VERIFICATION OF AXI4LITE

ECE 593 Fundamentals of Pre-Silicon Validation
Final Project Report

Abstract - Verified an Axi4 Lite design using Object Oriented Programming verification approach. Overall design consists of an interconnect, master and slave modules. During verification we were able to find several bugs in the design such as interconnect arbitration driving only a single slave, incorrect strobe.

GROUP 1:
Amrutha Anil (amrutha@pdx.edu)
Manjari Rajasekharan (manjari@pdx.edu)
Durganila Anandhan (anandhan@pdx.edu)

Table of Contents

1.Introduction

1.1. About Axi4-Lite

AXI-4 Lite Protocol, The Advanced eXtensible Interface (AXI), part of the ARM Advanced Microcontroller Bus Architecture is a parallel high-performance, synchronous, high-frequency, multi-master, multi-slave communication interface, mainly designed for on-chip communication.

The AMBA specification defines three AXI4 protocols:

● AXI4: A high performance memory mapped data and address interface. Capable of Burst access to memory mapped devices.

● AXI4-Lite: A subset of AXI, lacking burst access capability. Has a simpler interface than the full AXI4 interface. Main features are that the address is a traditional Address/Data - does not support burst (single address, single data) and supports only 32 or 64 bits data width.

● AXI4-Stream: A fast unidirectional protocol for transferring data from master to slave.

There are 5 different channels between the master and the slave.

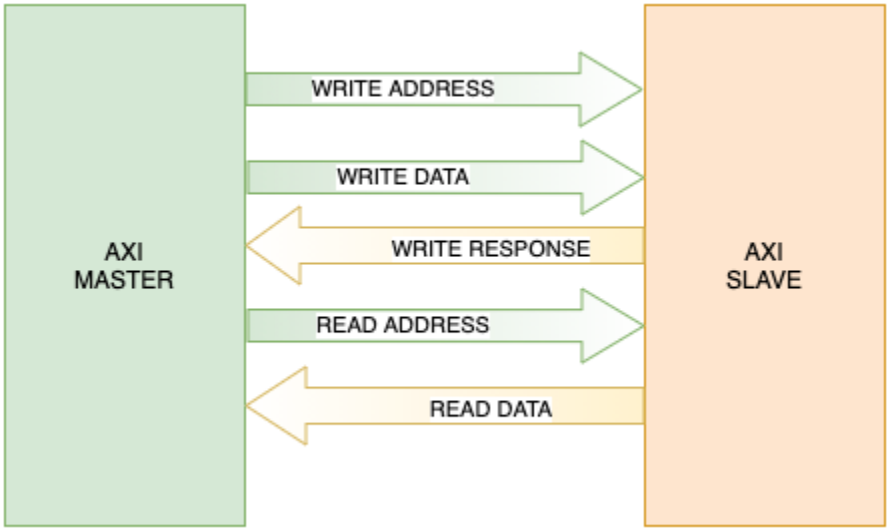


Fig 1: Channels between AXI4-Lite Master and Slave

In AXI4 Lite protocol, each channel uses the same VALID/READY handshake to transfer control and data information. The source generates the VALID signal to indicate when the data or control information is available. The destination generates the READY

signal to indicate that it accepts the data or control information. Transfer occurs only when both the VALID and READY signals are HIGH.

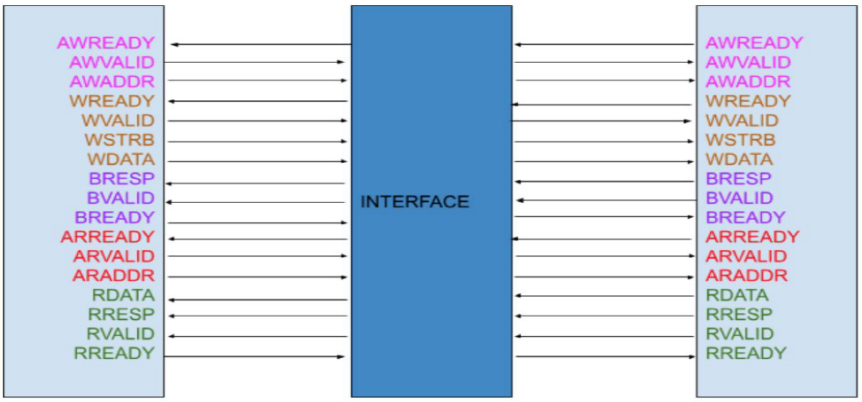
Source sends the next DATA or de-asserts VALID. Destination de-asserts READY if it is no longer able to accept DATA.

1.2. Architecture and Signal Description

The design chosen for verification consisted of an interconnect, two masters and two slaves as well as an interface developed using systemverilog constructs. Each channel has its own set of respective signals. The slave writes the data received from master to a memory. The memory range supported for the slave is parameterised. The design is 4 byte aligned. Below are the signals in AXi4-Lite.

Write Address Channel	Write Data Channel	Write Response Channel	Read Address Channel	Read Data Channel
AWREADY	WREADY	BRESP	ARREADY	RDATA
AWVALID	WVALID	BVALID	ARVALID	RRESP
AWADDR	WSTRB	BREADY	ARADDR	RVALID
	WDATA			RREADY

The signals are encapsulated using an interface. The master drives the address and controls the slave via the interface. Slave and master modports are defined in the interface to direct the signals. Below are the signals directed between master and slave through the interface.



2.Verification Requirements

The design for verification was taken from https://github.com/mmxsrup/axi4-interface.

The design chosen is an AXI4-Lite with multiple masters and slaves connected using an interconnect. After verification we found that the interconnect does not follow the protocol and it does not connect to multiple slaves. So we decided to disable interconnect and verify a single master and slave.

2.1. Verification Levels

AMBA AXI4-Lite is a bus interface. We have chosen to perform a system level verification. All the interconnections between different modules as well as the interface will be verified. Top module instantiates all the lower level components and will be tested for connectivity and functionality.

2.2. Functions

The following functions will be verified:

- Write transactions between master and slave
- Read transaction between master and slave
- Reset operation
- Back to Back operations
- Multiple byte data transfers

The following functionalities will not be verified:

- Protection types will not be verified
- Address width restricted to 12 bits
- Interconnect

2.3. Specific Tests & Methods

2.3.1. Type of Verification

A blackbox approach is chosen to verify the design thoroughly. The test stimulus will be driven into the DUT and a scoreboard will be used to verify the outputs.

2.3.2. Verification Strategy

An Object Oriented testbench using SystemVerilog was developed for verification. We have made use of a test factory so that the user will be able to select the type of test that is required to run. For synchronization mailboxes are used. Mailbox is used to transfer transactions between generator and driver as well as monitor and scoreboard. Threads were used to concurrently run the generator, driver, coverage, etc in the environment. We have used deterministic and constrained random strategies to verify the design. In order to check the correctness of the design deterministic tests were used to verify corner/edge cases as well as to test the basic functionality. The

chosen test will generate transactions which will be sent to the generator. Generator will send the transactions to the driver via a mailbox and the driver will drive the DUV via a virtual interface. Monitor will receive the transactions from the interface and send them to the scoreboard via a mailbox. Scoreboard will verify the correctness of the results against a reference model.

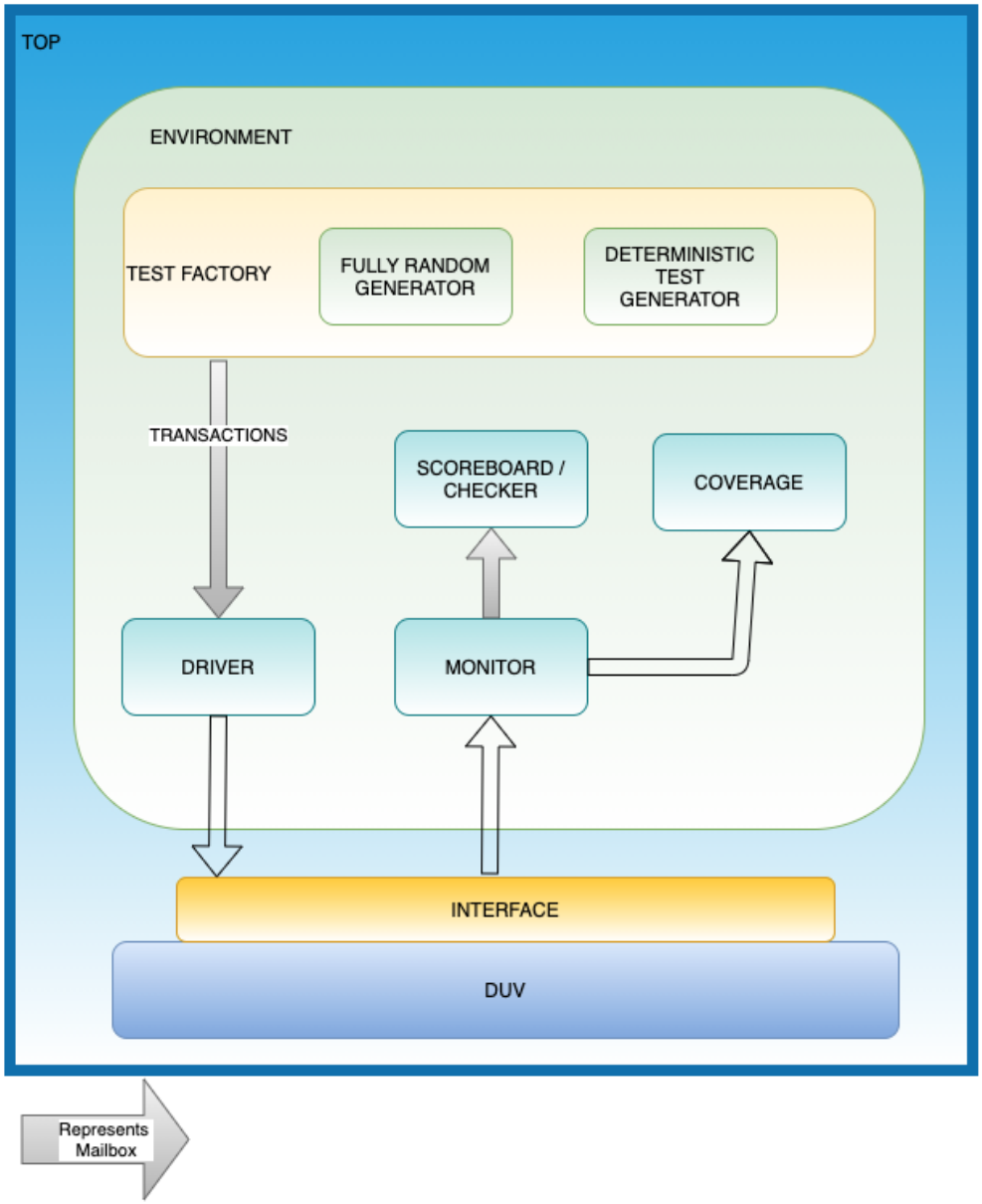


Fig 2: Verification Architecture

Users can select a DEBUG MODE during run time to run the test. If the debug mode is on, then the debug messages will be outputted. Users can also select the number of transactions or runs they want to perform during run time.

A package file is used to encapsulate the global variables, structs etc as well as to include all the necessary HVL files.

2.3.3. Abstraction Level

The verification will be transaction level and the lower level driver will drive cycle specific values to the DUV interface. A transaction class is created with all the signals such as data, address, start write, start read etc which can be sent together as a transaction to the driver.

2.3.4. Checking

The verification at the unit level level is very important to check the functionality of each unit. The possibility of finding and fixing a bug is easier and occupies less time while checked at the unit level. All possible cases are checked at the unit level.

The system-level checking is *on-the-fly* and *transaction-based*. For the read/writes, the data are read-from/written-to the local memory which acts as the reference model for the verification. The *save_val* task reads the write dataline and stores the value to the local memory, whereas the *check_val* task reads the read dataline and compares with the data in the local memory to confirm if the values are the same. Whenever there is any discrepancy, the score is incremented.

Some of the checking are as follows:

1. The order of write from a master is the same received by slave
2. Burst is always maintained as one
3. The read/write from a master is assigned to the correct slave (a write request from master is sent to slave)
4. Write address is x only when in reset
5. Write address value is x only when in reset

2.4. Coverage

The intention is to get the coverage of random and deterministic tests and add more tests to cover all the test scenarios for the design code. The covergroups are customized to cover all possible values of each signal and distributed into expected and illegal bins. Various coverpoints were coded for the combination of read & write transactions of data & address, Master and Slave FSM components with the reset, and the state transitions of Master and Slave.

The covergroups and coverpoints are as follows:

1. Covergroups for each of the channels between Master and Slave:
   - Write Address
   - Write Data
   - Read Address
   - Read Data
   - Write Response
2. Covergroups for Master & Slave reads and writes:
   - Master Read
   - Master Write

- Slave Read
- Slave Write
- Master Read with reset
- Master Write with reset
- Slave Read with reset
- Slave Write with reset

3. Covergroups for each of the channels with reset.
   - Ready and valid signals of Write Address with reset
   - Ready and valid signals of Write Data with reset
   - Ready and valid signals of Read Address with reset
   - Ready and valid signals of Read Data with reset
   - Ready and valid signals of Write Response with reset

2.5. Scenarios

Single operations	
1	Write 4 bytes to a location
2	Write 3 bytes to a location
3	Write 2 bytes to a location
4	Write to two locations
5	Read 4 bytes from a location
6	Read 3 bytes from a location
7	Read 2 bytes from a location

Multiple Operations:	
1	Write and read address -'0 and 2047 i.e edge addresses
2	Write to locations from 0 to 2048 and read in reverse order
3	Write and read data - all zeros & all ones
4	Consecutive writes (overwriting) to same location followed by a read
5	Modifying alternate locations and reading all the locations
6	Reset after write
7	Reset after Read

8	Both read and write start signal asserted
9	Neither read and write start signal asserted
10	Read when reset asserted
11	write when reset asserted

Error Scenarios:	
1	Write to an invalid address
2	Read from an invalid address

3.Project Management

3.1. Tools

- QuestaSIM
- Modelsim PE Student Edition
- Visual Studio Code

3.2. Risks/Dependencies

- The short schedule is a risk factor for the completion of the project.
- Development of the environment using OOP is new to the team members and is a potential risk
- The plan does not have any other potential risks or dependencies.

3.3. Resources

Team consists of three members:

1. Amrutha Anil (amrutha@pdx.edu)
2. Durganila Anandhan (anandhan@pdx.edu)
3. Manjari Rajasekharan (manjari@pdx.edu)

3.4. Schedule

05/14/2021	Unit level testing
05/19/2021	System level testing
05/20/2021	Enhanced testbench infrastructure by adding transaction, generator, driver and other classes and interface
05/23/2021	Updated testbench infrastructure to a complete object oriented testbench
05/25/2021	Added randomized test-cases
05/27/2021	Added deterministic test-cases
05/31/2021	Created a test factory
06/02/2021	Reviewed waveforms
06/02/2021	Added coverage and checker
06/03/2021	Created code cross reference and report

4.Verification Environment

Directory Structure

File	Description	Relative Path
axi_lite_master.sv	Drives address and controls to the slave via interface.	.DUT/axi_lite_master.sv
axi_lite_slave.sv	Writes data received from master to a memory. Sends data from the requested address to the master.	.DUT/axi_lite_slave.sv
axi_lite_interconnect.sv	Connects multiple slaves and masters.	.DUT/axi_lite.interconnect.sv

Table 1: DUV Files

File	Description	Relative Path
transaction.sv	Contains all the random variables and other variables required to generate a transaction. Some of the variables are constrained.	.TB/transaction.sv
top.sv	Top module where the DUVs are instantiated. Environment class is executed in this class.	.TB/top.sv
testFactory.sv	Allows user to select the testbench during run time.	.TB/testFactory.sv
scoreboard.sv	Compares the BFM's address lines and the testbench local memory.	.TB/scoreboard.sv
monitor.sv	Monitors the virtual interface and send the data to scoreboards class via mailbox.	.TB/monitor.sv
generator.sv	Parent class for test classes. Contains pure virtual execute task to be defined in all the child classes.	.TB/generator.sv
driver.sv	The data send by test classes are received by the driver	.TB/driver.sv

	class using mailbox and drives stimulus to DUV via virtual interface.	
axi_lite_coverage.sv	Functional coverage groups and bins.	.TB/axi_lite_coverage.sv
axi_env.sv	Objects of generator, driver, monitor, scoreboard, testfactory and coverage are created. Execute task is called to run the handles concurrently.	.TB/axi_env.sv
fully_random_test.sv	Contains fully random test stimulus. Uses mailbox to send the transactions to driver.	.TB/TestTypes/fully_random_test.sv
deterministic_tests.sv	Contains different interesting sequence for DUV inputs, edge cases and error conditions	.TB/TestTypes/deterministic_test.sv

Table 2: Testbench Files

File	Description	Relative Path
axi_lite_pkg.sv	Contains shared parameters, structures and enums used in DUV and testbench	./axi_lite_pkg
axi_lite_if.sv	Encapsulates all the signals required by the DUV.	./axi_lite_if.sv

Table 2: Shared files

testFactory.sv

Purpose:
Allows users to select the testbench during run time.

Class Instantiations:
None

Other functions called by the class:
None

Variables:
None

Tasks & Functions:

Task	Purpose

`static function generator Get_TestType (string testType,`
`  mailbox mb_generator2driver,`
`  logic debugMode,`
`  int numTransactions);`
 	
`fully_random_test fully_random_test_h;`
 	
`case (testType)`
`  "full_random" : begin`
`  fully_random_test_h = new(mb_generator2driver, debugMode, numTransactions);`
`  return fully_random_test_h;`
`  end`
	To get the handler of test selected by the user

transaction.sv

Purpose:
Used to generate transactions of various kinds. Contains random variables and fixed variables required to generate a transaction.

Class Instantiations:
None

Other functions called by the class:
None

Variables and constraints:

	Purpose

`logic reset_n;`
 	
`rand addr_t addr;`
`rand data_t data;`
`bit start_read;`
`bit start_write;`
`data_t rd_data;`
	To generate random resets to drive into DUV
	
	Generate read and write address
	Generate write data
	To start a read
	To start a write
	To get read data

`constraint addr_range {`
`addr inside{[0 : 4098]};`
`}`
 	
`constraint dist_data_cn {`
`data dist {`
`0 := 40,`
`[1: data_range] := 60};`
`}`
	To constraint the address within a range
	
	To constraint data within a range

Tasks & Functions:
None

fully_random_test.sv

Purpose:
Contains fully random test stimulus. It uses mailbox to send the transactions to driver.

Class Instantiations:
None

Other functions called by the class:
None

Variables and handlers:

Variable	Purpose
`transaction txn;` `transaction generate_pkt;`	transaction object to send the transaction to mailbox transaction object to generate random transactions

Tasks & Functions:

```
function new(mailbox mb_generator2driver, logic debugMode, int numTransactions);
    super.new(mb_generator2driver, debugMode, numTransactions);
```

Constructor that calls the super class new function.

```
task execute();
    repeat(numTransactions) begin
```

Task to generate the random transactions

```
task driver_send(
                input
                logic reset_n,
                addr_t addr,
                data_t data,
                logic start_read,
                logic start_write
);
    txn                 = new();
    txn.reset_n         = reset_n;
    txn.addr            = addr;
    txn.data            = data;
    txn.start_read      = start_read;
    txn.start_write     = start_write;

    if(debugMode)
        $display($time," fully_random_test.driver_send: reset_n %b, addr %h, data %h read %b, write %b",
                    reset_n, addr, data, start_read, start_write);
```

Task to send the generated transaction to mailbox which will be used by driver to drive the stimulus to interface.

generator.sv

Purpose:
Parent class for test classes. Contains pure virtual execute task to be defined in all the child classes.

Class Instantiations:
None

Other functions called by the class:
None

Variables and handlers:

Variable	Purpose

`mailbox mb_generator2driver`
`logic debugMode`
`int numTransactions`
	Mailbox for driving transactions from generator to driver
	To get the debug mode
	To get the number of transactions to run

Tasks & Functions:

```
pure virtual task execute();
```

Ensures all the child classes define execute definition

```
function new(mailbox mb_generator2driver, logic debugMode, int numTransactions);
```

Constructor for generator class

driver.sv

Purpose:
The data send by test classes are received by the driver class using mailbox and drives stimulus to DUV via virtual interface.

Class Instantiations:
None

Other functions called by the class:
None

Variables and handlers:

Variable	Purpose

`mailbox mb_generator2driver`
`logic debugMode`
`int numTransactions`
	Mailbox for driving transactions from generator to driver
	To get the debug mode
	To get the number of transactions to run

Tasks & Functions:

```
task execute();
    txn = new();

    forever begin
        mb_generator2driver.get(txn);
        drive_master(txn);
    end
endtask
```

Execute task to get the transaction from transaction mailbox generated by test classes.

```
task drive_master(transaction txn);
    // set writes
    bfm0.addr           = txn.addr;
    bfm0.data           = txn.data;
    bfm0.start_write    = txn.start_write;
    bfm0.start_read     = txn.start_read;
```

Task to drive the transaction to interface. It follows the DUV pin level protocols.

```
function new(mailbox mb_generator2driver, virtual axi_lite_if bfm0, logic debugMode);
```

Constructor to get the interface, user inputs and mailbox from generator.

scoreboard.sv

Purpose:
Compares the BFM's address lines and the testbench local memory and outputs the score.

Class Instantiations:

```
transaction         txn;
```

Other functions called by the class:

```
this.mb_monitor2scoreboard  = mb_monitor2scoreboard;
this.bfm                    = bfm;
txn                         = new();

mb_monitor2scoreboard.get(txn);
```

Variables and handlers:

Variable	Purpose
`mailbox mb_monitor2scoreboard;`	Mailbox for driving transactions from generator to driver.
`virtual axi_lite_if bfm;`	Virtual interface to get data from interface.
`transaction txn;`	Transaction object to send data to scoreboard
`int score = 0;`	To store score.
`logic [DATA_WIDTH-1:0] local_mem[BUFFER_SIZE];`	Store values for comparison.
`int i;`	Temporary variable for a for loop.

Tasks & Functions:

```
function new (mailbox mb_monitor2scoreboard,  virtual axi_lite_if bfm);
```

Function to create instances of mailbox, interface and transaction and to initialise the local memory locations to 0.

```
protected task save_val();
```

Function to read the write data line and store the value to local memory.

```
protected task check_val();
```

Function to read the read data line and check the local memory to confirm.

```
task execute();
```

Function to check read line and write line and compute the score for the scoreboard.

monitor.sv

Purpose:
Monitors the virtual interface and send the data to scoreboards class via mailbox.

Class Instantiations:

```
//Instantiate the BFM:
axi_lite_if bfm0(.aclk(aclk), .areset_n(areset_n));

//Instantiate the DUT master and slave:

axi_lite_master #(addr0) master0 (
        .m_axi_lite(bfm0.master)
);

axi_lite_slave slave0 (
  .s_axi_lite(bfm0.slave)
);
```

Other functions called by the class:

```
env_h = new(bfm0, test_type, debugMode, numTransactions);
env_h.execute();
```

Variables and handlers:

Variable	Purpose
`mailbox mb_monitor2scoreboard;`	Mailbox to send data to scoreboard
`virtual axi_lite_if bfm0;`	Virtual interface to get data from interface
`transaction txn;`	Transaction object to send data to scoreboard
`logic debugMode;`	Object for debug mode

Tasks & Functions:

```
task execute();
    forever begin
        sampleData();
    end
endtask
```

Task which runs forever and calls sample data task.

```
task sampleData();
    @(posedge bfm0.aclk);
```

Task to sample data from virtual interface and send to the scoreboard via mailbox at every positive edge of the clock

axi_lite_coverage.sv

Purpose:
To create the coverage groups and bins to obtain functional coverage.

Class Instantiations:
None

Other functions called by the class:
None

Variables and handlers:

Variable	Purpose
`virtual axi_lite_if bfm;`	Virtual interface to get data from interface

Tasks & Functions:

```
function new (virtual axi_lite_if b);
    cg_Read_Address = new();
    cg_Read_Data = new();
    cg_Write_Address = new();
    cg_Write_Data = new();
    cg_Write_Response = new();
    cg_Reset_Signal = new();
    cg_Master_FSM = new();
    cg_Slave_FSM = new();

    this.bfm = b;

endfunction : new
```

Function to create instances of the covergroups

```
task execute();
    forever begin : sampling_block
        @(posedge bfm.aclk);
        cg_Read_Address.sample();
        cg_Read_Data.sample();
        cg_Write_Address.sample();
        cg_Write_Data.sample();
        cg_Write_Response.sample();
        cg_Reset_Signal.sample();
        cg_Master_FSM.sample();
        cg_Slave_FSM.sample();
    end : sampling_block

endtask : execute
```

Function to sample the covergroups at the posedge of clock.

axi_env.sv

Purpose:

To create the environment with the objects of generator, driver, monitor, scoreboard, testfactory and coverage. The *execute* task is called to run the handles concurrently.

Class Instantiations:

```
generator           generator_h;
driver              driver_h;
monitor             monitor_h;
scoreboard          scoreboard_h;
axi_lite_coverage   coverage_h;
testFactory         testFactory_h;
```

Other functions called by the class:

```
generator_h     = testFactory_h.Get_TestType(testType, mb_generator2driver, debugMode, numTransactions);
driver_h        = new(mb_generator2driver, bfm0, debugMode);
monitor_h       = new(mb_monitor2scoreboard, bfm0, debugMode);
scoreboard_h    = new(mb_monitor2scoreboard, bfm0);
coverage_h      = new(bfm0);

monitor_h.execute();
scoreboard_h.execute();
generator_h.execute();
driver_h.execute();
coverage_h.execute();
```

Variables and handlers:

Variable	Purpose
`virtual axi_lite_if bfm0;`	Virtual interface to get data from interface
`string testType;`	Variable for testType
`logic debugMode;`	Variable for debug mode
`int numTransactions;`	Variable for number of transactions

Tasks & Functions:

```
function new (virtual axi_lite_if bfm0, string testType, logic debugMode, int numTransactions );
```

Function to create instance of interface and the variables.

```
task execute();
```

Function to create new objects of generator, driver, monitor, scoreboard, testfactory and coverage and run the *execute* function of each of these objects.

deterministic_tests.sv

Purpose:
Contains different interesting sequences for DUV inputs, edge cases and error conditions.

Class Instantiations:
None

Other functions called by the class:
None

Variables and handlers:

Variable	Purpose
transaction txn	transaction object to send the transaction to mailbox

Tasks & Functions:

```
function new(mailbox mb_generator2driver, logic debugMode, int numTransactions);
    super.new(mb_generator2driver, debugMode, numTransactions);
```

Constructor that calls the super class new function.

```
task execute();
    int i,j;

     //write and read 1 byte data
    driver_send(1'b1, 12'h4, 8'hEF, '0, '1);
    #10;
    driver_send(1'b1, 12'h4, '0, '1, '0);
    #10;

    //write and read 2 bytes of data
    driver_send(1'b1, 12'h4, 16'hBEEF, '0, '1);
    #10;
    driver_send(1'b1, 12'h4, '0, '1, '0);
    #10;

    //write and read 3 bytes data
```

Task to generate the directed tests

```
task driver_send(
                input
                logic reset_n,
                addr_t addr,
                data_t data,
                logic start_read,
                logic start_write
);
    txn                 = new();
    txn.reset_n         = reset_n;
    txn.addr            = addr;
    txn.data            = data;
    txn.start_read      = start_read;
    txn.start_write     = start_write;

    if(debugMode)
        $display($time," fully_random_test.driver_send: reset_n %b, addr %h, data %h read %b, write %b",
                    reset_n, addr, data, start_read, start_write);
```

Task to send the generated transaction to the mailbox which will be used by the driver to drive the stimulus to the interface.

top.sv

Purpose:
Top module where the DUVs are instantiated. Environment class is executed in this class.

Class Instantiations:

```
//Instantiate the BFM:
axi_lite_if bfm0(.aclk(aclk), .areset_n(areset_n));

//Instantiate the DUT master and slave:

axi_lite_master #(addr0) master0 (
        .m_axi_lite(bfm0.master)
);

axi_lite_slave slave0 (
  .s_axi_lite(bfm0.slave)
);
```

Other functions called by the class:

```
env_h = new(bfm0, test_type, debugMode, numTransactions);
env_h.execute();
```

Variables and handlers:

Variable	Purpose
`logic aclk;` `logic areset_n;`	Clock Active low reset

`string test_type`
`logic debugMode`
`int numTransactions`
	Get which test type to run from user during runtime
	Get user input if the run is in debug mode
	Get the number of transactions/runs to be performed

`environment env_h;`	Environment object

Tasks & Functions:

```
task InitialReset();
  areset_n = 0;
  repeat(10) @(posedge aclk);
  areset_n = 1;
```

To perform the initial reset

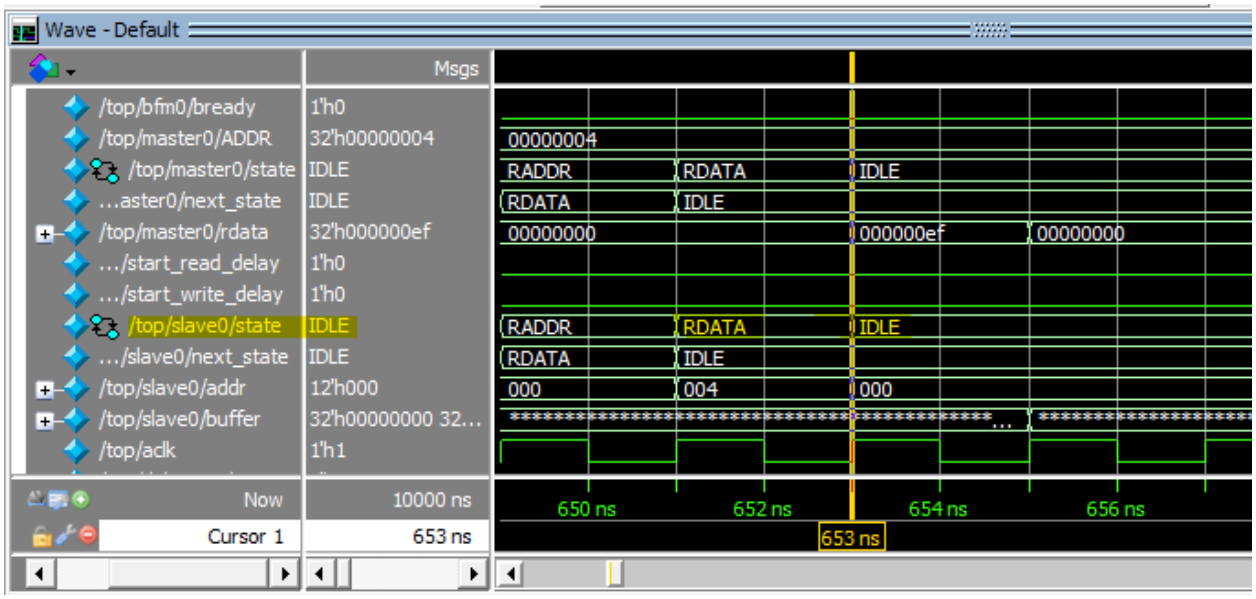
5.Results and Coverage

5.1. Bugs Found

1. The interconnect is not working as per the protocol requirements. So we had to strip down the design to only 1 master and 1 slave and complete the verification.
2. AXI4-Lite supports strobes, but the design always writes a 0 to the strobe.
3. Bug in the simulator:

During read operation, there isn't any incorrect state transition from IDLE to RDATA with respect to Slave, but the correct transition from RDATA to IDLE. The simulator still throws the error with the illegal bin of coverage (`illegal_bins sr_illegal7 = (IDLE => RDATA);`) as follows:

```
# ** Error: (vsim-8564) Illegal transition bin was hit at value=RDATA. The bin
counter for the illegal bin '\/top_sv_unit::axi_lite_coverage::cg_Reset_Signal
.Slave_Read_FSM_Reset.sr_illegal7' is 1.
#    Time: 653 ns  Iteration: 1  Region: /top_sv_unit::environment::execute
```



5.2. Transcripts

Debug messages are generated if the debug mode is turned on by the user.

```
153 deterministic_tests.driver_send: reset_n 1, addr 00000007, data 11 read 0, write 1
153 driver.drive_master addr 00000007, data 11, start_write 1 start_read 0
163 deterministic_tests.driver_send: reset_n 1, addr 00000008, data 12 read 0, write 1
163 driver.drive_master addr 00000008, data 12, start_write 1 start_read 0
173 deterministic_tests.driver_send: reset_n 1, addr 00000009, data 13 read 0, write 1
173 driver.drive_master addr 00000009, data 13, start_write 1 start_read 0
183 deterministic_tests.driver_send: reset_n 1, addr 0000000a, data 14 read 0, write 1
183 driver.drive_master addr 0000000a, data 14, start_write 1 start_read 0
193 deterministic_tests.driver_send: reset_n 1, addr 0000000b, data 15 read 0, write 1
193 driver.drive_master addr 0000000b, data 15, start_write 1 start_read 0
203 deterministic_tests.driver_send: reset_n 1, addr 0000000c, data 16 read 0, write 1
203 driver.drive_master addr 0000000c, data 16, start_write 1 start_read 0
213 deterministic_tests.driver_send: reset_n 1, addr 0000000d, data 17 read 0, write 1
213 driver.drive_master addr 0000000d, data 17, start_write 1 start_read 0
223 deterministic_tests.driver_send: reset_n 1, addr 0000000e, data 18 read 0, write 1
223 driver.drive_master addr 0000000e, data 18, start_write 1 start_read 0
233 deterministic_tests.driver_send: reset_n 1, addr 0000000f, data 19 read 0, write 1
233 driver.drive_master addr 0000000f, data 19, start_write 1 start_read 0
243 deterministic_tests.driver_send: reset_n 1, addr 00000010, data 1a read 0, write 1
243 driver.drive_master addr 00000010, data 1a, start_write 1 start_read 0
253 deterministic_tests.driver_send: reset_n 1, addr 00000011, data 1b read 0, write 1
253 driver.drive_master addr 00000011, data 1b, start_write 1 start_read 0
263 deterministic_tests.driver_send: reset_n 1, addr 00000012, data 1c read 0, write 1
263 driver.drive_master addr 00000012, data 1c, start_write 1 start_read 0
273 deterministic_tests.driver_send: reset_n 1, addr 00000013, data 1d read 0, write 1
273 driver.drive_master addr 00000013, data 1d, start_write 1 start_read 0
283 deterministic_tests.driver_send: reset_n 1, addr 00000014, data 1e read 0, write 1
```

5.3. Coverage Report

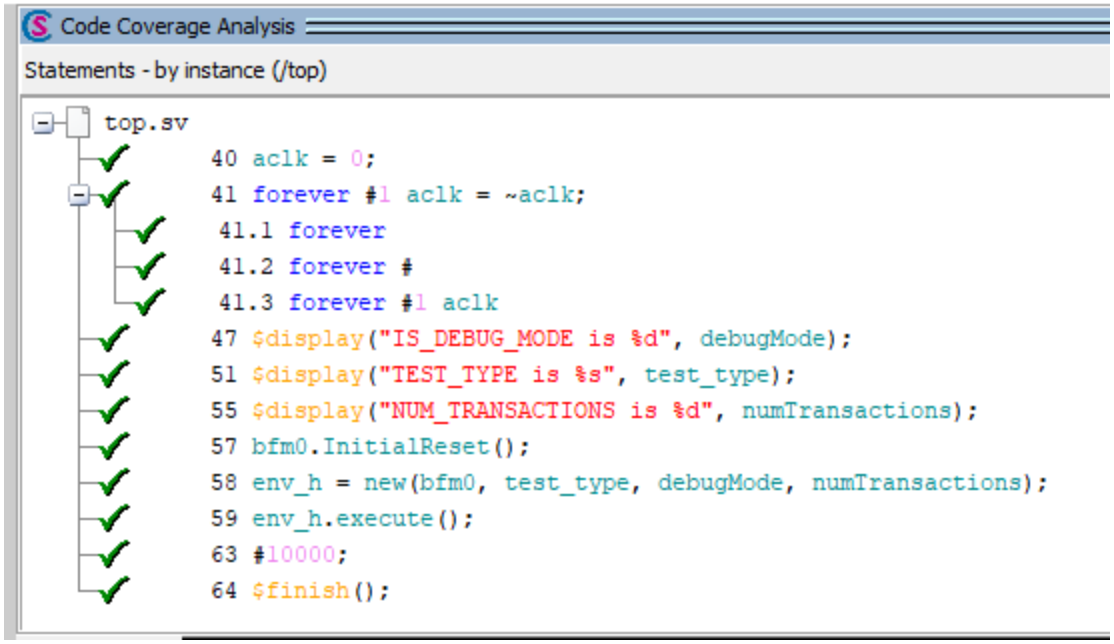
The code coverage and the functional coverage are obtained from the simulation of the code.

5.3.1. Code coverage

All the coverage types are enabled using the option argument *+cover=bcesxf*.

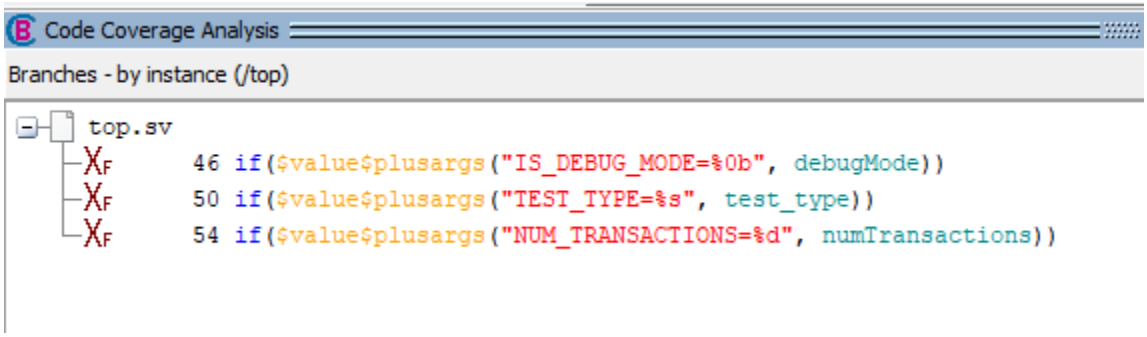
1. **Statement coverage:**

   The 100% statement coverage shows that all parts of the code have run during the simulation. This is important because during the simulation, there might be some part of the code which never gets run, possibly skipped during branches.



2. **Branch coverage:**

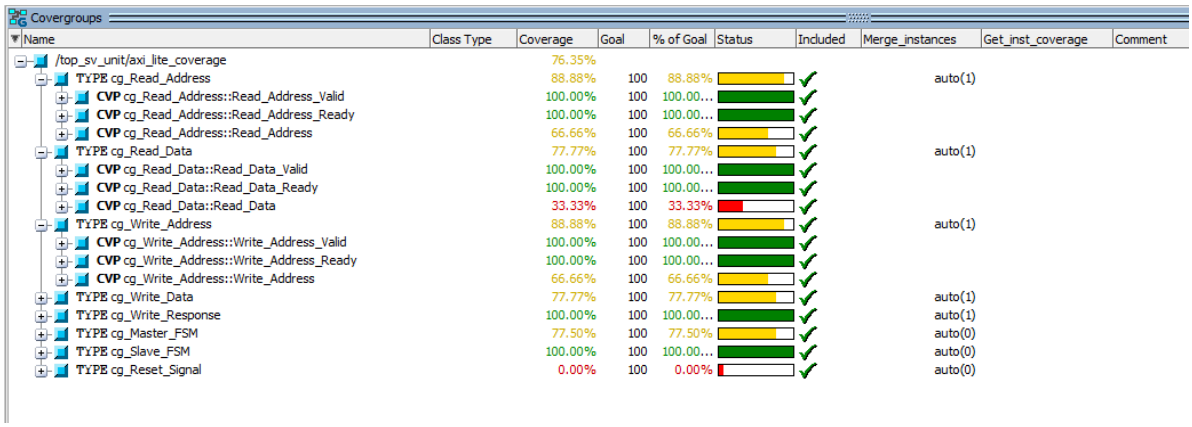
The branch coverage is to show if the code has taken up the respective branch. In this image, the command line arguments were provided during the run time and hence these branches were not taken, but these branch conditions have been verified.



5.3.2. Functional Coverage:

We could get functional coverage of about 76% for fully random tests and about 92% for deterministic tests. This shows the tests written cover most of the testing scenarios.

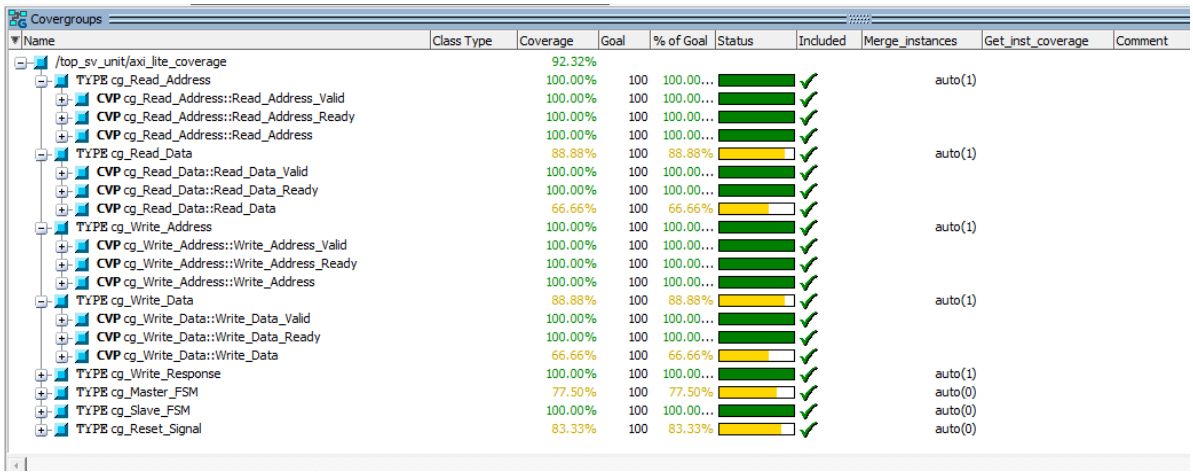
With fully random tests:



Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage	Comment
/top_sv_unit/axi_lite_coverage		76.35%							
TYPE cg_Read_Address		88.88%	100	88.88%			auto(1)		
CVP cg_Read_Address::Read_Address_Valid		100.00%	100	100.00...					
CVP cg_Read_Address::Read_Address_Ready		100.00%	100	100.00...					
CVP cg_Read_Address::Read_Address		66.66%	100	66.66%					
TYPE cg_Read_Data		77.77%	100	77.77%			auto(1)		
CVP cg_Read_Data::Read_Data_Valid		100.00%	100	100.00...					
CVP cg_Read_Data::Read_Data_Ready		100.00%	100	100.00...					
CVP cg_Read_Data::Read_Data		33.33%	100	33.33%					
TYPE cg_Write_Address		88.88%	100	88.88%			auto(1)		
CVP cg_Write_Address::Write_Address_Valid		100.00%	100	100.00...					
CVP cg_Write_Address::Write_Address_Ready		100.00%	100	100.00...					
CVP cg_Write_Address::Write_Address		66.66%	100	66.66%					
TYPE cg_Write_Data		77.77%	100	77.77%			auto(1)		
TYPE cg_Write_Response		100.00%	100	100.00...			auto(1)		
TYPE cg_Master_FSM		77.50%	100	77.50%			auto(0)		
TYPE cg_Slave_FSM		100.00%	100	100.00...			auto(0)		
TYPE cg_Reset_Signal		0.00%	100	0.00%			auto(0)		

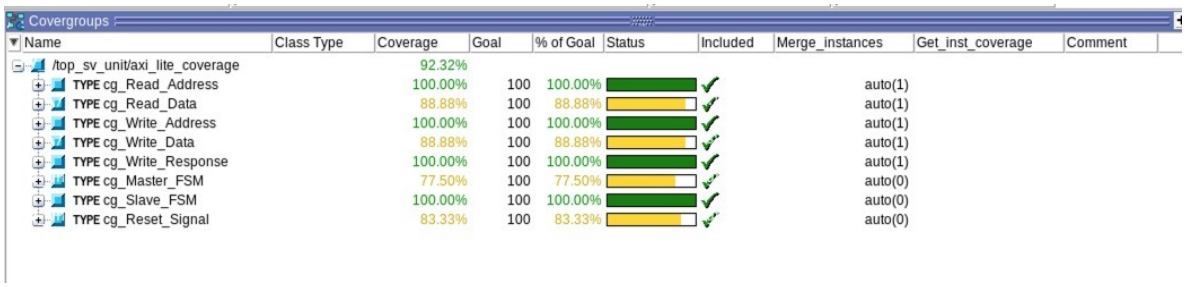
With Deterministic tests:



Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage	Comment
/top_sv_unit/axi_lite_coverage		92.32%							
TYPE cg_Read_Address		100.00%	100	100.00...			auto(1)		
CVP cg_Read_Address::Read_Address_Valid		100.00%	100	100.00...					
CVP cg_Read_Address::Read_Address_Ready		100.00%	100	100.00...					
CVP cg_Read_Address::Read_Address		100.00%	100	100.00...					
TYPE cg_Read_Data		88.88%	100	88.88%			auto(1)		
CVP cg_Read_Data::Read_Data_Valid		100.00%	100	100.00...					
CVP cg_Read_Data::Read_Data_Ready		100.00%	100	100.00...					
CVP cg_Read_Data::Read_Data		66.66%	100	66.66%					
TYPE cg_Write_Address		100.00%	100	100.00...			auto(1)		
CVP cg_Write_Address::Write_Address_Valid		100.00%	100	100.00...					
CVP cg_Write_Address::Write_Address_Ready		100.00%	100	100.00...					
CVP cg_Write_Address::Write_Address		100.00%	100	100.00...					
TYPE cg_Write_Data		88.88%	100	88.88%			auto(1)		
CVP cg_Write_Data::Write_Data_Valid		100.00%	100	100.00...					
CVP cg_Write_Data::Write_Data_Ready		100.00%	100	100.00...					
CVP cg_Write_Data::Write_Data		66.66%	100	66.66%					
TYPE cg_Write_Response		100.00%	100	100.00...			auto(1)		
TYPE cg_Master_FSM		77.50%	100	77.50%			auto(0)		
TYPE cg_Slave_FSM		100.00%	100	100.00...			auto(0)		
TYPE cg_Reset_Signal		83.33%	100	83.33%			auto(0)		

Combined Coverage

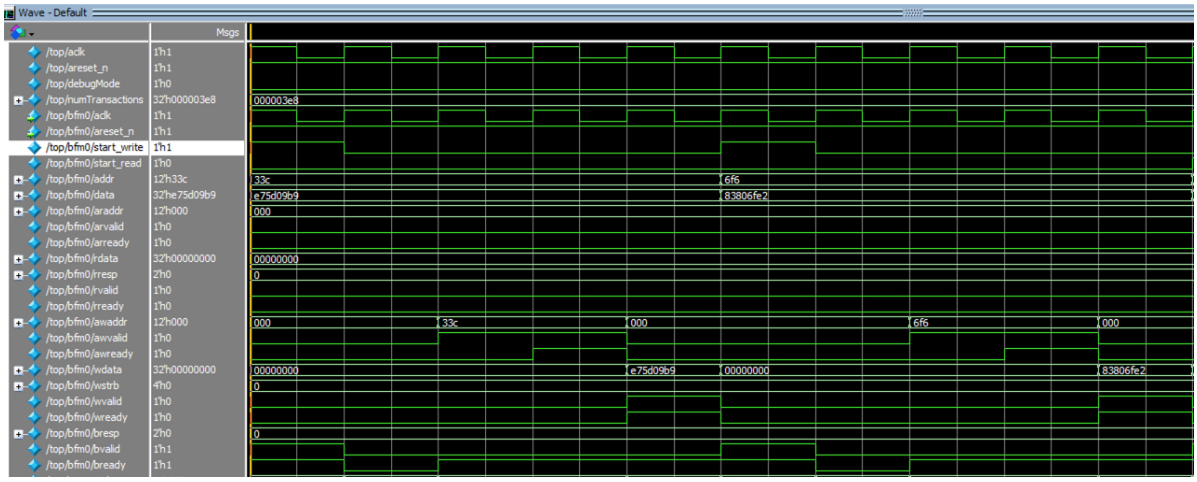


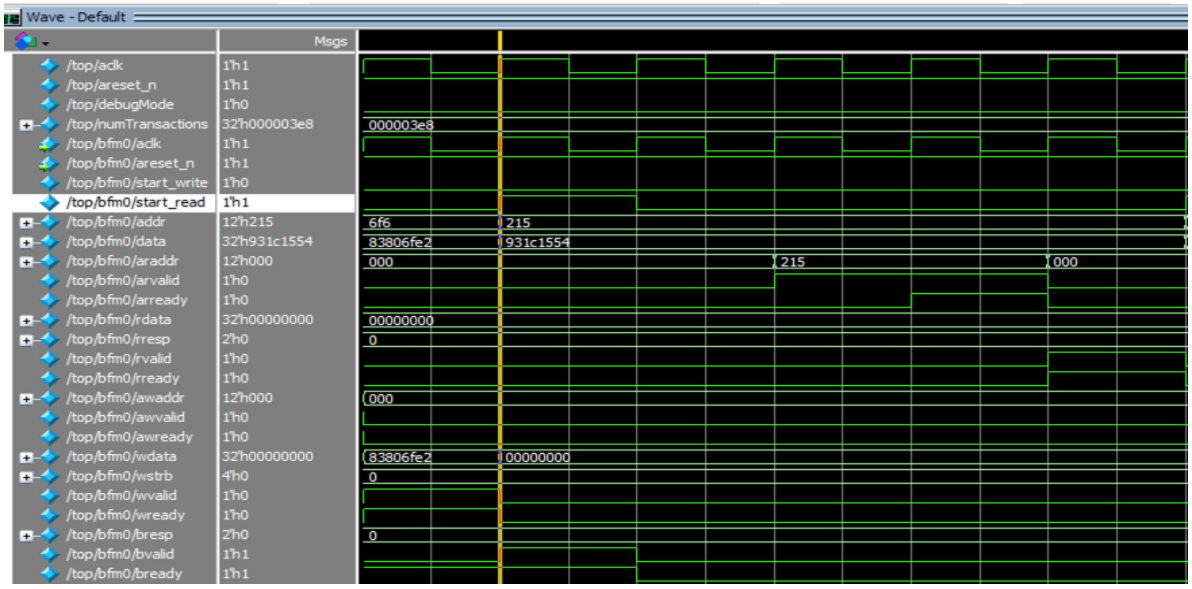
Covergroups

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instances	Get_inst_coverage	Comment
/top_sv_unit/axi_lite_coverage		92.32%							
TYPE cg_Read_Address		100.00%	100	100.00%			auto(1)		
TYPE cg_Read_Data		88.88%	100	88.88%			auto(1)		
TYPE cg_Write_Address		100.00%	100	100.00%			auto(1)		
TYPE cg_Write_Data		88.88%	100	88.88%			auto(1)		
TYPE cg_Write_Response		100.00%	100	100.00%			auto(1)		
TYPE cg_Master_FSM		77.50%	100	77.50%			auto(0)		
TYPE cg_Slave_FSM		100.00%	100	100.00%			auto(0)		
TYPE cg_Reset_Signal		83.33%	100	83.33%			auto(0)		

5.4. Wave Forms

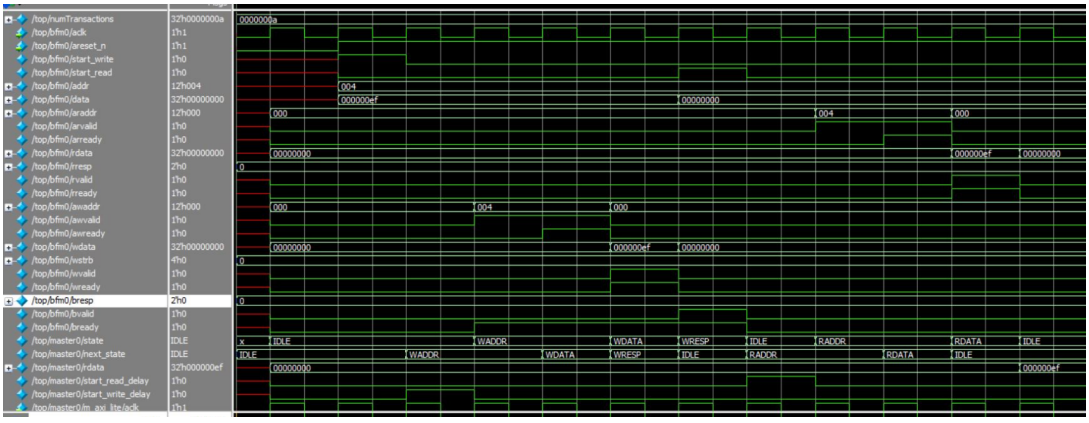
Fully Random tests:



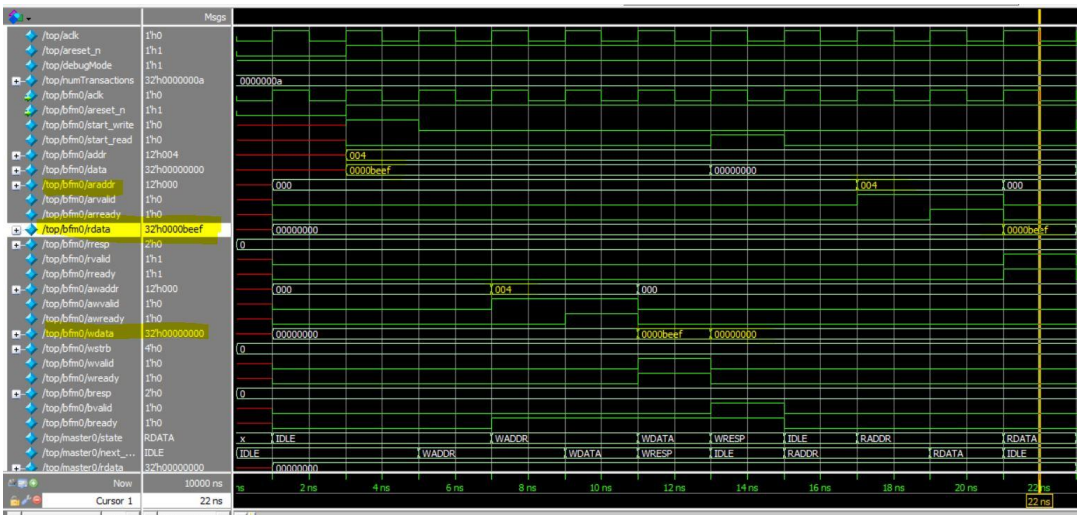


Deterministic cases:

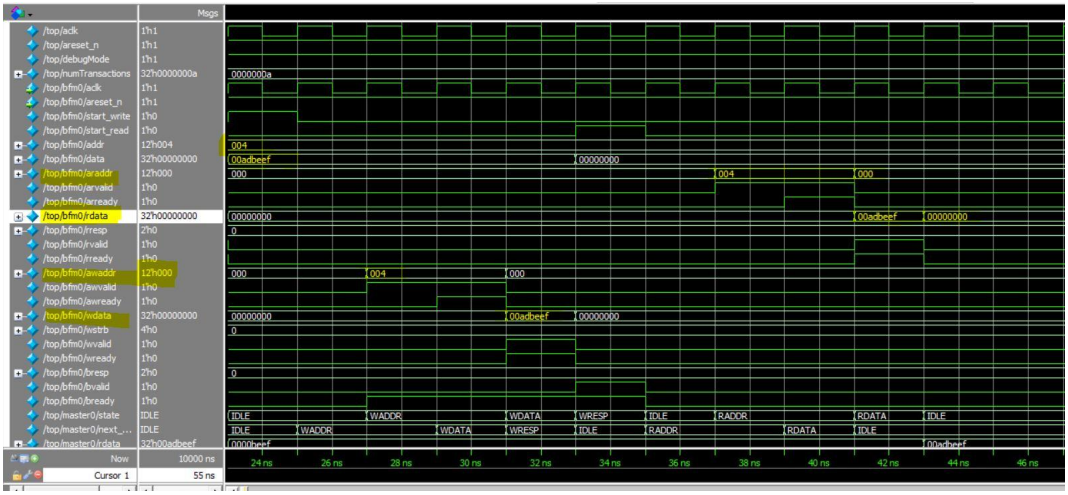
1. simple write and read txns - 1 byte



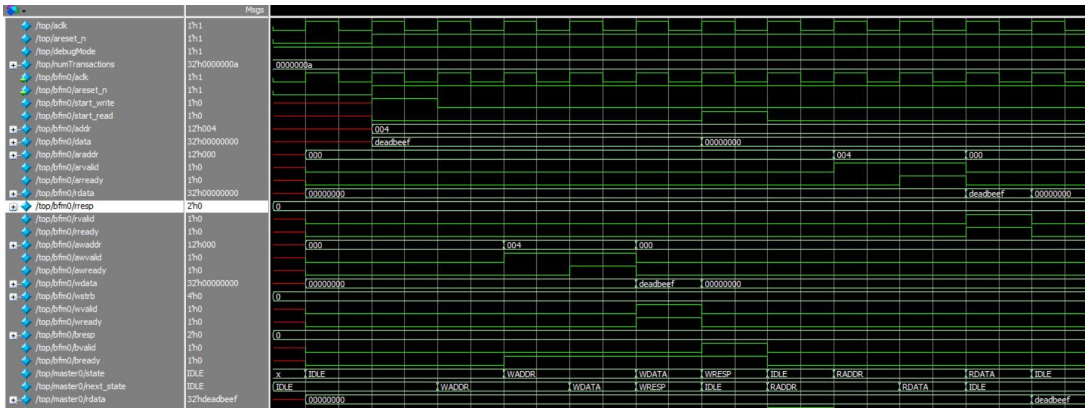
2. simple write and read txns - 2 bytes



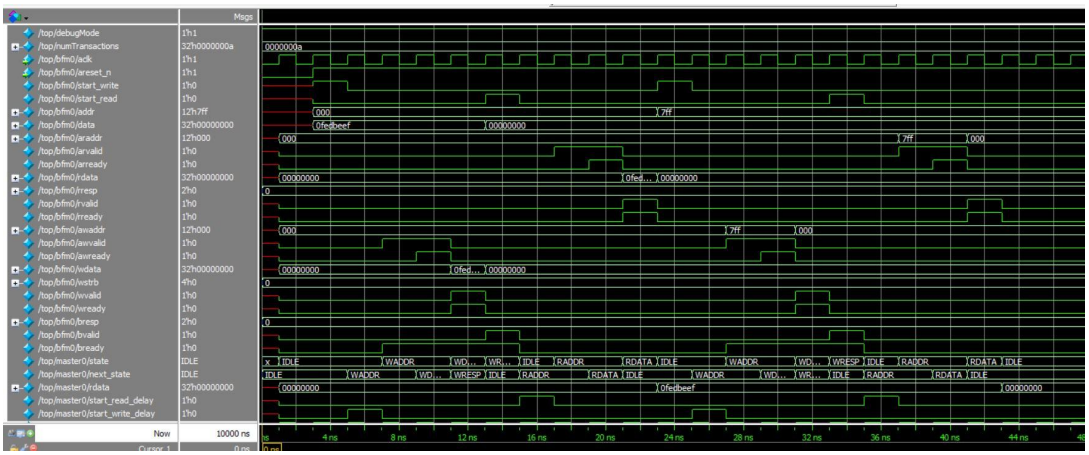
3. simple write and read txns - 3bytes



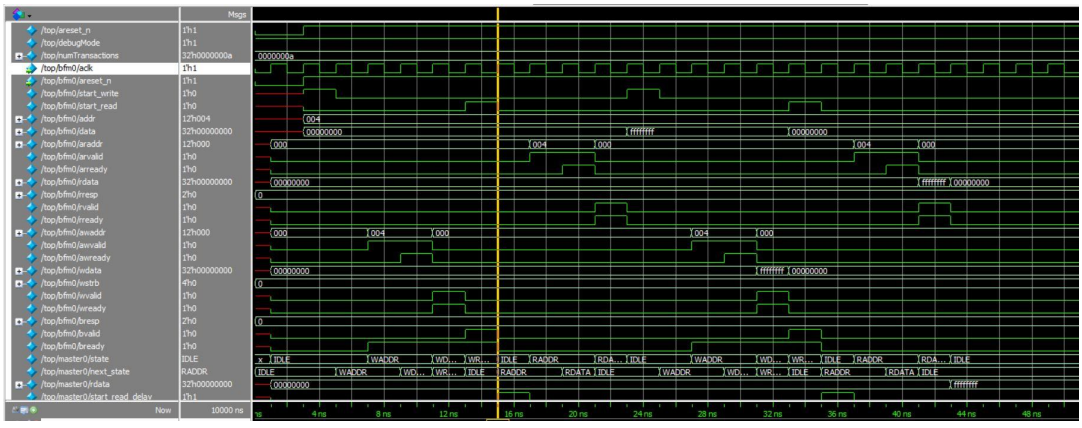
4. simple write and read txns - 4 bytes



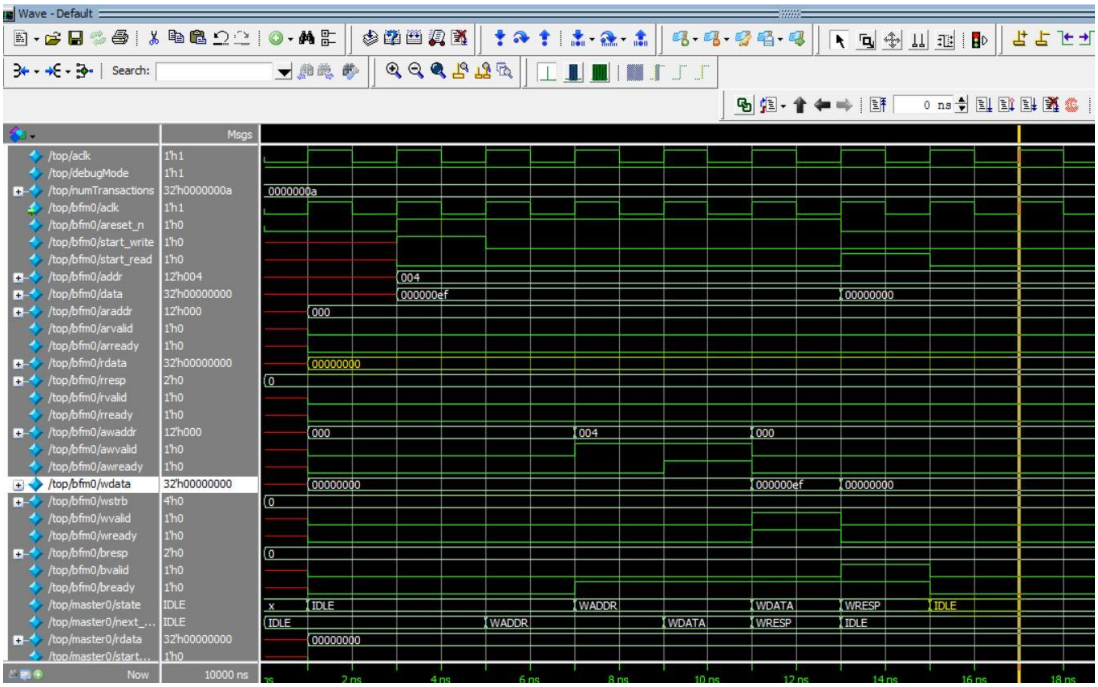
5. Writing to edge-addresses



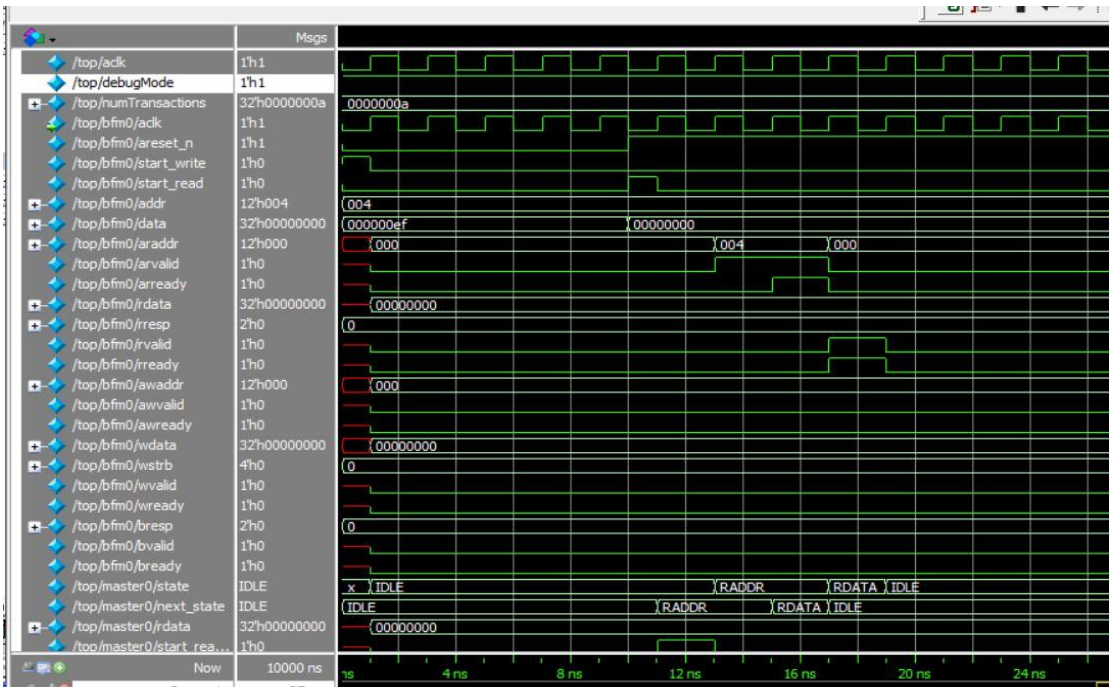
6. Writing all zeroes and ones



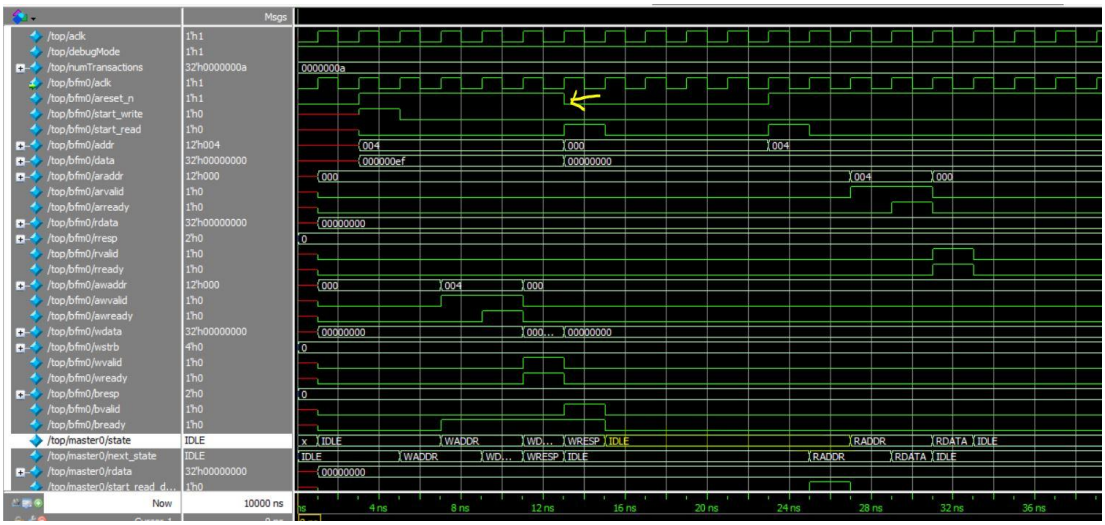
7. read during reset



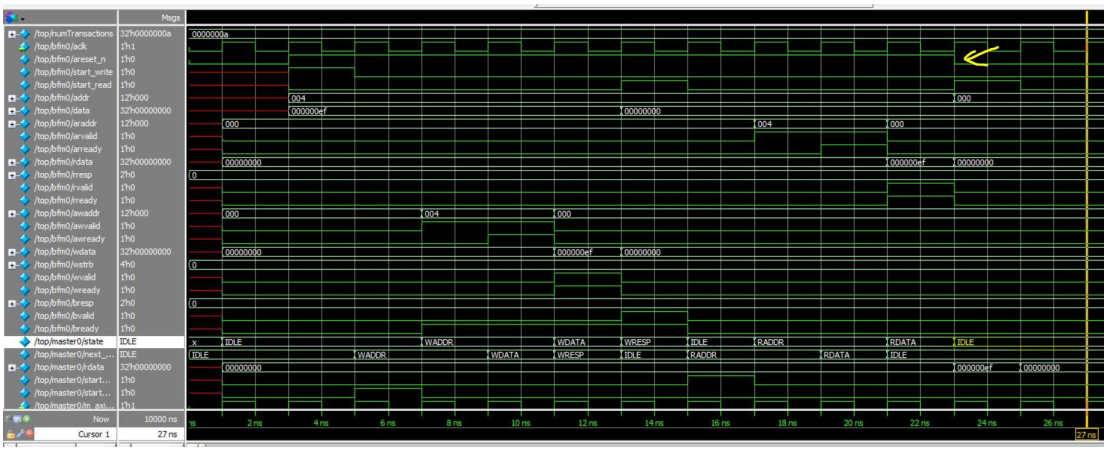
8. write during reset



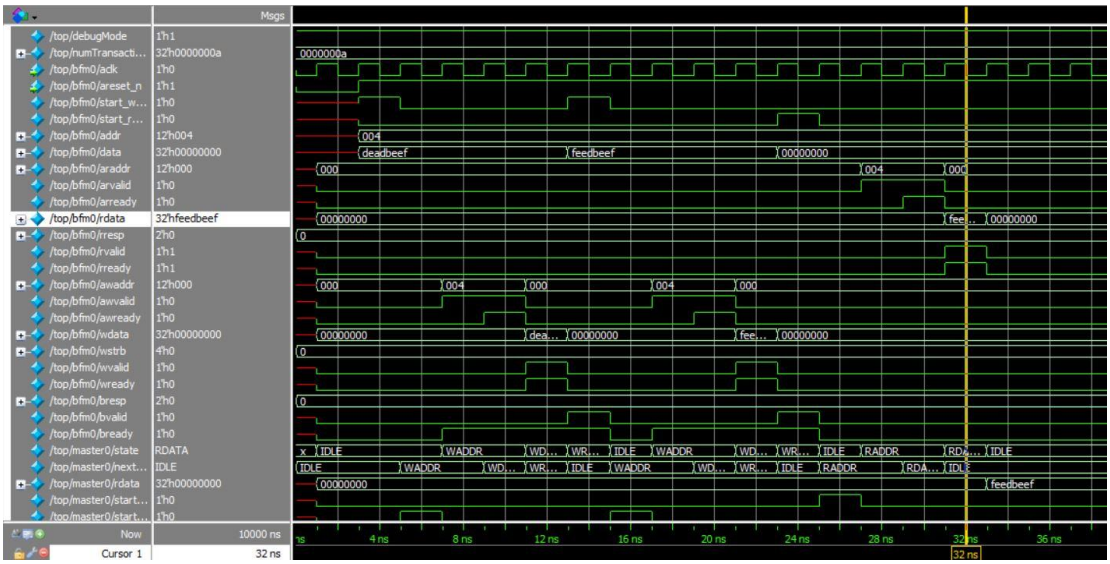
9. reset after write



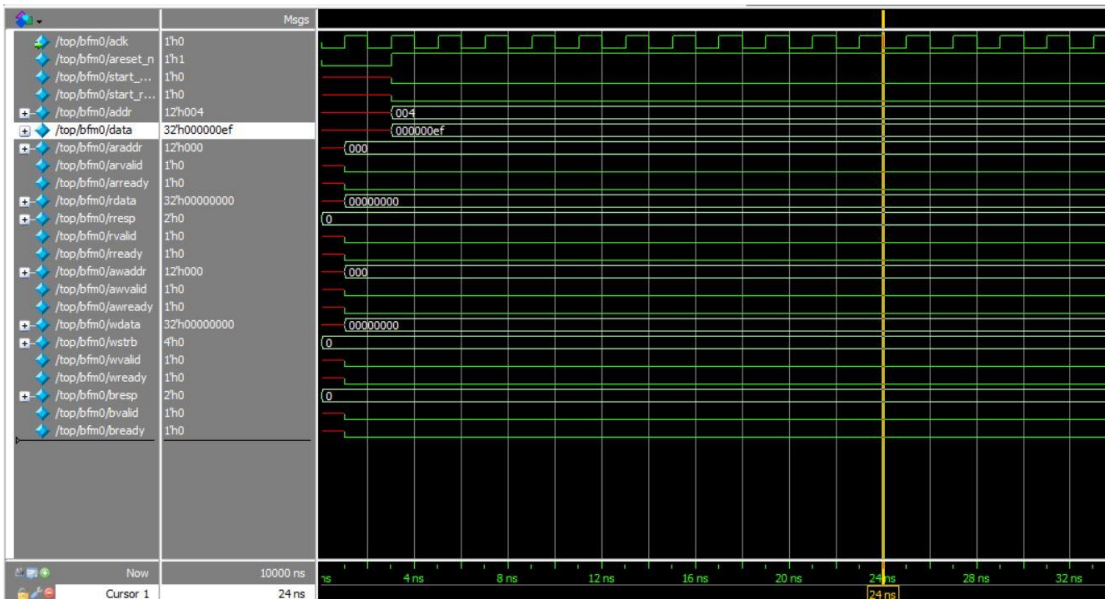
10. reset after read



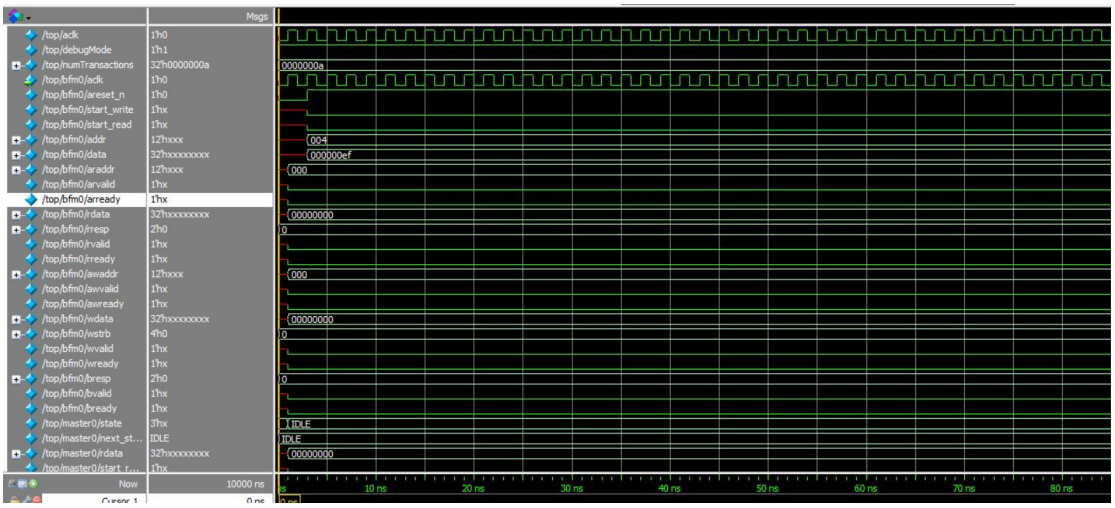
11. Read after over-writing a location



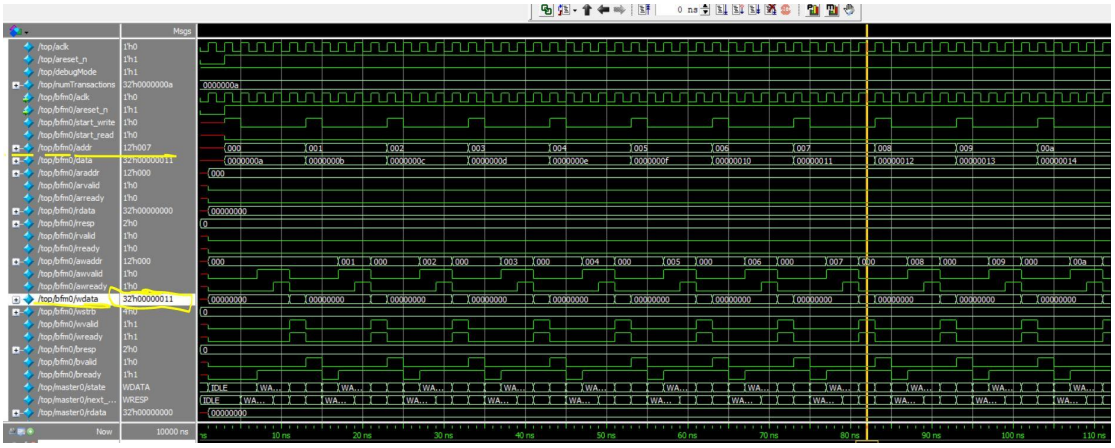
12. neither read nor write



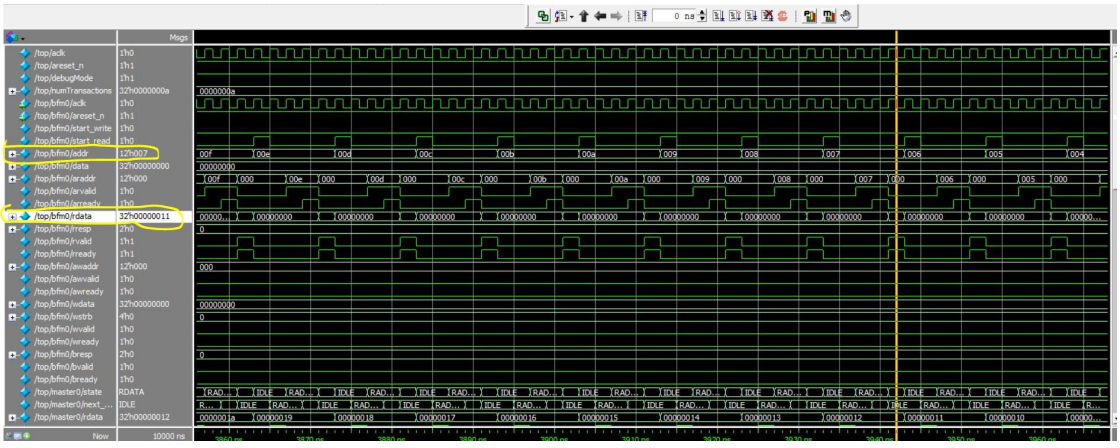
13. simultaneous read write



14. writing a range of locations and reading in reverse order

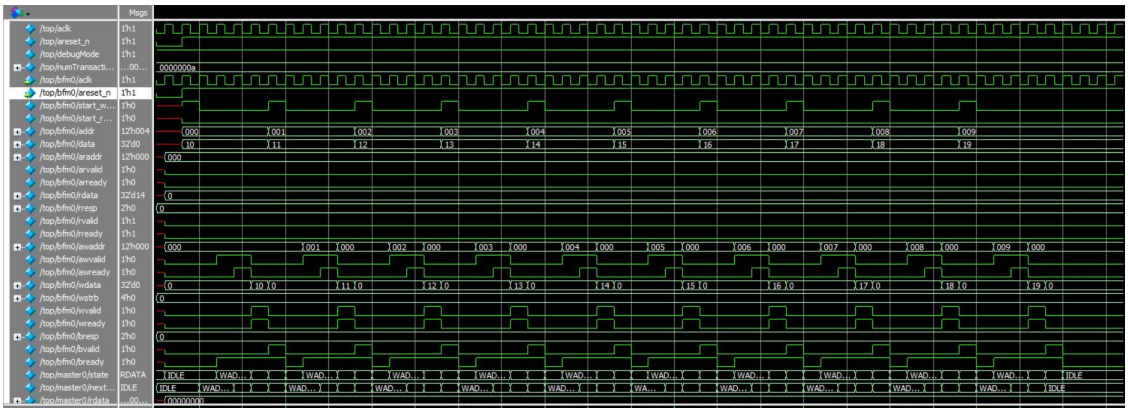


writing locations from 0 to 10

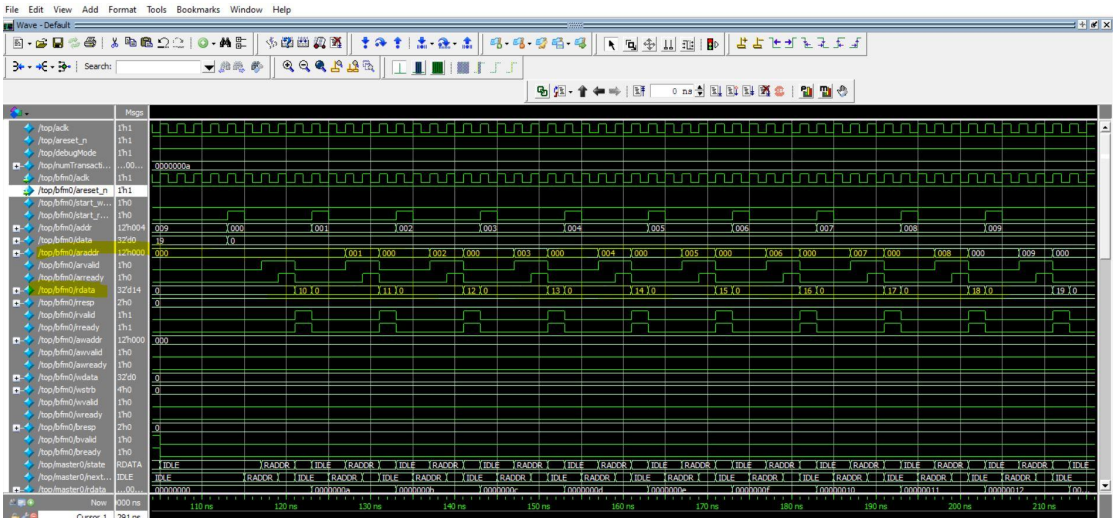


reading locations from 10 to 0

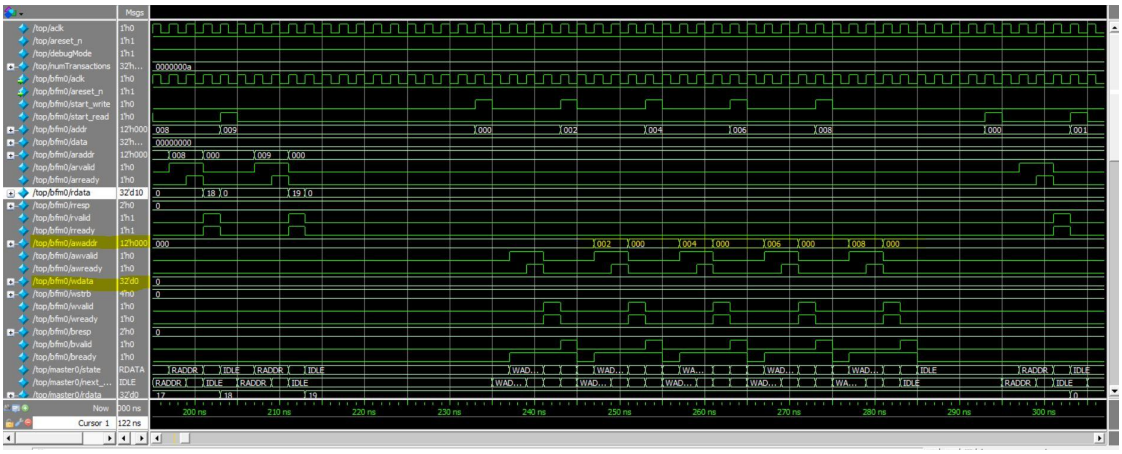
15. Modifying alternate locations and reading entire locations



writing locations j= 0 to 10 with data “j+10”



reading locations from 0 to 10 to confirm the data written previously



modifying locations 0,2,4,6,8 to data “0"

6.Conclusion

6.1. Resource Time

Total of 90 hours were taken for the overall verification of the design. It includes the time taken to debug the interconnect issue as well as development of the testbench architecture.

6.2. Challenges Encountered

1. The original intention was to verify multiple masters and slaves using interconnect. We were facing issues with the results generated and had to understand the interconnect RTL implementation. After many hours of debug we found that the interconnect that comes along with the design is not working as per the protocol requirements. So we had to proceed with one master and slave without an interconnect.
2. There was difficulty in setting up the test factory and had to figure out how the test types were to be separated.

6.3. Future Plans

1. Convert the verification to a UVM based environment
2. Fix the interconnect issue in the source design code and verify with multiple masters and slaves as per the initial plan.

7.Reference

- https://www.realdigital.org/doc/a9fee931f7a172423e1ba73f66ca4081
- http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf
- https://www.ripublication.com/ijaer17/ijaerv12n17_32.pdf
- https://developer.arm.com/documentation/dui0534/b
- Dr. Tom Schubert's Lectures and notes on ECE 593: Fundamentals of Pre-Silicon Validation
- Comprehensive Functional Verification by [Bruce Wile, John C. Goss, Wolfgang Roesner]
- The Uvm Primer: A Step-By-Step Introduction to the Universal Verification by Ray Salemi.