Verification of Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C) Protocols

Ravi S[1], Seeram Jahnavi[2], Veerla Nageswara Rao[2], Tammana Tarak Ram[2], Shaik Suhail Ahmed[2]

[1]Associate Professor, Department of ECE, Seshadri Rao Gudlavalleru Engineering College, India
[2]Student, Department of ECE, Seshadri Rao Gudlavalleru Engineering College, India

Abstract:
The Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C) protocols are widely used for communication between microcontrollers, sensors, and other digital devices. The correctness and reliability of these protocols are essential for proper system functioning. Therefore, it is necessary to verify these protocols thoroughly to ensure that they are error-free.
In this paper, a novel verification environment is proposed for the verification of SPI and I2C protocols using SystemVerilog. Since, SystemVerilog incorporates Object oriented Programming (OOPs) concept in Verilog programming language, stimulus generation and its application to the DUT are done at higher abstraction level. Further, the proposed approach involves creating verification environments using the Universal Verification Methodology (UVM) framework and verifying the protocols' functionality and performance.
Key Word*: Code coverage; functional coverage; SPI; I2C; System Verilog;*

I. Introduction

The Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I2C) protocols are widely used for communication between microcontrollers, sensors, and other digital devices. In particular SPI protocol is used in ARM, AVR, and Power PC Microcontrollers for communication purpose[1]. Communication protocols are critical for ensuring reliable and efficient data transfer between devices, and any errors in the communication process can lead to system failures[2]. Therefore, it is essential to verify the correctness and reliability of these protocols.

Verification is a crucial aspect of the development process, and it helps to identify and fix errors early in the design cycle. SystemVerilog is a hardware description and verification language that provides constructs for creating complex designs and verifying their functionality[3]. The Universal Verification Methodology (UVM) is a framework [6] built on top of SystemVerilog, which provides a standard methodology for creating and verifying verification environments. SPI protocol was introduced by Motorola, while I2C was introduced by Philips[4].

In this paper, we propose a methodology for verifying the SPI and I2C protocols using SystemVerilog and UVM. The methodology involves creating functional models of the protocols and verifying their functionality and performance using directed testing, random testing, and assertion-based testing. We will also perform performance testing to ensure that the protocols meet their timing and bandwidth requirements.

The project aims to identify and fix errors in the communication process, including deadlocks, livelocks, and other types of errors that can occur due to timing or other issues. The verification process will also ensure that the protocols adhere to their specifications and provide reliable and efficient communication between devices. The outcome of this project will be a verified model of the SPI and I2C protocols using SystemVerilog and UVM, which can be used to identify and fix errors in the communication process. The methodology proposed in this project can also be extended to other communication protocols to ensure their correctness and reliability.

II. SPI Protocol

The Serial Peripheral Interface (SPI) protocol is a synchronous serial communication interface used for short-distance communication between microcontrollers, sensors, and other digital devices. SPI allows for the transfer of data between devices at high speeds, making it a popular choice for applications that require high data rates. SPI uses a master-slave architecture, where the master initiates and controls the communication, and the slave responds to the master's commands. The master generates a clock signal, and the slave devices synchronize their data transfer with the clock.

SPI is a versatile protocol and can support various data formats, including single and multi-byte data transfers, as well as different clock frequencies. However, SPI does not provide a standardized protocol for addressing devices, error detection, or flow control, making it less suitable for long-distance communication or complex systems.
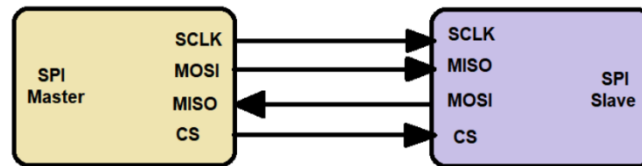

Fig 1 : SPI with single master and slave.

The communication between the master and slave devices occurs over four lines:
1. **MOSI (Master Output Slave Input) -** The MOSI line is used by the master device to send data to the slave device. The master drives the MOSI line with the data it wants to transmit, and the slave reads this data from the MOSI line.
2. **MISO (Master Input Slave Output) -** The MISO line is used by the slave device to send data to the master device. The slave drives the MISO line with the data it wants to transmit, and the master reads this data from the MISO line.
3. **SCK (Serial Clock) -** The SCK line is used to synchronize the data transfer between the master and the slave devices. The master generates clock pulses on the SCK line, and both the master and slave devices use this clock to control the timing of data transfer.
4. **SS (Slave Select) -** The SS line is used by the master device to select which slave device it wants to communicate with. The master drives the SS line low to select the slave device, and drives it high to deselect the device.

The number of slave devices that can be connected to a master in an SPI network depends on the hardware configuration of the system. The SPI protocol does not specify a maximum limit on the number of slaves that can be connected, but the number of available chip select lines on the master device usually limits the number of slaves.

Each slave device in an SPI network requires a separate chip select (SS) line from the master device. The master selects a particular slave device by driving its SS line low while keeping all other SS lines high. Therefore, the maximum number of slave devices that can be connected to a master is equal to the number of available SS lines on the master device.
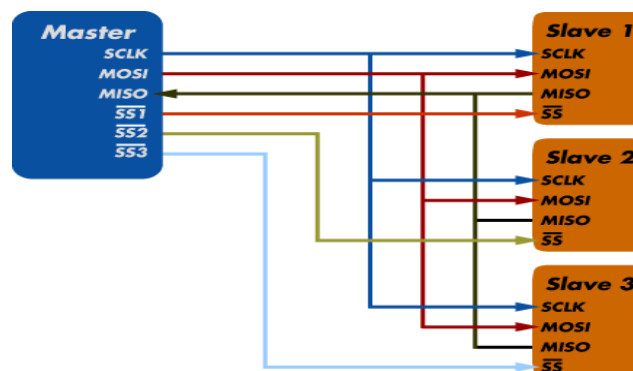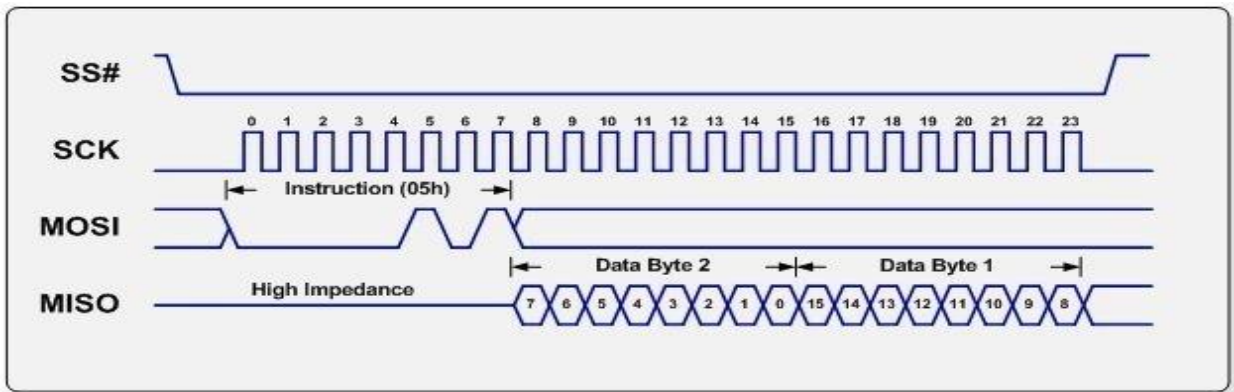

Fig 2: SPI with single mster and multiple slaves.

Fig 3: Data transfer in SPI protocol.

III. I2C PROTOCOL

I2C (Inter-Integrated Circuit) is a synchronous, multi-master and multi-slave serial communication bus protocol. I2C protocol uses two bidirectional lines, SDA (serial data) and SCL (serial clock), to transmit data between devices. Devices on the I2C bus are identified by unique 7-bit or 10-bit addresses, which are sent at the beginning of each communication transaction.

The basic operation of I2C protocol involves a master device, which initiates communication, and one or more slave devices, which respond to commands from the master. The master sends a start condition on the bus, followed by the slave address and a read/write bit to indicate the direction of communication. The master then sends or receives data from the slave, and the transaction is completed with a stop condition on the bus.

I2C supports multiple masters on the same bus, as well as multiple slaves, and allows for communication at varying data rates. It is commonly used in applications such as sensors, displays, and memory devices.
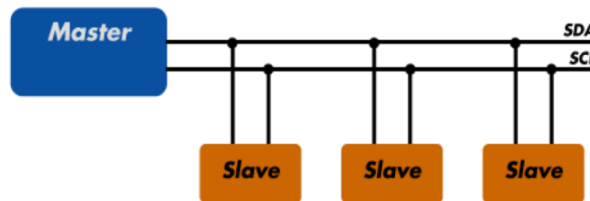

Fig 4: I2C protocol with single master and multiple slaves configuration.

The I2C protocol also includes a number of features for error detection and correction, including checksums and retries. These features help ensure the reliability and integrity of the data being transmitted over the bus.
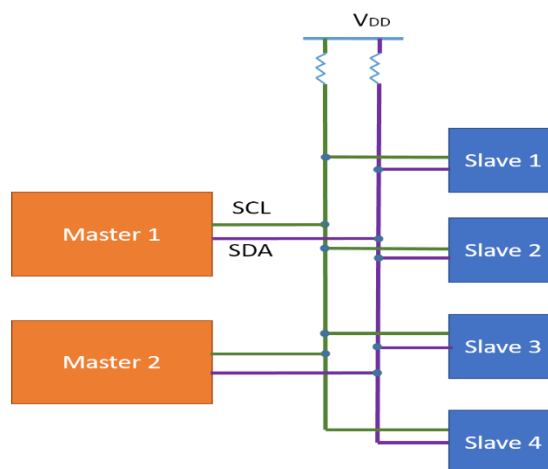

Fig 5: I2C protocol with multiple masters and multiple slaves configuration.
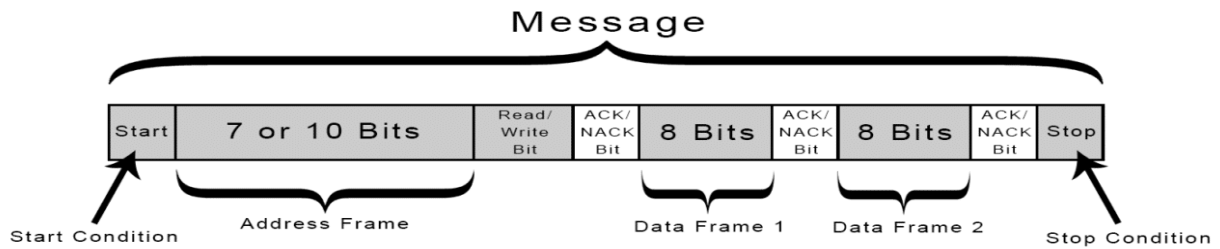
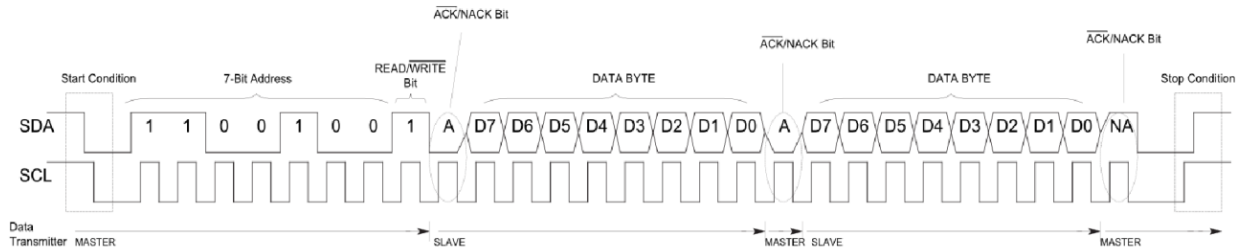**Fig 6(a): I2C frame structure with the Address and Data frame.**

Fig 6(b): Data transfer sequence in I2C protocol.

For data transmission in I2C protocol, the master device drives the SDA line to send the data byte to the slave device, while the SCL line is toggled to clock the data. Each data byte is acknowledged by the receiver (either the master or the slave) with an ACK or NACK signal on the SDA line[5].

For data reception, the slave device drives the SDA line to send the data byte to the master device, while the SCL line is toggled by the master to clock the data. After each byte is received, the master sends an ACK or NACK signal on the SDA line to acknowledge or reject the data. The process continues until all the data bytes have been transferred, after which the master sends a stop condition on the bus to indicate the end of the communication transaction. I2C also supports multi-byte data transfers, where the master or slave can send or receive multiple bytes of data in a single transaction. The multi-byte transfer is accomplished by keeping the SDA line stable during the ACK/NACK cycle and clocking the data on the SCL line. The verification environment in the proposed apporach verifies the I2C module for four operational modes namely, Master tarnsmit, master receive, Slave transmit, and slave receive.

IV. Proposed methodology

Verification in recent days becomes more effective by using a test environment which contains multiple interactive blocks. Some of these blocks include, generator for stimulus generation, driver to force inputs to the DUT, monitor for observing the results of DUT, score board that compares the results of DUT with the results of the reference model, and generating report which shows possible errors and the corresponding code line number for easy debugging, etc. Various blocks of verification environment and the relation between those blocks are illustrated in Figure 7. Functional details of each of these blocks are described as follows:

❖ **DUT:** DUT refers to the hardware design, which is written in Verilog or VHDL. DUT is a term that is commonly used in post-fabrication silicon validation. Pre validation is also known as Design Under Verification, or DUV for short.

❖ **Interface:** If the design had hundreds of port signals, connecting, maintaining, and re-using those signals would be difficult. Instead, we can group all of the design's I/O ports into a container that serves as an interface to the DUT. This interface can then be used to drive the design with values.
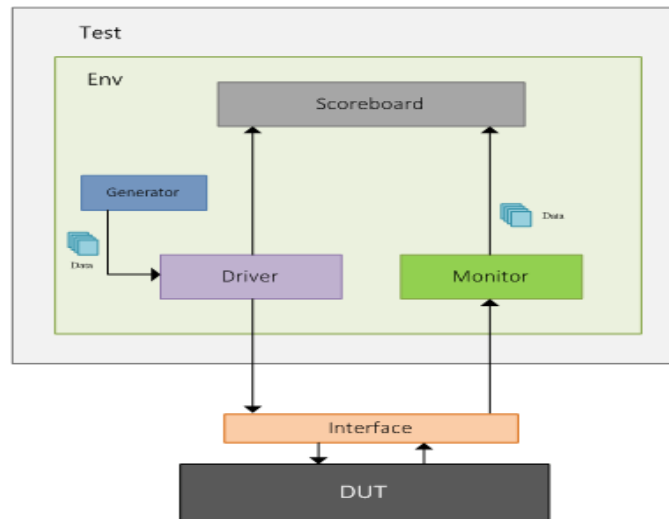
Fig 7: Components of Verification environment

❖ **Driver:** The driver is the verification component that performs the pin-wiggling of the DUT through an interface. When the driver has to drive some input values into the design, it simply needs to invoke this pre-defined task in the interface without knowing the time relationship between these signals. The time information is defined within the interface's task. This is the amount of abstraction required to increase the flexibility and scalability of testbenches. If the interface changes in the future, the new driver can call the same task and drive signals in a different way.

❖ **Generator:** The generator is a verification component that can generate and submit valid data transactions to the driver. The driver can then simply drive the data supplied by the generator over the interface. Data transactions are implemented as class objects, as seen in the graphic above by the blue squares. It is the driver's responsibility to obtain the data object and translate it into a format that the DUT can understand.

❖ **Monitor:** Up to this point, it has been discussed how data is driven to the DUT. But that's halfway done because our primary goal is to validate the design. The DUT processes input data and delivers the output pins the result. The processed data is picked up by the monitor, converted into a data object, and sent to the scoreboard.

❖ **Scoreboard:** The Scoreboard might consist of a reference model that performs same as the DUT. This model correlates with the DUT's intended behaviours. This reference model receives input from the DUT. So, if the DUT has a functional difficulty, the output of the DUT will differ from the output of our reference model. So comparing the design outputs to the reference model will tell us if the design has a functional flaw. Normally, this is done on the scoreboard

❖ **Environment:** In the environment class we can include multiple components. As new components can be inserted into the same environment for the future project, it increases the flexibility and scalability of the verification.

❖ **Test:** Keeping in mind that there will likely be hundreds of tests, it is not practical to directly alter the environment for each test, the test will create an environment object and configure it as it sees fit. Instead, we prefer environment settings that can be adjusted for every test. The test will be more effective and have better control over stimulus creation as a result.

In EDA tools such as EDA Playground [8], simulation of these protocols can be done using digital simulation techniques. This involves creating a simulation model of the protocol using Verilog or VHDL, and then simulating the model to test its functionality and performance. The simulation can be done at the signal level, where the behavior of individual signals is analysed, or at the transaction level, where the behavior of the entire protocol is analysed [9]. Further, using Universal Verification Methodology (UVM), provides an additional benefit of developing verification environment without making any changes in the design under test (DUT) [10].
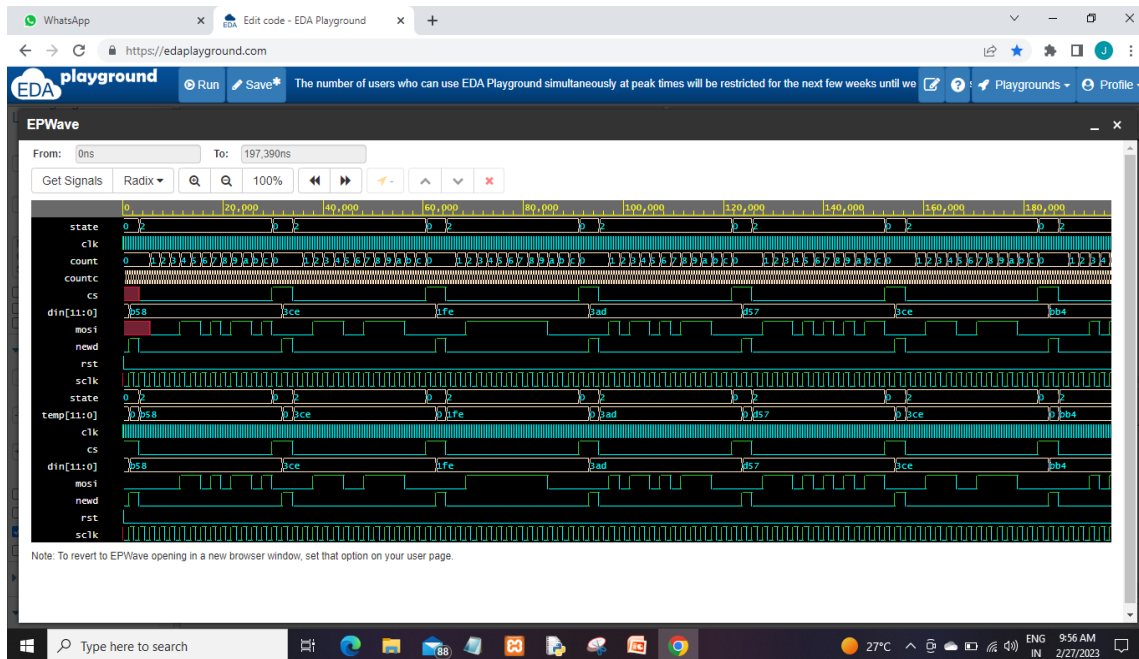
Fig 8(a): Simulation result of SPI protocol obtained through EDA Playground

- **State:** state refers to a specific condition or phase that the SPI module is in during a transaction.
- **Clk:** In SPI verification, the clock signal (clk) plays a critical role as it synchronizes the data transfer between the master and slave devices. The clock signal determines the timing of when data is transmitted and received on the MOSI and MISO signals.
- **Sclk:** Sclk (Serial Clock) is a key signal in the SPI (Serial Peripheral Interface) protocol, which is used to synchronize data transfer between a master device and a slave device. The Sclk signal is generated by the master device and is used to control the timing of the data transfer.
- **newd:** New data is the data that is to be transmitted.
- **CS**: CS (chip select) line represents particular slave device to be selected.
- **din[11:0]:** din[11:0] in SPI verification refers to the input data signal that is transmitted from the master device to the slave device over the MOSI signal line. The Din signal is a vector signal, with a width of 12 bits, and is typically used to transmit data from the master device to the slave device.
- **Count:** count refers to the number of clock cycles that are used to transfer data between the master and slave devices. The count is typically used to specify the length of the SPI transaction and to ensure that the correct number of bits or bytes is transferred.
- **rst:** reset signal can be used to initialize the SPI module and to ensure that it is in a known state before starting a transaction. The reset signal is typically used to bring the SPI module into a default state and to clear any internal registers or variables that may affect the behavior of the module.
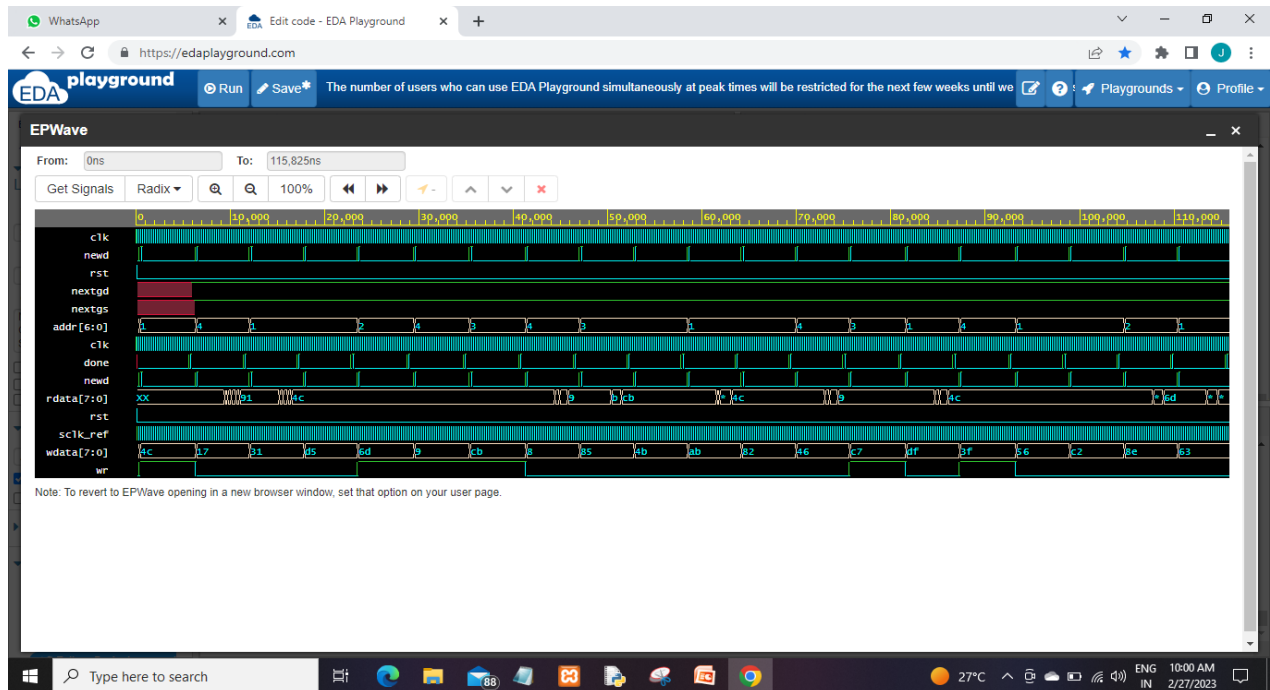
Fig 8(b): Simulation result of I2C protocol obtained through EDA Playground.

- **nextgd:** Nextgd (Next Generation Data) is a signal used in I2C (Inter-Integrated Circuit) verification to indicate the next data byte that will be transferred on the I2C bus. Nextgd is typically generated by the I2C master and is used to indicate the direction of the data transfer, as well as the address of the slave device being accessed.

- **nextgs:** Nextgs (Next Generation Start) is a signal used in I2C verification to indicate the next START condition that will be generated on the I2C bus. Nextgs is typically generated by the I2C master and is used to initiate a new data transfer on the I2C bus.

- **rdata:** rdata (Read Data) in I2C verification refers to the data that is being read from the slave device by the I2C master. The rdata signal is typically driven by the slave device in response to a read command from the I2C master [7].

- **wdata:** wdata (Write Data) in I2C verification refers to the data that is being written from the I2C master to the slave device on the I2C bus. The wdata signal is typically driven by the I2C master after sending the START condition and addressing the slave device.

- **Done:** Done in I2C verification refers to a signal that indicates the completion of a transaction between the I2C master and slave devices.

V. Conclusion

In this paper, a Systemverilog based verification environment is developed for two most commonly used wired communication protocols namely, SPI and I2C. The verification environment checks both functional coverage and code coverage for different modes of transmitting and receiving data. Particularly the operation is verified under full-duplex mode with single master and multiple-slave configurations. It is observed from the simulation results that the reusable SPI and I2C protocols are verified of its functionality using Systemverilog. The performance of the verification procedure is estimated in terms of functional coverage and code coverage. That is the RTL code of the DUT is covered by test cases. This verification work can be extended by considering verification of other advanced communication protocols[11, 12].

References

[1]. Chetan N, and R. Krishna, "Verification of SPI protocol Single Master Multiple Slaves using System verilog and Universal Verification Methodology (UVM)", International Journal of Engineering Research and Applications, Vol. 11, Issue 7, July 2021.
[2]. Ananthula Srinivas, M.Kiran Kumar, and Jugal Kishore Bhandari., Design and Verification of Serial Peripheral Interface", International Journal of Engineering development and Research, Vol. 1, Issue 3, Dec. 2014, pp. 130 -136.

[3]. System Verilog Tutorial: https://www.chipverify.com/systemverilog/systemverilog tutorial
[4]. F.Leens, "An Introduction to I2C and SPI Protocols," IEEE Instrumentation & Measurement Magazine, pp. 8-13, February 2009, DOI: 10.1109/MIM.2009.4762946
[5]. M.Sukhanya, and K.Gavaskar, "Functional Verification Environment for I2C Master Controller using System Verilog", 2017 4th International Conference on Signal Processing, Communications and Networking (ICSCN - 2017), March 16 – 18, 2017, Chennai, INDIA, DOI: 10.1109/ICSCN.2017.8085732
[6]. R.K Vaishnavi, S Bindu, and Sheik Chandbasha, "Design and Verification of APB Protocol by using System Verilog and Universal Verification Methodology," International Research Journal of Engineering and Technology (IRJET), vol. 06, no. 06, pp. 23950072, 2019.
[7]. Purvi Mulani , "Verification of I2C DUT using System Verilog", International Journal of Advanced Engineering Technology, IJAET, Vol. 1, Issue 3, Dec. 2020.
[8]. EDA Playground URL : https://edaplayground.com
[9]. G. Tumbush and Chris Spear. SystemVerilog for Verification, Third Edition: A Guide to Learning the Testbench Language Features. Springer, 2012.
[10]. Mohamed Azheruddin and Anand MJ., "Development of Verification Environment for I2C Controller Using System Verilog and UVM, International Journal of Computer Science and Mobile Computing, IJCSMC, Vol. 8, Issue. 5, May 2019, pg.100 – 108
[11]. Prashant Dwivedi, Neha Mishra, Amit SinghRajput, "Assertion & Functional Coverage Driven Verification of AMBA Advance Peripheral Bus Protocol Using System Verilog", International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT), 2021.
[12]. Vaishnavi RK, Design and Verification of APB Protocol by using System Verilog and Universal Verification Methodology, International Research Journal of Engineering and Technology (IRJET), Vol. 6, issue 6, June 2019.