EE 8350 Advanced Verification Methodologies for VLSI Systems
LAB 6

OBJECTIVE: To learn about the monitor and scoreboard, and their implementations in UVM.

In this lab, we will use a **RAM** RTL code having the following properties:

RTL Port Interface:

Signal	**Port name**	**Width (bits)**	**Direction**	**Description**
Clock	clk	1	Input	Clock signal
Address	address	8	Input	Provided by the testbench for read/write requests
Data	data	16	Bidirectional	Bidirectional data port for read/write requests
Read request	rd_req	1	Input	Asserted along with a valid address for a read request
Write request	wr_req	1	Input	Asserted along with valid address and data for a write request
Read valid	rd_valid	1	Output	Asserted along with a valid data for a read response

Table 1

The RAM will have the capability to store 256 data words, each of 16-bit width.

A read transaction uses the following protocol:

Cycle 1: rd_req input signal is asserted along with a valid **address**.

Cycle 2: rd_valid output signal is asserted along with valid **data** in the bidirectional data port.

A write transaction uses the following protocol:

Cycle 1: wr_req input is asserted along with valid **data** and **address**. No acknowledgement is provided.

We will build a monitor and scoreboard for the above-described RTL module.

UVM Monitor:
A monitor is a passive entity that samples the DUT signals through a virtual interface and converts the signal level activity to the transaction level. The monitor samples DUT signals but does not drive them. The monitor should have an analysis port (a TLM port) and a virtual interface handle that points to the DUT signals. TLM ports are used to facilitate communication between modules.

We will begin by building a **modport** for the monitor. Since this design (the RAM) has a clock signal, we are use a clocking block as a part of the interface.

Clocking block: A clocking block identifies a clock signal and captures the timing and synchronization requirements of the other signals being modeled. A clocking block groups a set of signals that are synchronous to a specified clock edge.

Step 1: Define a clocking block and a modport for the driver and monitor, respectively. Add the following code in the file ***ram_interface.sv***:

```
clocking driver_cb @ (negedge clk);
      output address;
      inout data;
      output rd_req;
      output wr_req;
      input rd_valid;
  endclocking : driver_cb

  modport driver_if_mp (clocking driver_cb);

  clocking monitor_cb @ (negedge clk);
      input address;
      input data;
      input rd_req;
      input wr_req;
      input rd_valid;
  endclocking : monitor_cb

  modport monitor_if_mp (clocking monitor_cb);
```

Note: Observe the directions of the signals in the clocking block. In the driver, the direction of signals is opposite to those of the RTL port list because driver drives the signals to the DUT. In the monitor modport, the signals are all input. This is because the monitor only samples the signals.

Before going to the monitor code, we will learn about TLM (Transaction Level Modeling) interfaces in UVM.

TLM interface:

A TLM interface is used to connect components in order to transfer data, in the form of transactions, between them. It consists of one or more methods used to transport information, typically entire transactions (i.e., objects) at a time. For example, a sequencer communicates the transactions to be driven to the DUT using a port-export mechanism. A monitor can send transactions from the DUT to the scoreboard using an analysis port mechanism.

Advantages:

A TLM interface provides a useful abstraction by allowing the communication to happen at the transaction level. Designs that make use of TLM ports and exports to communicate are both reusable and modular.

UVM provides ports, imports, exports and analysis ports for connecting components via the TLM interfaces.

Mechansim:

Any TLM communication involves two components, the producer and the consumer. The producer generates transactions and the consumer receives the transactions.

TLM put operation

- A put() method allows a producer to send a transaction to a consumer.

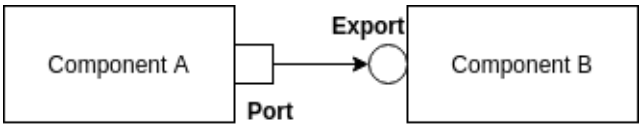

Figure 1

Component A - Producer
Component B - Consumer

Component A initiates sending the packet to component B using a put() method.
Component B is configured to simply accept the data it gets from component A.

The Port in Component A is of the class type: uvm_blocking_put_port
The Export in Component B is of the class type: uvm_blocking_put_imp
The implementation of the put() method is defined in Component B.

TLM get operation

- A get() method allows a consumer to request a transaction from a producer.

Figure 2

Component A - Producer
Component B - Consumer

Component B requests data from component A using the get() method.

The Port in Component B is of the class type: uvm_blocking_get_port
The Export in Component A is of the class type: uvm_blocking_get_imp
The implementation of the get() method is defined in Component A.

In other words, a port is always an initiator of a request and an export is always a provider of that request.

Communication between a Driver and a Sequencer:

A UVM driver and a UVM sequencer are connected using a UVM sequence_item port and an export. The port and export are defined in the uvm_driver and uvm_sequencer base classes.

- The UVM driver instantiates the seq_item_port, and the UVM sequencer instantiates the seq_item_export.
- The seq_item_port of the UVM driver is an object of the **uvm_seq_item_pull_port** class.
- The seq_item_export of the UVM sequencer is an object of the **uvm_seq_item_pull_imp** class.
- They are connected together in the connect phase.
  Example: **`driver.seq_item_port.connect( seqr.seq_item_export );`**
- In the driver, when seq_item_port.get_next_item() is called, the task calls the get_next_item() of the seq_item_export, which in turn calls the get_next_item() of uvm_sequencer.
- The driver gets the transaction from the sequencer and drives it to the DUT.
- Similarly, when seq_item_port.item_done() is called, the function calls item_done() of the seq_item_export, which in turn calls item_done() of uvm_sequencer.

Communication between the Driver/Monitor and the Scoreboard:

An **analysis port** is used to perform non-blocking broadcasts of transactions. A non-blocking broadcast implies that a stream of transactions is sent through the port regardless of whether a component is connected to the port. It is used by drivers and monitors to broadcast

transactions to scoreboards and coverage collectors. More than one component can be connected to each such port.

The **uvm_analysis_port** consists of a single function, write(). The component connected to the analysis port should provide an implementation of the write() method.

In the testbench provided for this lab, the scoreboard requires two analysis exports - one from the driver (i.e., the expected packet) and one from the monitor (i.e., the actual packet from the DUT). We use the **`uvm_analysis_imp_decl** macro to define two write functions. The driver and monitor will call the corresponding write methods to send the transaction to the scoreboard. This will be explained with an example while describing the scoreboard code.

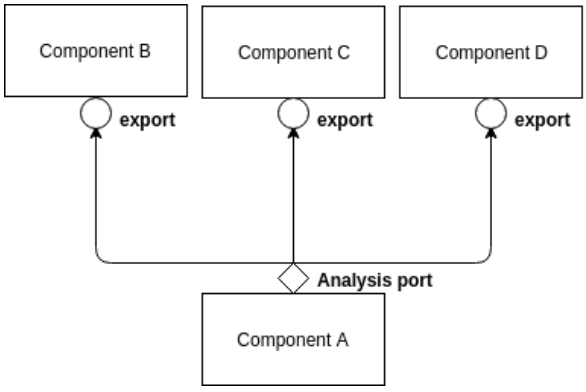

Figure 3

Component A - A driver or a monitor
Component B,C,D – A scoreboard, a coverage collector, etc.

Now for the actual driver and monitor code:

Step 2: In the **ram_driver.sv** file:

- Declare the analysis port to broadcast results to the scoreboard. An analysis port is a TLM interface used for communication between two modules (here, they are the driver and the scoreboard). Copy the following code:

```
uvm_analysis_port #(ram_transaction) Drv2Sb_port;
```

- In the build_phase, instantiate the analysis port Drv2Sb_port. Copy the following code:

```
Drv2Sb_port = new("Drv2Sb",this);
```

- In the run phase, as seen in last lab, the driver gets the transaction from the sequencer and drives it to the RTL. Here, the driver asserts the **rd_req** or **wr_req** signal based on whether it is a read or write transaction.

- Broadcast the transaction to the scoreboard:

```
Drv2Sb_port.write(req);
```

Step 3: In the ***ram_monitor.sv*** file:

- Register the monitor in the UVM factory.

```
`uvm_component_utils(ram_monitor)
```

- Declare a virtual interface:

```
virtual ram_interface ram_vif;
```

- Declare the analysis port to broadcast results to the scoreboard. An analysis port is a TLM interface used for communication between two modules (here, they are the monitor and the scoreboard). Copy the following code:

```
uvm_analysis_port #(ram_transaction) Mon2Sb_port;
```

- In the build phase, get the virtual interface from the config database:

```
if(!uvm_config_db#(virtual ram_interface)::get(this, "",
"ram_vif", ram_vif)) begin
      `uvm_error("", "uvm_config_db::get failed")
end
```

- Instantiate the analysis port Mon2Sb_port. Copy the following code:

```
Mon2Sb_port = new("Mon2Sb",this);
```

- In the run phase, the interface signals must be sampled and packed into an object of type **ram_transaction** and sent to the scoreboard for comparison with the expected packet. Declare an object of type ram_transaction and instantiate it.

```
ram_transaction trans;
trans = new ("trans");
```

  In a forever loop, continuously monitor the interface signals. If **`rd_valid`** is asserted, pack the sampled data and address into a read transaction. If **`wr_req`** is asserted, pack the sampled data and address into a write transaction. Broadcast the generated packet to the scoreboard using the **`Mon2Sb_port`**.

```
fork
      forever begin
      @(ram_vif.monitor_if_mp.monitor_cb)
      begin
```

```
            if(ram_vif.monitor_if_mp.monitor_cb.rd_valid)
            begin
                  trans.address =
ram_vif.monitor_if_mp.monitor_cb.address;
                  trans.data = ram_vif.monitor_if_mp.monitor_cb.data;
                  trans.rd_req = 1'b1;
                  Mon2Sb_port.write(trans);
            end
            if(ram_vif.monitor_if_mp.monitor_cb.wr_req)
            begin
                  trans.address =
ram_vif.monitor_if_mp.monitor_cb.address;
                  trans.data = ram_vif.monitor_if_mp.monitor_cb.data;
                  trans.wr_req =
ram_vif.monitor_if_mp.monitor_cb.wr_req;
                  Mon2Sb_port.write(trans);
            end
      end
      end
join
```

UVM Scoreboard:

The scoreboard will check the correctness of the DUT by comparing the DUT output with the expected values. The scoreboard will receive the expected transactions from the driver. It will also receive the actual (i.e., values from the DUT) transactions from the monitors implemented inside agents. A driver/monitor and scoreboard will communicate via TLM ports and exports.

In our scoreboard design, we will receive packets from the driver and the monitor, and we will store them in various FIFOs. Write and read transactions from the driver will be stored in two different FIFOs. Also, write and read transactions from the monitor will be stored in two different FIFOs. Hence, we need a total of four FIFOs - two for the driver and two for the monitor.
For the scoreboard's checking functionality, we will get packets from the driver write FIFO and from the monitor read FIFO, and then compare the data in both of these packets. This will ensure that the data sent by the driver during the write operation and the data sent by the RTL during the read operation have the same values.

Now for the actual scoreboard code:

Step 4: In the ***ram_scoreboard.sv*** file:

- We need 2 import ports in the scoreboard, for the expected packet which is sent by the driver and for the received packet which is coming from the monitor. Both ports will have an implementation function named write(). In order to differentiate between both of these write functions, we will need to provide different function names for them. **`uvm_analysis_imp_decl** macros are used for that purpose.

- The macro **`uvm_analysis_imp_decl(<_portname>)** is used to declare the uvm_analysis_imp_<_portname> class. The macro creates **write_<_portname>()**. This method has to be defined as per our requirements.
- The following macros should be placed outside of the class:

```
`uvm_analysis_imp_decl(_mon_trans)
`uvm_analysis_imp_decl(_drv_trans)
```

- Register the scoreboard in the UVM factory. (This and the following points are to be written inside of the class.)

```
`uvm_component_utils(ram_scoreboard);
```

- Declare two objects of the ***ram_transaction*** class - one for handling transactions from the driver and other from the monitor.

```
ram_transaction trans, input_trans;
```

- Declare two implementation ports.

```
uvm_analysis_imp_mon_trans #(ram_transaction,ram_scoreboard) Mon2Sb_port;
uvm_analysis_imp_drv_trans #(ram_transaction,ram_scoreboard) Drv2Sb_port;
```

- Declare TLM FIFOs to store the actual and expected transaction values.

```
uvm_tlm_fifo #(ram_transaction)  drv_wr_fifo;
uvm_tlm_fifo #(ram_transaction)  drv_rd_fifo;
uvm_tlm_fifo #(ram_transaction)  mon_wr_fifo;
uvm_tlm_fifo #(ram_transaction)  mon_rd_fifo;
```

- Instantiate the analysis ports and FIFOs in the build_phase.

```
Mon2Sb_port = new("Mon2Sb",  this);
Drv2Sb_port = new("Drv2Sb",  this);
drv_wr_fifo       = new("drv_wr_fifo", this,8);
drv_rd_fifo       = new("drv_rd_fifo", this,8);
mon_wr_fifo       = new("mon_wr_fifo", this,8);
mon_rd_fifo       = new("mon_rd_fifo", this,8);
```

- Two tasks need to be written in the scoreboard. One is called when the driver broadcasts a transaction to the scoreboard and other is called when the monitor broadcasts a transaction to the scoreboard. They are named using the convention **write_<string passed in uvm_analysis_imp_decl macro>.** The functionality of the task is to collect the transaction from the driver or monitor and add it to the corresponding FIFO.

```
function void write_drv_trans (ram_transaction input_trans);
```

```
        // Put the write values in the EXPFIFO
        if(input_trans.wr_req == 1)
        begin
        void'(drv_wr_fifo.try_put(input_trans));
        end

        //Put the read values in the EXPFIFO
        if(input_trans.rd_req == 1)
        begin
        void'(drv_rd_fifo.try_put(input_trans));
        end

    endfunction : write_drv_trans
    function void write_mon_trans (ram_transaction trans);

        // Try putting the write values in the EXPFIFO
        if(trans.wr_req == 1)
        begin
        void'(mon_wr_fifo.try_put(trans));
        end

        // Try putting the read values in the EXPFIFO
        if(trans.rd_req == 1)
        begin
        void'(mon_rd_fifo.try_put(trans));
        end

    endfunction : write_mon_trans
```

- In the run_phase task, get the values from the driver write and monitor read FIFOs and compare them. Flag an error if there is any mismatch. Declare transactions to get them from a FIFO:

```
ram_transaction exp_trans, out_trans;
```

Get the expected transaction from the driver write FIFO:

```
drv_wr_fifo.get(exp_trans);
```

Get the actual transaction from the monitor read FIFO:

```
mon_rd_fifo.get(out_trans);
```

Compare them and display the result:

```
if (exp_trans.data == out_trans.data) begin
        `uvm_info("PASS ", $sformatf("Actual=%h Expected=%h \n",
```

```
        out_trans.data, exp_trans.data), UVM_LOW)
      end
      else begin
        `uvm_error("ERROR", $sformatf("Actual=%d Expected=%d \n",
        out_trans.data, exp_trans.data))
end
```

Step 5: In **ram_env.sv**,

- Declare all of the environment components:

```
ram_agent agent;
ram_monitor mon;
ram_scoreboard sb;
```

- In the build phase, instantiate the components:

```
agent = ram_agent::type_id::create("agent", this);
mon   = ram_monitor::type_id::create("mon",this);
sb    = ram_scoreboard::type_id::create("sb",this);
```

- In the connect phase, connect the driver's analysis port to the scoreboard analysis implementation port.

```
agent.driver.Drv2Sb_port.connect(sb.Drv2Sb_port);
```

- Connect the monitor's analysis port to the scoreboard analysis implementation port.

```
mon.Mon2Sb_port.connect(sb.Mon2Sb_port);
```

Step 6: Run the simulation using the following command and take a screenshot of the simulation result.

```
vcs -cm line+cond+fsm -Mupdate +v2k -sverilog -timescale=1ns/10ps
+incdir+$UVM_HOME/src $UVM_HOME/src/uvm.sv $UVM_HOME/src/dpi/uvm_dpi.cc -
CFLAGS -DVCS top.sv ram_interface.sv  ram.v -l compile.log
+vcs+dumpvars+verilog.vpd -debug_all

./simv -cm line+cond+fsm +UVM_TESTNAME=ram_test
```

Step 7: Run the following command to view the waveform:

dve -vpd vcdplus.vpd

Step 8: The dve tool opens upon running the previous command.
Select ***"top"*** from the ***hierarchy*** column and click the + button beside it.
Select "***ram_if"*** subtab.
Select all signals that appears under the ***variable*** column.
Press ***Ctrl + 4*** to view the signals in the form of a waveform.
Use the □ , 2 , ½ option on dve to zoom into one read and one write operation.
Go to ***File->print***
Select ***Image***.
Select ***ok***.
The image gets stored in the same directory as your simulation run. Attach the image in your report. You may choose to the show the read/write operations in the same image or in different images - either way is acceptable.

A read operation image should include:
- Address
- Data
- Rd_req being asserted
- Rd_valid being asserted
- Clock cycles

A write operation image should include:
- Address
- Data
- Wr_req being asserted
- Clock cycles

Make sure that the numerical values are clearly visible in the image(s).

Questions:

1. Note that in **ram_sequence.sv**, the top sequence named **ram_sequence** generates two subsequences, **rd_sequence** and **wr_sequence**. Observe how these subsequences are started. What is m_sequencer and what is it used for?
2. Make the following change in **ram_interface.v**
   clocking driver_cb @ (negedge clk);
   output address;
   inout data;
   output rd_req;
   output #2 wr_req;
   input rd_valid;
   endclocking : driver_cb
   Follow steps 6,7,8. What difference do you see in the write transaction?

3. What is a TLM FIFO and what is its function?
4. State whether a port or an export must be defined in locations 1 and 2 in the following figure:

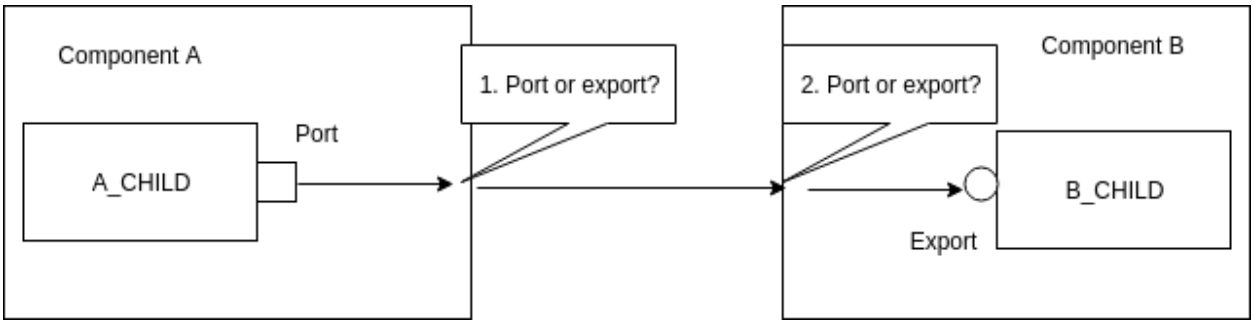

Figure 4

What is to be turned in?

A report (pdf file) consisting of the screenshot of Step 6, the waveform image(s) as described in Step 8, the waveform image(s) for question 2 and the answers to the above questions.

References used in the preparation of this lab:

1. http://www.verificationguide.com
2. https://www.edaplayground.com
3. http://testbench.in
4. http://www.asic-world.com
5. https://verificationacademy.com
6. http://systemverilog.us/driving_into_wires.pdf
7. http://www.eetimes.com/document.asp?doc_id=1276112
8. http://www.embedded.com/print/4004083
9. https://colorlesscube.com/uvm-guide-for-beginners
10. https://www.doulos.com
11. http://www.chipverify.com/uvm