EE 8350 Advanced Verification Methodologies for VLSI Systems
LAB 8

OBJECTIVE: To learn about UVM sequence arbitration and active/passive agents.

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

RTL Port Interface:

Signal	**Port name**	**Width (bits)**	**Direction**	**Description**
Clock	clk	1	Input	Clock signal
Ready Input	ready_in	1	Input	Indicate that the Input is ready to be read
Data Input	data_in	8	Input	Input data bus
Reset	reset	1	Input	Reset signal
Port Number	port_num	2	Input	Port number to which the input data must be routed to
Ready Output	ready_out_(n)	1	Output	Indicate that the output is ready at port number 'n'
Data Output	data_out_(n)	8	Output	Output data bus of port number 'n'

Table 1

RTL Behaviour:
The Switch drives the input data (**data_in**) to the output port (**data_out)** that is specified by the port number (**port_num**). The inout data is considered valid if the **ready_in** signal is asserted. The switch doesn't add any propagation delay, i.e. the output toggles in the same cycle as the input. The output data in each port is considered valid if the corresponding ready signal (**ready_out**) is asserted.

RTL Block Diagram:

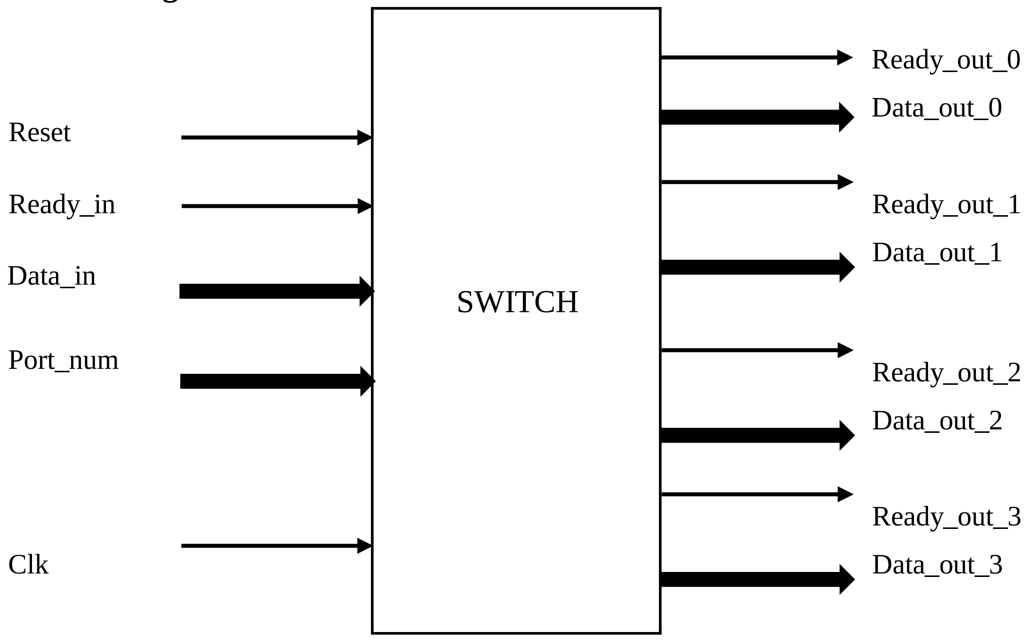

Fig 1

Testbench Architecture:

When used in a VLSI chip, a switch is used to route data packets from one component to another. We can infer that the input and output signals of the switch are connected to at least two different components. This suggests that the switch signals can be divided into two types of interfaces:

- input interface – consisting of the input signals
- output interface – consisting of the output signals

Step 1: In switch_input_interface.sv,

- Define the input interface signals.

```
logic reset;
logic ready_in;
logic [7:0] data_in;
logic [1:0] port_num;
```

- Define the clocking block for the driver. Note: We drive the signals to the DUT on the negedge of clock.

```
clocking driver_cb @ (negedge clk);
  output reset;
  output ready_in;
  output data_in;
  output port_num;
endclocking : driver_cb
```

- Define the driver modport.

```
modport driver_if_mp (clocking driver_cb);
```

- Define the clocking block for the monitor. Note: Here, we monitor signals driven to the DUT. This interface is for a monitor in the active agent (to be explained later).

```
clocking monitor_cb @ (negedge clk);
  input reset;
  input ready_in;
  input data_in;
  input port_num;
endclocking : monitor_cb
```

- Define the monitor modport.

```
modport monitor_if_mp (clocking monitor_cb);
```

Step 2: In switch_output_interface.sv:

- Define the output interface signals. Note: Here, we monitor signals driven from the DUT. This interface is for a monitor in the passive agent (to be explained later).

```
logic ready_out;
logic [7:0] data_out;
```

- Define the clocking block for the monitor.

```
clocking monitor_cb @ (negedge clk);
  input ready_out;
  input data_out;
endclocking : monitor_cb
```

- Define the monitor modport.

```
modport monitor_if_mp (clocking monitor_cb);
```

Step 3: In top.sv,

- Instantiate the interfaces.

```
switch_input_interface switch_input_if(clk);
switch_output_interface switch_output_if[3:0](clk);
```

  Note that we have created an array of output interfaces consisting of 4 entries (one for each port).

- Set the output interface in the configuration data base using a generate block.

```
generate
  genvar i;
  for(i =0; i < 4; i =i+1) begin
     initial
     begin

       uvm_config_db#(virtual  switch_output_interface)::set(null,
"*", $sformatf("switch_output_vif_%0d",i), switch_output_if[i]);

     end
  end
  endgenerate
```

- Set the input interface in the configuration database (in the initial block).

```
uvm_config_db#(virtual   switch_input_interface)::set(null,   "*",
"switch_input_vif", switch_input_if);
```

To verify the proper working of the switch, the testbench drives the input data along with the appropriate port number through the input interface and ensures that the same data is received at the correct port in the output interface. To realize this, we need a driver at the input interface and 4 monitors at the output interfaces (i.e., one for each port). We know that the driver and monitor are a part of the agent. UVM allows us to differentiate between an agent at an input or output interface using the concept of active and passive agents.

Active Agent:

- Drives stimulus to the DUT.
- Instantiates a driver, sequencer and monitor.
- In UVM, an agent can be configured to be active by setting the **is_active** variable to UVM_ACTIVE.

Passive Agent:

- Only monitors the signals from the DUT; it doesn’t drive them.
- Instantiates only the monitor.
- Used for checking and coverage only.
- In UVM, an agent can be configured to be passive by setting the **is_active** variable to UVM_PASSIVE.

The testbench architecture using active/passive agents is as shown below:

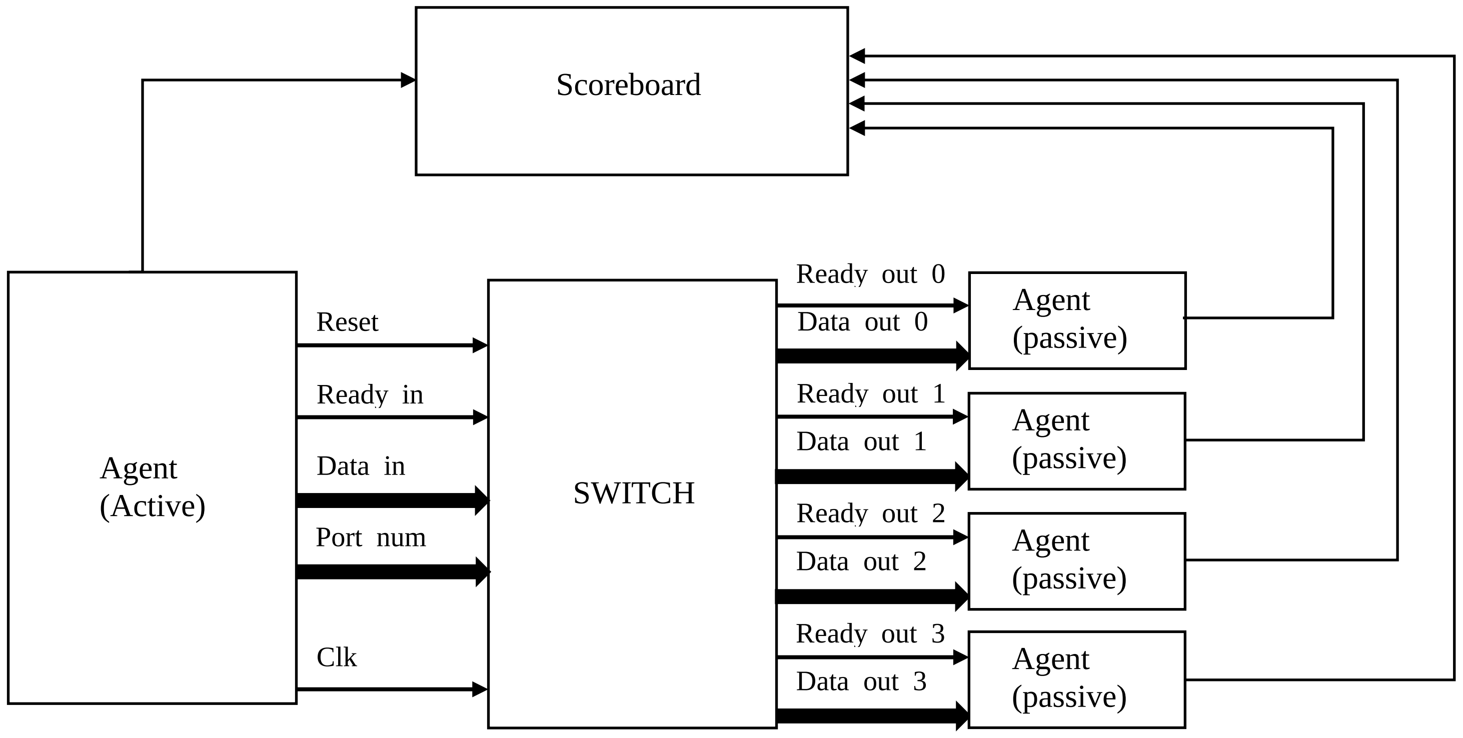

Fig 2

The scoreboard is used to compare the data packet driven to the input interface through the active agent with the data packet monitored by the passive agents from the output interfaces.

Let us build the input agent (active).

Step 4: In switch_agent_input.sv,

- Define the driver, monitor and sequencer.

```
switch_driver driver;
uvm_sequencer#(switch_transaction) sequencer;
switch_monitor_input mon;
```

- Instantiate the driver, monitor and sequencer in the build_phase.

```
driver = switch_driver::type_id::create("driver", this);
sequencer =
uvm_sequencer#(switch_transaction)::type_id::create("sequencer",
this);
mon   = switch_monitor_input::type_id::create("mon",this);
```

- Connect the sequencer to the driver in the connect_phase.

```
driver.seq_item_port.connect(sequencer.seq_item_export);
```

Step 5: In switch_driver.sv,

- Define the virtual interface to get the input interface.

```
        virtual switch_input_interface switch_vif;
```

- Get the input interface reference from the configuration database.

```
          if(!uvm_config_db#(virtual  switch_input_interface)::get(this,
      "", "switch_input_vif", switch_vif)) begin
            `uvm_error("", "uvm_config_db::get failed")
          end
```

- In the run_phase (inside the forever loop), get the next item from sequencer.

```
seq_item_port.try_next_item(req);
```

- Drive the packets to the DUT. The following code also counts the number of packets driven to each port and displays the value.

```
if(req != null)
begin
    // Wiggle pins of DUT
    @(switch_vif.driver_if_mp.driver_cb)
    begin
        switch_vif.driver_if_mp.driver_cb.ready_in <= 1'b1;
        switch_vif.driver_if_mp.driver_cb.data_in <= req.data_in;
        switch_vif.driver_if_mp.driver_cb.port_num <= req.port_num;
    end

    //Calculate the number of packets driven in each sequence
    case(req.port_num)
      2'b00 : begin
                seq_1++;
              end
      2'b01 : begin
                seq_2++;
              end
      2'b10 : begin
                seq_3++;
              end
      2'b11 : begin
                seq_4++;
              end
    endcase

    `uvm_info("DRIVER",$sformatf("seq_1=%0d      seq_2=%0d      seq_3=%0d
seq_4=%0d",seq_1,seq_2,seq_3,seq_4),UVM_HIGH);
```

```
        seq_item_port.item_done();
    end
    else
    begin
        //Reset the ready_in after driving packet
        @(switch_vif.driver_if_mp.driver_cb)
        begin
            switch_vif.driver_if_mp.driver_cb.ready_in <= 1'b0;
        end
    end
```

Step 6: In switch_monitor_input.sv,

- Define the virtual interface to get the input interface.

```
        virtual switch_input_interface switch_vif;
```

- Get the input interface reference from the configuration database.

```
          if(!uvm_config_db#(virtual  switch_input_interface)::get(this,
      "", "switch_input_vif", switch_vif)) begin
            `uvm_error("", "uvm_config_db::get failed")
          end
```

- Send the packet to the scoreboard, in the run_phase.

```
                switch_transaction trans;
                trans = new ("trans");
                fork
                  forever begin
                    @(switch_vif.monitor_if_mp.monitor_cb)
                    begin
                    if(switch_vif.monitor_if_mp.monitor_cb.ready_in)
                         begin
                           trans.data_in =
      switch_vif.monitor_if_mp.monitor_cb.data_in;
                           Mon2Sb_port_input.write(trans);
                         end
                    end
                  end
                join
```

Now, let us build the output agent (passive).

Step 7: In switch_agent_output.sv,

- Define the monitor.

```
switch_monitor_output mon;
```

- Instantiate the monitor in the build_phase.

```
mon   = switch_monitor_output::type_id::create("mon",this);
```

Step 8: In switch_monitor_output.sv,

- Define the virtual interface to get the input interface.

```
virtual switch_output_interface switch_vif;
```

- Get the output interface reference from the configuration database.

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

- Send the packet to the scoreboard, in the run_phase.

```
         switch_transaction trans;
         trans = new ("trans");
         fork
           forever begin
             @(switch_vif.monitor_if_mp.monitor_cb)
             begin
                  if(switch_vif.monitor_if_mp.monitor_cb.ready_out)
                  begin
                    trans.data_out =
switch_vif.monitor_if_mp.monitor_cb.data_out;

                    Mon2Sb_port_output.write(trans);
                  end
             end
           end
         join
```

Step 9: In switch_env.sv,

- Declare input and output agents. Note that we have defined an array of output agents.

```
switch_agent_input input_agent;
switch_agent_output output_agent[4];
```

- Declare an array of virtual output interfaces.

```
virtual switch_output_interface switch_vif[4];
```

- Instantiate the input agent in the build_phase and set the is_active field to UVM_ACTIVE.

```
input_agent = switch_agent_input::type_id::create("input_agent", this);
input_agent.is_active = UVM_ACTIVE;
```

- Instantiate the output agents using a for loop.

```
   for(int i =0; i < 4; i++) begin

         string out_agent_name = $sformatf("output_agent_%0d",i);

         output_agent[i] =
switch_agent_output::type_id::create(out_agent_name, this);
```

- For each output agent created, get the corresponding output interface.

```
         if(!uvm_config_db#(virtual switch_output_interface)::get(this,
"", $sformatf("switch_output_vif_%0d",i), switch_vif[i])) begin
           `uvm_error("", "uvm_config_db::get failed")
         end
```

- Set the output interface in the configuration database such that only the corresponding output agent can get it.

```
         uvm_config_db#(virtual switch_output_interface)::set(null,
$sformatf("uvm_test_top.env.%0s.mon",out_agent_name), "switch_vif",
switch_vif[i]);
```

- End the for loop.

```
      end
```

- Points to note here:
  - In top.sv, we set the output interface in the configuration database under the names **`switch_output_vif_0, switch_output_vif_1`** and **`switch_output_vif_0, switch_output_vif_3.`**
  - In the above step, we get the interfaces from the configuration database and set them back under same name **`(switch_vif)`** but with different instance names (agent instances). This ensures that only the specified instance can get the interface.
  - We have to do the above steps because the agent objects are not available in the top file.
- Connect the input agent's driver analysis port to the scoreboard in the connect_phase.

```
input_agent.mon.Mon2Sb_port_input.connect(sb.Drv2Sb_port);
```

- Connect each output agent's monitor analysis port to the scoreboard in the connect_phase.

```
for(int i =0; i < 4; i++) begin
  output_agent[i].mon.Mon2Sb_port_output.connect(sb.Mon2Sb_port);
end
```

The scoreboard code is similar to what we wrote in Lab 6.

Before we discuss the sequence item, sequence and the sequencer of the testbench, we will discuss one of the important features of UVM – Sequence Arbitration.

UVM Sequence Arbitration:
As we have seen in the earlier labs, the UVM driver is used to provide stimulus to the DUT through signal/pin level transitions. The inputs driven to the DUT are provided by the UVM sequence item or transaction. The UVM sequencer does the job of transferring sequence items from a sequence to the driver.

To verify any DUT, it is often necessary to provide inputs using multiple sequences. For example, we required two sequences (read and write) in Lab 6 in order to perform read/write tasks on the RAM module. It is a good practice to conglomerate sequence items verifying similar features of the DUT into one sequence. A well architected testbench would contain many such sequences. It is important to generate multiple sequence items in each sequence to increase the chances of achieving 100% functional coverage and thereby thoroughly verifying the DUT.

In this type of scenario, multiple sequences compete at the same time to get the allocation of the sequencer. In UVM, the uvm_sequencer has a built-in mechanism to arbitrate between multiple sequences that are running concurrently. This feature of the UVM sequencer is called Sequence Arbitration and it determines which sequence gets the grant to send its transaction items to the driver.

UVM provides 6 different arbitration mechanisms. The set_arbitration() task of the UVM sequencer is used to select one of these six schemes. The name and functionality of each arbitration scheme is given below.

ARBITRATION SCHEME	**FUNCTIONALITY**
UVM_SEQ_ARB_FIFO	Starts the sequences in first-in, first-out order.
UVM_SEQ_ARB_WEIGHTED	Higher priority given for higher weights.
UVM_SEQ_ARB_RANDOM	Start sequences in random order. Weights are not considered.
UVM_SEQ_ARB_STRICT_RANDOM	Higher priority for higher weights. Sequences having the same weight are picked at random.
UVM_SEQ_ARB_STRICT_FIFO	Higher priority for higher weights. Sequences having the same weight are processed in a FIFO order.
UVM_SEQ_ARB_USER	Create a user-defined arbitration scheme.

TABLE 2

In this lab, we will construct 4 sequences. Each sequence will generate transactions to one of the four output ports. All sequences will use the same sequencer to send the transactions to the driver. We will work with the various arbitration schemes in the above table.

Step 10: In switch_sequence.sv,

- Declare the transaction fields in the switch_transaction class.

```
bit [1:0] port_num;
rand bit [7:0] data_in;
bit [7:0] data_out;
```

- Generate transactions to port 0 by adding the following code to the body task of the sequence_1 class.

```
repeat(8)
begin

   req = switch_transaction::type_id::create("req");

   start_item(req);
```

```
    if (!req.randomize()) begin
      `uvm_error("SEQUENCE_1", "Randomize failed.");
    end

    req.port_num = 2'b00;

    finish_item(req);
end
```

- Repeat the above step in sequences 2, 3 and 4 by changing the port_num to 2’b01, 2’b10 and 2’b11, respectively.
- In the main_sequence, declare the four sequences defined above.

```
sequence_1 seq_1;
sequence_2 seq_2;
sequence_3 seq_3;
sequence_4 seq_4;
```

- In the new function, instantiate the four sequences.

```
seq_1 = sequence_1::type_id::create("seq_1");
seq_2 = sequence_2::type_id::create("seq_2");
seq_3 = sequence_3::type_id::create("seq_3");
seq_4 = sequence_4::type_id::create("seq_4");
```

- In the task body, set the arbitration scheme for the default sequencer.

```
m_sequencer.set_arbitration(UVM_SEQ_ARB_WEIGHTED);
```

- Start the sequences with a defined priority.

```
fork
  seq_1.start(m_sequencer,this,100);
  seq_2.start(m_sequencer,this,200);
  seq_3.start(m_sequencer,this,400);
  seq_4.start(m_sequencer,this,200);
join
```

Step 11: In switch_test.sv,

- Declare the instances of the environment and the main sequence.

```
switch_env env;
main_sequence main_seq;
```

- Instantiate the environment and the main_sequence.

```
env = switch_env::type_id::create("env", this);
main_seq = main_sequence::type_id::create("main_seq");
```

- Start the sequence by passing the sequencer as the argument.

```
main_seq.start(env.input_agent.sequencer);
```

Step 12: Run the simulation using the following command and take a screenshot of the simulation result. (There is no need to include all of the print statements, just include the report statements showing there is no error.)

```
      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 switch_input_interface.sv switch_output_interface.sv
switch.v -l compile.log -debug_all

      ./simv -cm line+cond+fsm +UVM_TESTNAME=switch_test
+ntb_random_seed=<your student ID> +UVM_VERBOSITY=UVM_HIGH
```

Step 13: Run the following command to view the waveform:

```
      dve -vpd vcdplus.vpd
```

Step 14: The dve tool opens upon running the previous command.
Select ***"top"*** from the ***hierarchy*** column and click the + button beside it.
Select the "***sw0"*** subtab.
Select the following signals that appear under the ***variable*** column.

clk
data_in
port_num
data_out_0
data_out_1
data_out_2
data_out_3

Press ***Ctrl + 4*** to view the signals in the form of a waveform.
Use the □ , 2 , ½ option on dve to zoom in/out.
Go to ***File->print***
Choose the ***image*** option under ***destination*** subdivision.
Name the image.
Set the ***Begin time*** = 0
Set the ***End time*** = 40000
Set the ***Time range per page*** = 10000
Select ***ok***.

Four images get stored in the same directory as for your simulation run. Attach the images in your report. Make sure that the numerical values are clearly visible in the image(s).

Step 15: Repeat steps 12-14 using each of the following arbitration schemes:

UVM_SEQ_ARB_FIFO,
UVM_SEQ_ARB_RANDOM,
UVM_SEQ_ARB_STRICT_RANDOM and
UVM_SEQ_ARB_STRICT_FIFO

Questions:

1. What is the difference between get_next_item() and get()?
2. Why do we need to use a generate block to set the output interfaces in the configuration database in top.sv?
3. Rank the order in which the 8 transactions for each sequence get driven for each of the arbitration schemes.
   For example, if seq_1 finishes driving all of its packets first followed by seq_2, seq_3 and seq_4, they are ranked 1, 2, 3, 4, respectively.

Sequence	WEIGHTED	FIFO	RANDOM	STRICT_RANDOM	STRICT_FIFO
Seq_1					
Seq_2					
Seq_3					
Seq_4					

What is to be turned in?

A report (pdf file) consisting of the screenshot of Step 12, the waveform image(s) as described in Step 14 for all the arbitration schemes 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://www.eetimes.com/document.asp?doc_id=1276112
7. http://www.embedded.com/print/4004083
8. https://colorlesscube.com/uvm-guide-for-beginners
9. https://www.doulos.com
10. http://www.chipverify.com/uvm