Sumeet Pawar
Surakshith Reddy Mothe
Seth Rohrbach
ECE593

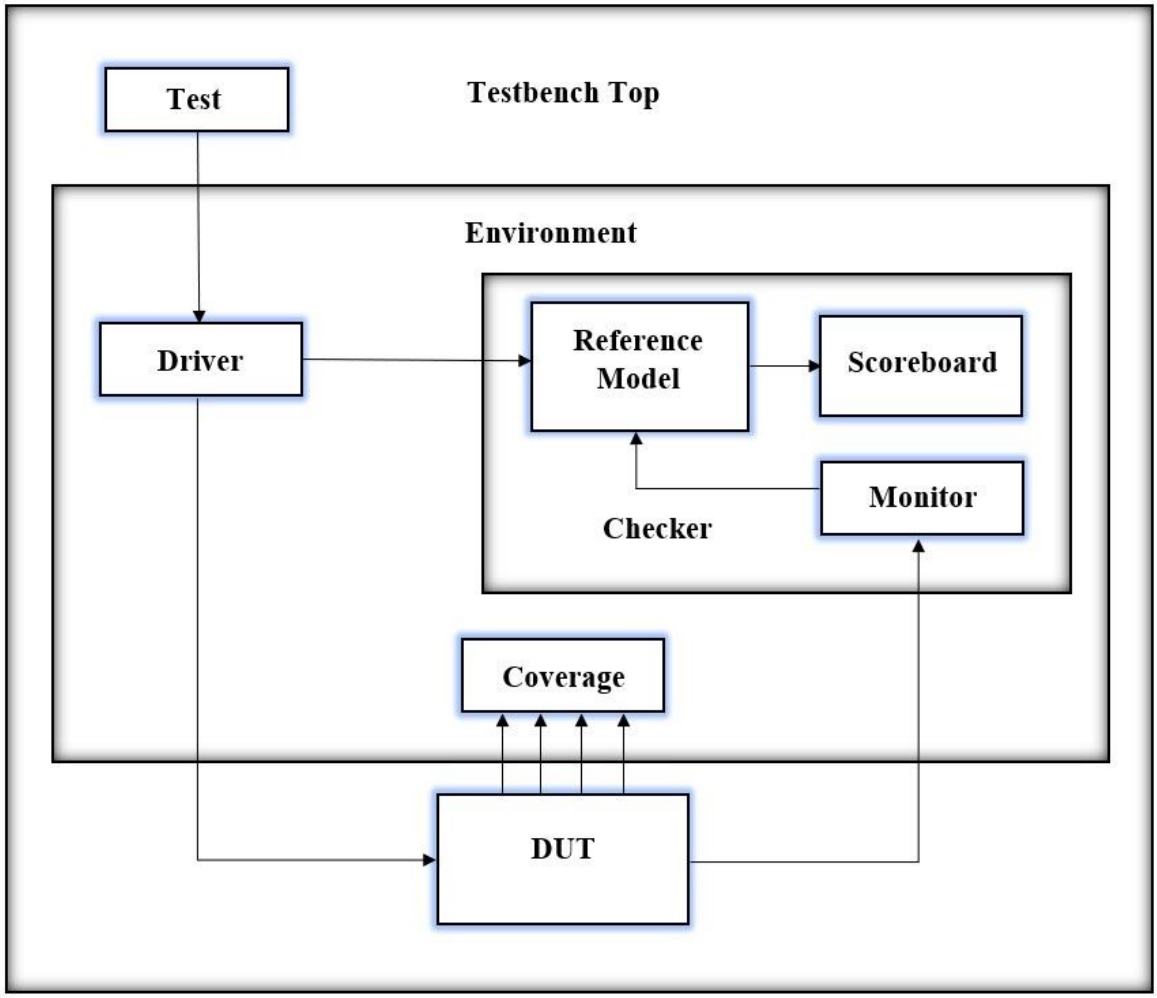
AXI4-Lite Verification Report

Abstract

Our team verified all functionality of the AXI4-Lite bus communication protocol with some code coverage of the design as well. Basic functionality was completely verified by our tests but ideal code coverage was hampered by SystemVerilog limitations and time constraints. Our longest test run showed 74.85% code coverage with a runtime of 1E10; the majority of uncovered space is additional address space.

I. Verification Techniques

i) Testbench components:



The above mentioned keywords are associated with the following:
1) Testbench Top: '**a_OOTB_TOP**' **module** defined in the '**axi4_ootb_top.sv**' file.
2) Environment: '**axi4_environment**' **class** defined in the '**axi4_env.sv**' file.
3) Driver: '**axi4_tester**' **class** defined in the '**axi4_tb_objs.sv**' file.
4) Checker: '**axi4_checker**' **class** defined in the '**axi4_checker_obj.sv**' file.
5) Coverage: '**axi4_Coverage**' **class** defined in the '**axi4_Coverage.sv**' file.
6) DUT: '**axi4_lite_master**' **module** defined in '**axi4_lite_master.sv**' file and '**axi4_lite_slave**' **module** defined in '**axi4_lite_slave.sv**' file.

ii) Testbench component interactions:

The top level testbench module contains the DUT instance and the environment class. The environment contains the Driver class, Checker class and the Coverage class.

The tests are provided to the Driver class from the top level testbench. The Driver class drives these tests into the DUT. The outputs that are generated by the DUT are observed by the monitor which is present in the Checker class. The monitor will provide the results produced by the DUT to the Reference model. Along with this, the reference model also receives the test cases from the Driver class which are being driven into the DUT. The reference model, upon receiving these test cases from the driver, will produce the expected output on-the-fly. This expected output is compared with the output generated by the DUT. Upon a mismatch between the expected output and the output generated by the DUT, the score (which is initially set to 0) is incremented.

When the tests are being run, the Coverage class will gather the coverage data from the DUT. This coverage data will reveal un-exercised portions of design code. The functional coverage will reveal missing functionalities in the design.

II. Verification Results

Our verification strategy employs several deterministic tests, a fully random test, and several constrained random tests for corner cases we identified. The deterministic tests include writing sequentially to 4k memory locations, reading sequentially from 4k memory locations, reading from and writing to an out of bounds address, writing to and reading from the same location at 4k different addresses concurrently, and reading from and writing from 4k different locations concurrently. Constrained random tests include randomly reading from and randomly writing to the 4k different local memory addresses that SystemVerilog allowed us to instantiate. We have also included fully randomized read and write tests.

Because the AXI4 protocol is relatively simple, we implemented all of the tests under the same class and executed them sequentially using the same environment in order to improve coverage. For a more complicated design, it would make sense to implement the tests as separate objects and execute them individually; however, we decided to execute them in the same environment in an attempt to maximize coverage.

Our best coverage was 74.85%. The main difficulty in achieving higher coverage is the 32 bit address space. We encountered two problems when attempting to get better coverage of the address space: simulation time, and instantiating local memory in the testbench or checker objects. Our simulations encountered problems when we tried to allocate more than 4k of local memory in a file for testing purposes, and we did not have time to try to find a solution to allow us to increase that. For execution time, simply running the simulation for longer would allow us to increase our coverage.

Future work should first address these two problems. Increasing simulation time is an obvious first step to improving coverage. The local memory limitations for SystemVerilog likely also have a solution. Past that, verification could be improved with the addition of parameters to our tests. This could allow the testbench to be used if the AXI4-Lite Bus protocol changes. It is also possible that our testbench could work for other serial protocols.

III. Labour Recap

Sumeet Subhash Pawar:
I spent around 3 hours per week from week 6 through 8 and around 10 hours per week during week 9 and 10. So a total of 30 hours have been spent by me during this entire term on this project. During the initial weeks, the efforts were focused on collaboratively discussing and choosing an appropriate design to verify, checking the waveforms to see if the design is working correctly and coming up with an initial verification plan. Later on, I extensively worked on coming up with the Coverage class which required thoughtful analysis of the coverage data that we want to gather. I also worked on putting this coverage class together with the other files in the hierarchy. This was a cumbersome process for me because it was the first time I was working on creating a hierarchical testbench structure. In the end, I was able to come up with a solution to get this coverage class working with other classes in the hierarchy. The remaining time and effort was spent on revising the verification plan, preparing the verification report, running simulations to enhance coverage, and debugging and resolving errors collaboratively with my other teammates.

Surakshith Reddy Mothe:
I have spent around 4 hours a week from week 6 through week 9, and 12 hours in week 10 which sums up to a total of around 26-30 hours during the course of the term. The initial time was spent on reading the specification, coming up with the test scenarios to include in the verification plan along with my teammates. Since it's our first time working with classes I took time to clearly understand all the basics of OOPs constructs and tried a few examples before implementing them in the project. At first, I worked on developing a bfm and finalized it with the consent of my other teammates. Then I worked on developing covergroups along with other teammates and have finally come up with a rigorous coverage class. I have also worked on developing deterministic tests in the form of tasks and random read or write tasks that use the rand type variables to perform random read and random write transactions. Regular communication with other teammates helped me in developing more tests and properly understanding how we can link all the classes for the proper function of the test environment.

Seth Rohrbach:
I would estimate I spent 3-4 hours per week on this project for weeks 6 through 9, and in week 10 I spent more like 12 hours on this project. This totals to 24-28 hours or so in the second half of this quarter. The majority of this time was spent working on various classes for the object oriented test bench. I also spent a considerable amount of time putting the various objects together. The basic structure of the test bench was not particularly challenging, but the details of the object oriented syntax and the way they went together was more difficult than I expected. In particular, making sure the various files knew about the other objects was cumbersome and could be quite frustrating, and I think that the way UVM takes care of that for you would be a major boon. Looking sideways in the hierarchy was also something that took a little bit of thought and I had not done before, because in prior testbenches I would generally just instantiate everything I wanted access to in the top level module and I could look downwards through the hierarchy to see any signals needed.

IV. Challenges Encountered

This project presented a number of challenges for our team. No members of our team had any prior experience with object oriented programming. Our initial verification plans had to be slightly scaled back when we encountered some difficulty with various new syntax, functionality, and implementation details of the object oriented test bench.

One of the primary challenges we faced was implementing our verification plan as an object oriented test bench. This was really the goal of the project, but because no member of our team had any prior experience with object oriented programming, this meant in many ways we were re-learning how to do things that would otherwise be trivial. This definitely slowed our progress in general.

A specific problem that objects and classes in SystemVerilog caused us was how to share the class with the entire design. This caused major headaches at several points, particularly because the SystemVerilog compiler did not necessarily provide verbose warnings. In particular, we eventually discovered that using the `include directive essentially appends the `included file at the end of the current file. This could multiply define critical signals or modules, leading to extremely vague error messages from the compiler. Eventually we were able to include all of our class definitions by `including them in the package containing other definitions, but this would prove to be an imperfect solution.

An unrelated challenge which ended up conflicting with our `include solution was looking sideways in the hierarchy. Because our test drivers, checker, and coverage classes were effectively all parallel with each other and the DUV, as well as below the BFM in the hierarchy, it was slightly problematic to look at signals in the AXI4 master and slave modules. We ended up using $root.hierarchy syntax in order to look sideways in the hierarchy; however, the SystemVerilog compiler does not allow files `included in packages to use the $root call, which conflicted with our ability to `include all of the class definitions in the package. We ended up using a tree structure to work around this, but I do not think it is a perfect solution. It makes us appreciate what UVM test bench code provides.

V. Code Leveraged

The only pre-existing code which we leveraged was the code obtained from the design team who implemented the AXI4_lite protocol design.

Our verification team initially tried to verify it's basic functionality with a standard test environment and taking a look at its waveforms so as to ensure that the design is being compiled and is working before our team could move forward with creating a much robust verification plan to rigorously verify the design. After a healthy testing in the available time frame, our team has suggested what more can be done in the 'Future Plans' section to reach a verification complete phase.

VI. Bugs

There was one bug we encountered with SystemVerilog. The simulation tools would not allow us to use larger than a 4k array. This slightly complicated functional code verification, although using the 4k array does provide reasonable confidence that the design is working when combined with monitoring the BFM for addresses outside the 4k space.

At this time we have not injected any intentional bugs into the design or into our test bench due to time constraints. If time permitted we planned to expand testing by injecting bugs into the AXI4-Lite master and slave modules.

VII. Testbench Code Cross Reference

i) axi4_lite_Defs.sv:

The axi4_lite_Defs.sv file contains the axi4_lite_Defs **package**. This package is imported in all our other testbench files as well as the DUT files. The details of this package are as follows:

1) axi4_lite_Defs: This package contains the global definitions such as parameters for the AXI4 Lite Bus design which can be imported in other files of our object oriented testbench. The contents of this package are:

i) Address width and Data width of the bus defined as parameters. Both the Address and Data width are 32-bits.

ii) Enumerated state types which represent the states that the Master and Slave FSM Design can attain during a Read/Write operation. These states are IDLE, ADDR, DATA, RESP.

iii) This package also includes the axi4_tb_objs.sv file and axi4_checker_obj.sv file. This would later on help in connecting all the files together in a hierarchy in the top level testbench by simply importing this package.

ii) axi4_lite_bfm.sv:

This file contains an **interface** axi4_lite_bfm that contains all the interface signals between the master and the slave including the master_if and slave_if modports to properly declare the directions of the interface signals. This interface is the crux of the testbench environment as it is accessed by all the classes namely axi4_tester class, axi4_checker class, axi4_coverage class and the axi4_environment class by declaring it as virtual in each class which in turn makes sure that all the components interfere with common signals. The testbench environment drives and monitors the signals declared in this bfm interface. This interface contains the following signals:

1) **ARREADY**: Slave asserts this signal high to indicate it is ready to receive the address on the bus.

2) **RDATA:** Contains the data provided by the slave for a read transaction.

3) **RVALID:** Slave asserts this signal high to indicate the data present in the channel is high.

4) **ARADDR:** Contains the read address provided by the master for a read transaction.

5) **ARVALID:** Master asserts this signal high to indicate the read address is valid.

6) **RREADY** : Master asserts this signal high to indicate it is ready to receive data from the slave.

7) **AWREADY**: Slave asserts this signal high on the write address channel to indicate it is ready for a write transaction.

8) **AWVALID:** Master asserts this signal high to indicate the write address is valid.

9) **WVALID:** Master asserts this signal high to indicate the write data is valid.

10) **WREADY:** Slave asserts this signal high on the write data channel to indicate it is ready for a write transaction.

11) **WDATA:** Contains the write data provided by the master for a write transaction.

12) **AWADDR:** Contains the write address provided by the master for a write transaction.

13) **BREADY:** Master asserts this signal high to indicate it is ready to receive a response from the slave.

14) **BVALID:** Slave asserts this signal high to indicate a valid response on the write response channel.

This interface also has a ports list that has a clock signal namely **ACLK** and an asynchronous active low reset signal namely **ARESETN** that the master and slave work on.

iii) axi4_Coverage.sv:

The axi4_Coverage.sv file contains a single **class** called 'axi4_Coverage'. This class contains all the covergroups for obtaining the code and functional coverage data. There are a total of 8 covergroups defined in this class with their respective coverpoints. Additionally, there is a function 'new' and a task 'execute' also defined in this class. The details of the contents of the class are as follows:

1) **Covergroups:**

i) cg_Read_Address: This covergroup is for sampling the Read Address channel signals. All the coverpoints here sample the signals only when Reset signal is NOT provided. It contains the following 3 **coverpoints**:

a) **Read_Address_Valid:** This is the coverpoint for Read Address Valid signal. It contains the following 2 **bins**:
   - **ARVALID_High**: This bin is hit when ARVALID signal is asserted high.
   - **ARVALID_Low:** This bin is hit when ARVALID signal is asserted low.

b) **Read_Address_Ready:** This is the coverpoint for Read Address Ready signal. It contains the following 2 **bins**:
   - **ARREADY_High:** This bin is hit when ARREADY signal is asserted high.
   - **ARREADY_Low:** This bin is hit when ARREADY signal is asserted low.

c) **Read_Address:** This is the coverpoint for Read Address. It contains the following 3 **bins**:
   - **ARADDR_First_Location:** This bin is hit when Read Address has a value of 0, which is our first location in the memory.
   - **ARADDR_Last_Location:** This bin is hit when Read Address has a value of 4096, which is our last location in the memory.
   - **ARADDR_range[]:** This is an automatic bin which creates unique bins for the address range from location 1 to location 4095. These bins would be hit when Read Address has a value of in the range 1 to 4095.

ii) cg_Read_Data: This covergroup is for sampling the Read Data channel signals. All the coverpoints here sample the signals only when Reset signal is NOT provided. It contains the following 3 **coverpoints**:

a) **Read_Data_Valid:** This is the coverpoint for Read Data Valid signal. It contains the following 2 **bins**:
   - **RVALID_High:** This bin is hit when RVALID signal is asserted high.
   - **RVALID_Low:** This bin is hit when RVALID signal is asserted low.

b) **Read_Data_Ready:** This is the coverpoint for the Read Data Ready signal. It contains the following 2 **bins**:
   - **RREADY_High:** This bin is hit when RREADY signal is asserted high.

- **RREADY_Low:** This bin is hit when RREADY signal is asserted low.

c) **Read_Data:** This is the coverpoint for Read Data. It contains the following 3 **bins**:
   - **RDATA_All_Zeros:** This bin is hit when Read Data has a value of 0.
   - **RDATA_All_Ones:** This bin is hit when Read Data has a value of 4096.
   - **RDATA_range[]:** This is an automatic bin which creates unique bins for the data range from location 1 to location 4095. These bins would be hit when Read Data has a value of in the range 1 to 4095.

iii) cg_Write_Address: This covergroup is for sampling the Write Address channel signals. All the coverpoints here sample the signals only when Reset signal is NOT provided. It contains the following 3 **coverpoints**:

a) **Write_Address_Valid:** This is the coverpoint for Write Address Valid signal. It contains the following 2 **bins**:
   - **AWVALID_High**: This bin is hit when AWVALID signal is asserted high.
   - **AWVALID_Low:** This bin is hit when AWVALID signal is asserted low.

b) **Write_Address_Ready:** This is the coverpoint for Write Address Ready signal. It contains the following 2 **bins**:
   - **AWREADY_High:** This bin is hit when AWREADY signal is asserted high.
   - **AWREADY_Low:** This bin is hit when AWREADY signal is asserted low.

c) **Write_Address:** This is the coverpoint for Write Address. It contains the following 3 **bins**:
   - **AWADDR_First_Location:** This bin is hit when Write Address has a value of 0, which is our first location in the memory.
   - **AWADDR_Last_Location:** This bin is hit when Write Address has a value of 4096, which is our last location in the memory.
   - **AWADDR_range[]:** This is an automatic bin which creates unique bins for the address range from location 1 to location 4095. These bins would be hit when Write Address has a value of in the range 1 to 4095.

iv) cg_Write_Data: This covergroup is for sampling the Write Data channel signals. All the coverpoints here sample the signals only when Reset signal is NOT provided. It contains the following 3 **coverpoints**:

a) **Write_Data_Valid:** This is the coverpoint for Write Data Valid signal. It contains the following 2 **bins**:
   - **WVALID_High:** This bin is hit when WVALID signal is asserted high.
   - **WVALID_Low:** This bin is hit when WVALID signal is asserted low.

b) **Write_Data_Ready:** This is the coverpoint for the Write Data Ready signal. It contains the following 2 **bins**:
   - **WREADY_High:** This bin is hit when WREADY signal is asserted high.

- **WREADY_Low:** This bin is hit when WREADY signal is asserted low.

c) **Write_Data:** This is the coverpoint for Write Data. It contains the following 3 **bins**:
   - **WDATA_All_Zeros:** This bin is hit when Write Data has a value of 0.
   - **WDATA_All_Ones:** This bin is hit when Write Data has a value of 4096.
   - **WDATA_range[]:** This is an automatic bin which creates unique bins for the data range from location 1 to location 4095. These bins would be hit when Write Data has a value of in the range 1 to 4095.

v) cg_Write_Response: This covergroup is for sampling the Write Response channel signals. All the coverpoints here sample the signals only when Reset signal is NOT provided. It contains the following 2 **coverpoints**:

a) **Write_Response_Valid:** This is the coverpoint for the Write Response Valid signal. It contains the following 2 **bins**:
   - **BVALID_High:** This bin is hit when BVALID signal is asserted high.
   - **BVALID_Low:** This bin is hit when BVALID signal is asserted low.

b) **Write_Response_Ready:** This is the coverpoint for the Write Response Ready signal. It contains the following 2 **bins**:
   - **BREADY_High:** This bin is hit when BREADY signal is asserted high.
   - **BREADY_Low:** This bin is hit when BREADY signal is asserted low.

vi) cg_Master_FSM: This covergroup is for sampling the Master FSM transitions. All the coverpoints here sample the signals only when Reset signal is NOT provided. It contains the following 2 **coverpoints**:

a) **Master_Read_FSM:** This is the coverpoint for sampling the Master FSM transitions on a Read operation. It contains the following 11 **bins**:
   - **mr1:** This bin will be hit when the Master FSM transitions from IDLE to ADDR.
   - **mr2:** This bin will be hit when the Master FSM transitions from ADDR to DATA.
   - **mr3:** This bin will be hit when the Master FSM transitions from DATA to RESP..
   - **mr4:** This bin will be hit when the Master FSM transitions from RESP to IDLE.
   - **mr_sequence:** This bin will be hit when the Master FSM transitions from IDLE to ADDR to DATA to RESP and back to IDLE. This represents the expected sequence of Master FSM state transitions for a read operation.
   - **mr_illegal1:** This is an illegal bin which will be hit when the Master FSM transitions from DATA to ADDR. A hit on this bin will cause a runtime error.
   - **mr_illegal2:** This is an illegal bin which will be hit when the Master FSM transitions from RESP to DATA. A hit on this bin will cause a runtime error.
   - **mr_illegal3:** This is an illegal bin which will be hit when the Master FSM transitions from RESP to ADDR. A hit on this bin will cause a runtime error.

- **mr_illegal4:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to DATA. A hit on this bin will cause a runtime error.
- **mr_illegal5:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to RESP. A hit on this bin will cause a runtime error.
- **mr_illegal6:** This is an illegal bin which will be hit when the Master FSM transitions from ADDR to RESP. A hit on this bin will cause a runtime error.

b) **Master_Write_FSM:** This is the coverpoint for sampling the Master FSM transitions on a Write operation. All the coverpoints here sample the signals only when Reset signal is NOT provided. It contains the following 11 **bins**:
   - **mw1:** This bin will be hit when the Master FSM transitions from IDLE to ADDR.
   - **mw2:** This bin will be hit when the Master FSM transitions from ADDR to DATA.
   - **mw3:** This bin will be hit when the Master FSM transitions from DATA to RESP..
   - **mw4:** This bin will be hit when the Master FSM transitions from RESP to IDLE.
   - **mw_sequence:** This bin will be hit when the Master FSM transitions from IDLE to ADDR to DATA to RESP and back to IDLE. This represents the expected sequence of Master FSM state transitions for a write operation.
   - **mw_illegal1:** This is an illegal bin which will be hit when the Master FSM transitions from DATA to ADDR. A hit on this bin will cause a runtime error.
   - **mw_illegal2:** This is an illegal bin which will be hit when the Master FSM transitions from RESP to DATA. A hit on this bin will cause a runtime error.
   - **mw_illegal3:** This is an illegal bin which will be hit when the Master FSM transitions from RESP to ADDR. A hit on this bin will cause a runtime error.
   - **mw_illegal4:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to DATA. A hit on this bin will cause a runtime error.
   - **mw_illegal5:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to RESP. A hit on this bin will cause a runtime error.
   - **mw_illegal6:** This is an illegal bin which will be hit when the Master FSM transitions from ADDR to RESP. A hit on this bin will cause a runtime error.

vii) cg_Slave_FSM: This covergroup is for sampling the Slave FSM transitions. All the coverpoints here sample the signals only when Reset signal is NOT provided. It contains the following 2 **coverpoints**:

c) **Slave_Read_FSM:** This is the coverpoint for sampling the Slave FSM transitions on a Read operation. It contains the following 11 **bins**:
   - **sr1:** This bin will be hit when the Slave FSM transitions from IDLE to ADDR.
   - **sr2:** This bin will be hit when the Slave FSM transitions from ADDR to DATA.
   - **sr3:** This bin will be hit when the Slave FSM transitions from DATA to RESP..
   - **sr4:** This bin will be hit when the Slave FSM transitions from RESP to IDLE.

- **sr_sequence:** This bin will be hit when the Slave FSM transitions from IDLE to ADDR to DATA to RESP and back to IDLE. This represents the expected sequence of Slave FSM state transitions for a read operation.
- **sr_illegal1:** This is an illegal bin which will be hit when the Slave FSM transitions from DATA to ADDR. A hit on this bin will cause a runtime error.
- **sr_illegal2:** This is an illegal bin which will be hit when the Slave FSM transitions from RESP to DATA. A hit on this bin will cause a runtime error.
- **sr_illegal3:** This is an illegal bin which will be hit when the Slave FSM transitions from RESP to ADDR. A hit on this bin will cause a runtime error.
- **sr_illegal4:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to DATA. A hit on this bin will cause a runtime error.
- **sr_illegal5:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to RESP. A hit on this bin will cause a runtime error.
- **sr_illegal6:** This is an illegal bin which will be hit when the Slave FSM transitions from ADDR to RESP. A hit on this bin will cause a runtime error.

d) Slave_Write_FSM: This is the coverpoint for sampling the Slave FSM transitions on a Write operation. It contains the following 11 **bins**:

- **sw1:** This bin will be hit when the Slave FSM transitions from IDLE to ADDR.
- **sw2:** This bin will be hit when the Slave FSM transitions from ADDR to DATA.
- **sw3:** This bin will be hit when the Slave FSM transitions from DATA to RESP..
- **sw4:** This bin will be hit when the Slave FSM transitions from RESP to IDLE.
- **sw_sequence:** This bin will be hit when the Slave FSM transitions from IDLE to ADDR to DATA to RESP and back to IDLE. This represents the expected sequence of Slave FSM state transitions for a write operation.
- **sw_illegal1:** This is an illegal bin which will be hit when the Slave FSM transitions from DATA to ADDR. A hit on this bin will cause a runtime error.
- **sw_illegal2:** This is an illegal bin which will be hit when the Slave FSM transitions from RESP to DATA. A hit on this bin will cause a runtime error.
- **sw_illegal3:** This is an illegal bin which will be hit when the Slave FSM transitions from RESP to ADDR. A hit on this bin will cause a runtime error.
- **sw_illegal4:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to DATA. A hit on this bin will cause a runtime error.
- **sw_illegal5:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to RESP. A hit on this bin will cause a runtime error.
- **sw_illegal6:** This is an illegal bin which will be hit when the Slave FSM transitions from ADDR to RESP. A hit on this bin will cause a runtime error.

viii) cg_Reset_Signal: This covergroup is for sampling all the above signals and FSM transitions when a Reset signal is provided. All the coverpoints here sample the signals only when Reset signal is provided. It contains the following 14 **coverpoints**:

a) Read_Address_Valid_Reset: This is the coverpoint for Read Address Valid signal to sample the ARVALID signal during a reset operation. It contains the following 2 **bins**:

- **ARVALID_Low_Reset**: This bin is hit when ARVALID signal is asserted low.
- **ARVALID_High_Reset_illegal:** This is an illegal bin which is hit when ARVALID signal is asserted high.

b) **Read_Address_Ready_Reset:** This is the coverpoint for Read Address Ready signal to sample the ARREADY signal during a reset operation. It contains the following 2 **bins**:
   - **ARREADY_Low_Reset**: This bin is hit when ARREADY signal is asserted low.
   - **ARREADY_High_Reset_illegal:** This is an illegal bin which is hit when ARREADY signal is asserted high.

c) **Read_Data_Valid_Reset:** This is the coverpoint for Read Data Valid signal to sample the RVALID signal during a reset operation. It contains the following 2 **bins**:
   - **RVALID_Low_Reset**: This bin is hit when RVALID signal is asserted low.
   - **RVALID_High_Reset_illegal:** This is an illegal bin which is hit when RVALID signal is asserted high.

d) **Read_Data_Ready_Reset:** This is the coverpoint for Read Data Ready signal to sample the RREADY signal during a reset operation. It contains the following 2 **bins**:
   - **RREADY_Low_Reset**: This bin is hit when RREADY signal is asserted low.
   - **RREADY_High_Reset_illegal:** This is an illegal bin which is hit when RREADY signal is asserted high.

e) **Write_Address_Valid_Reset:** This is the coverpoint for Write Address Valid signal to sample the AWVALID signal during a reset operation. It contains the following 2 **bins**:
   - **AWVALID_Low_Reset**: This bin is hit when AWVALID signal is asserted low.
   - **AWVALID_High_Reset_illegal:** This is an illegal bin which is hit when AWVALID signal is asserted high.

f) **Write_Address_Ready_Reset:** This is the coverpoint for Write Address Ready signal to sample the AWREADY signal during a reset operation. It contains the following 2 **bins**:
   - **AWREADY_Low_Reset**: This bin is hit when AWREADY signal is asserted low.
   - **AWREADY_High_Reset_illegal:** This is an illegal bin which is hit when AWREADY signal is asserted high.

g) **Write_Data_Valid_Reset:** This is the coverpoint for Write Data Valid signal to sample the WVALID signal during a reset operation. It contains the following 2 **bins**:
   - **WVALID_Low_Reset**: This bin is hit when WVALID signal is asserted low.
   - **WVALID_High_Reset_illegal:** This is an illegal bin which is hit when WVALID signal is asserted high.

h) **Write_Data_Ready_Reset:** This is the coverpoint for Write Data Ready signal to sample the WREADY signal during a reset operation. It contains the following 2 **bins**:
   - **WREADY_Low_Reset**: This bin is hit when WREADY signal is asserted low.
   - **WREADY_High_Reset_illegal:** This is an illegal bin which is hit when WREADY signal is asserted high.

i) **Write_Response_Valid_Reset:** This is the coverpoint for Write Response Valid signal to sample the BVALID signal during a reset operation. It contains the following 2 **bins**:
   - **BVALID_Low_Reset**: This bin is hit when BVALID signal is asserted low.
   - **BVALID_High_Reset_illegal:** This is an illegal bin which is hit when BVALID signal is asserted high.

j) **Write_Response_Ready_Reset:** This is the coverpoint for Write Response Ready signal to sample the BREADY signal during a reset operation. It contains the following 2 **bins**:
   - **BREADY_Low_Reset**: This bin is hit when BREADY signal is asserted low.
   - **BREADY_High_Reset_illegal:** This is an illegal bin which is hit when BREADY signal is asserted high.

k) **Master_Read_FSM_Reset:** This coverpoint samples the Master FSM state transitions for a Read operation when a Reset signal is provided. It contains the following 12 **bins**:
   - **mr_reset1:** This bin will be hit when the Master FSM transitions from ADDR to IDLE.
   - **mr_reset2:** This bin will be hit when the Master FSM transitions from DATA to IDLE.
   - **mr_reset3:** This bin will be hit when the Master FSM transitions from RESP to IDLE.
   - **mr_illegal1:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to ADDR.
   - **mr_illegal2:** This is an illegal bin which will be hit when the Master FSM transitions from ADDR to DATA.
   - **mr_illegal3:** This is an illegal bin which will be hit when the Master FSM transitions from DATA to RESP.
   - **mr_illegal4:** This is an illegal bin which will be hit when the Master FSM transitions from DATA to ADDR.
   - **mr_illegal5:** This is an illegal bin which will be hit when the Master FSM transitions from RESP to DATA.
   - **mr_illegal6:** This is an illegal bin which will be hit when the Master FSM transitions from RESP to ADDR.
   - **mr_illegal7:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to DATA.
   - **mr_illegal8:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to RESP.

- **mr_illegal9:** This is an illegal bin which will be hit when the Master FSM transitions from ADDR to RESP.

l) **Master_Write_FSM_Reset:** This coverpoint samples the Master FSM state transitions for a Write operation when a Reset signal is provided. It contains the following 12 **bins**:
   - **mw_reset1:** This bin will be hit when the Master FSM transitions from ADDR to IDLE.
   - **mw_reset2:** This bin will be hit when the Master FSM transitions from DATA to IDLE.
   - **mw_reset3:** This bin will be hit when the Master FSM transitions from RESP to IDLE.
   - **mw_illegal1:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to ADDR.
   - **mw_illegal2:** This is an illegal bin which will be hit when the Master FSM transitions from ADDR to DATA.
   - **mw_illegal3:** This is an illegal bin which will be hit when the Master FSM transitions from DATA to RESP.
   - **mw_illegal4:** This is an illegal bin which will be hit when the Master FSM transitions from DATA to ADDR.
   - **mw_illegal5:** This is an illegal bin which will be hit when the Master FSM transitions from RESP to DATA.
   - **mw_illegal6:** This is an illegal bin which will be hit when the Master FSM transitions from RESP to ADDR.
   - **mw_illegal7:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to DATA.
   - **mw_illegal8:** This is an illegal bin which will be hit when the Master FSM transitions from IDLE to RESP.
   - **mw_illegal9:** This is an illegal bin which will be hit when the Master FSM transitions from ADDR to RESP.

m) **Slave_Read_FSM_Reset:** This coverpoint samples the Slave FSM state transitions for a Read operation when a Reset signal is provided. It contains the following 12 **bins**:
   - **sr_reset1:** This bin will be hit when the Slaver FSM transitions from ADDR to IDLE.
   - **sr_reset2:** This bin will be hit when the Slave FSM transitions from DATA to IDLE.
   - **sr_reset3:** This bin will be hit when the Slave FSM transitions from RESP to IDLE.
   - **sr_illegal1:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to ADDR.
   - **sr_illegal2:** This is an illegal bin which will be hit when the Slave FSM transitions from ADDR to DATA.

- **sr_illegal3:** This is an illegal bin which will be hit when the Slave FSM transitions from DATA to RESP.
- **sr_illegal4:** This is an illegal bin which will be hit when the Slave FSM transitions from DATA to ADDR.
- **sr_illegal5:** This is an illegal bin which will be hit when the Slave FSM transitions from RESP to DATA.
- **sr_illegal6:** This is an illegal bin which will be hit when the Slave FSM transitions from RESP to ADDR.
- **sr_illegal7:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to DATA.
- **sr_illegal8:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to RESP.
- **sr_illegal9:** This is an illegal bin which will be hit when the Slave FSM transitions from ADDR to RESP.

n) **Slave_Write_FSM_Reset:** This coverpoint samples the Slave FSM state transitions for a Write operation when a Reset signal is provided. It contains the following 12 **bins**:
   - **sw_reset1:** This bin will be hit when the Slave FSM transitions from ADDR to IDLE.
   - **sw_reset2:** This bin will be hit when the Slave FSM transitions from DATA to IDLE.
   - **sw_reset3:** This bin will be hit when the Slave FSM transitions from RESP to IDLE.
   - **sw_illegal1:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to ADDR.
   - **sw_illegal2:** This is an illegal bin which will be hit when the Slave FSM transitions from ADDR to DATA.
   - **sw_illegal3:** This is an illegal bin which will be hit when the Slave FSM transitions from DATA to RESP.
   - **sw_illegal4:** This is an illegal bin which will be hit when the Slave FSM transitions from DATA to ADDR.
   - **sw_illegal5:** This is an illegal bin which will be hit when the Slave FSM transitions from RESP to DATA.
   - **sw_illegal6:** This is an illegal bin which will be hit when the Slave FSM transitions from RESP to ADDR.
   - **sw_illegal7:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to DATA.
   - **sw_illegal8:** This is an illegal bin which will be hit when the Slave FSM transitions from IDLE to RESP.
   - **sw_illegal9:** This is an illegal bin which will be hit when the Slave FSM transitions from ADDR to RESP.

2) **Functions:**

i) new: This function will be called in the environment class that is defined in the 'axi4_env.sv' file. Calling this 'new' function will create an object of our axi4_Coverage class. The function 'new' instantiates all the 8 covergroups that are defined above. It also maps the bfm instantiated in this class with the bfm shared by other classes.

3) **Tasks:**

i) execute: This task will be called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. This task samples all our covergroups on every positive edge of the clock.

iv) axi4_tb_objs.sv:

This file contains **axi4_tester class** that drives the bfm interface master signals by taking the role of a master. It contains a **constraint** "legal" that constrains the read address(Read_Address) and write address(Write_Address) to legal values. Additionally, this class contains 14 **tasks** and a **function** "new" which are briefly explained below:

1) **Functions:**
   i) new: This function will be called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. Calling this 'new' function will create an object of our axi4_tester class. The function 'new' maps the bfm instantiated in this class with the bfm shared by other classes.

2) **Tasks:**
   i) rand_write_op: This task will be called in the 'execute' task in the axi4_tester class while the 'execute' task is called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. This task performs a read operation at a random address generated by the rand type variable Read_Address.

   ii) rand_write_op: This task will be called in the 'execute' task in the axi4_tester class while the 'execute' task is called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. This task performs a write operation at a random address with random data generated by the rand type variables Write_Address and Write_Data respectively.

   iii) Readtask: This task will be called in the tasks listed below as **v - xiii** which inturn are called in the 'execute' task in the axi4_tester class while the 'execute' task is called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. This task accepts a single argument namely R_address that contains the read address to perform a read transaction at that address.

   iv) Writetask: This task will be called in the tasks listed below as **v - xiii** which inturn are called in the 'execute' task in the axi4_tester class while the 'execute' task is called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. This task accepts two arguments namely W_address and W_Data that contains the write address and write data to perform a write transaction at that address with write data.

   v) Write_Read_IntermediateLocation_Test: This task will be called in the 'execute' task in the axi4_tester class while the 'execute' task is called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. This task calls 'Readtask' and 'Writetask' to perform a single write and then single read from the same intermediate location.

   vi) Write_Read_FirstLocation_Test: This task will be called in the 'execute' task in the axi4_tester class while the 'execute' task is called in the 'axi4_environment' class that is defined

in the ‘axi4_env.sv’ file. This task calls ‘Readtask’ and ‘Writetask’ to perform a single write and then single read from the first location.

vii) Write_Read_LastLocation_Test: This task will be called in the ‘execute’ task in the axi4_tester class while the ‘execute’ task is called in the ‘axi4_environment’ class that is defined in the ‘axi4_env.sv’ file. This task calls ‘Readtask’ and ‘Writetask’ to perform a single write and then single read from the last location.

viii) Write_Read_Sametime_SameLocation_Test: This task will be called in the ‘execute’ task in the axi4_tester class while the ‘execute’ task is called in the ‘axi4_environment’ class that is defined in the ‘axi4_env.sv’ file. This task calls ‘Readtask’ and ‘Writetask’ to perform single read and single write at the same time to the same location.

ix) Write_Read_Sametime_DifferentLocation_Test: This task will be called in the ‘execute’ task in the axi4_tester class while the ‘execute’ task is called in the ‘axi4_environment’ class that is defined in the ‘axi4_env.sv’ file. This task calls ‘Readtask’ and ‘Writetask’ to perform single read and single write at the same time to different locations.

x) Multiple_Writes_Multiple_Reads_ConsecutiveLocations_Test: This task will be called in the ‘execute’ task in the axi4_tester class while the ‘execute’ task is called in the ‘axi4_environment’ class that is defined in the ‘axi4_env.sv’ file. This task calls ‘Readtask’ and ‘Writetask’ to perform multiple writes first to consecutive locations and then multiple reads from the same consecutive locations.

xi) Multiple_Writes_Multiple_Reads_RandomLocations_Test: This task will be called in the ‘execute’ task in the axi4_tester class while the ‘execute’ task is called in the ‘axi4_environment’ class that is defined in the ‘axi4_env.sv’ file. This task calls ‘Readtask’ and ‘Writetask’ to perform multiple writes first to random(non-consecutive) locations and then multiple reads from the same random(non-consecutive) locations.

xii) Multiple_Writes_Single_Read_SameLocation_Test: This task will be called in the ‘execute’ task in the axi4_tester class while the ‘execute’ task is called in the ‘axi4_environment’ class that is defined in the ‘axi4_env.sv’ file. This task calls ‘Readtask’ and ‘Writetask’ to perform multiple writes to the same location and finally read from the same location.

xiii) Outofboundary_Memory_Access_Test: This task will be called in the ‘execute’ task in the axi4_tester class while the ‘execute’ task is called in the ‘axi4_environment’ class that is defined in the ‘axi4_env.sv’ file. This task calls ‘Readtask’ and ‘Writetask’ to try accessing a memory location which is off the limits of the available memory.

xiv) execute: This task is called in the ‘axi4_environment’ class that is defined in the ‘axi4_env.sv’ file. This task calls all the tasks listed above namely:

a) rand_write_op

b) rand_write_op
c) Readtask
d) Writetask
e) Write_Read_IntermediateLocation_Test
f) Write_Read_FirstLocation_Test
g) Write_Read_LastLocation_Test
h)Write_Read_Sametime_SameLocation_Test
i)Write_Read_Sametime_DifferentLocation_Test
j)Multiple_Writes_Multiple_Reads_ConsecutiveLocations_Test,
k)Multiple_Writes_Multiple_Reads_RandomLocations_Test,
l)Multiple_Writes_Single_Read_SameLocation_Test,
m)Outofboundary_Memory_Access_Test.

v) axi4_checker_obj.sv:

This file contains **axi4_checker** class that monitors the bfm signals and saves the write data in a local memory("Reference model") during a write transaction and compares the data present at the corresponding address location in the local memory with the data read from the slave during a read transaction while keeping track of number of mismatches and updating them in a score variable. This class contains a **monitor, scoreboard and a reference model**. Additionally, this class contains **3 tasks** and a **function** "new" which are briefly explained below:

1) **Functions:**
   i) new: This function will be called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. Calling this 'new' function will create an object of our axi4_checker class. The function 'new' maps the bfm instantiated in this class with the bfm shared by other classes.

2) **Tasks:**
   i) save_val: This task will be called in the 'execute' task in the axi4_checker class while the 'execute' task is called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. This task saves the write data(WDATA) at the corresponding address in the local memory when the write address is valid(AWADDR high) during a write transaction.

   ii) check_val: This task will be called in the 'execute' task in the axi4_checker class while the 'execute' task is called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. This task compares the data read from the slave(RDATA) with that of local memory at the corresponding read address during a valid read transaction(RVALID high) and updates the score in case of a mismatch.

   iii) execute: This task is called in the 'axi4_environment' class that is defined in the 'axi4_env.sv' file. This task calls all the tasks listed above namely:
   a) save_val
   b) check_val

vi) axi4_env.sv:

The ‘axi4_Coverage.sv’ file is included in this ‘axi4_env.sv’ file. The ‘axi4_env.sv’ file contains a single **class** called ‘axi4_environment’. In this environment class, class handles are created for the ‘axi4_tester’ class (defined in the ‘axi4_tb_objs.sv’ file), ‘axi4_Coverage’ class (defined in the ‘axi4_Coverage.sv’ file) and ‘axi4_checker’ class (defined in the ‘axi4_checker_obj.sv’ file). Additionally, there is a function ‘new’ and a task ‘execute’ also defined in this class. The details of the contents of the class are as follows:

1) **Functions:**
   i) new: This function ‘new’ maps the bfm instantiated in this class with the bfm shared by other classes.

2) **Tasks:**
   i) execute: This task creates the objects of the ‘axi4_tester’, ‘axi4_Coverage’ and ‘axi4_checker’ class by calling new() on the handles of these classes that were declared in this ‘axi4_environment’ class. By doing this, the function ‘new’ declared in the ‘axi4_tester’, ‘axi4_Coverage’ and ‘axi4_checker’ classes is being called here. Additionally, this task also calls the task ‘execute’ defined in the ‘axi4_tester’, ‘axi4_Coverage’ and ‘axi4_checker’ classes.

vii) axi4_ootb_top.sv:

The ‘axi4_env.sv’ file is included in this ‘axi4_ootb_top.sv’ file. This ‘axi4_ootb_top.sv’ file contains a **module** called ‘a_OOTB_TOP’. In this module, the BFM (‘axi4_lite_bfm’), Master (‘axi4_lite_master’) and Slave (‘axi4_lite_slave’) design modules are instantiated. The clock signal is defined and a handle of the ‘axi4_environment’ class is created. In this ‘a_OOTB_TOP’ module, we put our DUT in the default mode by asserting LOW the active low reset signal for 20 time units and then de-asserting the reset signal. An object of the ‘axi4_environment’ class is created by calling new() on the environment class handle that is declared in this class. Finally, the ‘execute’ task defined in the ‘axi4_environment’ class is called to perform testing of the DUT.

VIII. Future Plans

As mentioned prior, with additional time we could improve verification and coverage by increasing simulation time and solving the problem related to local memory arrays in SystemVerilog classes.

We could also improve coverage by including more parameters and constraints on our randomization. To go along with this, we could improve the testing environment by allowing simulation time selection of which variety of tests will be run. As the number of tests available increases, it would make less sense to execute them all simultaneously.

Another potential way to improve coverage would be to enhance the DUV to provide command ports to the AXI4 master module. At that point, we could include those ports to the BFM and drive the command ports in the master module, allowing the master module to handle all communications with the slave module directly. Our current verification strategy was to drive the lines on the BFM and control both the slave and master module in that way. We chose this strategy due to the expectations and constraints of the project, but with a little time the DUV's master module could be improved to provide a testing environment that would more accurately represent a real world use of the design.