International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 09 Issue: 09 | Sep 2022 www.irjet.net p-ISSN: 2395-0072

Design and Verification of the UART and SPI protocol using UVM

Vivekananda T[1], Mahesh Kumar N[2]

[1]PG Student (M.Tech in VLSI Design & Embedded Systems), Department of Electronics and Communication Engineering, Dayananda Sagar College of Engineering, Bengaluru, Karnataka
[2]Associate Professor, Department of Electronics and Communication Engineering, Dayananda Sagar College of Engineering, Bengaluru, Karnataka

---

Abstract - *Communication protocol plays an important role in organizing communication between the devices. These protocols have specific set of rules and agreed upon between the devices to achieve successful communication. UART and SPI are two widely used protocols in serial communication. This paper includes the design of the UART and SPI functional module SV hardware description language. Functional verification of the UART and SPI is performed with the help of the Universal Verification Methodology. The reusable UVM testbench architecture is designed to drive the randomized stimuli to the unit under test to check the functional correctness, by comparing the collected response to the intended response using the scoreboard mechanism. Reusability of the stimulus will reduce the overall execution time.*

Key Words: UART-Universal Asynchronous Receiver-Transmitter, SPI-Serial Peripheral Interface, UVM-Universal Verification Methodology, SV-SystemVerilog, Code coverage analysis

1. INTRODUCTION

In VLSI chip design process, verification of the design plays a very important role. The design must be tested for different test cases to verify its functionality. It takes a lot of effort, time for the verification of integrated circuits and its efficiency measure is a tedious job. The complexity of verification increases with the complexity of the device's design [1]. The verification of complex designs consumes approximately 60-70% of the product development life cycle, and cannot be achieved by the traditional directed testing method [2]. Verification of IC's using Verilog/ SystemVerilog lacks the test bench's reusability for environment and also consumes more time. Universal Verification Methodology is the standard form of Verification [1]. The UVM has numerous class libraries which enable reuse of the environment across the projects. In this paper, we have used a two communication protocol designs to do verification using the UVM. One is UART and another one is SPI. Communication protocols can be divided into two categories: Inter system protocols and Intra system protocols. Inter system protocol is implemented in communication between two distinct devices, such as a computer and a microcontroller kit through inter bus system. UART is the one of the inter system communication protocol. UART stands for Universal Asynchronous Receiver-Transmitter. UART is a hardware communication protocol that is primarily used for data transmission over a long distance and between the devices with high reliability. It can be used in both transmission and receiving of a data serially in asynchronous way [1]. UART is an asynchronous, means communication between the devices without using clock signal. It has two pins, one is transmitter and another is receiver. Parallel data is converted into serial format by UART transmitter after receiving from system bus and transmits. Serial data is converted back into parallel format by UART receiver and send it to system bus. For transmission and receiving data between two UART modules, only two wires are required.

Intra system protocol is implemented in communication between two devices on the printed circuit board or SoC's. SPI is the one of the intra system communication protocol. SPI stands for Serial Peripheral Interface. SPI is primarily used for data transmission over a short distance with high speed and reliability. SPI is a synchronous, full-duplex interface with a master slave architecture, which uses clock for the communication. The rising or falling edge of the clock is used to synchronise the data from the master or slave. Data can be sent or received simultaneously. The SPI comes in either 3-wire or 4-wire interface. The clock signal produced by the master serves as the synchronizer for data transmission.

2. OVERVIEW OF UART AND SPI

2.1 UART

UART is one of the most widely used hardware communication protocol. It supports bi-directional, asynchronous and serial data transmission. UART can communicate in the simplex, half-duplex and full-duplex modes. One way transmission is allowed in simplex mode. Either transmitter or receiver is allowed transmission at a time in half duplex mode; a receiver will be in idle state if the transmitter is transmitting data, and vice versa. Both way transmissions is allowed in a full duplex mode; both transmitter and receiver work at the same time [3]. UART typically contains two wires: a transmitter (Tx) and a receiver (Rx) as shown in the fig -1.

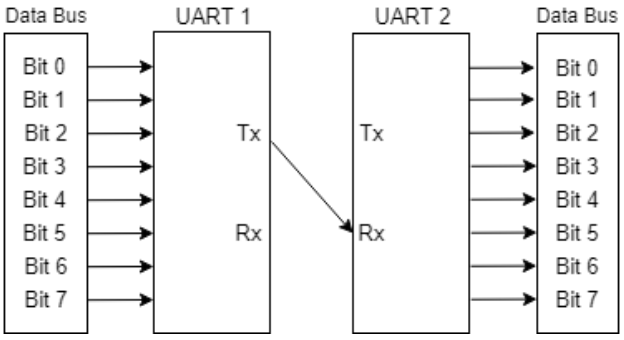


Fig -1: Universal Asynchronous Receiver Transmitter

The transmitting UART converts parallel data into serial data, once received from the data bus. The receiving UART translates back to parallel data after receiving serial data bit by bit. No clock synchronization between transmitter (Tx) and receiver (Rx), this means they have to agree beforehand to a clock frequency. This is done by setting the baud rate for UART Tx and UART Rx. Transmission of data speed is measured as a baud rate, which is represented in bits per second (bps). UART can works in different baud rates like 1200, 2400, 4800, 9600, and 115200 bps. Tx and Rx must function at same baud rate in the both UARTs [1] [3]. Data is sent in the form of packets.

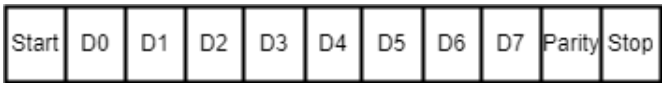


Fig -2: UART frame format

Frame format of a UART is shown in the fig -2. It consists of start bit, stop bit and parity bit. The beginning and finish of a message are represented by the start and stop bits. At the receiving end, the parity bit acts as the check bit. Parity bit is optional and depends on requirements.

Communication in UART is initiated by transmitter in the form of frame. Start, stop, and parity bits are added to construct a data frame. The data frame is then transmitted serially to Tx pin. Rx pin reads the data frame and removes the start and stop bit. Actually length of the data is 5 to 8 bits [4].

2.2 SPI

SPI is one of the most widely used communication protocol. SPI stands for Serial Peripheral Interface. Serial Peripheral Interface supports synchronous, full duplex communication and serial data transmission with high speed over the short distance. Short distance means communication between the devices on the same board. SPI is mainly used for communication between the microcontrollers/micro-processors and peripherals [6]. SPI protocols requires clock signal for synchronization between the transmission and reception during the communication. SPI protocol works in the master-slave configuration. SPI supports single master-single slave and single mater-multiple saves architectures. SPI supports bi-directional, synchronous and serial data transmission. Transmission takes place on synchronous clock signal generated by the master. SPI needs just four lines for the communication to happen between the master and the slave [7].

Fig -3: Serial Peripheral Interface

SPI single master-slave architecture is shown in the fig -3. SPI consists of the four signals: serial clock (SCK), chip select/ slave select (CS), master in slave out (MISO) and master out slave in (MOSI). The rising or falling edges of the clock are used to send and receive data from the master and slave, respectively. The signal description of the serial peripheral interface

Master In Slave Out (MISO): Using a MISO line, the master chip receives data from the slave. Transmits data serially one direction with MSB bit sent first. The MISO line will be in a high impedance state if the slave chip is not chosen.

Master Out Slave In (MOSI): Using a MOSI line, the master chip transmits data to the slave chip. MSB bit is sent first and in single direction serially during data transmission.

Serial Clock (SCK): Serial clock is used to synchronise data between the master & slave and thr MOSI & MISO signal line. A byte of data can be exchanged between a slave and master device in eight clock cycles. SCK is generated by the master device and is used as the slave device's input clock.

Chip/Slave Select (CS): The slave chip is selected using the chip select line of slave. It should stay active low during data transfer [6-8].

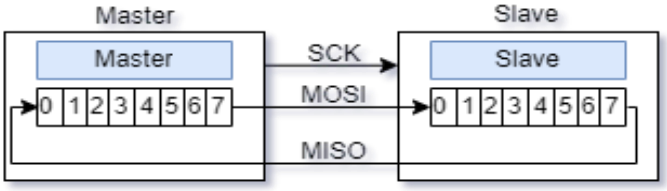


Fig - 4: SPI 8 bit data transmisson

After Communication in the SPI is initiated by the master and configures the clock. Master uses the clock configure to set the frequency which must be less than or equal to slave

device's maximum frequency. The master will select the particular slave device for communication through the slave's CS line to low state. MOSI line is used to shift the data from shift register to slave and slave shifts the data into shift register. MISO line is used to shift the data from slave to master in the same fashion as MOSI as shown in the fig -4 [6].

3. UNIVERSAL VERIFICATION METHODOLOGY

Verification is a process of determining whether the designed module meets the specific requirements in the design of IC's. UVM is a standard methodology for verifying integrated circuit designs. Abbreviation of UVM is Universal Verification Methodology. UVM is mainly derived from the open verification methodology (OVM). The OVM was created as a result of a joint effort between Cadence Design Systems and Mentor Graphics to integrate the ideas of Universal Reuse Methodology (URM) and Advanced Verification Methodology (AVM). UVM uses SystemVerilog for verification. Additionally, Synopsys joined and combined the Reference Verification Methodology (RVM) and OVM methodologies to create Universal Verification Methodology (UVM), which was approved by the Accellera committee to become the new standard to be used for the functional verification of integrated circuit designs in 2009 [2]. Based on the requirement of the project, some of the points considered in the UVM are code re-usability, build the verification environment, random stimulus generation and verify the design under test using the environment. Fig -5 shows the typical UVM testbench architecture.

The major three categories of classes at UVM are as follows.

uvm_object: Base class in the UVM for uvm data and other classes for operational method. uvm_component and uvm_transaction are inherited from it.

uvm_transaction: Serves as the root base class for uvm components and utilized in the generation of stimuli.

uvm component: are essentially static and remain throughout the simulation. Top, Agent, Sequence Item, Sequence, Test, Environment, Driver, Sequencer, Monitor, Scoreboard are the uvm_components.

The various phases of all the uvm components are

Build phase: Based on configuration and factory settings, it creates and configures component hierarchies.

Connect phase: Establishes connection between the ports of various uvm components with one another.

Run phase: Implement as a task for uvm components which is presence during the entire run-time.

Extract phase: Acquires data from scoreboards and functional coverage monitors and process it.

Check phase: Checks whether DUT is behaving properly and identify errors during execution of testbench.

Report phase: Displays simulation results and also saves results to a file.

Final phase: Termination of the simulation.

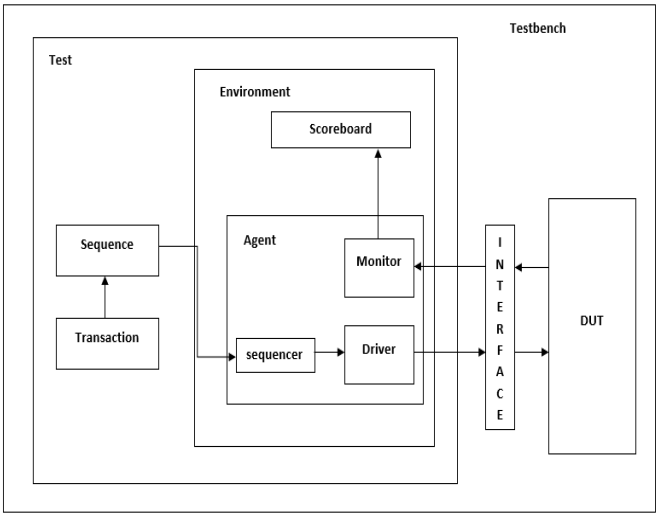


Fig -5: UVM testbench architecture

UVM testbench architecture has the following components.

Testbench: Instantiates the both unit under test and test class. And also establishes connection between them.

Test: Configures the testbench, construction of a higher level lower in the hierarchy will serve as the initial testbench parts development measure and instantiates the sequence to begin the stimulus.

Environment: Verification components like scoreboard and agent are grouped together for reusability.

Agent: Incorporates monitor, driver and sequencer as a signal entity through the TLM interface.

Sequencer: The task of the sequencer is to guide transactions (also known as sequence items) that are generated in sequence to the driver or vice versa.

Driver: Retrieves data from sequencer and sends the same to DUT and reference model through interface.

Monitor: Samples the unit being tested and reference model, records the data present in transactions, and then compares it.

Scoreboard: Consists of checkers to verify the design functionality.

Sequence: Generating and receiving the data items in sequence to and from the driver is done.

Sequence item: Acts as a placeholder for the procedure that the monitor will check on DUT signals.

Code coverage analysis gives feedback to the report file regarding the execution of statements, branches, conditions, and expressions in the source code and measures the amount of logic that is toggled during execution. Code coverage mainly includes statement coverage, condition coverage, branch coverage, expression coverage, focused expression coverage and toggled coverage.

4. RESULTS

Functional modules for the above discussed protocols are designed using SystemVerilog HDL, compiled and simulated using QuestaSim. UVM verification and Code coverage analysis are performed on the designed functional modules using QuestaSim 10.7c include with UVM 1.2.

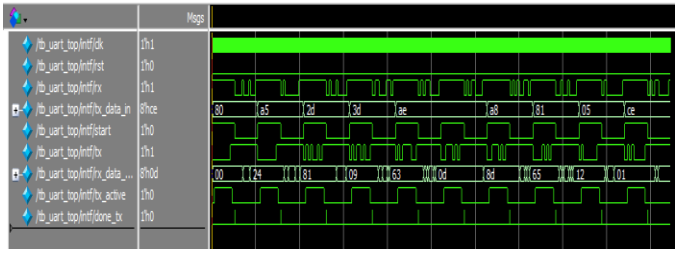


Fig -6: Simulated waveform of UART

Above fig -6 shows the simulated waveform of the UART data transaction between transmitter and receiver. Transmission of the data happens on the positive edge of the clock cycle.

```
#
# UVM_INFO uart_driver.sv(32) @ 0: uvm_test_top.env.agent.driv [] ----------------------------------------------
# UVM_INFO uart_driver.sv(33) @ 0: uvm_test_top.env.agent.driv []   Transaction No. = 0
# UVM_INFO uart_driver.sv(43) @ 261405: uvm_test_top.env.agent.driv []   start = 1,   tx_data_in = 80,  done_tx = 1
# UVM_INFO uart_driver.sv(44) @ 261405: uvm_test_top.env.agent.driv [] [TRANSACTION]::TX PASS
# UVM_INFO uart_driver.sv(63) @ 523815: uvm_test_top.env.agent.driv []   Expected data = 24,   Obtained data = 24
# UVM_INFO uart_driver.sv(66) @ 523815: uvm_test_top.env.agent.driv [] [TRANSACTION]::RX PASS
# UVM_INFO uart_driver.sv(67) @ 523815: uvm_test_top.env.agent.driv [] ----------------------------------------------
# UVM_INFO uart_driver.sv(32) @ 523815: uvm_test_top.env.agent.driv [] ----------------------------------------------
# UVM_INFO uart_driver.sv(33) @ 523815: uvm_test_top.env.agent.driv []   Transaction No. = 1
# UVM_INFO uart_driver.sv(43) @ 784225: uvm_test_top.env.agent.driv []   start = 1,   tx_data_in = a5,  done_tx = 1
# UVM_INFO uart_driver.sv(44) @ 784225: uvm_test_top.env.agent.driv [] [TRANSACTION]::TX PASS
# UVM_INFO uart_driver.sv(63) @ 1046635: uvm_test_top.env.agent.driv []   Expected data = 81,   Obtained data = 81
# UVM_INFO uart_driver.sv(66) @ 1046635: uvm_test_top.env.agent.driv [] [TRANSACTION]::RX PASS
# UVM_INFO uart_driver.sv(67) @ 1046635: uvm_test_top.env.agent.driv [] ----------------------------------------------
# UVM_INFO uart_driver.sv(32) @ 1046635: uvm_test_top.env.agent.driv [] ----------------------------------------------
# UVM_INFO uart_driver.sv(33) @ 1046635: uvm_test_top.env.agent.driv []   Transaction No. = 2
# UVM_INFO uart_driver.sv(43) @ 1307045: uvm_test_top.env.agent.driv []   start = 1,   tx_data_in = 2d,  done_tx = 1
# UVM_INFO uart_driver.sv(44) @ 1307045: uvm_test_top.env.agent.driv [] [TRANSACTION]::TX PASS
# UVM_INFO uart_driver.sv(63) @ 1569455: uvm_test_top.env.agent.driv []   Expected data = 9,   Obtained data = 9
# UVM_INFO uart_driver.sv(66) @ 1569455: uvm_test_top.env.agent.driv [] [TRANSACTION]::RX PASS
# UVM_INFO uart_driver.sv(67) @ 1569455: uvm_test_top.env.agent.driv [] ----------------------------------------------
# UVM_INFO uart_driver.sv(32) @ 1569455: uvm_test_top.env.agent.driv [] ----------------------------------------------
```

Fig -7: UVM report of UART data transaction between tx and rx

UVM report in the fig -7 gives the transaction details like transaction numbers, transaction of the data between the transmitter and receiver passed or failed. Here, we have used the 10 transactions. Once the data is transmitted successfully, will get the message [TRANSACTION]::TX PASS. After the data is received successfully, expected data and obtained data is compared in the scoreboard of the UVM. If the both the data are same, will get the message [TRANSACTION]::RX PASS. Data is chosen randomly using the concept of randomization.

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   74
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# []    71
# [RNTST]     1
# [TEST_DONE]     1
# [UVM/RELNOTES]     1
#
# ** Note: $finish    : C:/questasim64_10.7c/verilog_src/uvm-1.2/src/base/uvm_root.svh(517)
#    Time: 5229195 ps  Iteration: 78  Instance: /tb_uart_top
```

Fig -8: UVM report summary of UART

UVM report summary in fig -8 gives the details of WARNING, ERROR, FATAL, number of test case, iteration and time taken during the UVM verification.

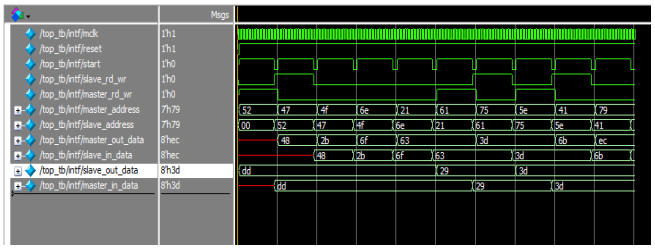


Fig -9: Simulated Waveform of SPI

Above fig - 9 shows the simulated waveform of the serial peripheral interface data exchange between the master and slave during MISO and MOSI mode of operation.

```
#
# UVM_INFO test.sv(19) @ 0: uvm_test_top [test] IN RUN PHASE OF TEST
# UVM_INFO scoreboard.sv(24) @ 0: uvm_test_top.i_env.i_scb [scoreboard] IN RUN PHASE OF SCOREBOARD
# UVM_INFO agent.sv(30) @ 0: uvm_test_top.i_env.i_agent [agent] IN RUN PHASE OF AGENT
# UVM_INFO sequencer.sv(13) @ 0: uvm_test_top.i_env.i_agent.i_sequencer [sequencer] IN RUN PHASE OF SEQUENCER
# UVM_INFO sequence1.sv(13) @ 0: uvm_test_top.i_env.i_agent.i_sequencer@@i_seq [sequence1] Executing sequence
# UVM_INFO driver.sv(22) @ 100: uvm_test_top.i_env.i_agent.i_driver [driver] IN RUN PHASE OF DRIVER
# UVM_INFO monitor.sv(27) @ 100: uvm_test_top.i_env.i_agent.i_monitor [monitor] IN RUN PHASE OF MONITOR
# UVM_INFO scoreboard.sv(31) @ 1950: uvm_test_top.i_env.i_scb [scoreboard] -----RD_WR Match-----
# UVM_INFO scoreboard.sv(32) @ 1950: uvm_test_top.i_env.i_scb [scoreboard] Master sent RD_WR:1  Slave received RD_WR:1
# UVM_INFO scoreboard.sv(40) @ 1950: uvm_test_top.i_env.i_scb [scoreboard] -----ADDRESS Match-----
# UVM_INFO scoreboard.sv(41) @ 1950: uvm_test_top.i_env.i_scb [scoreboard] Master sent ADDRESS:52  Slave received ADDRESS:52
# UVM_INFO scoreboard.sv(49) @ 1950: uvm_test_top.i_env.i_scb [scoreboard] -----MISO data Match-----
# UVM_INFO scoreboard.sv(50) @ 1950: uvm_test_top.i_env.i_scb [scoreboard] Master received MISO data:dd  Slave sent MISO data:dd
# UVM_INFO scoreboard.sv(58) @ 1950: uvm_test_top.i_env.i_scb [scoreboard] -----MOSI data Match-----
# UVM_INFO scoreboard.sv(59) @ 1950: uvm_test_top.i_env.i_scb [scoreboard] Master sent MOSI data:0  Slave received MOSI data:0
# UVM_INFO sequence1.sv(13) @ 2050: uvm_test_top.i_env.i_agent.i_sequencer@@i_seq [sequence1] Executing sequence
# UVM_INFO scoreboard.sv(31) @ 3950: uvm_test_top.i_env.i_scb [scoreboard] -----RD_WR Match-----
# UVM_INFO scoreboard.sv(32) @ 3950: uvm_test_top.i_env.i_scb [scoreboard] Master sent RD_WR:0  Slave received RD_WR:0
# UVM_INFO scoreboard.sv(40) @ 3950: uvm_test_top.i_env.i_scb [scoreboard] -----ADDRESS Match-----
# UVM_INFO scoreboard.sv(41) @ 3950: uvm_test_top.i_env.i_scb [scoreboard] Master sent ADDRESS:47  Slave received ADDRESS:47
# UVM_INFO scoreboard.sv(49) @ 3950: uvm_test_top.i_env.i_scb [scoreboard] -----MISO data Match-----
# UVM_INFO scoreboard.sv(50) @ 3950: uvm_test_top.i_env.i_scb [scoreboard] Master received MISO data:dd  Slave sent MISO data:dd
```

Fig -10: UVM report of SPI's data transaction between master and slave

UVM report in the fig -10 gives the successful data transfer between master and slave and vice-versa. Transaction details like RD_WR Match, address Match, MISO data Match and MOSI data Match are shown on successful data exchanged between the master and slave

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  100
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [RNTST]     1
# [TEST_DONE]     1
# [UVM/RELNOTES]     1
# [UVMTOP]     1
# [agent]     1
# [driver]     1
# [monitor]     1
# [scoreboard]    81
# [sequence1]    10
# [sequencer]     1
# [test]     1
#
# ** Note: $finish    : C:/questasim64_10.7c/verilog_src/uvm-1.2/src/base/uvm_root.svh(517)
#    Time: 2005 ns  Iteration: 73  Instance: /top_tb
# 1
```

Fig -11: UVM report summary of SPI

UVM report summary in fig -11 gives the details of WARNING, ERROR, FATAL, number of test case, iteration and time taken during the UVM verification.

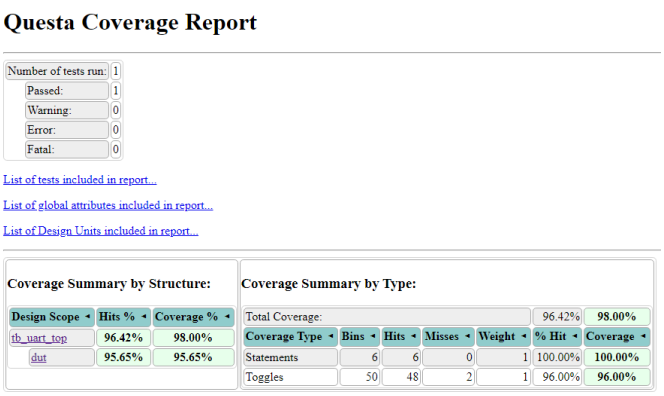


Fig -12: Code coverage report of UART

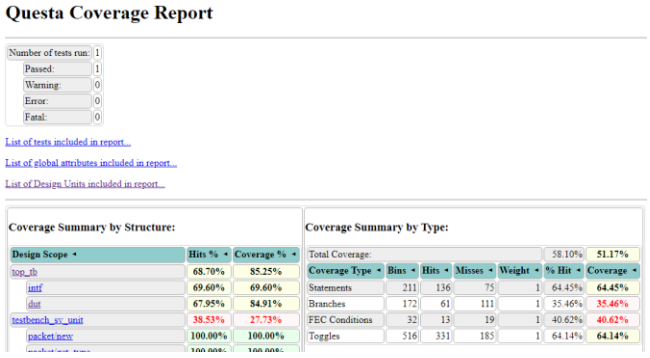


Fig -13: Code coverage report of SPI

Fig -12 and fig -13 shows the code coverage report of UART and SPI functional modules.

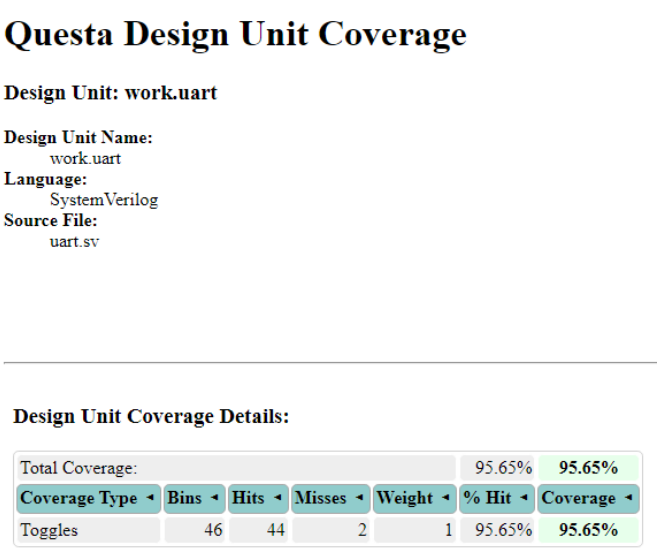


Fig -14: Code coverage details of UART functional module

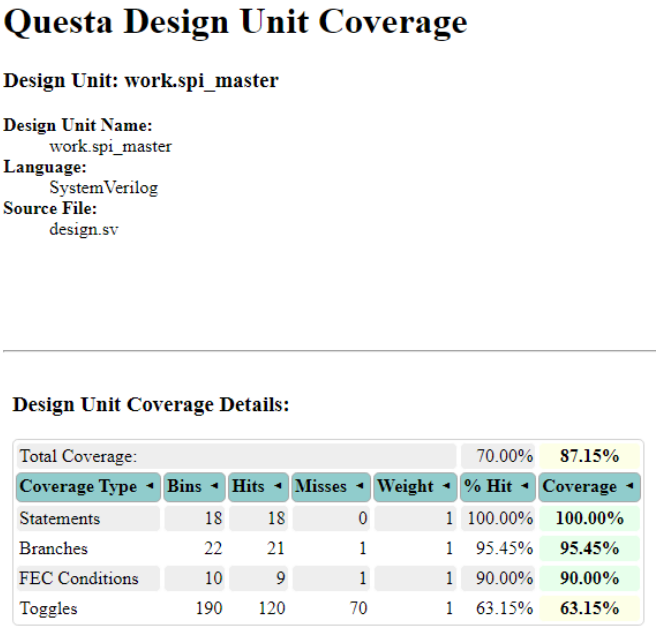


Fig –15: Code coverage of SPI master

Fig -15 shows the code coverage details of SPI master functional module.

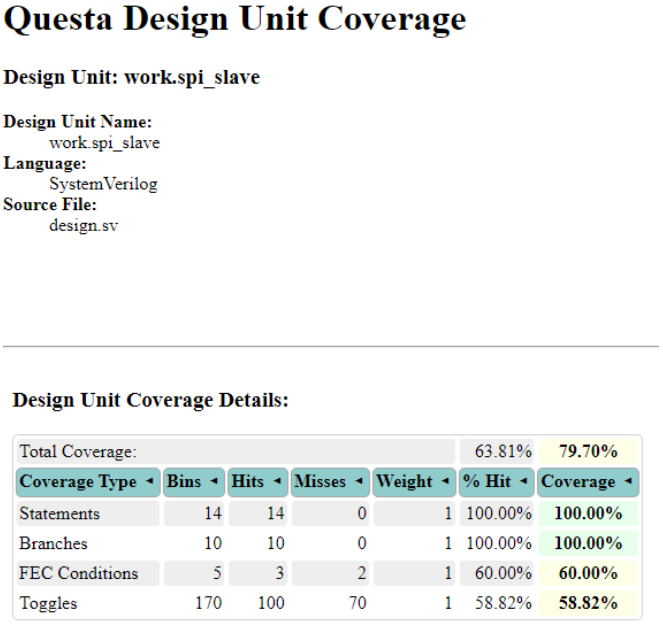
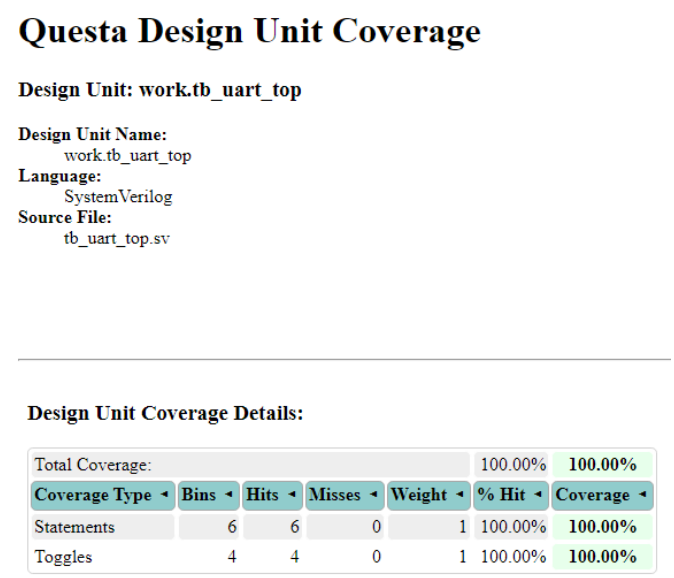


Fig -16: Code coverage of SPI Slave

Fig -16 shows the code coverage details of SPI slave functinal module.



Questa Design Unit Coverage

Design Unit: work.tb_uart_top

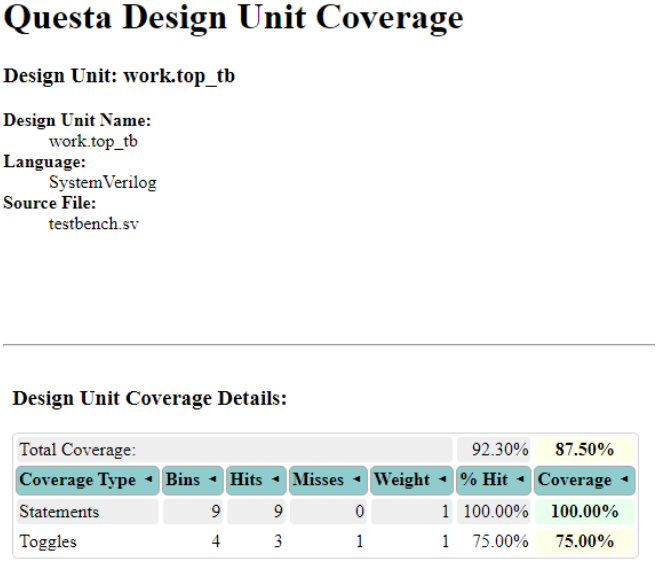
Design Unit Name:
work.tb_uart_top
Language:
SystemVerilog
Source File:
tb_uart_top.sv

Design Unit Coverage Details:

Total Coverage:					100.00%	100.00%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	6	6	0	1	100.00%	100.00%
Toggles	4	4	0	1	100.00%	100.00%

Fig -17: Code coverage of UART testbench

Fig -17 shows the code coverage details of UART testbench



Questa Design Unit Coverage

Design Unit: work.top_tb

Design Unit Name:
work.top_tb
Language:
SystemVerilog
Source File:
testbench.sv

Design Unit Coverage Details:

Total Coverage:					92.30%	87.50%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	9	9	0	1	100.00%	100.00%
Toggles	4	3	1	1	75.00%	75.00%

Fig -18: Code coverage of SPI testbench

Fig -18 shows the code coverage details of the SPI testbench

5. CONCLUSIONS

This paper contains the brief discussion of the UART and SPI protocols. The work includes the design of the functional modules of UART and SPI using SystemVerilog HDL. UVM testbench environment is built for the designed modules. Functional modules are simulated using QuestaSim 10.7c. UVM verification and code coverage analysis of designed modules are performed using the QuestaSim 10.7c pre-compiled with UVM 1.2.

REFERENCES

[1] Priyanka B., Gokul M.R., Nigitha A., & Poomica., J. (2021). "Design of UART Using Verilog And Verifying Using UVM". 2021 7th International Conference on Advanced Computing and Communication Systems (ICACCS), 1, 1270-1273.

[2] B. Vineeth; B. Bala Tripura Sundari, "UVM Based Testbench Architecture for Coverage Driven Functional Verification of SPI Protocol", 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), 2018 sep19

[3] Ashok Kumar Gupta, Ashish Raman, Naveen Kumar, and Ravi Ranjan "Design and Implementation of High-Speed Universal Asynchronous Receiver and Transmitter (UART) " 2020 7th International Conference on Signal Processing and Integrated Networks (SPIN) IEEE 20 April 2020

[4] Kashyap B, Ravi V. "Universal Verification Methodology Based Verification of UART Protocol" InJournal of Physics: Conference Series 2020 Dec 1 (Vol. 1716, No. 1, p. 012040). IOP Publishing.

[5] Pallavi Polsani, V. Priyanka B., Y. Padma Sai, "Design and Verification of Serial Peripheral Interface (SPI) Protocol" International Journal of Recent Technology and Engineering (IJRTE) ISSN: 2277-3878 (Online), Volume-8 Issue-6, March 2020

[6] Dr.Punith Kumar M B, Sreekantesha H N, "Design and Verification of SPI Core Using UVM" Journal of Emerging Technologies and Innovative Research (JETIR) JETIR May 2019, Volume 6, Issue 5

[7] Kulkarni A, Sakthivel SM. "UVM methodology based functional Verification of SPI Protocol" In Journal of Physics: Conference Series2020 Dec 1 (Vol. 1716, No. 1, p. 012035). IOP Publishing

[8] Ni W, Wang X. "Functional coverage-driven UVM-based UART IP verification" In2015 IEEE 11th International Conference on ASIC (ASICON) 2015 Nov 3 (pp. 1-4). IEEE.

[9] Yamini R, & Ramya M V. (2020). "Design and Verification of UART using System Verilog" International Journal of Engineering and Advanced Technology (IJEAT), 9(5), 1208–1211

[10] Roopesh, D Siddesha K, Kavitha Narayan B M "RTL DESIGN AND VERIFICATION OF SPI MASTER-SLAVE USING UVM" International Journal of Advanced Research in Electronics and Communication Engineering Volume 4, Issue 8, August 2015.

[11] Y. Fang and X. Chen, "Design and Simulation of UART Serial Communication Module Based on VHDL", 2011 3rd International Workshop on Intelligent Systems and Applications, pp. 1-4, 2011.