is getting saturated, new architectures are introduced to build up more sophisticated SoC devices which made the chip functional operation much more complicated [1]. This lead to increase the burden of verification engineer, where simple old approaches became inefficient anymore [2]. The UVM (Universal Verification Methodology) initiated by Accellera [3] is a highly abstracted standardized verification methodology. The framework provides a group of organized SV (System Verilog) classes aiming to enable full control on the DUT (Design Under Test).

Since current conventional SoCs contain blocks communicating with the embedded target processor, it became much more elusive to verify both hardware design and firmware driver on the embedded target as early as possible. This work introduces a unified automated flow where both hardware and firmware are verified together in the UVM testbench. This is achieved by introducing a new type of UVM agents entitled semi-active agents.

This paper is organized as follows: Section II shows a list of related work. Section III gives details about the embedded target flow. Section IV illustrates the proposed verification flow. A case study deploying the proposed approach is shown in Section V. Finally, Section VI shows the paper conclusion and future work.

II. Background and Related Work

The authors in [4] proposed a reusable UVM environment aiming to verify different types of SoC buses. The paper lacks details about the deployed verification methodology in order to include the blocks interacting with the verified buses on the SoC level. Another reusable mechanism is proposed in [5] targeting different types of flash memory controllers. The main target is to find the common features between different controllers to generate common scenarios. Despite comparing between different controllers, the verification approach is performed on the block level not the chip level.

Similarly, Hybrid Memory Cube (HMC) memory controller is verified in [6]. In [7], a proposed block level reusable testbench based on UVM is deployed for the sake of verifying synchronous FIFO. The authors in [8] proposed another block level environment aiming to verify the DVB (Digital Video Broadcast) module. The verification approach in the aforementioned work is based on applying UVM agents without including the firmware driver in their verification flow.

Jiayi et al. proposed a new environment in [9] targeting the verification of RISC-V integrated in full SoC environment. The case study for the deployed verification environment is the DMA, where the authors illustrated the UVM environment built to verify it. In the active state, the sequence is derived from the sequencer, meanwhile in the passive state, the sequence is derived by the RISC-V core. The paper does not provide any details about the checking mechanism in the case of the passive state. The limitations in this approach are twofold: First, the firmware code running on the RISC-V core includes either directed test vector or simple randomized data, so it lacks automation. Second, if passive agents are used, the scoreboard will have no details about the reference data executed by the firmware code on the RISC-V core. If the sampled data from the monitor placed at the RISC-V core is considered as the reference, this data might be erroneous as there are multiple operations performed between the core and the boot memory. The proposed verification flow aims to solve the aforementioned problems in an automated way by introducing a new type of UVM agents entitled semi-active agents.

III. Embedded Target Flow

The conventional SoC system as illustrated in Figure 1 consists of the following building blocks:

- **Processor**: For executing firmware driver code and controlling all other blocks.
- **Boot ROM (Read Only Memory)**: Non volatile on-chip memory used to store bootable image.
- **RAMs (Random Access Memories)**: Two volatile on-chip RAMs are used to serve the processor; the I-RAM (Instruction-RAM) and D-RAM (Data-RAM).

The processor interacts with all memories using high speed interfaces such as: AMBA-AXI (ARM Advanced Microcontroller Bus Architecture Advanced eXtensible Interface) [10], AMBA-AHB (AMBA Advanced High-performance Bus) [11], and STBus [12]. If the SoC system contains multiple masters communicating with multiple slaves, a bus matrix shall be used for the arbitration mechanism.

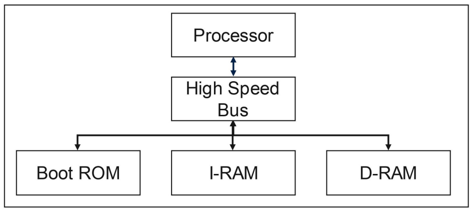


Fig. 1: SoC level system

On power up, the processor fetches the configurable startup address or the static address mapped to the boot ROM. The loaded boot image guides the processor to the next step to either fetch an external flash memory to extract the firmware driver image and pass it the I-RAM to be executed, or fetch the previously loaded code on I-RAM directly. The processor then starts to execute the I-RAM code while using the D-RAM, then interacts with other slaves in the SoC.

IV. Proposed Verification Approach

Verifying a specific block in the SoC system mentioned earlier is performed on two phases. The RTL verification phase, which involves verifying the RTL design without the deployment of firmware driver. The flow involves two levels of verification:

- **Block Level Verification**: Which involves verifying the RTL block solely without including any other SoC level blocks. The main target at this level is to verify the block functionality in all possible cases by driving the input directly to the block without concentrating on verifying the timing diagram requirements between various blocks.
- **SoC Level Verification**: Based on prior knowledge of the created block level test-cases, this flow stubs all the blocks interacting with the intended DUT. The main target of the scenarios created at this level is to verify the correct DUT behavior while driving the input based on the intended timing diagram from other blocks.

The second phase is the firmware driver verification. After performing the steps mentioned earlier, the firmware driver code is verified using the following two approaches:

- **FPGA Verification Flow**: The firmware driver is compiled and the generated ELF (Executable Loading Format) is converted to hex format to be loaded on the processor embedded in the FPGA board. Verification is performed using the SWD (Serial Wire Debug) port [13] where register values are read and functionality of the code is verified correctly. The drawbacks of this approach are threefold: First, the long FPGA bit file generation cycle leading to large turn around time if an issue is detected in the RTL. Second, debugging the issues on FPGA level needs probe insertion on specific signals in the design which not only increases the bit file generation time, but also allocates extra chip area for the probe memory. Third, verifying using the SWD is very basic process and is done manually.
- **Simulation Based Verification Flow**: The firmware driver hex file is also loaded on the processor in the simulation flow. The DUT input is derived from the firmware driver side and passed to the reference model to generate the intended output, then propagated to the SV-side through file I/O. This flow does not suffer from the issues in the FPGA flow, however it has from the following limitations: First, randomization in C-side is very elementary and lacks complex constraints present in SV. Second, the approach also lacks the ability to apply the same sequences developed in the SoC level verification to obtain high coverage results.

The proposed approach defines a complete UVM environment capable of combining the RTL block level and SoC level with the ability for verifying the firmware driver code using the same random sequences in an easy automated way. The proposed UVM environment operates in two modes: Stubbing mode, where the UVM agent operates first in active/re-active state to mimic the behavior of the processor while interacting with the DUT as illustrated in Figure 2. The sequencer generates the random transaction, then passes it to both reference model DPI (Direct Programming Interface) [14] functions and UVM driver which in terms forwards it to the DUT. Then, the monitor samples transactions performed by the driver and passes it to the scoreboard.

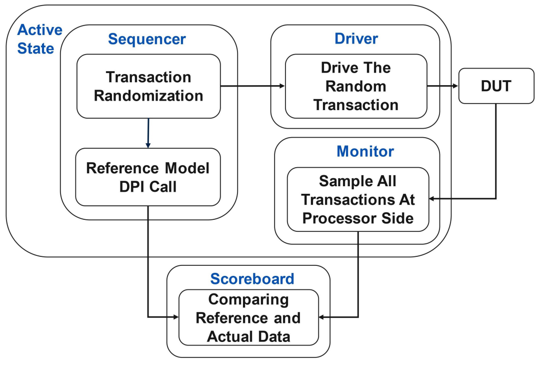
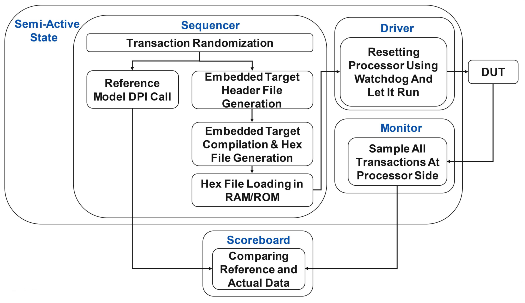

Active State
Sequencer
Transaction Randomization
Reference Model DPI Call
Driver
Drive The Random Transaction
DUT
Monitor
Sample All Transactions At Processor Side
Scoreboard
Comparing Reference and Actual Data

Fig. 2: Agent active state

In order to verify the functionality of the firmware driver, a new state for the UVM agent is proposed entitled semi-active/semi-reactive state where the processor is involved without stubbing as mentioned in Figure 3. The sequencer starts by randomizing the transaction and passing the input to the reference model through the DPI functions. To overcome the drawbacks in the techniques mentioned earlier, the sequencer generates C-header file on the fly containing the randomized data from the SV side, then it compiles the generated header file with the firmware driver file list to generate the embedded target hex file with the aid of $\$system$ system task to call GCC compiler. The hex file is stored in an array using $\$readmemh$ system task, then it is loaded in either the ROM or RAM using backdoor access to their full hierarchical path. This is achieved by passing the array content and the hierarchical name to a DPI function that calls VPI (Verification Programming Interface) subroutines [14].

The driver in this agent state is only used for resetting the processor through the watchdog timer. The processor afterwards starts loading either the ROM or RAM code and executes the firmware code instructions without any interference from the UVM driver. The monitor shall sample all transactions performed by the processor including both opcode and data transactions then passes them to the scoreboard. Since the UVM driver has no control on the input derived to the DUT, the reference data is derived from the sequencer not the driver.


Semi-Active State
Sequencer
Transaction Randomization
Reference Model DPI Call
Embedded Target Header File Generation
Embedded Target Compilation & Hex File Generation
Hex File Loading in RAM/ROM
Driver
Resetting Processor Using Watchdog And Let It Run
DUT
Monitor
Sample All Transactions At Processor Side
Scoreboard
Comparing Reference and Actual Data
Fig. 3: Agent semi-active state

V. Case Study

In order to realize the proposed verification criteria, a complete SoC is studied. As illustrated in figure 4, the system consists of two masters and single slave. The first master is a processor, meanwhile the other master is a generic controller. Both masters are communicating with the encryption engine slave through high speed AMBA AHB bus [11].

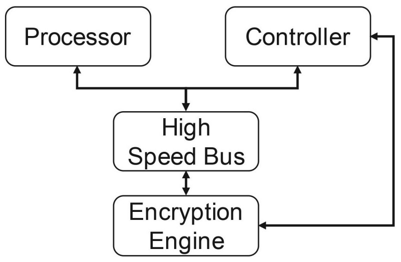


Fig. 4: Studied SoC design

The encryption engine is not only capable of applying confidentiality for encryption/decryption mechanism, but also supports integrity mode for verifying authenticated users. The supported symmetric encryption/decryption algorithms are: AES (Advanced Encryption Standard) [15] and DES (Data Encryption Standard) [16]. As illustrated in Figure 5, the engine consists of RIF (Register Interface) for processor configuration, input and output memories for interaction with the controller.

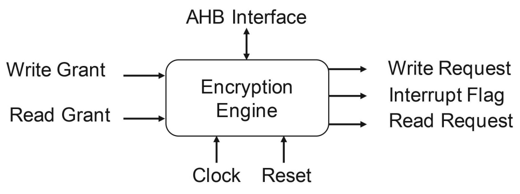


Fig. 5: Encryption engine block

Interaction between the encryption engine and the controller is performed using the following side band channel signals:

- **Write Request**: Encryption engine is asking the controller to start writing plain text data.
- **Write Acknowledgment**: Controller is informing the encryption engine that write request is accepted.
- **Write Grant**: Controller is acknowledging the end of write phase.
- **Read Request**: Encryption engine is asking the controller to start reading cipher text data.
- **Read Acknowledgment**: Controller is informing the encryption engine that read request is accepted.
- **Read Grant**: Controller is acknowledging the end of read phase.

The communication scheme between both masters and the encryption engine is performed as illustrated in Figure 6. The processor starts to configure the engine parameters through its RIF including the encryption scheme, the plain text size, mode of operation either confidentiality or integrity, and encryption/decryption algorithm. The processor then asserts the engine start signal to initiate the confidentiality/integrity operation. The engine in terms signals the controller with side band write request signal. The controller then responds with write acknowledgment whenever it becomes ready for plain text data transmission to the engine. After finishing data transmission, the controller asserts the write grant.

In case of confidentiality mode, the engine starts generating the cipher text data in the output memory, then signals the controller with read request. The controller responds with read acknowledgment when being ready and starts to read the cipher data, then eventually asserts the read grant. Meanwhile, in case of integrity mode, the engine generates the digital signature in the RIF, then the processor reads the generated signature. Eventually, the engine asserts the done register and asserts interrupt on the processor NVIC (Nested Vector Interrupt Control) ports.

Verifying the slave engine mentioned earlier is performed using two interfaces: AHB interface for any AHB transaction and crypto interface for side band channel signals. Communication between UVM classes and the concrete interfaces connected to the DUT is performed using virtual interface

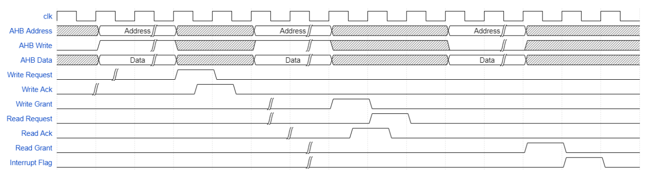


Fig. 6: Encryption engine waveform

instances as illustrated in Figure 7. The UVM $config\_db$ database class is used to propagate the initialized virtual interface instances from the top level module to the corresponding driver and monitor.

The UVM environment is built using three UVM agents:

- **Processor Agent**: The agent is responsible for RIF configuration and capturing the digital signature. It has two states: reactive and semi-reactive. In the reactive state, the sequencer forwards the random data to the driver and the DPI reference model function. Side band channel signals sampled from the monitor are passed to the sequencer to control the sequence. This is performed through either separate broadcast or unidirectional UVM TLM channel established between both UVM components. The signal value is retrieved at the sequencer side using the get method provided by the UVM TLM FIFO. The differences in case of semi-reactive state are: The hex file generation is performed by the processor and the limited usage for the UVM driver for remapping if needed and resetting the processor. Since the monitor in the semi-reactive state shall sample all the transactions performed by the physical processor, an address decoder is used to exclude the op-code transactions and focus on the encryption engine transactions.
- **Controller Agent**: The agent is responsible for passing the plain text data to the engine and read the cipher data in case of confidentiality mode. Similar to the processor agent, both states are enabled. However, the side band channel signals are different.
- **Crypto Agent**: The main purpose of using this agent is to verify the functionality of the bus matrix. It has only passive state to forward the sampled data to the scoreboard. Since this agent has the visibility on all transactions passed from the two masters and the slave, it is used to broadcast the transactions to the subscriber.

The scoreboard consists of four queues buffering all reference and actual transactions from corresponding agents. Checking mechanism is performed at the following spots:

- **Processor Side**: Where reference data is generated by the processor sequencer and actual data is sampled by the processor monitor.
- **Controller Side**: Where reference data is generated by the controller sequencer and actual data is sampled by the controller monitor.
- **Crypto Side**: Where reference data is generated by either the processor sequencer or the controller sequencer and actual data is sampled by crypto monitor.
- **SW Driver Side**: Where the reference data is dumped in the C-header file to be compiled along with other firmware driver files to generate the hex file. In such case the firmware is able to assert on the resultant data compared with the reference data dumped earlier from the sequencer.

Random test pattern generation and test scenarios deployment are performed using a virtual sequence controlling children sequences running on the corresponding children sequencers. The UVM base test class can be configured to run three families of virtual sequences: Stubbing the processor and the controller, where both processor and controller agents are in re-active state; Stubbing the controller only, where the processor agent is in semi-reactive state and the controller agent is in re-active state; No stubs, where both processor and controller agents are in semi-reactive state.

Switching between various virtual sequence families is performed on the command line using $\$plusargs$ system function propagating the sequence type as a string to the base test class. Since the children sequences must be configured based on the sequence type, the string variable is propagated from the base test class to the children sequences using the $resource\_db$ database class.

Timing diagram verification is performed through System Verilog Assertions (SVAs), where concurrent assertions are written in a separate module instantiated under the top level testbench module.

VI. Conclusion And Future Work

The proposed verification methodology provides the privilege to verify both hardware design and firmware driver within the same automated UVM environment in the simulation phase. This leads to reduce both the verification life cycle and the burden on firmware test engineer. Deploying the proposed approach on the full chip design to verify E2E (End To End) scenarios shall be added in the future for system completeness.

VII. Acknowledgement

This work is funded by ST Microelectronics. The author would like to thank Mostafa Khamis and Omar Ibrahim for their comments that greatly improved the manuscript.

References

[1] H. Foster, “2018 FPGA Functional Verification Trends,” in 19th International Workshop on Microprocessor and SOC Test and Verification (MTV), Austin, TX, USA, Dec. 2018.

[2] S. Hosny and A. Baher, “Design Crawler: A Web Application for Digital Design Metadata Analysis,” 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV), Austin, TX, USA, Dec. 2019.

[3] Accellera Systems Initiative, “Universal Verification Methodology (UVM) 1.2 User Guide,” Oct. 2015.

[4] A. Hussien et al., “Development of a Generic and a Reconfigurable UVM-Based Verification Environment for SoC Buses,” in 31st International Conference on Microelectronics (ICM), pp. 195-198, Cairo, Egypt, Dec. 2019.

[5] K. Salah, “A Unified UVM Architecture for Flash-Based Memory,” in 18th International Workshop on Microprocessor and SOC Test and Verification (MTV), pp. 1-4, Austin, TX, Dec. 2017.

[6] N. K. Doshi, S. Suryawanshi, and G. N. Kumar, ”Development of generic verification environment based on UVM with case study on HMC controller,” in IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT), pp. 550-553, Bangalore, India, May 2016.

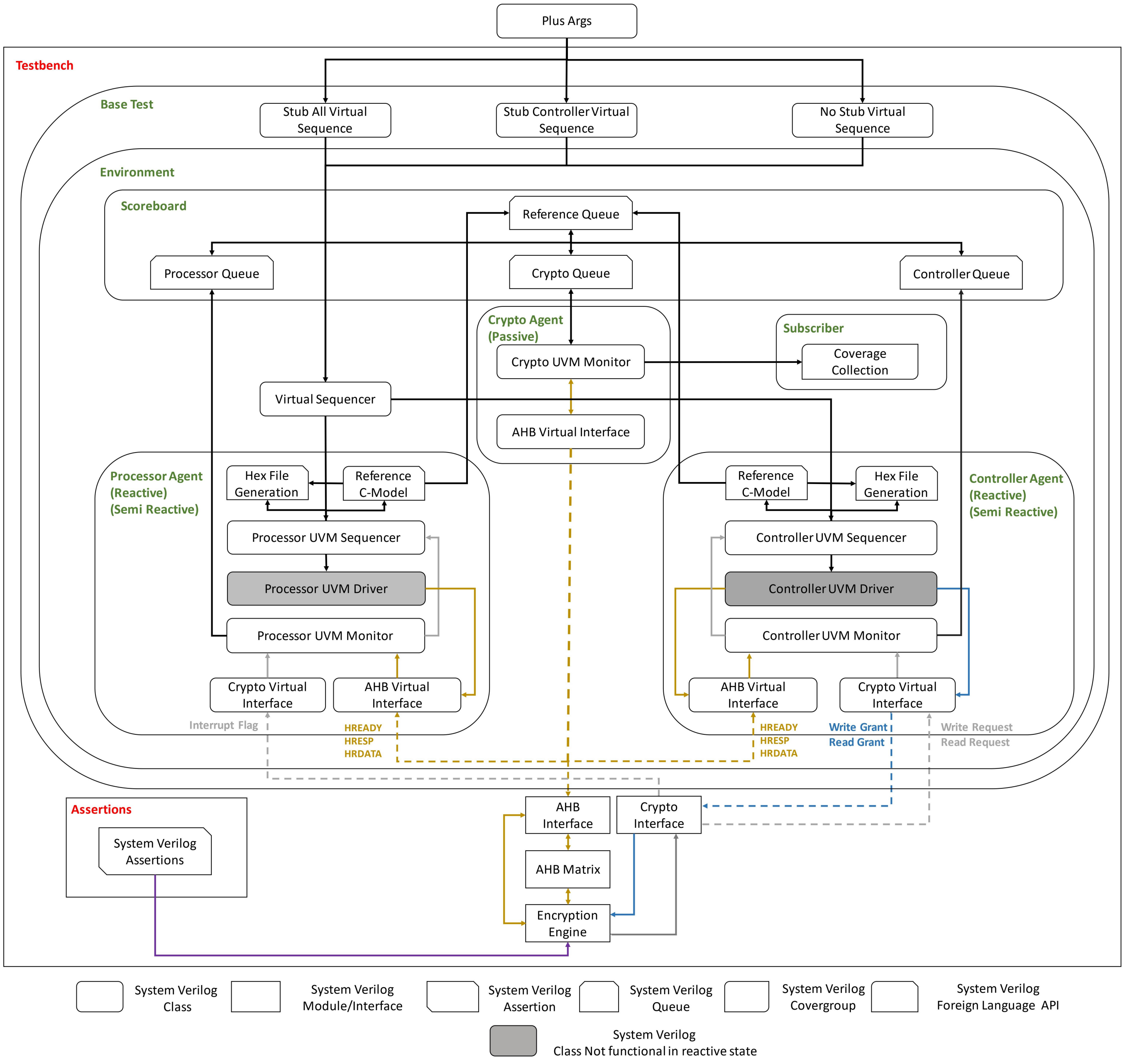


Fig. 7: Encryption engine verification environment

[7] T. M. Pavithran and R. Bhakthavatchalu, “UVM based testbench architecture for logic sub-system verification,” in International Conference on Technological Advancements in Power and Energy (TAP Energy), pp. 1-5, Kollam, India, Dec. 2017.
[8] M. F. U. Rahman and D. Naveen, “Verification of a Digital Video Broadcasting – Satellite to Handheld (DVB-SH) IP Using UVM,” in Second International Conference on Computational Intelligence & Communication Technology (CICT), pp. 462-467, Ghaziabad, India, Feb. 2016.
[9] J. Wang, N. Tan, Y. Zhou, T. Li and J. Xia, “A UVM Verification Platform for RISC-V SoC from Module to System Level,” in IEEE 5th International Conference on Integrated Circuits and Microsystems (ICICM), pp. 242-246, Nanjing, China, Oct. 2020.
[10] ARM Corporation, “AMBA AXI™ and ACE™ Protocol Specification,” Sep. 2003.
[11] ARM Corporation, “AMBA™ Specification,” Rev 2.0, June 1999.
[12] STMicroelectronics Corporation, Texas Instruments, “STBus communication system concepts and definitions UM0484 User manual,” Oct. 2012.
[13] ARM Corporation “ARM Debug Interface Architecture Specification,” V5.0, Aug. 2006.
[14] IEEE Computer Society and the IEEE Standards Association Corporate Advisory Group, “IEEE Standard for SystemVerilog Unified Hardware Design, Specification, and Verification Language 1800™-2012,” Feb 2013.
[15] F. J. D’souza and D. Panchal, “Advanced encryption standard (AES) security enhancement using hybrid approach,” in International Conference on Computing, Communication and Automation (ICCCA), pp. 647-652, Noida, India, May 2017.
[16] Seung-Jo Han, Heang-Soo Oh and Jongan Park, “The improved data encryption standard (DES) algorithm,” Proceedings of ISSSTA’95 International Symposium on Spread Spectrum Techniques and Applications, pp. 1310-1314, vol.3, Mainz, Germany, Aug. 2002.