Verification of ARM AMBA-3 AHB-lite Protocol

ECE-593 Fundamentals of Pre-Silicon Validation
Final Project Report
Spring -2020

AAYUSH RAVICHANDRAN [981120898]

KAUSTUBH MHATRE [974819541]

SAURABH WAMAN CHAVAN [911836716]

Contents

CHAPTER 1: INTRODUCTION **4**

1.1 PROTOCOL DESCRIPTION 4
1.2 SIGNAL DESCRIPTION 5
1.2.1 MASTER *5*
1.2.2 SLAVE *6*
1.2.3 GLOBAL SIGNALS *7*
1.2.4 DECODER *7*
1.2.5 MULTIPLEXOR *8*
1.3 OPERATION 8

CHAPTER 2: VERIFICATION PLAN **9**

2.1 VERIFICATION REQUIREMENTS 9
2.1.1 Verification Levels- *9*
2.1.2 Functions- *9*
2.1.3 Specific Tests & Methods- *10*
2.1.4 Coverage- *12*
2.1.5 Scenarios- *12*
2.2 PROJECT MANAGEMENT: 18
2.2.1 Tools- *18*
2.2.2 Risk/Dependencies- *18*
2.2.3 Resources- *19*
2.2.4 Schedule- *19*

CHAPTER 3: VERIFICATION ENVIRONMENT **20**

3.1 INTERFACE.SV 20
3.2 TRANSACTION.SV 21
3.3 GENERATOR.SV 24
3.4 DRIVER.SV 27
3.5 COVERAGE_AHB.SV 28
3.6 MONITOR.SV 31
3.7 SCOREBOARD.SV 31
3.8 ENV.SV [ENVIRONMENT CLASS] 32
3.9 TB_TOP.SV 33
3.10 TOP.SV 33
3.11 TOP_MAIN.SV 34
3.12 PROBLEMS FACED 34

CHAPTER 4: RESULTS AND COVERAGE **35**

4.1 RESULTS 35
4.2 VERIFIED AND NON-VERIFIED FUNCTIONS 38
4.4 WAVE OUTPUT 39
4.4 COVERAGE 41

CHAPTER 5: REFERENCES **45**

List of figures

Figure 1: Block diagram of AMBA-3 AHB-Lite protocol 4
Figure 2: Master Interface 5
Figure 3: Slave Interface 6
Figure 4: Decoder Interface 7
Figure 5: Multiplexor Interface 8
Figure 6: Single burst 39
Figure 7: INCR burst 39
Figure 8: WRAP4 burst 40
Figure 9: INCR4 burst 40
Figure 10: COVERGROUP report 44

CHAPTER 1: INTRODUCTION

1.1 PROTOCOL DESCRIPTION

AMBA AHB-Lite is a bus interface that supports a single bus master and provides high-bandwidth operation. It is used in high-performance synthesizable designs.

AHB-Lite implements the features required for high-performance, high clock frequency systems including:
• burst transfers
• single-clock edge operation
• non-tristate implementation
• wide data bus configurations, 64, 128, 256, 512, and 1024 bits.

The most common AHB-Lite slaves are internal memory devices, external memory interfaces, and high bandwidth peripherals.

The main component types of an AHB-Lite system are:
1. Master
2. Slave
3. Decoder
4. Multiplexor

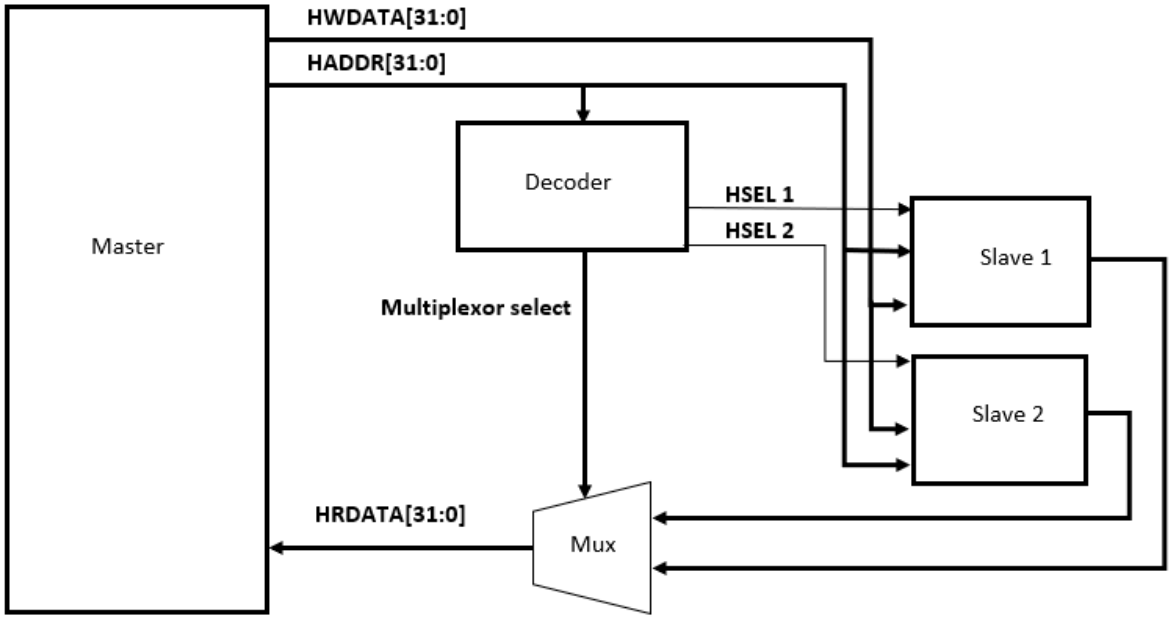

Figure 1: Block diagram of AMBA-3 AHB-Lite protocol

1.2 SIGNAL DESCRIPTION

1.2.1 MASTER

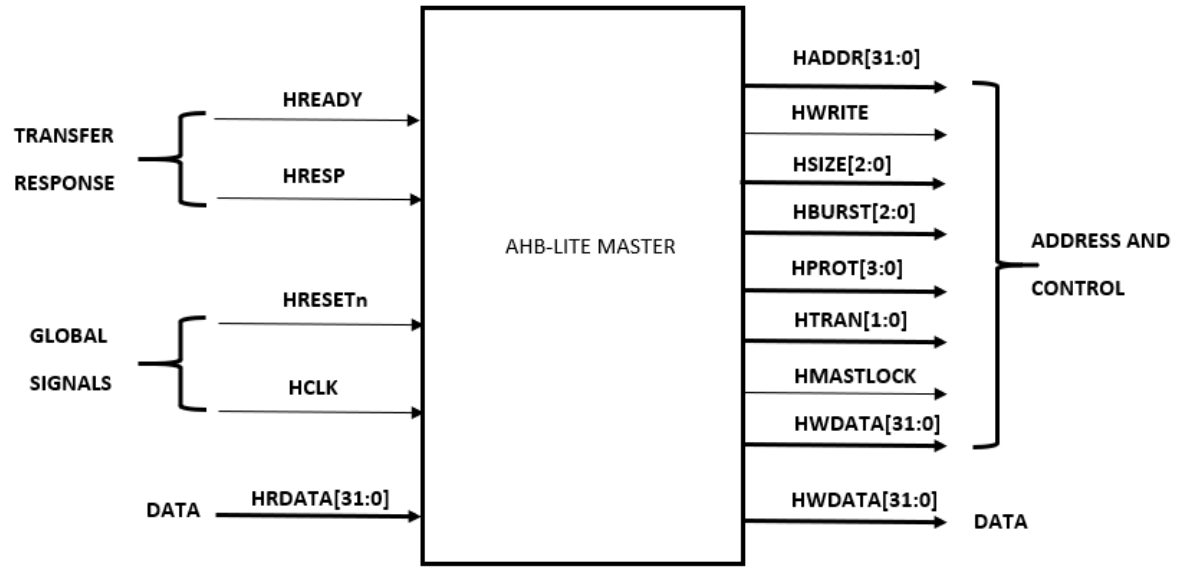

Figure 2: Master Interface

Name	Destination	Description
HADDR[31:0]	Slave and decoder	32 bit system address bus
HBURST[2:0]	Slave	Indicates if the transfer type is a burst of single type or burst of 4, 8 or 16 beats. Burst can be incrementing or wrapping(wrapping burst wrap around a address boundary)
HMASTLOCK	Slave	When asserted HIGH indicates that the transfer sequence is locked and current transfer is indivisible. No other transfer other than the locked sequence can be processed. Note: It is not implemented in the current design and verification
HPROT[3:0]	Slave	Provides additional information about the bus access. Note: It is not implemented in the current design and verification
HSIZE[2:0]	Slave	Indicates the transfer size. Typical sizes are, byte, half word or word. It must be equal or less than the size of the bus. It is used in conjunction with the HBURST and has the timing which is same as address bus.
HTRANS[1:0]	Slave	Indicates the transfer type is: 1. IDEAL 2. BUSY 3. NONSEQUENTIAL 4. SEQUENTIAL
HWDATA[31:0][a]	Slave	Indicates write data of minimum 32 bits from Master to Slave
HWRITE	Slave	Indicates direction of transfer. When HIGH the direction is from Master to Slave and when LOW it indicates read operation from slave to master.

1.2.2 SLAVE

A Slave selected using the HSELx signal using the Decoder controls the transfer progress. Slave response signalling consists of the following three types:

1. It can immediately complete the transfer
2. Insert WAIT states for the transfers to continue
3. Provide error signalling to indicate the transfer has failed

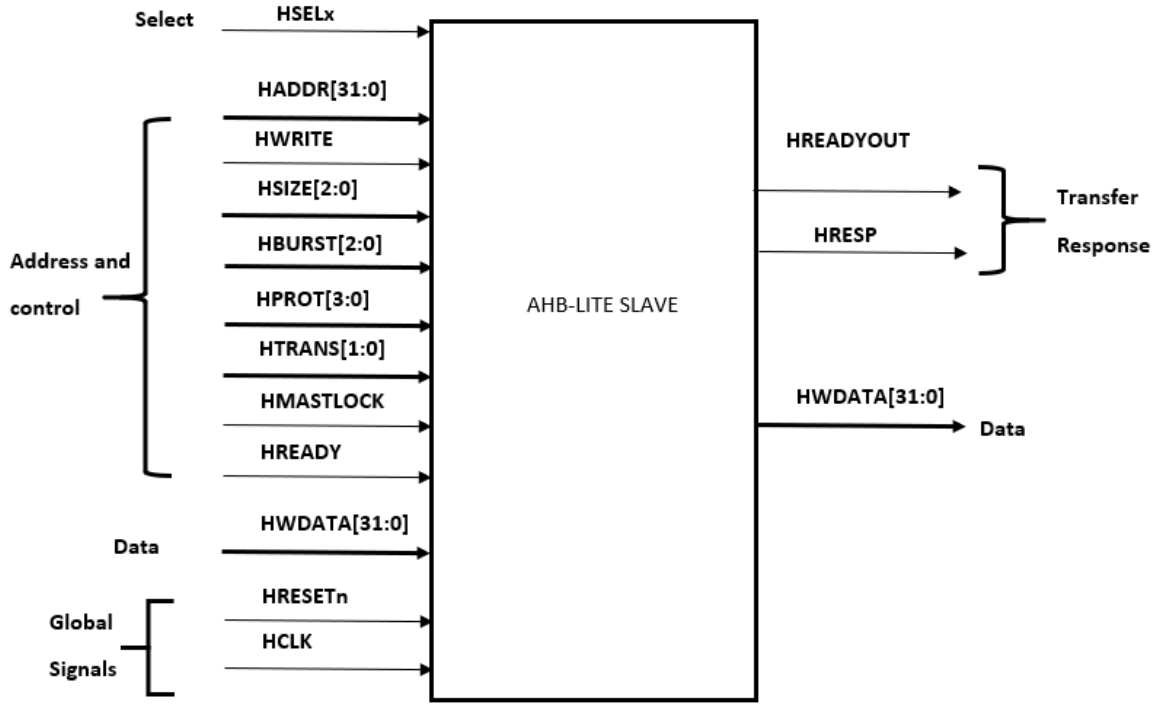

Figure 3: Slave Interface

Name	Destination	Description
HRDATA[31:0][a]	Multiplexor	During read operation, the data is transferred from slave to multiplexor and then from multiplexor to the master.
HREADYOUT	Multiplexor	When HIGH it indicates that the transfer has finished on the bus. When LOW the current transfer is extended.
HRESP	Multiplexor	Additional information can be provided on the transfer status. LOW indicates that the transfer status is OKAY HIGH indicates that the transfer status is ERROR

1.2.3 GLOBAL SIGNALS

Name	Destination	Description
HCLK	Clock source	It is a single clock signal. All signals are sampled at the rising edge of the clock. Output signal changes must occur after the rising edge of HCLK
HRESETn	Reset Controller	It is the only active LOW signal. It resets the bus and the system. The reset can be asserted asynchronously, it is deasserted synchronously after the rising edge of HCLK

1.2.4 DECODER

The central decoder provides HSELx slave select signal for each slave of interest for transfer.

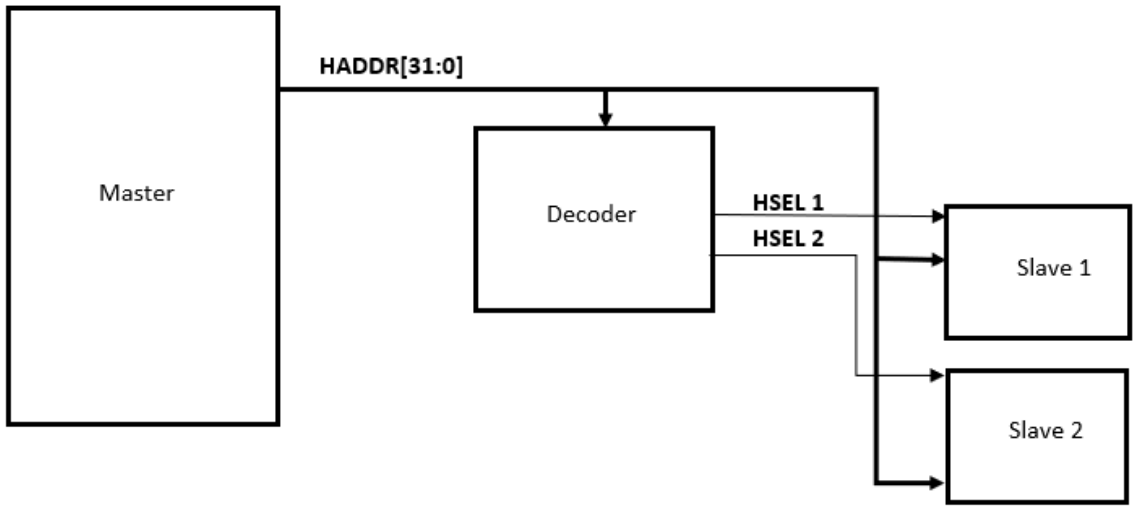

Figure 4: Decoder Interface

Name	Destination	Description
HSELx[a]	Slave	Each slave has this signal which indicates its selection. HREADY signal is monitored when a slave is selected initially to ensure the previous transfer completion

The decoder also provides the signal HSELx to the Multiplexor so that the Multiplexor routes the signals from the selected Slave to the Master.

1.2.5 MULTIPLEXOR

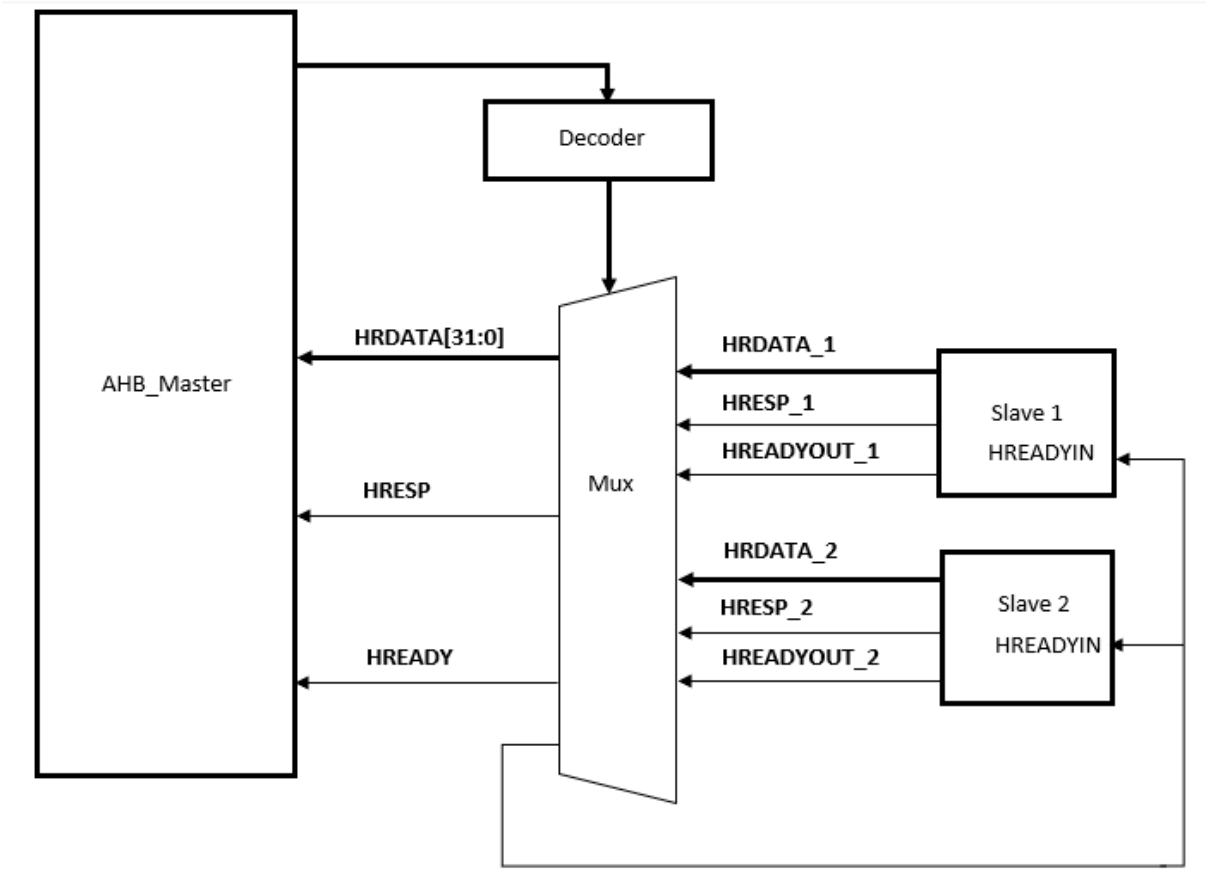

Figure 5: Multiplexor Interface

Name	Destination	Description
HRDATA[31:0]	Master	Read data bus as selected by the decoder
HREADY	Master and Slave	HREADY when asserted HIGH indicates that the previous transfer is complete
HRESP	Master	Response signal of the transfer selected by the decoder

1.3 OPERATION

The Master drives the control and address signals when it starts a transfer. The transfer consists of two phases:

1. Address phase: One address and control cycle
2. Data Phase: one or more cycles for the data

The transfers can be:

1. Single
2. Incremental burst
3. Wrapping burst which wrap at address boundaries

CHAPTER 2: VERIFICATION PLAN

2.1 Verification Requirements

2.1.1 Verification Levels-

The AMBA 3 AHB is a simple bus interface. In this particular design there are one master and two slaves. Hence, we only need to verify at two levels. One is where the smaller design unit level module will be individually verified and then a top-level verification of the complete interface.

1. **Unit level verification**

   This level of verification was chosen because it would be simple to thoroughly check the working and functionality of each individual design modules before we move to the top-level verification. The AHB lite bus design consist of the following unit level components that could be verified

   a) Memory Controller
   b) Decoder
   c) Multiplexer
   d) Memory module/ slave

2. **Chip level verification**

   This level of verification will test the interconnection between the different modules and that the entire interface is working as expected. Since the individual components were verified at the earlier level this makes verification at this level only verify the interworking between them. The top module instantiates all the unit level components and so is suitable for another level of verification. It needs to be verified for proper connectivity and functionality. Deterministic as well as random testcases are tested using class based testbench.

2.1.2 Functions-

The following are the functions and the appropriate action that are verified

- Different transfer types / burst operations (Single, INCR, INCR4, INCR8, INCR16, WRAP4, WRAP8, WRAP16) for both Read and Write operations.
- Waited operations such as master inserting Idle or Busy.
- Check if the two slave devices/ memory are being selected properly or not.
- Check if multiplexer is working or not.
- Check if memory module works as expected or not for both read and writes.

There is some functionality that won't be checked or some that have been limited because of the limitation in the design such as

- Protection types won't be verified.
- Locking operations won't be verified.
- Address and Data will be restricted to 8 bits.
- The multi slave functionality will only be tested for top level and after all individual unit levels have been tested.

2.1.3 Specific Tests & Methods-

1. **Type of Verification**

   A black box approach will be taken to thoroughly verify the design. The stimulus will be driven into the design and a self-checking scoreboard checks the output.

2. **Verification Strategy**

   A mix of deterministic and constrained random strategy will be used to verify the design. The memory controller, decoder, multiplexer and the memory module will be verified with deterministic tests. Similarly, for the top level a mix of the two methods will be used along with a self-checking testbench.

   In the unit level testing of the memory controller a write operation is followed by a read operation hence verifying both the operations. But for the top level a self-checking scoreboard performs all operations to its local memory array and then compares the output of the design to its own.

3. **Abstraction Level**

   For the top-level verification, the generator has cycle specific values for all master signals which are then driven into the interface by the driver.

4. **Checking**

   A golden vector and reference model type of self-checking testbench will be used to verify the design. The golden vector model will be easier to use for unit testing and for deterministic testing of the design modules. As we move to the top-level verification and perform many combinations of the operations it will be better to move to the reference model as we can just monitor the interface pins and check what results should be produced. The memory controller will be thoroughly checked to verify that the protocol is being followed as per the specification. To summarise.

   For the top-level verification, a reference model is used where it performs the operation and then compares its results to that of the design.

   For the unit level verification, a mix of the two methods are used. Like for the memory controller a write operation performed followed by a read

operation to the same address and hence its verified. For other modules expected results are compared with that of the design.

Some of the protocol specific checkers are as follows

When reset all signals must be at their default values
When master is in BUSY state the present operation is halted and it remains halted until master remains in the BUSY state
For IDLE transfer type no data transfer should occur and any pending transaction should be terminated
For BUSY transfer type when used the address and control signals should reflect the next transfer
Only burst of undefined length (INCR) can have a BUSY transfer as the last cycle of burst
The control information for a SEQ transfer type is identical to previous transfer (SEQ transfer type happens for burst operations after the first cycle where the control information is given along with address and data and remains SEQ till the operation is complete)
For SEQ transfer type the next address is addition of previous address plus the transfer size in bytes (the controller calculates the next address for the operation by adding the transfer size to the current address)
HSIZE signal should remain constant throughout the transfer (the transfer size should remain constant for the entire operation)
For wrapping burst transfer when address crosses the address boundary, they should wrap around
A master should not attempt to start incrementing burst that crosses a 1KB address boundary
All transfers in a burst must be aligned to the address boundary equal to the size of transfer
Master cannot perform a BUSY transfer immediately after a SINGLE operation
Transfers of fixed length burst types must terminate with a SEQ transfer
When a wait state is introduced the master can only change transfer type and address

2.1.4 Coverage-

Code and functional coverage will be used to find interesting points like what all different control signals were used, different operations performed, different types of operations performed and different transfer sizes and burst, etc as well as different combinations of the operations. As per the test plan come coverage components will be limited in some capacity such as address and data to 8 bits for reduced test cases and better results.

2.1.5 Scenarios-

Some of the scenarios are as follows

TEST NUMBER	TEST CASE NAME
	Single Read Operation
1	Read operation in Single burst mode when master is idle or busy and slave is ready (Here we perform a Read operation with HBURST = SINGLE and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = IDLE or BUSY for the second cycle)
2	Read operation in Single burst mode when master is ready and slave is ready (Here we perform a Read operation with HBURST = SINGLE and HTRANS = NONSEQ for the first cycle)
	Single Write Operation
3	Write operation in Single burst mode when master is idle or busy and slave is ready (Here we perform a Write operation with HBURST = SINGLE and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = IDLE or BUSY for the second cycle)
4	Write operation in Single burst mode when master is ready and slave is ready (Here we perform a Write operation with HBURST = SINGLE and HTRANS = NONSEQ for the first cycle)
	INCR Burst Read Operation

5	Read operation in INCR burst mode when master is not ready and slave is ready (Here we perform a Read operation with HBURST = INCR and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and then we introduce an HTRANS = IDLE or BUSY in one of the following cycles)
6	Read operation in INCR burst mode when both master and slave are ready (Here we perform a Read operation with HBURST = INCR and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next n cycles)
	INCR4 Burst Read Operation
7	Read operation in INCR4 burst mode when master is not ready and slave is ready (Here we perform a Read operation with HBURST = INCR4 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
8	Read operation in INCR4 burst mode when both master and slave are ready (Here we perform a Read operation with HBURST = INCR4 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 2 cycles)
	INCR8 Burst Read Operation
9	Read operation in INCR8 burst mode when master is not ready and slave is ready (Here we perform a Read operation with HBURST = INCR8 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
10	Read operation in INCR8 burst mode when both master and slave are ready (Here we perform a Read operation with HBURST = INCR8 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 6 cycles)

	INCR16 Burst Read Operation
11	Read operation in INCR16 burst mode when master is not ready and slave is ready (Here we perform a Read operation with HBURST = INCR16 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
12	Read operation in INCR16 burst mode when both master and slave are ready (Here we perform a Read operation with HBURST = INCR16 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 14 cycles)
13	Read operation in INCR, INCR4, INCR8 or INCR16 burst mode when both master and slave are ready and transaction happens beyond the range specified by the burst operation
	INCR Burst Write Operation
14	Write operation in INCR burst mode when master is not ready and slave is ready (Here we perform a Write operation with HBURST = INCR and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and then we introduce an HTRANS = IDLE or BUSY in one of the following cycles)
15	Write operation in INCR burst mode when both master and slave are ready (Here we perform a Write operation with HBURST = INCR and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next n cycles)
	INCR4 Burst Write Operation
16	Write operation in INCR4 burst mode when master is not ready and slave is ready (Here we perform a Write operation with HBURST = INCR4 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)

17	Write operation in INCR4 burst mode when both master and slave are ready (Here we perform a Write operation with HBURST = INCR4 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 2 cycles)
	INCR8 Burst Write Operation
18	Write operation in INCR8 burst mode when master is not ready and slave is ready (Here we perform a Write operation with HBURST = INCR8 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
19	Write operation in INCR8 burst mode when both master and slave are ready (Here we perform a Write operation with HBURST = INCR8 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 6 cycles)
	INCR16 Burst Write Operation
20	Write operation in INCR16 burst mode when master is not ready and slave is ready (Here we perform a Read operation with HBURST = INCR16 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
21	Write operation in INCR16 burst mode when both master and slave are ready (Here we perform a Read operation with HBURST = INCR16 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 14 cycles)
22	Write operation in INCR, INCR4, INCR8 or INCR16 burst mode when both master and slave are ready and transaction happens beyond the range specified by the burst operation

	WRAP4 Burst Read Operation
23	Read operation in WRAP4 burst mode when master is not ready and slave is ready (Here we perform a Read operation with HBURST = WRAP4 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
24	Read operation in WRAP4 burst mode when both master and slave are ready (Here we perform a Read operation with HBURST = WRAP4 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 2 cycles)
	WRAP8 Burst Read Operation
25	Read operation in WRAP8 burst mode when master is not ready and slave is ready (Here we perform a Read operation with HBURST = WRAP8 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
26	Read operation in WRAP8 burst mode when both master and slave are ready (Here we perform a Read operation with HBURST = WRAP8 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 6 cycles)
	WRAP16 Burst Read Operation
27	Read operation in WRAP16 burst mode when master is not ready and slave is ready (Here we perform a Read operation with HBURST = WRAP16 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
28	Read operation in WRAP16 burst mode when both master and slave are ready (Here we perform a Read operation with HBURST = WRAP16 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on

	for the next 14 cycles)
29	Read operation in WRAP4, WRAP8 or WRAP16 burst mode when both master and slave are ready and transaction happens beyond the range specified by the burst operation
	WRAP4 Burst Write Operation
30	Write operation in WRAP4 burst mode when master is not ready and slave is ready (Here we perform a Write operation with HBURST = WRAP4 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
31	Write operation in WRAP4 burst mode when both master and slave are ready (Here we perform a Write operation with HBURST = WRAP4 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 2 cycles)
	WRAP8 Burst Write Operation
32	Write operation in WRAP8 burst mode when master is not ready and slave is ready (Here we perform a Write operation with HBURST = WRAP8 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
33	Write operation in WRAP8 burst mode when both master and slave are ready (Here we perform a Write operation with HBURST = WRAP8 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 6 cycles)
	WRAP16 Burst Write Operation
34	Write operation in WRAP16 burst mode when master is not ready and slave is ready (Here we perform a Write operation with HBURST = WRAP16 and HTRANS =

	NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on but for some cycles in between HTRANS = INDE or BUSY is asserted and then the operation resumes till it's completed)
35	Write operation in WRAP16 burst mode when both master and slave are ready (Here we perform a Write operation with HBURST = WRAP16 and HTRANS = NONSEQ for the first cycle and then introduce an HTRANS = SEQ for the second cycle and so on for the next 14 cycles)
36	Write operation in WRAP4, WRAP8 or WRAP16 burst mode when both master and slave are ready and transaction happens beyond the range specified by the burst operation
	Different Combinations of Above operations
37	Write operations followed by Read operations (of the same Burst transfer type and of different types)
38	Read operations followed by Write operations
39	Read operations followed by other Read operations
40	Write operations followed by other Write operations

2.2 Project management:

2.2.1 Tools-

A. QuestaSim
B. Modelsim PE Students Edition 10.4a

2.2.2 Risk/Dependencies-

- This plan does not have any specific risks / dependencies.
- But the short schedule forms a part of risk factor for completion of project.
- The new way of implementing testbench environment can be a potential risk due to lack of familiarity to this implementation.

2.2.3 Resources-

The team will consist of the following three members

A. Aayush Ravichandran (981120898)
B. Kaustubh Herambh Mhatre (974819541)
C. Saurabh Waman Chavan (9811836716)

2.2.4 Schedule-

A. Week1- Unit level testing of unit level components
B. Week2- Self checking testbench for memory controller
C. Week3- Verify top level module & multiple slaves
D. Week4- Revisit verification plan check for missed test cases

CHAPTER 3: VERIFICATION ENVIRONMENT

3.1 interface.sv

1. This block encapsulates all the signal required by the DUV.

```
interface inf(input  HCLK);

    //Signals for interaction with DUT
    logic                           HRESETn;
    logic   [DATAWIDTH-1:0]         HWDATA;
    logic   [ADDRWIDTH-1:0]         HADDR;
    logic   [DATATRANFER_SIZE-1:0] HSIZE;
    BType_t                         HBURST;
    Trans_t                         HTRANS;
    logic                           HWRITE, HMASTLOCK, HREADY;
    logic   [3:0]                   HPROT;

    logic   [DATAWIDTH-1:0]         HRDATA;
    Response_t                      HRESP;

    modport DUT (
                    input  HCLK, HRESETn,
                    input  HWDATA,
                    input  HADDR,
                    input  HSIZE,
                    input  HBURST,
                    input  HTRANS,
                    input  HWRITE, HMASTLOCK, HREADY,
                    input  HPROT,
                    output HRDATA,
                    output HRESP
                );
```

3.2 transaction.sv

The transactions are based on the following BURST transfer and the transfer types defined in the protocol.

BURST

The protocol defines the BURSTS of 4, 8, and 16-beats, undefined burts, and single transfers. Support for incrementing and wrapping burst is provided

HBURST[2:0]	**Type**	**Description**
b000	SINGLE	Single burst
b001	INCR	Incrementing burst of undefined length
b010	WRAP4	4-beat wrapping burst
b011	INCR4	4-beat incrementing burst
b100	WRAP8	8-beat wrapping burst
b101	INCR8	8-beat incrementing burst
b110	WRAP16	16-beat wrapping burst
b111	INCR16	16-beat incrementing burst

TRANSFER TYPES

The transfer types supported in the protocol are as follows:

TYPE	DESCRIPTION
IDLE	It indicates that no data transfer is required. Master uses and IDLE transfer when it does not want to perform a data transfer.Slaves must always provide a zero wait state OKAY response to IDLE transfers and the transfer must be ignored by the slave
BUSY	The BUSY transfer enables masters to insert idle cycles in the middle of a burst. The next transfer cannot take place immediately. When a master uses the BUSY transfer type the address and control signals must reflect the next transfer in the burst. Slaves must always provide a zero wait state OKAY response to BUSY transfers and the transfer must be ignored by the slave.
NONSEQ	Indicates a single transfer or the first transfer of a burst. The address and control signals are unrelated to the previous transfer. Single transfers on the bus are treated as bursts of length one and therefore the transfer type is NONSEQUENTIAL.
SEQ	The remaining transfers in a burst are SEQUENTIAL and the address is related to the previous transfer. The control information is identical to the previous transfer. The address is equal to the address of the previous transfer plus the transfer size, in bytes, with the transfer size being signaled by the HSIZE[2:0] signals. In the case of a wrapping burst the address of the transfer wraps at the address boundary.

This contains class called txn

`rand logic HRESETn;` `rand logic [31:0] HWDATA;` `rand logic [31:0] HADDR;` `rand logic [2:0] HSIZE;` `rand logic [2:0] HBURST;` `logic [1:0] HTRANS;` `logic HWRITE;` `rand logic HMASTLOCK;` `rand logic HREADY;` `rand logic [3:0] HPROT;` `logic [31:0] HRDATA;` `logic HRESP;` `randc logic [3:0] test_case;`	Data members have been declared as rand/randc in order obtain random values.
`//Constraint on HWDATA` `constraint legal_data {` `HWDATA <= 239;` `HWDATA >= 0;` `}`	constraints for data members of HWDATA, HADDR, HSIZE, HMASTLOCK, HREADY and HBURST
`//Constraint on HADDR` `constraint legal_addr_s1{` `HADDR <= 239;` `HADDR >= 0;` `}`	For accessing Memory 1
`constraint legal_addr_s2{` `HADDR_s2 >= 256;` `HADDR_s2 <= 478;` `}`	For accessing Memory 2
`//Constraint on HSIZE` `constraint legal_size {` `HSIZE == 3'b001;` `}`	Keeping HSIZE constant (although not part of design implementation)

`` //Constraint on HMASTLOCK constraint legal_Mast_Loc { HMASTLOCK == 1'b0; } ```	Keeping HMASTLOCK constant (although not part of design implementation)
`` //Constraint on selecting type of operation constraint tests { test_case <= 15; test_case >= 0; } ```	For selecting testcase in generator

3.3 generator.sv

1. This file contains a class called gen

`class gen;` `  txn pkt;` `  txn t;`	Object of transaction class “txn” is created
`//RANDOM OPERATIONS` `repeat(40000)` `begin` `  pkt=new();` `  assert(pkt.randomize);` `  //pkt.print("Generator data and address ");` `  d[1] = pkt.HWDATA;` `  d[2] = pkt.HWDATA + 1;` `  d[3] = pkt.HWDATA + 2;` `  d[4] = pkt.HWDATA + 3;` `  d[5] = pkt.HWDATA + 4;` `  d[6] = pkt.HWDATA + 5;` `  d[7] = pkt.HWDATA + 6;` `  d[8] = pkt.HWDATA + 7;` `  d[9] = pkt.HWDATA + 8;` `  d[10] = pkt.HWDATA + 9;` `  d[11] = pkt.HWDATA + 10;` `  d[12] = pkt.HWDATA + 11;` `  d[13] = pkt.HWDATA + 12;` `  d[14] = pkt.HWDATA + 13;` `  d[15] = pkt.HWDATA + 14;` `  d[16] = pkt.HWDATA + 15;`	Object of transaction class txn is randomized

```
//Task to drive data to the driver from generator
task driver_send(
                    input
                        logic                           HRESETn,
                        logic   [DATAWIDTH-1:0]         HWDATA,
                        logic   [ADDRWIDTH-1:0]         HADDR,
                        logic   [DATATRANFER_SIZE-1:0] HSIZE,
                        BType_t                         HBURST,
                        Trans_t                         HTRANS,
                        logic                           HWRITE, HMASTLOCK, HREADY,
                        logic   [3:0]                   HPROT

                );
    t = new();
    t.HRESETn   =   HRESETn;
    t.HWDATA    =   HWDATA;
    t.HADDR     =   HADDR;
    t.HSIZE     =   HSIZE;
    t.HBURST    =   HBURST;
    t.HTRANS    =   HTRANS;
    t.HWRITE    =   HWRITE;
    t.HMASTLOCK =   HMASTLOCK;
    t.HREADY    =   HREADY;
    t.HPROT     =   HPROT;
    t.HRDATA    =   HRDATA;
    t.HRESP     =   HRESP;

    mbx.put(t);

endtask
```

The random values generated are passed to the driver class via mailbox. This is done inside a method called “driver_send”

2. The method driver_send is called in order to generate deterministic as well as random test cases.

Random testing

1. Various random packets are written as per the protocol BURSTS transfer and the transfer types.
2. The random packets are randomly selected from the transactor class where they are constrained based on the number of random packets written by us.
3. Following code snippet consists of the random packets for SINGLE and INCR burst transfer. In the same way packets are written for other BURST transfer types along with the respective transfer types as per the protocol norms.

```
unique case(pkt.test_case)
    4'd0: begin
        //SINGLE WRITE
        driver_send(1,  d[1],   pkt.HADDR,  0,  SINGLE,     NONSEQ,     1,  0,  1,  0);
        end
    4'd1: begin
        //SINGLE READ
        driver_send(1,      0,  pkt.HADDR,  0,  SINGLE,     NONSEQ,     0,  0,  1,  0);
        end
    4'd2: begin
        //INCR WRITE
        driver_send(1,      0,  pkt.HADDR,  0,    INCR,     NONSEQ,     1,  0,  1,  0);
        driver_send(1,   d[2],  pkt.HADDR,  0,    INCR,        SEQ,     1,  0,  1,  0);
        end
    4'd3: begin
        //INCR READ
        driver_send(1,          0,  pkt.HADDR,  0,    INCR,     NONSEQ,     0,  0,  1,  0);
        driver_send(1, pkt.HWDATA,  pkt.HADDR,  0,    INCR,        SEQ,     0,  0,  1,  0);
        end
```

Deterministic testing

1. The packets for deterministic testing are sent through driver_send function where a packet for example SINGLE burst trnasfer is being written for a particular address and then the packet for read from the same location is being driven. Similar packets are written for the other types of the BURST transfers with the respective transfer types as per the protocol.

```
//Single Write followed by Single read for a specific address
//$display("Single Write followed by Single read for a specific address");
driver_send(1,  d[1],           pkt.HADDR,  0,  SINGLE,     NONSEQ,     1,  0,  1,  0);
driver_send(1,  0, pkt.HADDR,   0,  SINGLE,     NONSEQ,     0,  0,  1,  0);

//INCR Write
//$display("INCR Write");
driver_send(1,   0,             pkt.HADDR,  0,    INCR,     NONSEQ,     1,  0,  1,  0);
driver_send(1,  d[2],           pkt.HADDR,  0,    INCR,        SEQ,     1,  0,  1,  0);

//INCR Read
//$display("INCR Read");
driver_send(1,  0,          pkt.HADDR,  0,    INCR,     NONSEQ,     0,  0,  1,  0);
driver_send(1,  pkt.HWDATA, pkt.HADDR,  0,    INCR,        SEQ,     0,  0,  1,  0);

//INCR4 Write
//$display("INCR4 Write");
driver_send(1,  0,          pkt.HADDR,  0,    INCR4,    NONSEQ,     1,  0,  1,  0);
for(int i = 1; i<=4; i = i + 1)
begin
    driver_send(1,  d[i], pkt.HADDR,    0,    INCR4,       SEQ,     1,  0,  1,  0);
end
```

3.4 driver.sv

1. This file contains the driver class

`` //Virtual interface to link send data to DUT virtual inf vif; //Transaction object to collect data sent by generator txn pkt; //Mailbox to connect to generator mailbox mailbox mbx=new(); ```	Declaration of virtual interface vif, object of transaction txn class and mailbox mbx
`` //New function function new(mailbox mbx,virtual inf vif); this.mbx=mbx; this.vif=vif; endfunction //Run task task run(); pkt=new(); forever begin mbx.get(pkt); //pkt.print("Driver address and data"); driver_item(pkt); end endtask ```	The randomized data sent by the gen class is received by the driver class via mailbox
`` //Task to drive data sent from the generator to the DUT via virtual interface task driver_item(txn t); vif.HRESETn = t.HRESETn; vif.HWDATA = t.HWDATA; vif.HADDR = t.HADDR; vif.HSIZE = t.HSIZE; vif.HBURST = t.HBURST; vif.HTRANS = t.HTRANS; vif.HWRITE = t.HWRITE; vif.HMASTLOCK = t.HMASTLOCK; vif.HREADY = t.HREADY; vif.HPROT = t.HPROT; @(posedge vif.HCLK); endtask ```	The received data is passed to the DUV via the virtual interface

3.5 coverage_ahb.sv

1. This class contains a coverage class
2. It reads data at every positive edge of the clock from the virtual interface "vinf" and checks coverage.

`//OKAY or ERROR response` `slave_response: coverpoint HRESP {` `bins OKAY = {1'b0};` `}`	bins for slave response. HRESP will always be 0 in this design.
`burst: coverpoint HBURST { //BURST transfers` `bins SINGLE = {3'b000};` `bins INCR = {3'b001};` `bins WRAP4 = {3'b010};` `bins INCR4 = {3'b011};` `bins WRAP8 = {3'b100};` `bins INCR8 = {3'b101};` `bins WRAP16 = {3'b110};` `bins INCR16 = {3'b111};` `}`	bins for various BURST transfers mentioned in the AHB protocol. Checks if all modes of transfer are covered.
`transfer_type: coverpoint HTRANS { //transfer types` `bins IDLE = {2'b00};` `bins BUSY = {2'b01};` `bins NONSEQ = {2'b10};` `bins SEQ = {2'b11};` `}`	bins for transfer types. Checks if the master has undergone all transfer types.
`ready_master_slave: coverpoint HREADY {` `bins READY = {1'b1};` `bins WAIT = {1'b0};` `}`	bins for ready signal
`reset: coverpoint HRESETn { //reset signal ACTIVE LOW` `bins RESET_asserted = {1'b0};` `bins RESET_deasserted = {1'b1};` `}`	bins for reset signal. Checks if reset is toggled.
`data_size : coverpoint HSIZE { //size of a transfer` `bins BYTE = {3'b000};` `ignore_bins SIZE_INVALID[] = {[3'b001:3'b111]};` `}`	bins for transfer size. HSIZE will always be 0 in this design

`master_read_data: coverpoint HRDATA {  //data for read operation` `bins READ_DATA[] = {[0:255]} iff (HWRITE==0);` `}`	bins for the constrained data during read operation
`//data for write operation` `master_broadcast_data: coverpoint HWDATA {` `bins WRITE_DATA[]  = {[0:255]} iff (HWRITE==1);` `}`	bins for the costrained data during write operation
`//address for read operation` `master_read_addr: coverpoint HADDR {` `bins READ_addr[]  = {[0:510]} iff (HWRITE==0);` `}`	bins for the constrained address during read operation
`//address for write operation` `master_write_addr: coverpoint HADDR {` `bins WRITE_addr[]  = {[0:510]} iff (HWRITE==1);` `}`	bins for the constrained address during write operation
`//read or write operations` `data_operations: coverpoint HWRITE {` `bins READ  = {1'b0};` `bins WRITE  = {1'b1};` `}`	bins for the HWRITE signal

`` //read write transitions read_write_transitions: coverpoint HWRITE { bins READ_WRITE = (0 => 1); bins WRITE_READ = (1 => 0); bins READ_WRITE_SINGLE = (0 => 1) iff (HBURST==SINGLE); bins WRITE_READ_SINGLE = (1 => 0) iff (HBURST==SINGLE); bins READ_WRITE_INCR = (0 => 1) iff (HBURST==INCR); bins WRITE_READ_INCR = (1 => 0) iff (HBURST==INCR); bins READ_WRITE_INCR4 = (0 => 1) iff (HBURST==INCR4); bins WRITE_READ_INCR4 = (1 => 0) iff (HBURST==INCR4); bins READ_WRITE_INCR8 = (0 => 1) iff (HBURST==INCR8); bins WRITE_READ_INCR8 = (1 => 0) iff (HBURST==INCR8); bins READ_WRITE_INCR16 = (0 => 1) iff (HBURST==INCR16); bins WRITE_READ_INCR16 = (1 => 0) iff (HBURST==INCR16); bins READ_WRITE_WRAP4 = (0 => 1) iff (HBURST==WRAP4); bins WRITE_READ_WRAP4 = (1 => 0) iff (HBURST==WRAP4); bins READ_WRITE_WRAP8 = (0 => 1) iff (HBURST==WRAP8); bins WRITE_READ_WRAP8 = (1 => 0) iff (HBURST==WRAP8); bins READ_WRITE_WRAP16 = (0 => 1) iff (HBURST==WRAP16); bins WRITE_READ_WRAP16 = (1 => 0) iff (HBURST==WRAP16); } ```	bins for the read write transitions during the various BURST transfers
`` //transitions based on the generator scenarios burst_transitions: coverpoint HBURST { bins SINGLE_SINGLE = (SINGLE => SINGLE); bins SINGLE_INCR4 = (SINGLE => INCR4); bins SINGLE_INCR8 = (SINGLE => INCR8); bins SINGLE_INCR16 = (SINGLE => INCR16); bins SINGLE_WRAP4 = (SINGLE => WRAP4); bins SINGLE_WRAP8 = (SINGLE => WRAP8); bins SINGLE_WRAP16 = (SINGLE => WRAP16); ```	bins for the BURST transitions as per the generator

Cross coverage

`` //all responses for read and write transitions must be OKAY read_write_okay: cross read_write_transitions, slave_response; ```	cross between slave response and the read write transitions to see the slave response being OKAY in all read write transitions
`` //all burst types can have different data size from 1byte to 4bytes data_size_burst: cross burst, data_size; ```	cross between various BURST transfers and data size To see the data size being 1byte

3.6 monitor.sv

1. This class contains a monitor class

```
//Sample task to send data to the scoreboard via mailbox
task sample();
    @(posedge vif.HCLK);
        t.HRESETn    =   vif.HRESETn;
        t.HWDATA     =   vif.HWDATA;
        t.HADDR      =   vif.HADDR;
        t.HSIZE      =   vif.HSIZE;
        t.HBURST     =   vif.HBURST;
        t.HTRANS     =   vif.HTRANS;
        t.HWRITE     =   vif.HWRITE;
        t.HMASTLOCK  =   vif.HMASTLOCK;
        t.HREADY     =   vif.HREADY;
        t.HPROT      =   vif.HPROT;
        t.HRDATA     =   vif.HRDATA;
        t.HRESP      =   vif.HRESP;

  mon2scr.put(t);
endtask
```

Data is read from the virtual interface and sent to the scoreboards class via mailbox at every positive edge of the clock

3.7 scoreboard.sv

1. It gets data from the monitor via a mailbox
2. This file has a reference model which consists of a memory. Based on the values the values of the data received from the monitor operations are performed on this memory.
3. During any read operations it is checked whether the data read by the DUV matches the data present in the memory of this file. Following code snippet is a read part of WRAP4 checker logic. The logic checks whether the data for the wrapped address which was written is the same in the read part.

```
WRAP4   :begin
          if(ability == 1)
          begin
              waddress[1:0] = wrap_address[1:0] + addr_incr;
              addr_incr = addr_incr + 1;
              if(memory[waddress] == t.HRDATA)
              begin
                  $display("SUCCESSFULL ::WRAP4:: Actual data = %h, Testbench data = %h, testbech address = %h", t.HRDATA, memory[waddress], waddress, $time);
              end
              else
              begin
                  $display("FAILURE ::WRAP4:: Actual data = %h, Testbench data = %h, testbech address = %h", t.HRDATA, memory[waddress], waddress, $time);
              end
          end
```

3.8 env.sv [environment class]

1. it consists of the env class

`` class env; //Object of generator gen g; //Object of driver driver d; //Object of monitor monitor m; //Object of scoreboard scoreboard scr; //Object of coverage coverage cov; ```	Objects of generator, driver, monitor, scoreboard and coverage are created.
`` //New function function new(virtual inf tif); this.vif=tif; endfunction //Run task task run(); g=new(mbx); d=new(mbx,vif); m=new(vif,mbx2); scr=new(mbx2,vif); cov=new(vif); //Concurrent operation fork g.run(); d.run(); m.run(); scr.run(); cov.run(); join endtask ```	linking of these objects together via mailbox and virtual interface Run task is called in each of these objects **concurrently via fork join**

3.9 tb_top.sv

1. It consists of a program block named tb_top

```
program tb_top(inf intf);

    //Object of environment class
    env e;

    initial begin
        e = new(intf);
        e.run();
    end
endprogram
```

1. program block has an input as interface
2. It declares an object of environment class, passes this input interface to the environment class and calls its run task

3.10 top.sv

1. This file consists a module named AHB and has an input as interface.

```
module AHB(inf.DUT intf);

    //Instance of DUT
    AHB_TOP qwerty  (
                        .HCLK(intf.HCLK),
                        .HRESETn(intf.HRESETn),
                        .HWDATA(intf.HWDATA),
                        .HADDR(intf.HADDR),
                        .HSIZE(intf.HSIZE),
                        .HBURST(intf.HBURST),
                        .HTRANS(intf.HTRANS),
                        .HWRITE(intf.HWRITE),
                        .HMASTLOCK(intf.HMASTLOCK),
                        .HREADY(intf.HREADY),
                        .HPROT(intf.HPROT),
                        .HRDATA(intf.HRDATA),
                        .HRESP(intf.HRESP)
                    );


endmodule
```

This module instantiates the AHB_TOP module and passes all the interface signals to this instance.

3.11 top_main.sv

1. It consists of a top module.

```
module top;

    bit clk,reset,select;
    inf pif(clk);
    AHB u0 (pif);
    tb_top p0 (pif);

    initial
    begin
        #1000000    ;
        $stop;
    end

    always #5 clk=~clk;

endmodule
```

1. This module is responsible to generate a clock signal
2. It creates an interface block and passes a clock signal to it
3. It creates an instance of AHB module, tb_top program block and passes the interface to both of them.

3.12 Problems faced

1. The first problem faced was the order in which the various files of class based test bench had to be included in the module. After studying the structure in which different files got compiled, the correct order of including files was found.
2. There are multiple test cases present in the top level test bench. However tha main question was where were these test cases going to be written (generator or driver). If the test cases were written in driver the way in which driver sent signals to the DUV had to be hardcoded. This would limit the resuability feature of the test bench. So the driver was kept generic and various deterministic/ random test cases were listed in the generator.
3. There were multiple issues faced while designing the scoreboard. Most of them were based on synchronization issues. Issues were also faced in designing checking logic for SINGLE read/write operations as both the address and data phase for these operations had HTRANS as NONSEQ.

CHAPTER 4: RESULTS AND COVERAGE

4.1 RESULTS

TEST NUMBER	TEST CASE NAME	TEST RESULT
	Single Read Operation	
1	Read operation in Single burst mode when master is idle or busy and slave is ready	TEST PASS
2	Read operation in Single burst mode when master is ready and slave is ready	TEST PASS
	Single Write Operation	
3	Write operation in Single burst mode when master is idle or busy and slave is ready	TEST PASS
4	Write operation in Single burst mode when master is ready and slave is ready	TEST PASS
	INCR Burst Read Operation	
5	Read operation in INCR burst mode when master is not ready and slave is ready	TEST PASS
6	Read operation in INCR burst mode when both master and slave are ready	TEST PASS
	INCR4 Burst Read Operation	
7	Read operation in INCR4 burst mode when master is not ready and slave is ready	TEST PASS
8	Read operation in INCR4 burst mode when both master and slave are ready	TEST PASS
	INCR8 Burst Read Operation	
9	Read operation in INCR8 burst mode when master is not ready and slave is ready	TEST PASS
10	Read operation in INCR8 burst mode when both master and slave are ready	TEST PASS
	INCR16 Burst Read Operation	
11	Read operation in INCR16 burst mode when master is not ready and slave is ready	TEST PASS

12	Read operation in INCR16 burst mode when both master and slave are ready	TEST PASS
13	Read operation in INCR, INCR4, INCR8 or INCR16 burst mode when both master and slave are ready and transaction happens beyond the range specified by the burst operation	TEST PASS
	INCR Burst Write Operation	
14	Write operation in INCR burst mode when master is not ready and slave is ready	TEST PASS
15	Write operation in INCR burst mode when both master and slave are ready	TEST PASS
	INCR4 Burst Write Operation	
16	Write operation in INCR4 burst mode when master is not ready and slave is ready	TEST PASS
17	Write operation in INCR4 burst mode when both master and slave are ready	TEST PASS
	INCR8 Burst Write Operation	
18	Write operation in INCR8 burst mode when master is not ready and slave is ready	TEST PASS
19	Write operation in INCR8 burst mode when both master and slave are ready	TEST PASS
	INCR16 Burst Write Operation	
20	Write operation in INCR16 burst mode when master is not ready and slave is ready	TEST PASS
21	Write operation in INCR16 burst mode when both master and slave are ready	TEST PASS
22	Write operation in INCR, INCR4, INCR8 or INCR16 burst mode when both master and slave are ready and transaction happens beyond the range specified by the burst operation	TEST PASS
	WRAP4 Burst Read Operation	
23	Read operation in WRAP4 burst mode when master is not ready and slave is ready	TEST PASS
24	Read operation in WRAP4 burst mode when both master and slave are ready	TEST PASS
	WRAP8 Burst Read Operation	

25	Read operation in WRAP8 burst mode when master is not ready and slave is ready	TEST PASS
26	Read operation in WRAP8 burst mode when both master and slave are ready	TEST PASS
	WRAP16 Burst Read Operation	
27	Read operation in WRAP16 burst mode when master is not ready and slave is ready	TEST PASS
28	Read operation in WRAP16 burst mode when both master and slave are ready	TEST PASS
29	Read operation in WRAP4, WRAP8 or WRAP16 burst mode when both master and slave are ready and transaction happens beyond the range specified by the burst operation	TEST PASS
	WRAP4 Burst Write Operation	
30	Write operation in WRAP4 burst mode when master is not ready and slave is ready	TEST PASS
31	Write operation in WRAP4 burst mode when both master and slave are ready	TEST PASS
	WRAP8 Burst Write Operation	
32	Write operation in WRAP8 burst mode when master is not ready and slave is ready	TEST PASS
33	Write operation in WRAP8 burst mode when both master and slave are ready	TEST PASS
	WRAP16 Burst Write Operation	
34	Write operation in WRAP16 burst mode when master is not ready and slave is ready	TEST PASS
35	Write operation in WRAP16 burst mode when both master and slave are ready	TEST PASS
36	Write operation in WRAP4, WRAP8 or WRAP16 burst mode when both master and slave are ready and transaction happens beyond the range specified by the burst operation	TEST PASS

4.2 Verified and Non-verified functions

Verified functions:

1. All BURST types.
2. All transfer types.
3. Reset operation when asserted LOW.
4. Write and Read back to back operations.
5. In between IDLE state transfer type insertion during BURST transfers.
6. In between BUSY state insertion during BURST transfers.

Non Verified Functions:

1. Locked transfers.
2. Protection control.
3. Slave response ERROR signalling.
4. Transfer size other than 1 byte.

4.4 Wave Output

SINGLE BURST

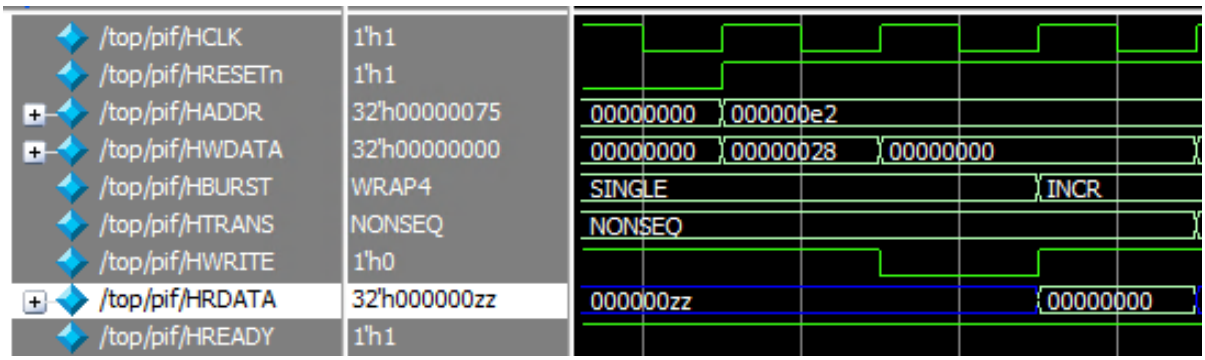

Figure 6: Single burst

1. In the first clock cycle which is the address phase the address 000000e2 and the control signals are given. The control signals are the HWRITE=1 [write operation] and the transfer type of NONSEQ as per the protocol.
2. In the second cycle the HWDATA write data 00000000 to the address location. The transfer type is NONSEQ during this data phase.
3. In the 3rd clock cycle the same address location is being given from which we will read the same data the HRDATA gives the data being written which is 00000000.

INCR BURST

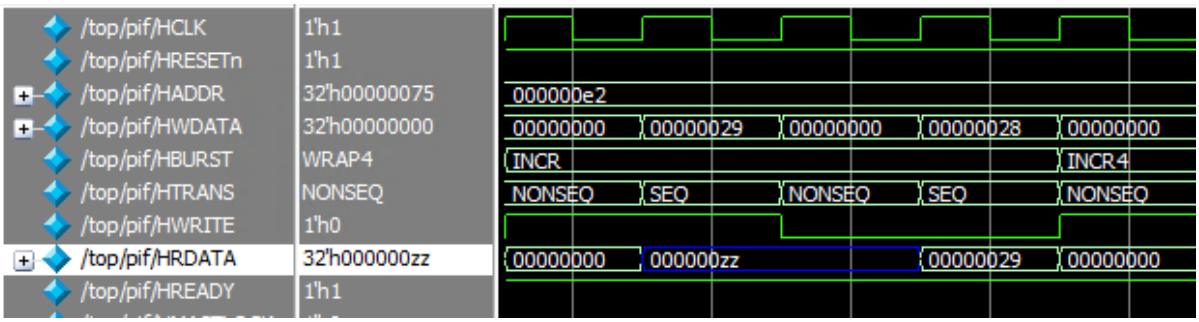

Figure 7: INCR burst

1. In the first clock cycle which is the address phase the address 000000e2 and the control signals are given. The control signals are the HWRITE=1 [write operation] and the transfer type of NONSEQ as per the protocol.
2. In the second cycle the HWDATA write data 00000029 to the address location. The transfer type is SEQ during this data phase.
3. In the 3rd clock cycle the same address location is being given from which we will read the same data being written.
4. In the 4th cycle the HRDATA gives the data being written which is 00000029.

WRAP 4 BURST

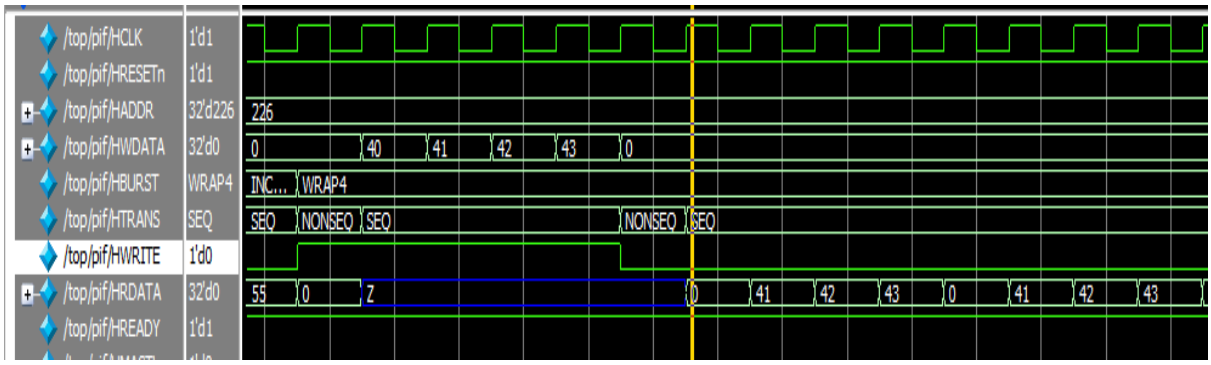

Figure 8: WRAP4 burst

1. In the first clock cycle which is the address phase the address 00000226 and the control signals are given. The control signals are the HWRITE=1 [write operation] and the transfer type of NONSEQ as per the protocol.
2. In the second cycle till the 5th cycle the HWDATA write data 00000040 through 00000043 is being written to the address location. The transfer type is SEQ during this data phase. The address is being wrapped after the last byte being written to the address location 00000226.
3. In the 6th clock cycle the address phase of the read operation will appear along with the control signal HWRITE=0 and the transfer type of NONSEQ as per the protocol.
4. From the 7th clock cycle till 10th clock cycle the HRDATA is being read which is 00000040 through 00000043 and the transfer type is SEQ as per the protocol.

INCR 4 BURST

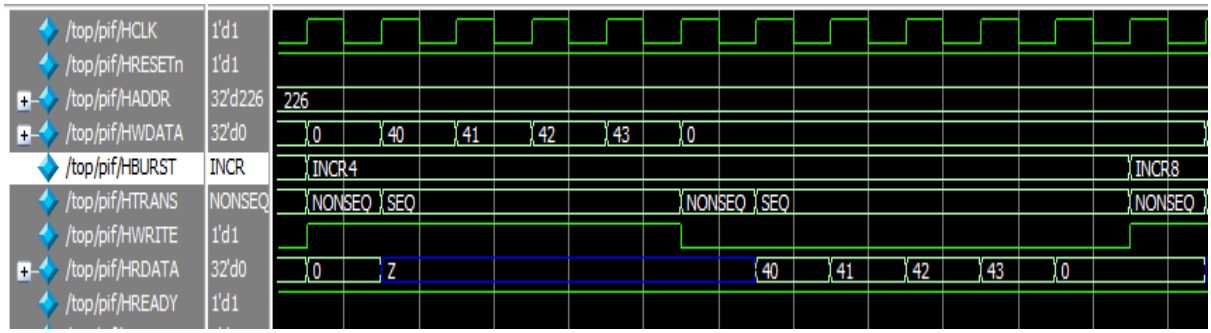

Figure 9: INCR4 burst

1. In the first clock cycle which is the address phase the address 00000226 and the control signals are given. The control signals are the HWRITE=1 [write operation] and the transfer type of NONSEQ as per the protocol.
2. In the second cycle till the 5th cycle the HWDATA write data 00000040 through 00000043 is being written to the address location 00000226 through 00000229. The transfer type is SEQ during this data phase.
3. In the 6th clock cycle the address phase of the read operation will appear along with the control signal HWRITE=0 and the transfer type of NONSEQ as per the protocol.
4. From the 7th clock cycle till 10th clock cycle the HRDATA is being read which is 00000040 through 00000043 and the transfer type is SEQ as per the protocol.

4.4 Coverage

COVERPOINT	COVERPOINT SPECIFICATION	COVERAGE
`//OKAY or ERROR response` `slave_response: coverpoint HRESP {` `bins OKAY = {1'b0};` `}`	bins for slave response	100%
`burst: coverpoint HBURST { //BURST transfers` `bins SINGLE = {3'b000};` `bins INCR = {3'b001};` `bins WRAP4 = {3'b010};` `bins INCR4 = {3'b011};` `bins WRAP8 = {3'b100};` `bins INCR8 = {3'b101};` `bins WRAP16 = {3'b110};` `bins INCR16 = {3'b111};` `}`	bins for various BURST transfers mentioned in the AHB protocol	100%
`transfer_type: coverpoint HTRANS { //transfer types` `bins IDLE = {2'b00};` `bins BUSY = {2'b01};` `bins NONSEQ = {2'b10};` `bins SEQ = {2'b11};` `}`	bins for transfer types	100%
`ready_master_slave: coverpoint HREADY {` `bins READY = {1'b1};` `bins WAIT = {1'b0};` `}`	bins for ready signal	100%
`reset: coverpoint HRESETn { //reset signal ACTIVE LOW` `bins RESET_asserted = {1'b0};` `bins RESET_deasserted = {1'b1};` `}`	bins for reset signal	100%
`data_size : coverpoint HSIZE { //size of a transfer` `bins BYTE = {3'b000};` `ignore_bins SIZE_INVALID[] = {[3'b001:3'b111]};` `}`	bins for transfer size	100%

`master_read_data: coverpoint HRDATA {  //data for read operation` `bins READ_DATA[] = {[0:255]} iff (HWRITE==0);` `}`	bins for the costrained data during read operation	99.21%
`//data for write operation` `master_broadcast_data: coverpoint HWDATA {` `bins WRITE_DATA[] = {[0:255]} iff (HWRITE==1);` `}`	bins for the costrained data during write operation	99.21%
`//address for read operation` `master_read_addr: coverpoint HADDR {` `bins READ_addr[] = {[0:510]} iff (HWRITE==0);` `}`	bins for the constrained address during read operation	46.96%
`//address for write operation` `master_write_addr: coverpoint HADDR {` `bins WRITE_addr[] = {[0:510]} iff (HWRITE==1);` `}`	bins for the constrained address during write operation	46.96%
`//read or write operations` `data_operations: coverpoint HWRITE {` `bins READ = {1'b0};` `bins WRITE = {1'b1};` `}`	bins for the HWRITE signal	100%

`` //read write transitions read_write_transitions: coverpoint HWRITE { bins READ_WRITE = (0 => 1); bins WRITE_READ = (1 => 0); bins READ_WRITE_SINGLE = (0 => 1) iff (HBURST==SINGLE); bins WRITE_READ_SINGLE = (1 => 0) iff (HBURST==SINGLE); bins READ_WRITE_INCR = (0 => 1) iff (HBURST==INCR); bins WRITE_READ_INCR = (1 => 0) iff (HBURST==INCR); bins READ_WRITE_INCR4 = (0 => 1) iff (HBURST==INCR4); bins WRITE_READ_INCR4 = (1 => 0) iff (HBURST==INCR4); bins READ_WRITE_INCR8 = (0 => 1) iff (HBURST==INCR8); bins WRITE_READ_INCR8 = (1 => 0) iff (HBURST==INCR8); bins READ_WRITE_INCR16 = (0 => 1) iff (HBURST==INCR16); bins WRITE_READ_INCR16 = (1 => 0) iff (HBURST==INCR16); bins READ_WRITE_WRAP4 = (0 => 1) iff (HBURST==WRAP4); bins WRITE_READ_WRAP4 = (1 => 0) iff (HBURST==WRAP4); bins READ_WRITE_WRAP8 = (0 => 1) iff (HBURST==WRAP8); bins WRITE_READ_WRAP8 = (1 => 0) iff (HBURST==WRAP8); bins READ_WRITE_WRAP16 = (0 => 1) iff (HBURST==WRAP16); bins WRITE_READ_WRAP16 = (1 => 0) iff (HBURST==WRAP16); } ```	bins for the read write transitions during the various BURST transfers	100%
`` //transitions based on the generator scenarios burst_transitions: coverpoint HBURST { bins SINGLE_SINGLE = (SINGLE => SINGLE); bins SINGLE_INCR4 = (SINGLE => INCR4); bins SINGLE_INCR8 = (SINGLE => INCR8); bins SINGLE_INCR16 = (SINGLE => INCR16); bins SINGLE_WRAP4 = (SINGLE => WRAP4); bins SINGLE_WRAP8 = (SINGLE => WRAP8); bins SINGLE_WRAP16 = (SINGLE => WRAP16); ```	bins for the BURST transitions as per the generator	100%

CROSS COVERAGE RESULTS

`` //all responses for read and write transitions must be OKAY read_write_okay: cross read_write_transitions, slave_response { ignore_bins other = binsof(read_write_transitions) && binsof(slave_response.ERROR); } ```	cross between slave response and the read write transitions	100%
`` //all burst types can have different data size from 1byte to 4bytes data_size_burst: cross burst, data_size; ```	cross between various BURST transfers and data size	100%

COVERGROUP report

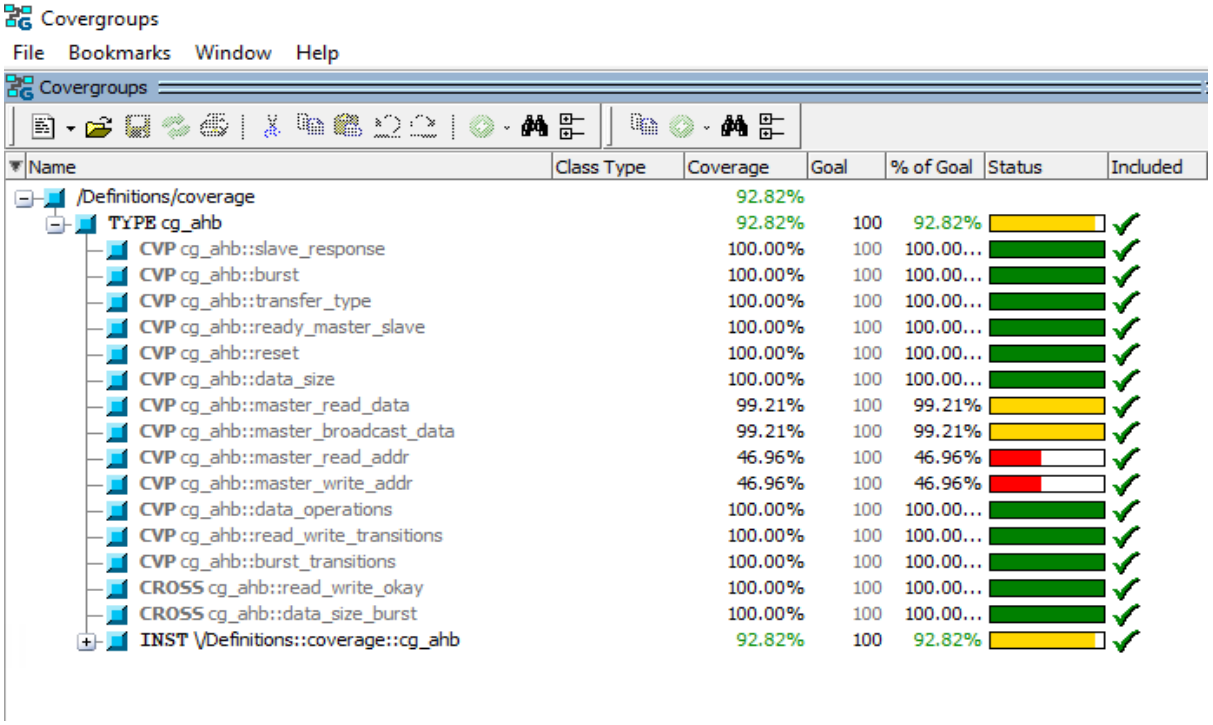

Figure 10: COVERGROUP report

Chapter 5: REFERENCES

1. AMBA® 3 AHB-Lite Protocol specification document by ARM.
2. Dr. Tom Schubert's Lectures and notes on ECE 593: Fundamentals of Pre-Silicon Validation [Spring 2020]
3. Comprehensive Functional Verification by [Bruce Wile, John C. Goss, Wolfgang Roesner]
4. The Uvm Primer: A Step-By-Step Introduction to the Universal Verification by Ray Salemi.