Journal of Harbin Engineering University
ISSN: 1006-7043

Vol 44 No. 8
August 2023

Design and Verification of AMBA Advanced Extensible Interface Memory Using Assertion Based Formal Checks

Dr Sujatha Hiremath, *Dept. of Electronics and Communication Engineering, RV College of Engineering,* Bengaluru, India

Arjumanth Farraj, *Dept. of Electronics and Communication Engineering, RV College of Engineering,* Bengaluru, India

Abstract—One of the hardest aspects of chip design is frequently functional verification. A given model's correct adherence to the specification is checked through functional verification. Additionally, it ensures that the implementation follows the specification. Functional verification has emerged as one of the major design process bottlenecks as a result of the quick increase in design size and complexity. The Advanced eXtensible Interface is designed as a collection of interdependent systems, each of which has a limited number of configurations or states. Finite State Machines (FSMs) consist of states and their transitions. The primary goal of the FSM is to check the flow of transactions in Network on Chip (NoC)s, where each transaction entering and exiting a subsystem must be secure, reliable, and pass through all of the state machine's different stages. The assertion-based formal checks for the design were supported by state machines for read and write situations. Assertions are used to verify design principles or requirements and produce errors or warnings when they fail. The AMBA AXI protocol is designed using SystemVerilog, and formal checks in the form of System Verilog assertions are used to examine the internal signals that provide read-write transactions of the memory that serves as the slave for the network on chip. The ability of the memory to react to different signals provided by the master is examined using the various burst transactions in terms of Fixed, Increment (INCR), and Wrap transactions. In this work, an AXI memory module is designed and verified using UVM and simulated using both vaivado and questasim.

Keywords—Finite State Machine, Advanced Extensible Interferface, Universal Verification Methodology, Network on Chip.

I. INTRODUCTION

These days, designs are becoming more intricate and employ several Intellectual Property (IP) cores. Advanced Extensible Interface (AXI) interconnection IP is one of these IPs and is one of the most used interconnect IPs. Reusable components are needed when systems grow bigger and faster in order to reduce the complexity and length of the design verification process. Specialized verification approaches, like the Universal Verification Methodology (UVM), are used to speed up the verification process. These approaches employ a certain coding style that allows the code to be used in any test environment based on the same methodology. The verification includes the write-address channel, write-data channel, write-response channel, read-address, and read-data—includes the verification of the memory transactions of advanced extensible interface. The test bench is designed to validate how an advanced extensible interface protocol's memory transactions work, including how data is moved between locations when reading and writing at the same site and at other locations.

II. ADVANCED MICRONCONTROLLER BUS ARCHITECTURE

The Arm-developed AMBA open standard for SoC design enables high-performance, modular, and reusable designs that function correctly the first time while consuming the least amount of silicon and power possible. The AXI protocol was developed to address the interface needs for a wide range of components while providing for flexibility in how those components are coupled. It was first built for high-frequency systems. AHB and APB are still backwards compatible with AXI, making it suitable for high-frequency, low-latency designs. AMBA is heavily utilized by a wide range of ASIC and SoC components. Networking SoCs, smartphones, and Internet of Things (IoT) subsystems are some of the devices that utilize these parts. Advanced verification approaches, such as UVM (Universal Verification

Methodology), is employed to increase the verification speed.These approaches employ certain methods of coding that enable the reuse of the source code in any verification environment that is based on the exact same approach.

Five legally distinct transfer streams are available on the AXI. The read address, the read data transaction (R), the write-address, the write-data transaction (W), and response (B) are all utilized in data transactions.

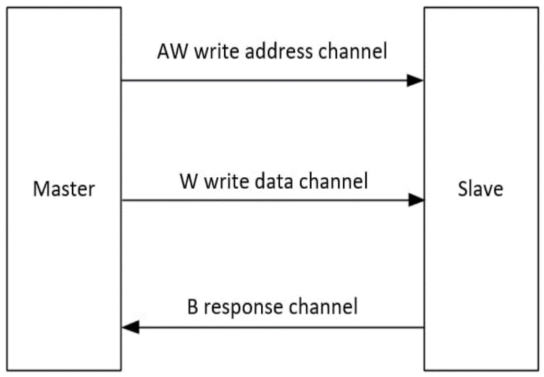
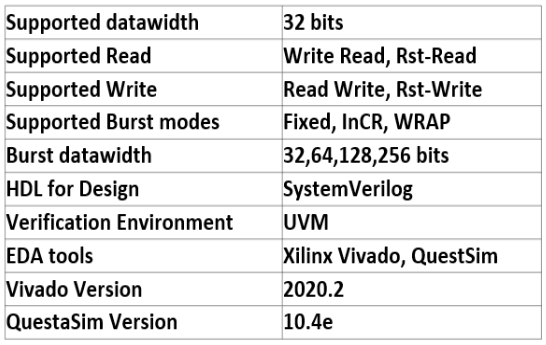


Figure 1 AXI BUS

III. The design specifications



Supported datawidth	**32 bits**
Supported Read	**Write Read, Rst-Read**
Supported Write	**Read Write, Rst-Write**
Supported Burst modes	**Fixed, InCR, WRAP**
Burst datawidth	**32,64,128,256 bits**
HDL for Design	**SystemVerilog**
Verification Environment	**UVM**
EDA tools	**Xilinx Vivado, QuestSim**
Vivado Version	**2020.2**
QuestaSim Version	**10.4e**

Figure 2: Design Specifications.

The Design specs consists of the supported bandwidth of 32bits, slave memory capacity of 2KB, supported read are Write Read and Reset Read, Supported Write are Read Write and Reset write, supported burst modes are Fixed, INCR and WRAP. Supported burst data width are 32, 64, 128, 256 bits. System Verilog is used as the preferred HDL and the EDA tools used are Xilinx Vivado and VCS Verdi.

A. *Verification Plan*

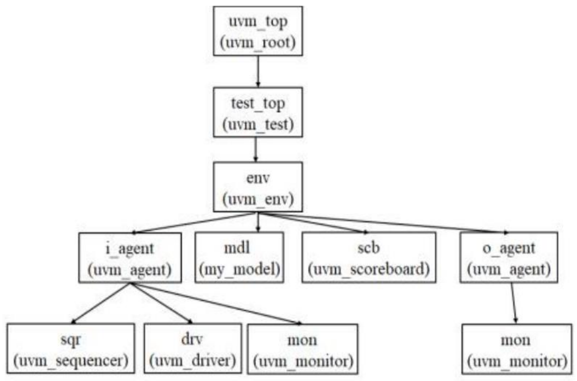


Figure 3 UVM Tree

The plan aids the verification engineer in comprehending how the verification should be performed. Introduction, presumptions,The plan includes an approach, tasks, resources, risks and timeframes, a list of test cases, a list of features to be tested, a list of entrance and exit criteria, as well as entry and exit criteria. Plan could be a document, a spreadsheet, or a plain text file. Additionally, it includes an explanation of the Testbench's architecture as well as a breakdown of each component's features.

IV. The UVM testbench Arhcitecture

The testbench was made utilizing the analysis and verification components that were taken from the UVM libraries. The test bench operates the DUV and produces random vectors. To check for DUV functionality, both the applied stimulus and the DUV reaction are tracked. The testbench additionally reports functional coverage using the coverage method as an indication for how far along the verification process is.

A. *UVM Package*

The libraries that are part of the UVM bundle can be used to build the layered UVM testbench framework and other verification components. There are three different classes:

1. object: It serves as the root class for all UVM data and hierarchical classes, defining methods for operations like create, copy, compare, and print that are performed often on all objects.
2. Component: All of UVM's significant verification components use it as their root

class. It offers interface to hierarchy, phasing, reporting, and the UVM factory and inherits every feature of a UVM object.

3. Transaction: All transactions used by UVM to generate and analyze stimuli use it as their base class. It offers a timing and recording interface and takes on all the characteristics of the UVM object.

B. Hierarchy

Numerous verification and analysis components make up the testbench's framework [5]. The functions of these items are described below

1. Sequence Item: It is the random stimulus that will be applied to the DUV's control layer, content, structure, and analysis data aspects. Motivated random testing makes use of the characteristics of the source data, which are classified as random factors.
2. Sequence: The process develops the sequence elements and randomizes them before transmitting them to the driver.
3. Sequencer: The sequence is executed by the sequencer, who also sends the order of items to the driver from the sequence where they are formed.
4. Driver: The driver transforms circuit state events from raw data fields of sequence items to control the design using interface.
5. Monitor: It collects data on DUT action at the individual pin level, transforms it into transfer level events using the TLM analysis ports, and sends those transactions through the scoreboard.
6. Agent: The agent is made up of the sequencer, driver, and monitor, whether they spawn all the components and excite the DUT to record the DUT responses, agents may be classified as passive or active.
7. Scoreboard: The transactions are compared to see if the DUT response matches the expected behavior. It determines whether a test scenario is successful or unsuccessful.
8. Environment: Each UVM verification component is housed in a single container called environment. The verification components can be altered in any way by defining an environment. On a UVM test bench, there may be several comparable situations.
9. Test: It establishes the circumstances and the sequence required to validate the different DUT identifying traits. Running an appropriate number of test cases causes the proper convergence of coverage and the full functional verification.

C. The Verification Plan

As it outlines the functional requirements of the design and identifies the many elements to be evaluated, the verification strategy is crucial to any verification effort. Furthermore, it specifies the test cases that must be run in order to fulfill the full design verification, as well as the coverage objectives.

Note: Write and read will be performed in the same transaction.

1. UVM Sequence item

All the signals required will be declared here. Write address, write response, read address and read response is declared as per the design. The design constraints are also added here and are applied to the id field for both read and write where the length is maintained same where a unique value will be randomized. Another constraint is added to the burst modes, here we have three types of burst modes i.e. Fixed, INCR and WRAP modes where awburst and airburst must be inside {0,1,2} where 0 indicates the fixed, 1 indicates increment mode and 2 indicates the WRAP mode. Similarly for valid signals, the read and write valid should not be equal for a transaction to be successful.

2. Rst_dut

The reset sequence is defined inside uvm_sequence which works on the transaction data. Inside the body task the reset of the dut takes place.

3. valid_wrrd_fixed

The write read transactions in fixed mode is also extended in uvm_sequence. Inside the body an object is created where the length of the burst transaction, type of burst and the size is mentioned. Burst mode has to be 0 for the design to work in fixed mode and followed by awsize is 2 which depicts that a maximum of 4 bytes is allowed for the transaction.

4. valid_wrrd_incr:

The write read transactions in fixed mode is also extended in uvm_sequence. Inside the body an object is created where the length of the burst transaction, type of burst and the size is mentioned. Burst mode has to be 1 for the design to work in fixed mode and followed by awsize is 2 which depicts that a maximum of 4 bytes is allowed for the transaction. With uvm_info, the message is displayed to make sure that the transaction is completed.

5. valid_wrrd_wrap:

The write read transactions in fixed mode is also extended in uvm_sequence. Inside the body an object is created where the length of the burst transaction, type of burst and the size is mentioned. Burst mode has to be 2 for the design to work in fixed mode and followed by awsize is 2 which depicts that a maximum of 4 bytes is allowed for the transaction. With uvm_info, the message is displayed to make sure that the transaction is completed.

6. err_wrrd_fix:

The error transactions in terms out of address transactions are sent and extends the uvm_sequence and similarly the three different burst mode transactions are verified for the error response.

UVM Classes

A. Class Driver:

wrrd_fixed_wr:The important signals for the read and write transactions are applied here with awvalid and wvalid to be high. The wait will be granted for the required number of wready and awready bits for the transactions to take place for the positive edges of the clock pulse.The interface is also declared here to get the access of the interface and the data container will be declared here to receive the transactions. Independent tasks will be created for rese for the positive edge of the clock pulse. During the write read write in fixed mode the reset will be removed by making it 1 followed by awvalid high and followed by declaring all the signals, awlen will be coming from sequencer and should be same, starting address is also mentioned for the fixed mode transactions to take place and the awburst to be 0 for fixed mode transactions. The wdata will be randomized with the rangefrom 0 to 10. The strobe signals are also declared to access all the bits. During the write stage the read valid has to be disabled and the read ready has to be disabled to deactivate read channel. last signal will be used to end the transaction. When bready and wlast is high and after the bvalid is high, the response of the transaction is received to indicate the status of the transaction. The negative edge of bready is always noted for the transaction to be successful and as soon as the positive edge of bready is depicted then the next transaction will be executed where this will be done as long as the burst length is reached. Post the write transactions, all the valid signals will be deactivated to end the transaction and wait for the response by enabling the bready and wlast signals.

wrrd_wrap_wr and wrrd_incr_wr: The only configuration change that is needed is to change the awburst to 1 for increment and 2 for wrap mode. The configuration begins by resetting the design, making the valid high and configuring all the mandatory signals for the write transactions to take place by turning off the read signals. Wait for the positive edge to complete burst transactions.

wrrd_fixed_rd, wrrd_incr_rd and wrrd_wrap_rd:During the read mode, arvalid and rvalid is made active. Till the positive edge of the rvalid is received depending on the number of transactions is available for the design to perform. During the positive edge of arready and at the end of the next clock tic, the next transaction is received. Both rvalid and arready as both are working on the same instance. Once all the transactions are read, the rlast signal is made active to receive the response of the transaction and finally the read is removed.

run_phase: All the specified tasks will be executed in the run phase. Using the seq_item_port.get_next_item, the transactions are received from the sequencer. The reset_dut task will perform the reset, all the various wr_rd_wr transactions with various burst modes are performed by calling the write task and read task separately with the memory slave. Finally the error write and read behaviors are also checked.

B. Class Monitor:
The monitor is used to store the data in an array during the write operation and during the read operation the comparison will be done where the size of the array is 32bits. Registering the class to a factory is done, in the build phase the access of the interface is granted. In the main phase of the monitor, works at the positive edge of the clock. Wready indicates the completion of write transaction where the data will be stored in the memory where we wait for awvalid and the array will be updated with the address of the next write with the data available on the wdata bus for a successful transfer the responseis 0 else there is an error transaction which is detected. The same address will used to access the array during the comparison of data written into the memory. The response is also collected at every instance of the transactions.

Task compare:

Here the signals err, rdresp and wrresp will be checked if err==0, rdresp==0 &wrresp==0 then the comparison will be success else the uvm_error will be displayed.

C. Class Agent:
Agent class will have the instance of driver, sequencer and monitor. In the build phase the object of all three will be created and using the connect phase the sequence_item_port of the driver will be connected with the sequence_item_export of the sequencer.

D. Class env:
Inside the env, the instance of the agent will be created and the main phase or run_phase the objection will be raised and dropped where the sequence will be called one by one.

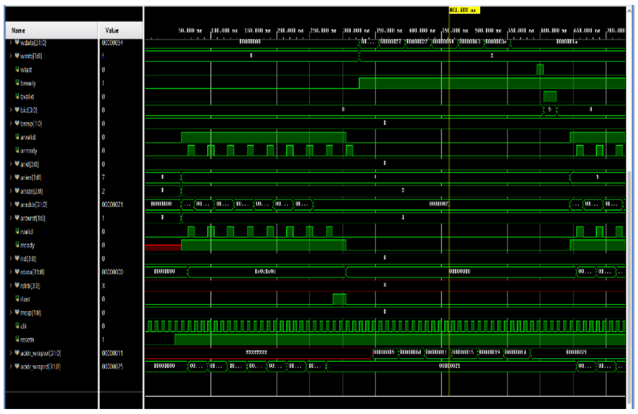


Figure 4 Increment mode burst type

SIMULATION AND RESULTS:

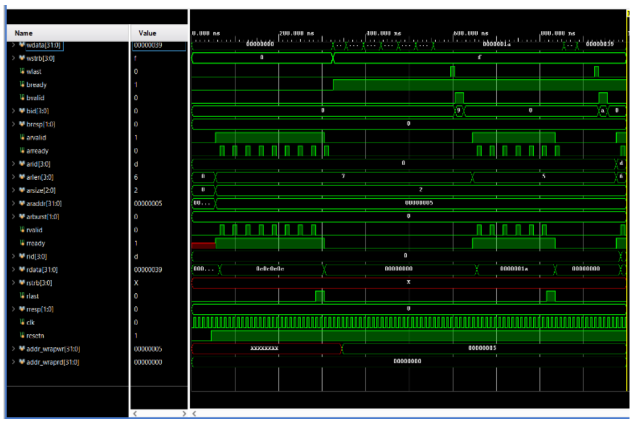


Figure 5: Valid Transactions of Fixed Mode

A. Valid transactions for fixed mode

- The fixed mode transactions under the reset mode are first checked where the time resolution at 1 ps, the reset is done with WR as 0, RD as 1, WRADDR as 5 and RDADDR as 5, WLEN as 3, RLEN as 7 and burst mode as Fixed mode, the data read will be 0 from the memory.

- After reset, with WR signal as 1 and start address as 5, the randomized data for 8 iterations will be written into the specified address location. The data written are 3c, 27, 22, 54, 63, 3e, 1a at the specified address location which is 5. After the transaction is complete, read process will begin by reading the data from the last specified address location as it is fixed burst mode.

B. Increment Mode

- During the increment mode the data is getting written into various address locations in an increment fashion with the starting address location as 5 and gets incremented with a write length as 4 with burst type as 1 for INCR mode, WR as 1 & RD as 0and the address locations are as follows: 5, 9, d, 11, 15, 19, 1d. The data written into these locations are 3c, 27, 22, 54, 63, 3e, 1a and here the transaction is completed for write. During the INCR read mode the data written in these locations will be read with starting location as 5 with transactions are done with RD as 1. The comparisons of the data read and written will done by the scoreboard and this will continue till the transaction is completed.

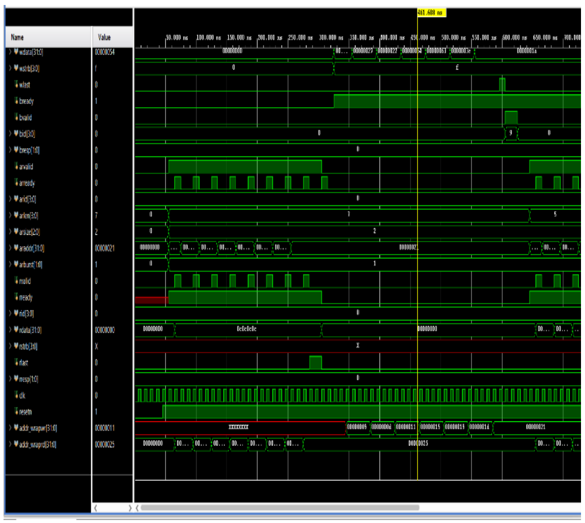


Figure 6 WRAP Mode transactions

C. WRAP Mode

- During the wrap mode, the transactions are very similar to the INCR mode. The only difference is that the address locations will be wrapped here by completing the transactions by writing to the 1st available address location. Similar to INCR and FIXED modes, the reset of the DUT will be done.The signals used in this mode are WR:1, RD:0, WRBUR:2, RDBUR:2, RADDR:5, WRADDR:5, WLEN:7, RLEN:7. During the write mode the address locations specified are 5, 9, d, 11, 15, 19, 1d, 1 for completing the wrap. The data written are 3c, 27, 22, 54, 63, 3e, 1a, f. Similarly, during the read mode, the data is read from the locations in the wrap mode and the data is retrieved from all the locations till the transactions are done. Finally, the scoreboard will compare the data written and read and throws the UVM_info about the data comparisons.

D. ASSERTIONS INSERTED IN THE DESIGN

Assertions are an important in terms of verification point of view, where the internal signals of the design are checked under various scenarios for which the DUT has to be stable. The various scenarios the properties are written for which the design is checked are: 1. during the positive edge of the clock the state of the design signal is checked at various timestamps to check whether the signals are toggling as expected or not, 2. During the positive edge of the clock, design signals are sampled at the rising edge of the sampled data, 3. Design handshake signals ack and req are checked wherein during the rising edge of request, the acknowledge should be active at the rising edge in non-overlapping mode., 4. During the rising edge of the en signal, data signal of the current iteration should be more than the previous value. During the level conditions as well as the behavior is checked. 5. By fixing the data signal to a fixed value and checking the behavior at various timestamps. 6. Disabling the signals at a particular data pattern is matched.

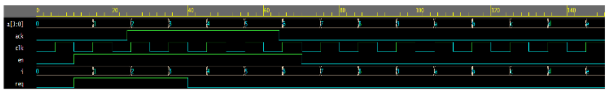


Figure 7: Assertion is checked at the level condition of the enable.

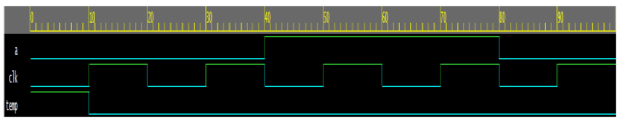


Figure 8: Assertions during the case 1

In case1, it is noted that the assertion gets failed for data signal a at 10ns and passes at 50ns during the positive edge of the clock. The assertion will pass whenever the signal is high only at the positive edge of the master clock.

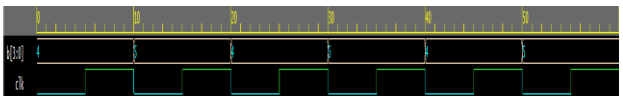


Figure 9 Assertion for data signal b under the sampling condition.

The assertion is checked for the sampled data of b signal at every rising edge of the same signal. Here a 4-bit data behavior is checked at various timestamps and the assertion passes for every iteration when the data signal goes from low to high.

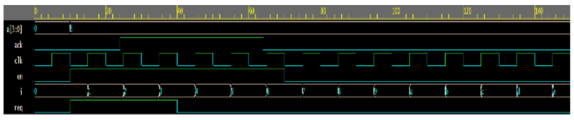


Figure 10: Assertion for rising edge of req, the acknowledge is checked.

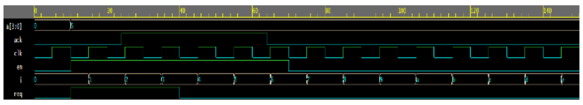


Figure 11: The behavior of data signals is checked for every clock edge.

The assertion for every rising edge of request, the behavior from the slave signal is checked. Assertion passes for every rising edge of req if the ack is high at the next rising edge of the clock.

Assertion is checked for every rising edge of the en (Enable) signal, the behavior of data is checked where in the value of the data at current iteration should be more than the previous clock edge. This is to check whether the incremental data insertions are happening in the design.

During the level condition the design signal, the assertion gets passed at the timestamps 25ns, 35ns, 45ns, 55ns, 65ns and 75ns and is depicted in the waveform shown in figure 10.

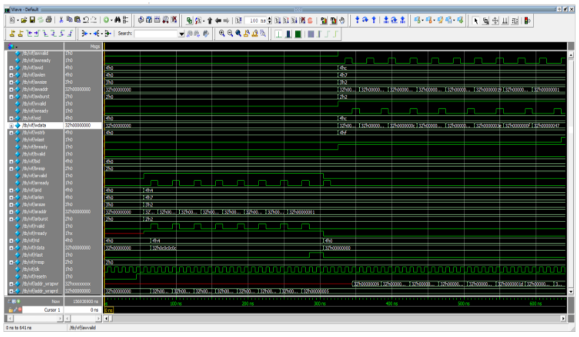


Figure 12: Integration of the memory slave with the design using Questasim

Figure 12 depicts about the usage of questasim for simulating the design along with the memory slave with all the interface signals showing the intended behavior.

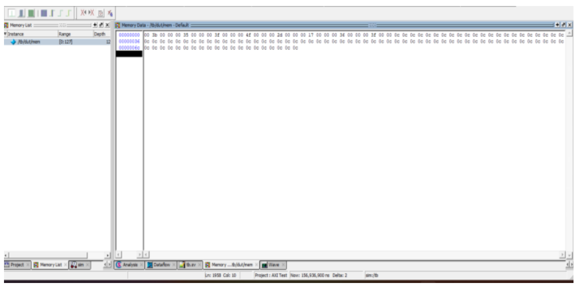


Figure 13 Memory by taking 128 bits memory space as an example for simulation

Memory is one of the most important part of a complete system for using which the intended tasks are performed for the end user. Here memory is used as the slave for performing various read, writes using the protocol which is one of the highly recommended interfaces in the SoC for high speed data transfer. The memory works as expected with the interface and the assertions have also passed with various scenarios making it very capable of performing important tasks. The burst transactions have also been exercised using the memory and the writes as well as the read happened using the axi interface with the design signals.

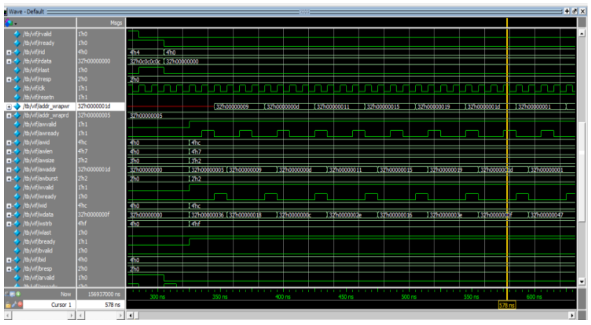


Figure 14: Figure showing the Design integrated along with the assertions with burst transactions of wrap mode

Figure 14 shows a snippet of burst mode transaction with wrap writing into the memory starting at the address 32’h00000009, 32’h0000000d, 32’h000000011, 32’h000000015, 32’h00000019, 32’h0000001d, 32’h00000001, which ends by wrapping the address as a completed envelope by writing the 1st address space available in the memory.

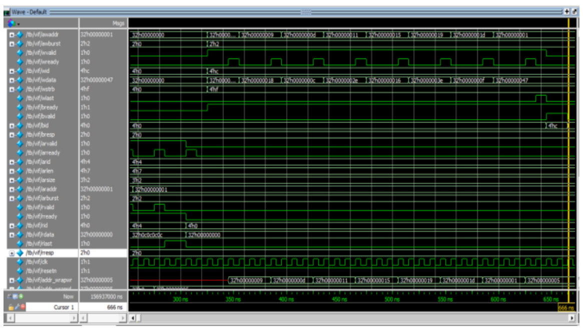


Figure 15 Burst transactions wrap read

The read transaction using the wrap mode is shown above starting from 350ns to 650ns.

CONCLUSION

In this paper, for the purpose of functionally verifying the AXI protocol, a UVM testbench is proposed.Through the methodical execution of pertinent test cases, the testbench completely functionally verified AXI and obtained 100% functional coverage. The simulation waveform shows how data can be successfully transferred between memory slave and AXI master modules using the Advanced Extensible Interface bus signals for various read and writes. The assertions were inserted in the testbench making it very reliable for the data behavior at various scenarios write – read, reset- read, read – write and reset – write at both edge and level conditions. The request, acknowledge, data signals at overlapping and non overlapping modes were also checked.Assertions also make sure that the signals are toggling as expected and hence helps in covering the signals during the coverage for both line and toggle. The entire design was done using the system verilog and the testbench environment was done UVM using Xilinx vivado 2020.02 and Questasim 10.4 e for simulating the design.

REFERENCES

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