JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

IP VERIFICATION OF SPI CONTROLLER FOR OPEN POWER PROCESSOR CORE BASED FABLESS SYSTEM ON CHIP (SoC)

[1]Mr.Bukkasam MadhuSudhan, [2]Mr.B.C.Vengamuni, [3]Dr.Gannera Mamatha
[1]MTech Scholar, [2]Assistant Professor (Adhoc), [3]Assistant Professor
[1, 2, 3] MTech in VLSI-SD,
[1, 2, 3] JNTUA College of Engineering Ananthapuramu, Ananthapuramu, India

Abstract: The Serial-Peripheral Interface (SPI) protocol is one of the most widely used bus protocols for connecting processors to peripheral devices with low/medium data transmission speeds. SPI architecture is used to communicate between multiple peripherals and the processor in a SoC application. The slave is subject to the master's power. The slave is represented by a sensor, monitor, or memory chip. A reusable logic or functionality unit, cell, or layout design that can be used in numerous chip designs is referred to as an intellectual property (IP) in the context of VLSI. These IPs are typically created with the intention of licensing them to other vendors. This IP verification of the SPI controller is done by writing test benches in System Verilog and UVM. This paper's aim is to verify Intellectual Property (IP) blocks and driver development of SPI controller for Open-Power processor A2O core-based fabless SoC connected through AXI4 interface. The methodology used for verifying is to develop Test benches in System Verilog and use them for Verification by using software like ModelSim Questa® and Vivado design suite-Xilinx®.

IndexTerms **- System Verilog, UVM, Testbench, SoC, SPI, AXI4.**

I. INTRODUCTION

Verification is the critical stage in the creation of a design. Nearly 80% of time in the design cycle is spent on verification. Technology requires a rapid and trustworthy verification mechanism in order to narrow the gap between supply and product demand. We are forced to create bigger, more capable, and more sophisticated designs by technological demands. High complexity designs are more prone to errors. Traditional verification techniques do not work well with them. The most common methodology for verifying intricate VLSI designs is UVM. UVM uses automation mechanisms including the production of random stimuli and Data and automation aspects like read, write, compare and copy are addressed by transaction-level modelling (TLM). UVM is an Accellera standard and includes numerous tool support, in contrast to other HDL languages. As AXI4 is a master and SPI is a slave, the development of a test bench for the ARM Advanced Microcontroller Bus Architecture (AMBA) AXI4 bus to SPI controller is to verify transactions between them. The authentication of write/read activities over the Bridge is justified by UVM verification. Verifying bridge transitions with UVM is a crucial goal, and testbench acts as the master for the AXI4 interface, which provides the required input signals. As a result, SPI interface performs as a slave, created to give appropriate signals to DUT, Which uses SPI along with AXI master. Accordingly, AXI interface controls the AXI master's reaction. The addition of a self-checking mechanism in the testbench was driven by the assertions at the interface for the integration of reusable environment into the tolerance detection approach. When a necessary condition (or conditions) is (are) broken, assertions identify errors as well as run-time fatal errors. The use of two different interfaces in place of one, especially for bridge nodes with independent clock mechanisms, allows synchronization to absorb the unique qualities of the bridge more quickly. The provided testbench supports reusable environment and works with all bridge transitions.

II. GOALS OF THE EXPERIMENT

Following are main goals of the experiment:
1) Creation of an environment for bridge protocol verification Included is a comprehensive Design under Test (DUT) scenarios assessment with the necessary set of cover points.
2) executing a large number of test cases just on testbench while using a self-checking approach to guarantee that UVM is running without any fatal errors, which makes debugging easier.
3) The use of 2 agents to track data movement from SPI to AXI4 Bridge in the reusable environment.
4) Complete functionality coverage to ensure that the testbench receives functional coverage. Coverage metrics go into detail about the design elements that are active during simulation.

5) Carrying out the necessary required assertions for assertion based verification (ABV). The objective is to stop any significant bugs from surviving through the SOC or ASIC development stage.
6) Creating a test bench to generate appropriate random stimuli for the detection of design faults that are currently active.

III. UVM TESTBENCH ARCHITECTURE

A. UVM Environment

When a DUV is get replaced, the UVM environment, also known as the Universal Verification Component (UVC), is designed to be reused. As the UVC unifies Agents, Driver, Sequencer, Monitor, Coverage checkers and Configuration, and independently for SPI and AXI environments that are compatible with the UVM semantics and base class. Fig. 1: The test bench architecture [1].

B. Protocol Universal Verification Components

The UVC for the proposed architecture of the AXI4 to SPI Controller are as follows:

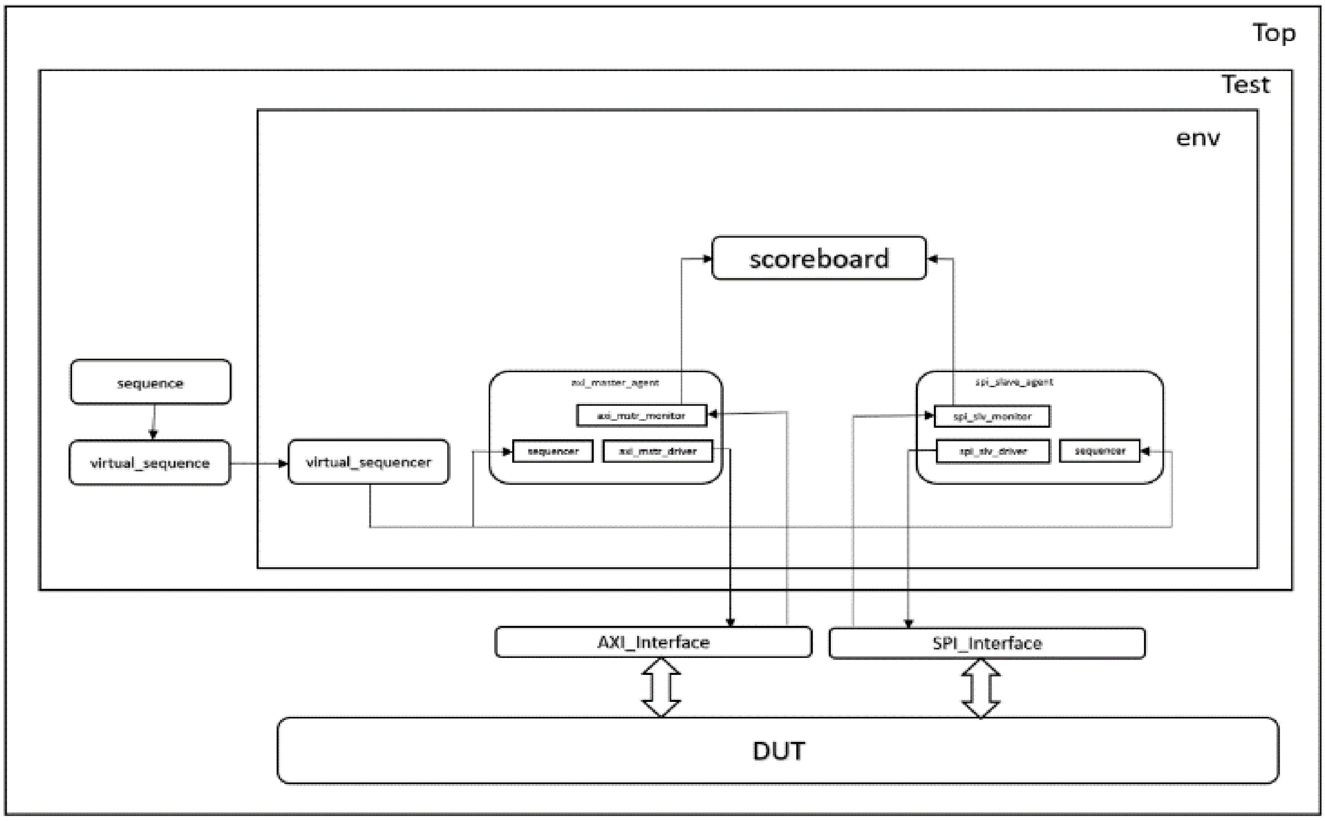


Fig. 1. Proposed Architecture of UVM testbench

1) SPI Sequencer: To deliver the randomized sequence item to the driver for the next transition of data using Transition level modelling, the sequencer is parameterized per sequence item (TLM). SPI slave receives input from the output of the AXI master signal while the sequencer creates random transactions for the slave SPI DUT [1].
2) AXI Sequencer: To deliver the randomized sequence items to the driver accompanying TLM, the AXI_sequencer is parameterized by seq item. AXI master receives input from SPI slave's output, and an AXI sequencer creates random transactions for AXI master and DUT.
3) Driver: To drive pin wiggles of the DUT via the interface, the driver transforms transition level data to pin level data. Drivers behave differently for SPI and AXI in each scenario. The bridge's SPI driver involves the SPI.
4) Monitor: By re-converting the abstract data through the analysis port and reporting the abstract transition, the monitor puts the abstract data back together. The conversion of pin level data to transaction level is carried out by the monitor as it monitors the data coming from the interface.
5) Agent: The term "agent" refers to a Combination of sequencer, monitor and driver.
6) Interface: Interface acts as a conduit between the environment and the DUT and includes logical requirements such as self-checking mechanisms. For interaction, it may additionally include bus functional models (BFM). There are two distinct interfaces for SPI and AXI in the suggested architecture, which is depicted in Fig. 1. Both interfaces function separately.
7) Coverage checker: For sampling and modification of cover groups meant for protocol function coverage, the coverage checker serves as an optional object. It is a component of the UVM environment not a part of the agent.

C. Coverage Metrics

Functional coverage matrices are used by the DUT during constraint random verification to determine whether the functionality and requirements have been met in accordance with the test plan or not. Additionally, it completes by thoroughly examining the design model's code using metrics for code coverage. The terms for controllability and observability are also covered. Controllability

is the ability to alter a design's code, functional model, or design structure by creating different random stimuli or virtual sequences via input pins.
In contrast, observability is the ability to track how designs, codes, or model structures affect output.

D. UVM sequence Arbitration

One sequence is easy to handle, but when numerous sequences are occurring at once, arbitration is necessary to stop a race condition known as Sequence arbitration. In this study, we employ the uvm sequence arb FIFO default arbitration technique.

E. Virtual sequence

It's essential to employ virtual sequence when using 2 interfaces and 2 agents. The top-level sequence for DUT configurations, which is the virtual sequence, correlates with the intermediate-level sub-sequences. Low-level or rudimentary sequences are handled by sub-sequences. The creation of transactions that are transmitted to the driver and the gathering of monitor responses are done by these low-level sequences. Virtual sequencer, which is an extension of uvm sequencer, passes virtual sequence. The agent's handle is transferred to the virtual sequencer in accordance with the rules.

IV. PHASES OF UVM

The verification methodology used is UVM. UVM introduces phases to establish the necessary functional routes and consistency in flow. All UVM modules currently in use perform these stages in the same hierarchical, top-to-bottom order. As seen in Fig. 2, the categorization provides three phases for UVM operation.

A. Build phase:

The testbench is connected and configured during this phase of construction with the intention of using a specific connection to execute the directional verification process. Build phases are of the three sub divided phases of build phase, connect phase, and end of elaboration phase.

B. Runtime phase:

The intermediate stage during which random stimulus creation and simulation occur. There are 13 sub-stages in all in the runtime phases.

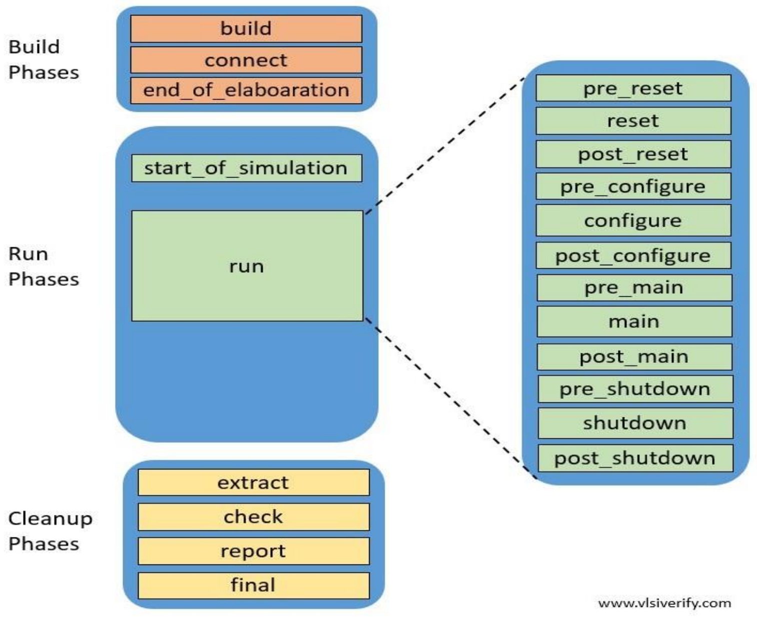


Fig. 2. Classification of UVM phases

C. Clean up phase:

The final phase of the UVM phasing mechanism is in charge of gathering and reporting the effects of the various test cases' results. The cleanup phase is divided into four sections: extract phase, check phase, report phase, and final phase.

V. WORKING OF TESTBENCH

According to Fig 3 and 4, the Testbench is divided into two top modules, HVL and HDL top. The main point of employing two top modules is to move the interface and RTL into one top module and the synthesizable portion of the testbench into the other. The unsynthesizable portion is moved into the HVL TOP, and the structure is given the label HDL TOP. Because it makes it possible to conduct lengthier tests rapidly. Depending on the mode of operation, this specific testbed can be utilized for both simulation and emulation. The transactions flow from the master virtual sequence and slave virtual sequence onto the axi4 and SPI I/F through the BFM Proxy and BFM, and the HVL TOP design is untimed. It also receives data from the monitor BFM and uses it to do checks using the scoreboard and coverage.

The design portion of HDL TOP is timed and synthesizable, and it also produces the clock and reset signals. In HDL TOP, there are Bus Functional Models (BFMs), which are synthesizable components of the drivers and monitors. BFMs also contain back pointers to their proxies, allowing them to call non-blocking methods that are specified in the proxies.

Tasks and operations that the driver and monitor proxy in the HVL calls within the drivers and monitors. Data is sent between the HDL TOP and HVL TOP in this way.

Since the clock is produced by the HDL TOP inside the emulator, HDL and HVL use transaction-based communication to provide information-rich transactions, which enables the emulator to operate at full speed.

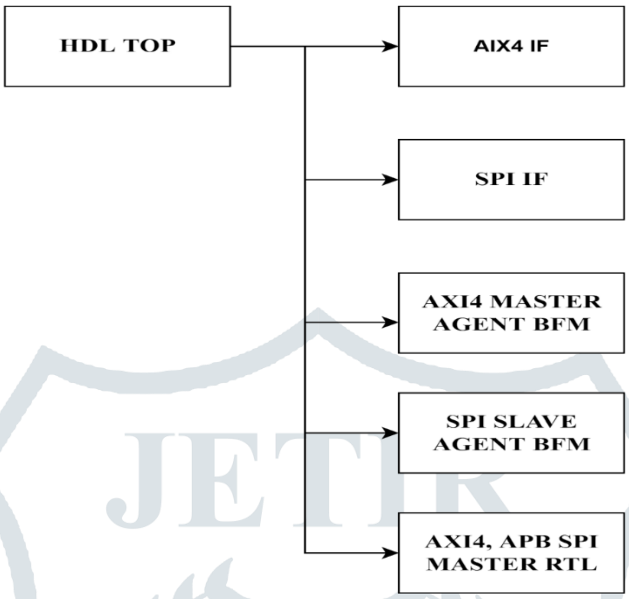


Fig. 3. HDL Top

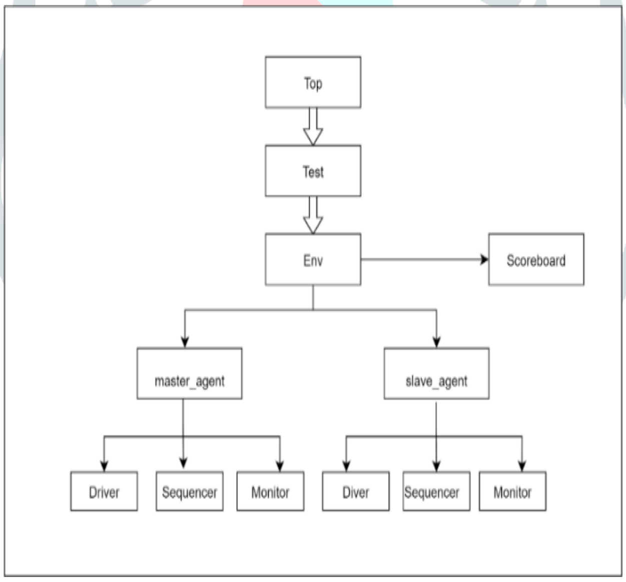


Fig. 4. HVL Top

The run test ("test name") method in top is used to start the entire tb_component set when a test is run.

VI. RESULTS AND DISCUSSIONS

A. Debugging tips:

As design complexity keeps rising, new problems with verification and debugging are emerging. Fortunately, fresh approaches and techniques (like UVM) have been developed to deal with rising design complexity. Although UVM adoption has the potential to increase productivity, there are more recent debugging issues that are unique to UVM that need to be resolved.

Here basic_write_read_reg_test was used as an example test case to demonstrate the AXI4 protocol's debugging flow below, and UVM HIGH verbosity was utilized to run all of the information.

B. Debug Flow:

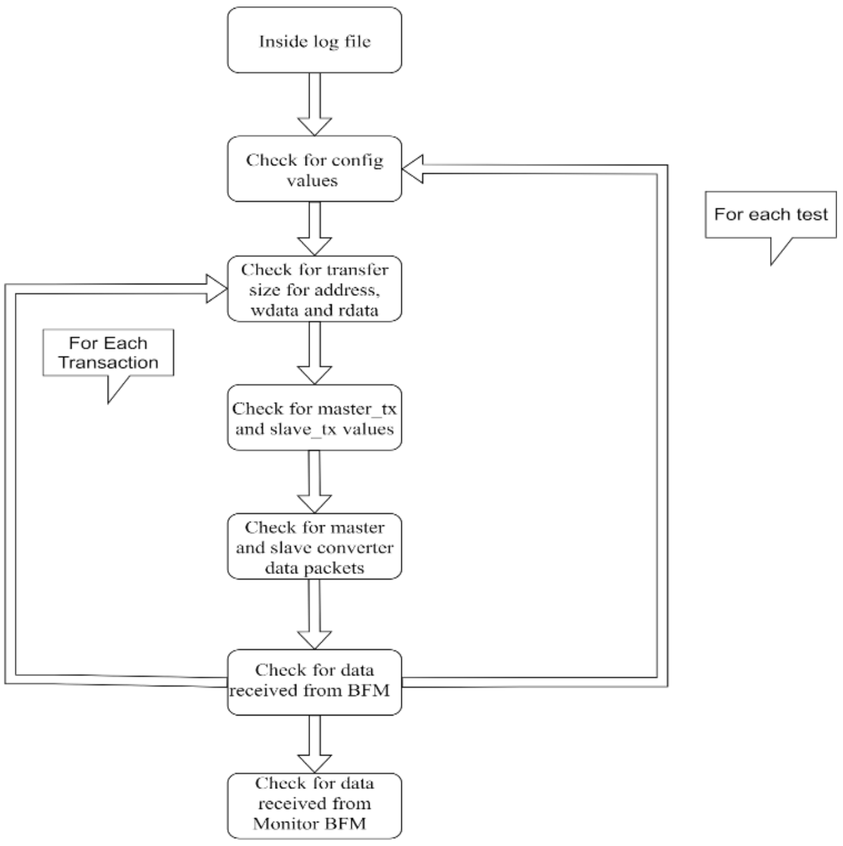

Inside log file
Check for config values
Check for transfer size for address, wdata and rdata
Check for master_tx and slave_tx values
Check for master and slave converter data packets
Check for data received from BFM
Check for data received from Monitor BFM
For each test
For Each Transaction

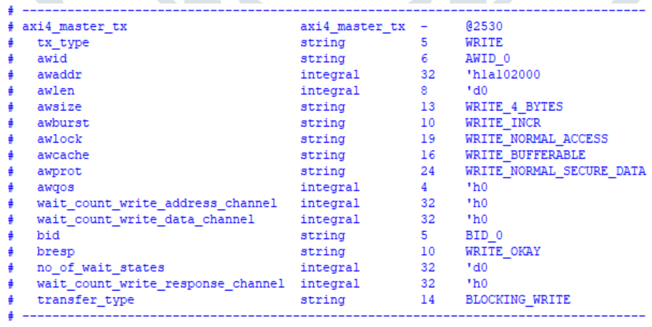
Fig. 5. Debug Flow

To get the debug flow shown in fig. 5, first access the log file that is located in the test folder that has been run. Then, follow the steps following.

C. Transaction values:

1. Master Transaction values:

Once the configuration variables and transfer size are accurate, verify whether the data will be communicated by the master tx class or the slave tx class.

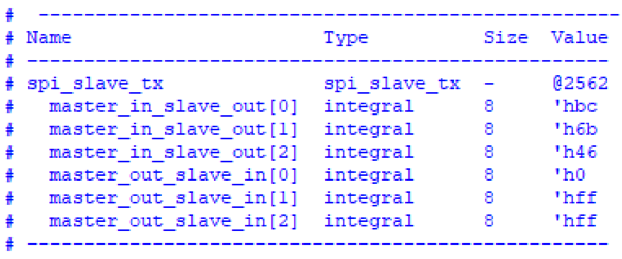


```
# ---------------------------------------------------------------------------------
# axi4_master_tx                         axi4_master_tx  -     @2530
#   tx_type                              string          5     WRITE
#   awid                                 string          6     AWID_0
#   awaddr                               integral        32    'h1a102000
#   awlen                                integral        8     'd0
#   awsize                               string          13    WRITE_4_BYTES
#   awburst                              string          10    WRITE_INCR
#   awlock                               string          19    WRITE_NORMAL_ACCESS
#   awcache                              string          16    WRITE_BUFFERABLE
#   awprot                               string          24    WRITE_NORMAL_SECURE_DATA
#   awqos                                integral        4     'h0
#   wait_count_write_address_channel     integral        32    'h0
#   wait_count_write_data_channel        integral        32    'h0
#   bid                                  string          5     BID_0
#   bresp                                string          10    WRITE_OKAY
#   no_of_wait_states                    integral        32    'd0
#   wait_count_write_response_channel    integral        32    'h0
#   transfer_type                        string          14    BLOCKING_WRITE
# ---------------------------------------------------------------------------------
```

Fig. 6. Master tx values

First, look to see if the transaction is idle. (Ex: pselx here equals 0 and penable is 1) When the pselx becomes high after the present is high based on the pclk edge, the data is sampled on the same clock edge.

2. Slave Transaction values:



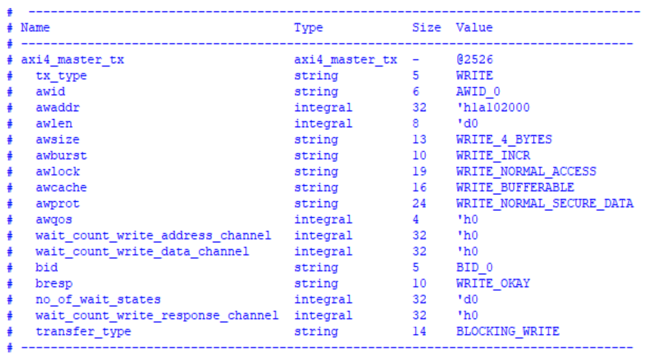
```
#  ---------------------------------------------------
# Name                      Type          Size  Value
# ---------------------------------------------------
# spi_slave_tx              spi_slave_tx  -     @2562
#   master_in_slave_out[0]  integral      8     'hbc
#   master_in_slave_out[1]  integral      8     'h6b
#   master_in_slave_out[2]  integral      8     'h46
#   master_out_slave_in[0]  integral      8     'h0
#   master_out_slave_in[1]  integral      8     'hff
#   master_out_slave_in[2]  integral      8     'hff
# ---------------------------------------------------
```

Fig. 7. Slave tx values

Data is transferred to master or slave BFMs after randomization. Depending on the configurations of the master, the master driver BFM will drive the awaddr, awvalid, awready, awdata signals and sample the awaddr, awvalid, awready, awdata; likewise, the slave driver BFM will drive the awaddr, awvalid, awready, awdata signal and sample the awaddr, awvalid, awdata.

D. Monitor values:

1. Master Monitor values:



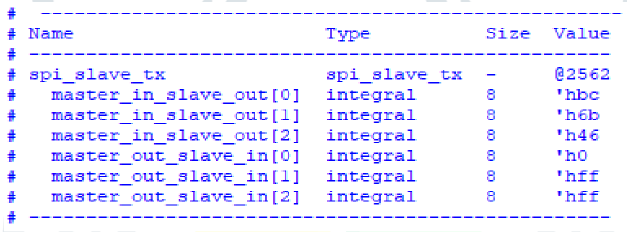
```
#  ---------------------------------------------------------------------------------
# Name                               Type           Size  Value
# --------------------------------------------------------------------------------
# axi4_master_tx                     axi4_master_tx  -     @2526
#   tx_type                          string         5     WRITE
#   awid                             string         6     AWID_0
#   awaddr                           integral       32    'h1a102000
#   awlen                            integral       8     'd0
#   awsize                           string         13    WRITE_4_BYTES
#   awburst                          string         10    WRITE_INCR
#   awlock                           string         19    WRITE_NORMAL_ACCESS
#   awcache                          string         16    WRITE_BUFFERABLE
#   awprot                           string         24    WRITE_NORMAL_SECURE_DATA
#   awqos                            integral       4     'h0
#   wait_count_write_address_channel integral       32    'h0
#   wait_count_write_data_channel    integral       32    'h0
#   bid                              string         5     BID_0
#   bresp                            string         10    WRITE_OKAY
#   no_of_wait_states                integral       32    'd0
#   wait_count_write_response_channel integral      32    'h0
#   transfer_type                    string         14    BLOCKING_WRITE
# --------------------------------------------------------------------------------
```

Fig. 8. Master Monitor values

The master driver BFM will print all of the signals that the master has driven, as well as sampled data. Both the master and the slave driver BFM will print every signal that the slave has driven as well as any sampled data.

The final data for both the master and slave BFMs must be identical. The monitor will capture the data once it has been driven or sampled, and it will publish the driven and sampled data in the request form or at the transaction level.

2. Slave Monitor values:

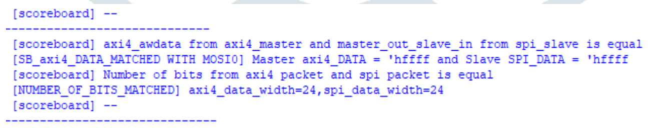


```
#  ----------------------------------------------------------
# Name                       Type          Size  Value
# ----------------------------------------------------------
# spi_slave_tx               spi_slave_tx  -     @2562
#   master_in_slave_out[0]   integral      8     'hbc
#   master_in_slave_out[1]   integral      8     'h6b
#   master_in_slave_out[2]   integral      8     'h46
#   master_out_slave_in[0]   integral      8     'h0
#   master_out_slave_in[1]   integral      8     'hff
#   master_out_slave_in[2]   integral      8     'hff
# ----------------------------------------------------------
```

Fig. 9. Slave Monitor values

E. Scoreboard Checks:

Finally, scoreboard checks are used to compare the awaddr and data width data between the master and slave sides.



```
 [scoreboard] --
----------------------------
 [scoreboard] axi4_awdata from axi4_master and master_out_slave_in from spi_slave is equal
 [SB_axi4_DATA_MATCHED WITH MOSI0] Master axi4_DATA = 'hffff and Slave SPI_DATA = 'hffff
 [scoreboard] Number of bits from axi4 packet and spi packet is equal
 [NUMBER_OF_BITS_MATCHED] axi4_data_width=24,spi_data_width=24
 [scoreboard] --
-----------------------------
```

Fig. 10. Scoreboard values

F. Waveforms Obtained:

The data is transferred from AXI4 master bus to SPI slave controller and the writing in spi slave done through WDATA signal showed in fig 6.

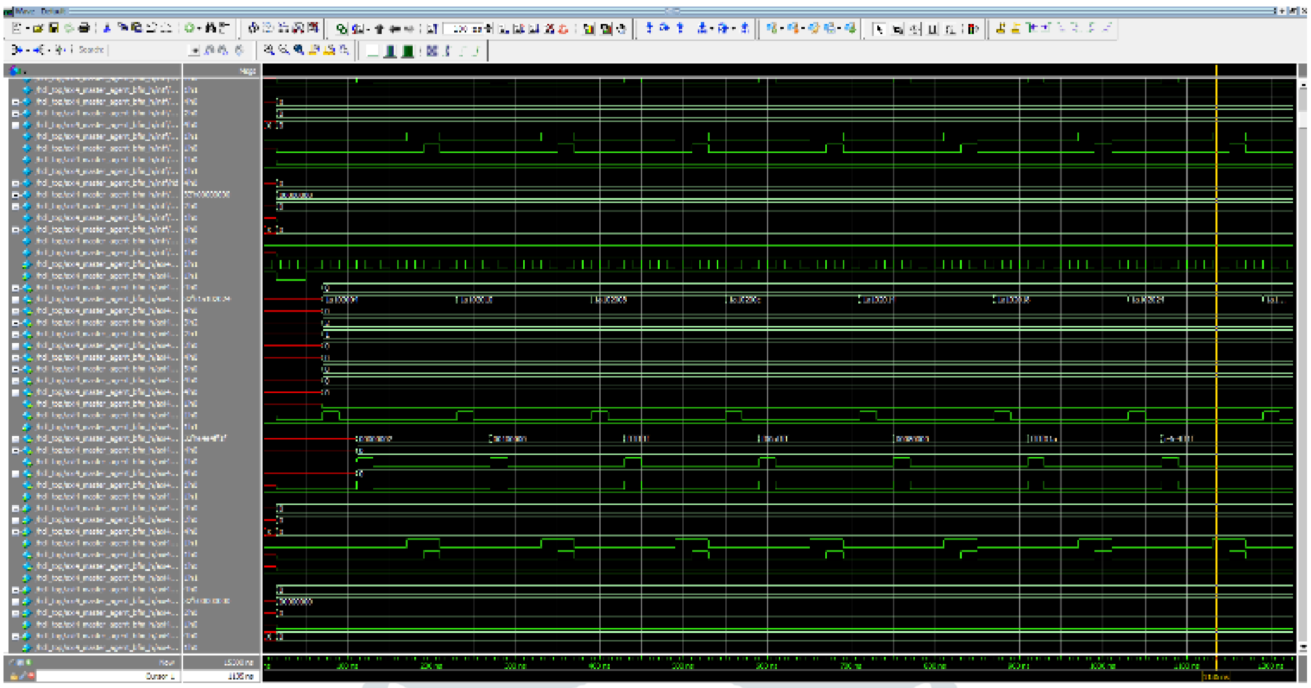


Fig. 11. Wave 1

Once the WDATA is high WREADY and WVALID signals goes high. After a successful WVALID signal we can say data is successfully transferred showed in fig 7.

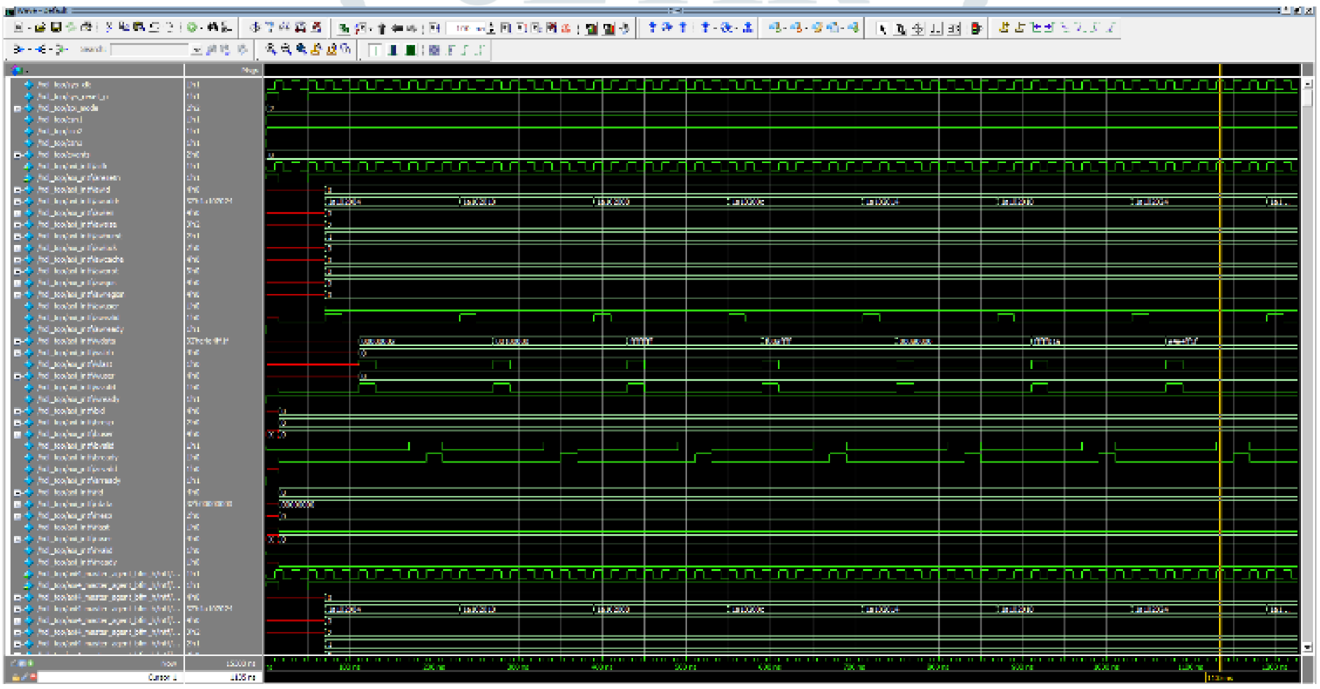


Fig. 12. Wave 2

VII. CONCLUSION AND FUTURE SCOPE

This work developed, simulated and verified of AXI4 SPI Controller whichutilized A2O processor based fabless SoC in SV and UVM using Questasim. The controller achieves the operational features. Using AXI4, this suggested design could enhance data transfer efficiency. In Xilinx Vivado, the design is developed and interfaced with A20, while Mentor Questa is used for simulation. This controller support high speed and short distance communication devices withthe help of the clocking schemes. The design can be extended to QSPI and DSPI for the double data rate and quad data rate forsingle clock to get the operations faster.

REFERENCES

[1] Gaurav Sharma, Lava Bhargava, Vinod Kumar “Self-Assertive Generic UVM Testbench for advanced verification of bridge IP’s”, IEEE, 2017.
[2] Martin Barnasconi, Manfred Dietrich, Karsten Einwich, and Thilo Vortler” UVM-SystemC-AMS Framework for System-Level Verification and Validation of Automotive Use Cases” IEEE Design & Test, 2015.
[3] J Francesconi. JA Rodriguez, M P Julian” UVM Based Test bench Architecture for Unit Verification,” 2014 Argentina Conference on MicroNanoelectronics Technology and Applications (EAMTA), July 2014.
[4] H Y Yang. ”Highly automated and efficient simulation environment with UVM,” IEEE VLSI Design, Automation and Test (VLSI-DAT), 2014.
[5] Gabe Moretti. ”Accelleras Support for ESL Verification and Stimulus Reused”, IEEE Design & Test 2016.