Cross Code Report for Verification of ARM AMBA-3 AHB-lite Protocol

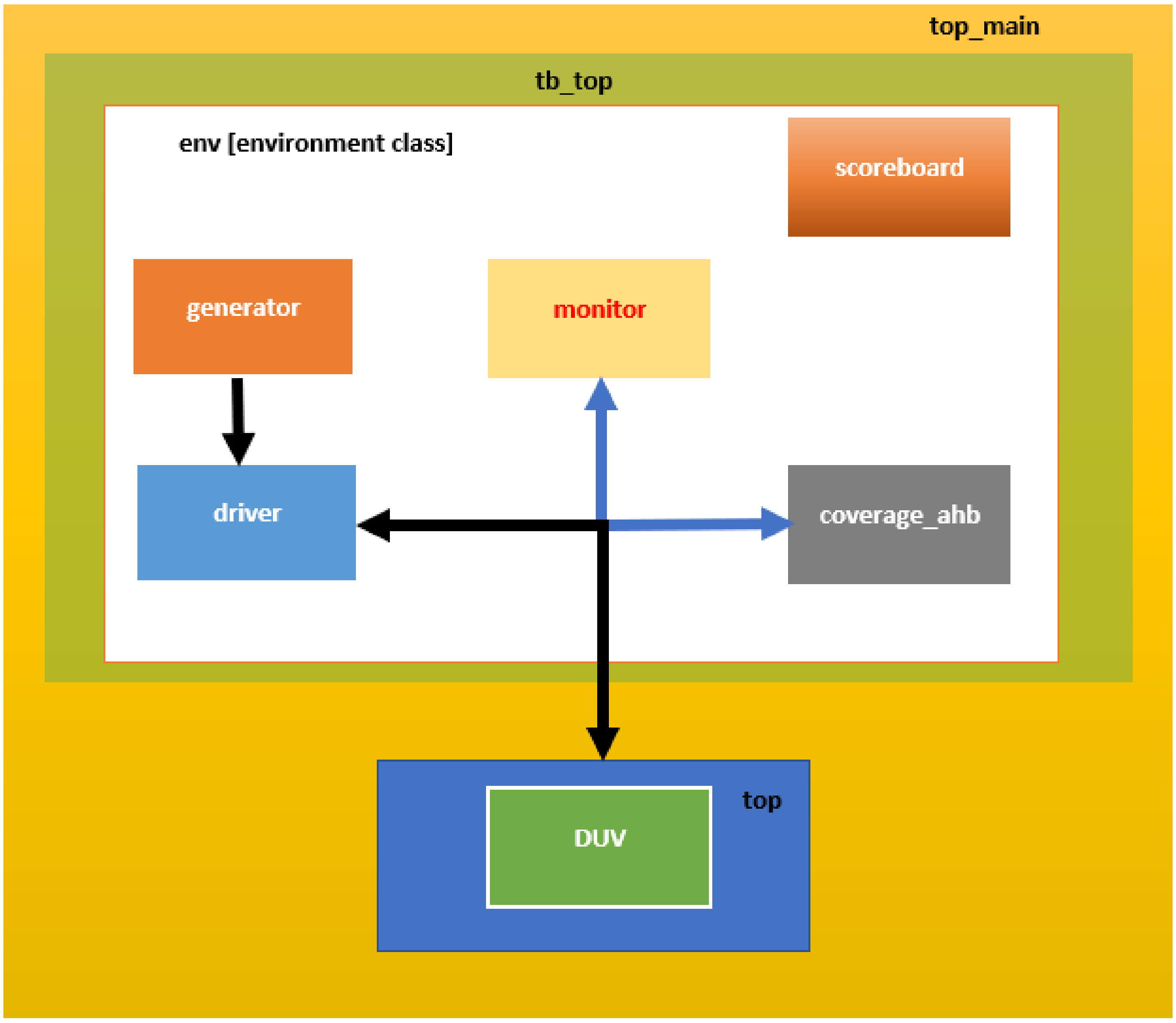
ECE-593 Fundamentals of Pre-Silicon Validation
Spring -2020

AAYUSH RAVICHANDRAN [981120898]

KAUSTUBH MHATRE [974819541]

SAURABH WAMAN CHAVAN [911836716]

Introduction to Verification Environment



In this project, we have a transaction class [txn] in which certain data types are declared as rand. Constraints are also present on the random data generated. There is a print() function to display the data and address to be written.

The generator class [gen] declares an object of the transaction class and randomizes it in order to get the random values. The generator class has a new function(). It also has a run task which is used to create the possible and deterministic and random test cases. There is another task called driver_send which is called by the run task to send the transaction to the driver via mailbox.

The driver class receives the transaction sent by the generator. It has a new() function. It also has a run task which calls another task called driver item which sends the data received from the mailbox to the DUV via virtual interface.

The monitor class snoops the transaction on this interface. Thus class has a new() function. It has a run task that calls another task called sample which sends the transaction that was snooped to the scoreboard via mailbox.

The scoreboard receives this data. It has a dummy memory present in it. By observing the transaction it receives it does read or write operations on the memory. Checking is done in the read operation by comparing the data read by the DUV and the data that is present in the dummy memory for that particular address.

The coverage class receives the transactions via a virtual interface. It has a covergroup named cg_ahb which has multiple coverpoints. It also has coverpoints for cross coverage. It has a new() function. It also has a run task which captures the transaction from the virtual interface at every positive edge of clock.

The environment class [env] creates objects of generator, driver, monitor, scoreboard and coverage_ahb and links them. The mailbox of generator is connected to the mailbox of the driver. The virtual interface of the driver is connected to virtual interface of monitor, scoreboard and coverage_ahb. The mailbox of monitor is connected to the mailbox of the scoreboard. The environment class has a new function(). It has a run task where it allocates memory to the differrent objects. This task has a fork join block which calls run task of all of these objects.

The tb_top is a program block which has an input of interface type. This program block creates an object of environment class and calls its run task.

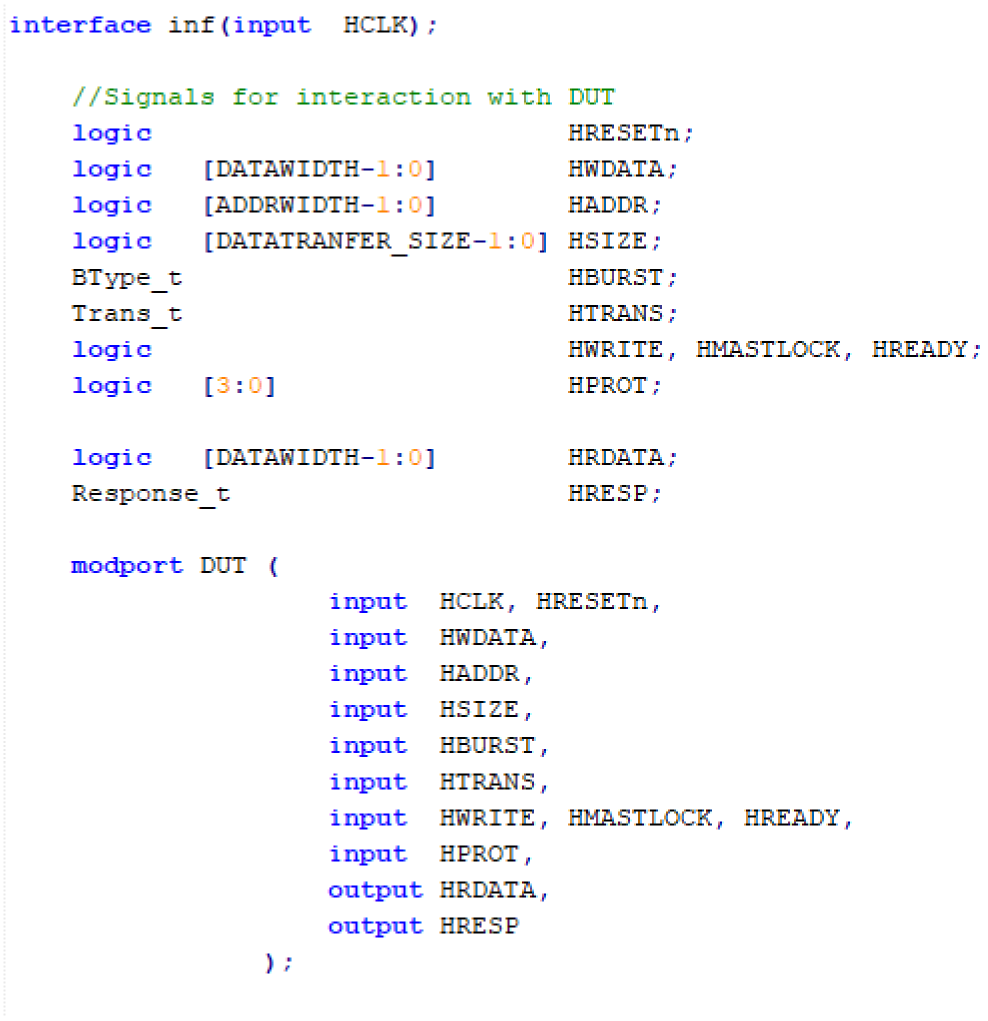
The AHB is a module present in top.sv file which encapsulates the entire DUV. It has input of interface type.

The top_main instantiatesthe AHB module and tb_top program block and passes interface to both of them. This top module is responsible for generation of clock signal.

The generator generates deterministic as well random transactions and passes it to the driver. The driver then passes this transaction on the interface. Monitor, coverage class and the DUV get the transaction from this interface. The monitor passes transaction to the scoreboard where checking happens.

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
                );
```

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)

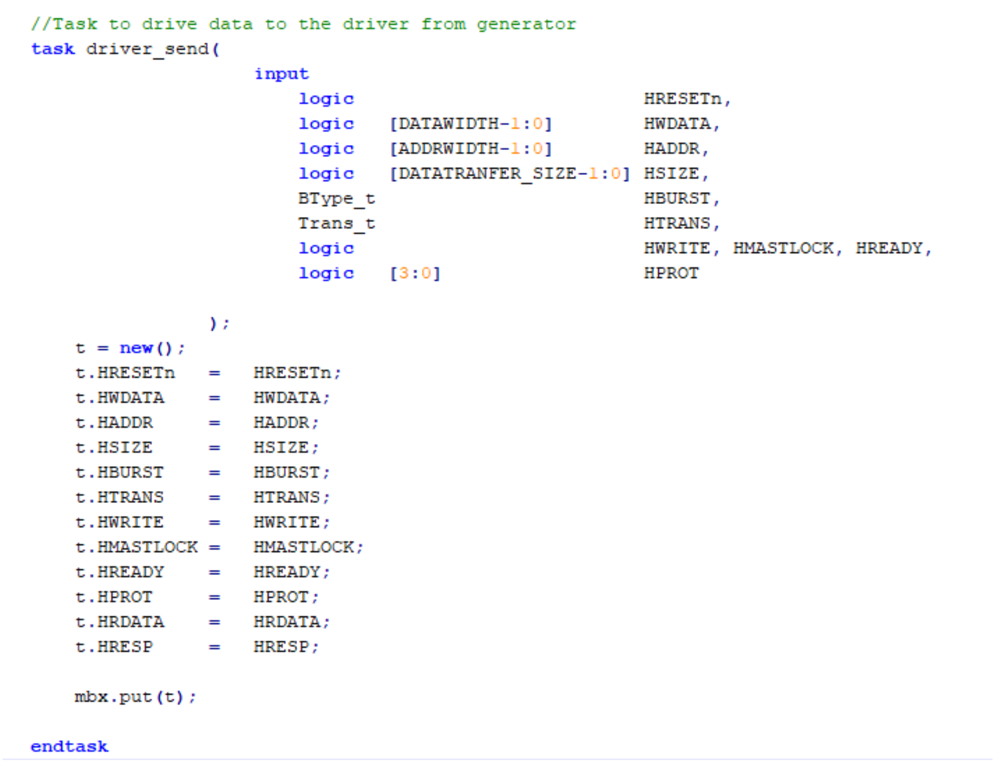
`//Constraint on selecting type of operation`
`constraint tests {`
`        test_case <= 15;`
`        test_case >= 0;`
`    }`
	For selecting testcase in generator

3 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

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

```
//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
```

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
```

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

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

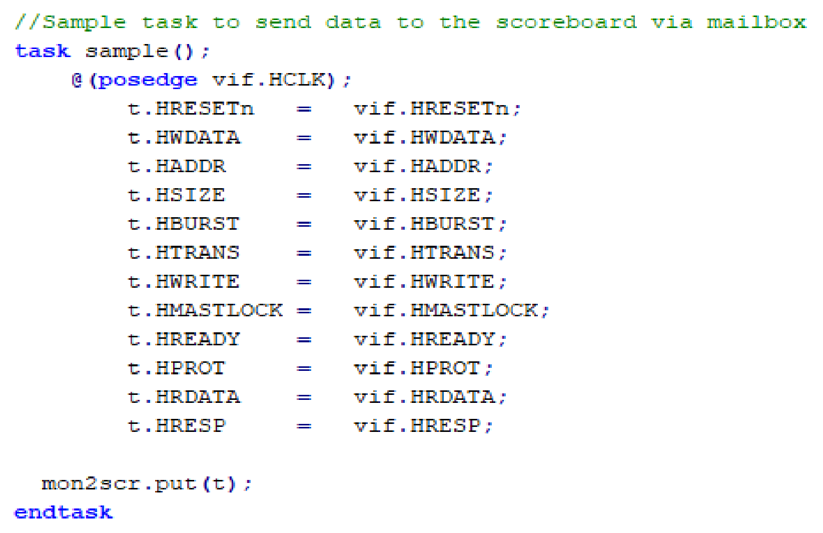
Cross coverage

`//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 as per the design for all burst types

6 monitor.sv

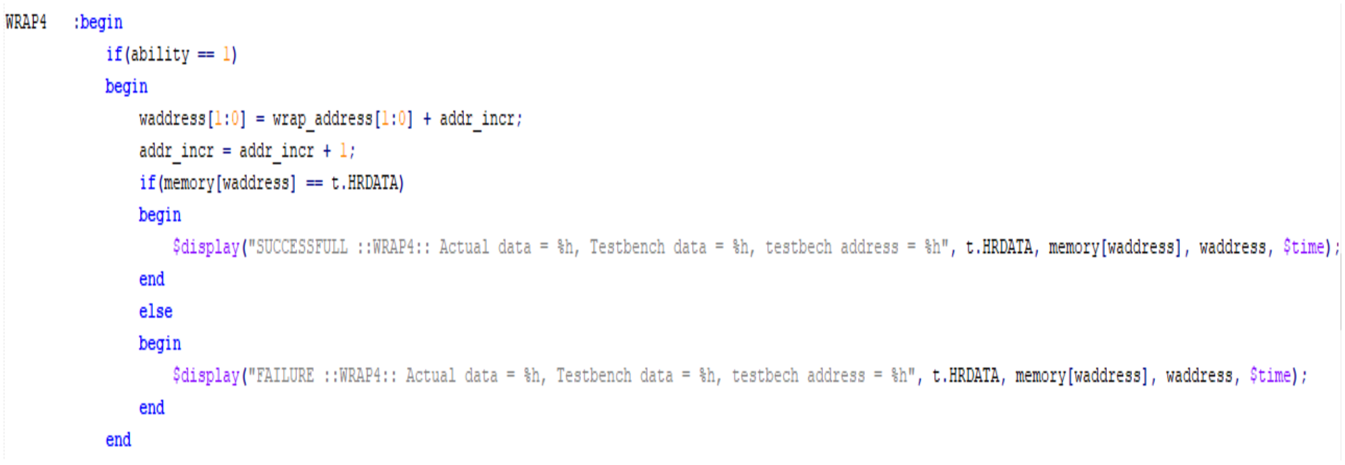
1. This class contains a monitor class

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

```
//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
```

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
```

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**

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

10 top.sv

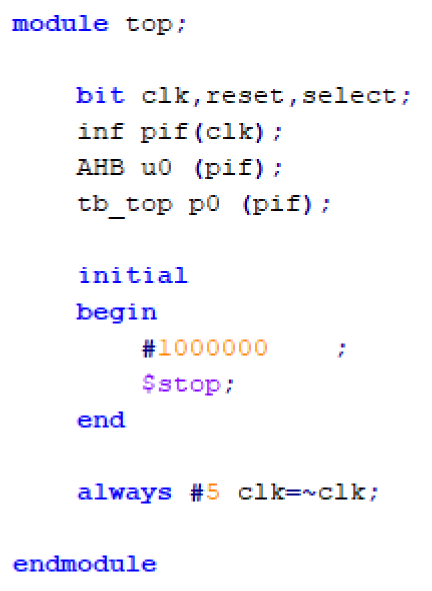
1. This file consists a module named AHB and has an input as interface.

```
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.

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.

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.