EE 8350 Advanced Verification Methodologies for VLSI Systems
LAB 7

OBJECTIVE: To learn about UVM subscriber, type and instance overriding in the UVM factory.

We will use a RAM example in this lab to illustrate the concepts.

UVM Subscriber:

This class receives transactions from a connected component. Making such a connection "subscribes" this component to any transactions sent out by the connected component.

Functional Coverage:

Functional coverage is a user-defined metric that measures how much of the design specification has been exercised during the verification process.

Covergroup:

This is a user-defined type. It is similar to a class where, the type definition is written once, and multiple instances of that type can be created in different situations. A covergroup is defined between the key words `covergroup` and `endgroup`. An instance of a covergroup is created using the new() operator.

A covergroup specification can include the following elements:

- A clocking event that synchronizes the sampling of coverage points
- A set of coverage points
- Cross coverage between different coverage points
- Optional formal arguments
- Coverage options

Coverpoint:

A coverpoint can be a variable or an expression. Each coverpoint has bins associated with it which keep track of the values that have occurred during the simulation.

The subscriber will sample the values of the fields of the incoming analysis transactions in a covergroup.

Functional Coverage for the RAM:

In our RAM example, the fields that are to be covered for complete functional verification are:

- Read request (**rd_req**)
- Write request (**wr_req**)
- Read and Write addresses (**address**)

This is to ensure that the RAM RTL is verified for both read and write requests. Also, we need to check if all address ranges are covered.

Step 1: In the file ***ram_subscriber.sv***, add the following:

- Register the subscriber in the UVM factory.

```
`uvm_component_utils(ram_subscriber)
```

- Declare variables to store read/write addresses, the read request and the write request.

```
bit wr_req;
bit [7:0] write_address;
bit rd_req;
bit [7:0] read_address;
```

- Define a covergroup and declare coverpoints inside the covergroup for read request, write request, read address and write address. Also, define bins for some of these coverpoints.

```
covergroup cover_ram;
      coverpoint wr_req;
      coverpoint write_address
       {
            bins low = {[8'h1:8'h3F]};
            bins med = {[8'h40:8'hBF]};
            bins high = {[8'hC0:8'hFF]};
       }
       coverpoint rd_req;
       coverpoint read_address
       {
            bins low = {[8'h1:8'h3F]};
            bins med = {[8'h40:8'hBF]};
            bins high = {[8'hC0:8'hFF]};
       }
endgroup
```

- Declare a virtual interface object.

```
virtual ram_interface ram_vif;
```

- Declare an analysis port to receive transactions from the monitor.

```
uvm_analysis_imp #(ram_transaction,ram_subscriber) aport;
```

- In the new function, call new() of the covergroup.

```
cover_ram = new();
```

- Get the virtual interface from uvm_config_db

```
            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 in the build phase.

```
            aport = new("aport",this);
```

- Implement a write function of the analysis port. In the write function, include the variables defined in the subscriber based on the values in the transaction sent from the monitor. Finally, call the sample method of the covergroup.

```
          function void write(ram_transaction t);
               if(t.wr_req)
               begin
               wr_req = t.wr_req;
               rd_req = t.rd_req;
               write_address = t.address;
               end
               if(t.rd_req)
               begin
               rd_req = t.rd_req;
               wr_req = t.wr_req;
               read_address = t.address;
               end
               cover_ram.sample();
           endfunction
```

Step 2: In the file **ram_env.sv**,

- Declare the subscriber object.

```
            ram_subscriber sub;
```

- In the build_phase, instantiate the subscriber object.

```
            sub   = ram_subscriber::type_id::create("sub",this);
```

- In the connect_phase, connect the monitor's analysis port to the subscriber's analysis implementation port.

```
            mon.aport.connect(sub.aport);
```

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

- Define an analysis port to broadcast to ram_subscriber.sv,

```
uvm_analysis_port #(ram_transaction) aport;
```

- Instantiate aport in the build_phase,

```
aport = new("aport",this);
```

- In the run_phase, send read and write transactions to the subscriber:

```
 //Send read transaction to subscriber
 aport.write(read_trans);

 //Send write transaction to subscriber
aport.write(write_trans);
```

Step 4: In the file **ram_testbench_pkg.sv**,

- Include ram_subscriber.sv in the package.

```
`include "ram_subscriber.sv"
```

Step 5: Run the simulation with the following command:

```
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 -debug_all

./simv -cm line+cond+fsm +UVM_TESTNAME=ram_test +ntb_random_seed=5258627
```

Note: The highlighted portion is for performing the functional coverage. You should be able to see a new directory called simv.vdb being created.

The directory simv.vdb contains the data collected by the coverage collector. Information regarding the coverpoints as well as the bins covered for each coverpoint in the current simulation are stored there. In order to generate a report that is understandable to the user, we will use the command called urg.

Step 6: Generate the functional coverage report using the following command:

```
urg -dir simv.vdb
```

This generates a new directory called urgReport. To view the report, open the dashboard.html in a web browser.

firefox urgReport/dashboard.html

You should be able to see the line, conditional, FSM and group coverage percentages. Line coverage denotes the number of lines of RTL that have been executed in the current simulation. Conditional coverage denotes the percentage of conditions in the RTL that are executed in the current simulation. For example, consider the condition:

```
if(a>b)
```

Suppose that this is a line in the RTL. This condition is said to be fully covered if it evaluates to both true and false during the simulation.

Group coverage is the coverage of the covergroups that you have defined. FSM coverage denotes the coverage of all states of an FSM in the RTL design. It is zero for this simulation, since there are no FSMs in our RTL design.

Clicking on the group coverage link will display all the covergroups that you have provided in the verification environment. Clicking on each covergroup link will list the corresponding coverpoints and their coverage information.

Step 7: Click on the **groups** tab. Click on **ram_testbench_pkg::ram_subscriber::cover_ram** to view the coverage for the coverpoints. Take a screen shot of the summary in **Summary for Group ram_testbench_pkg::ram_subscriber::cover_ram.**

Step 8: Obtain the waveform image as described in last lab. Also, take a screenshot of the simulation result.

UVM Factory:

Instead of using a new() function, in UVM we make calls to a look-up table to create the various components and transactions. This special look-up table is called the UVM factory. Entries into this look-up table are made by registering the components and transactions as they are defined. When you call the factory to create the requested component or transaction type, the factory itself will create the object by calling the constructor that was defined for that object. The factory also facilitates an object of one type to be substituted with an object of a derived type without having to change the structure of the testbench or having to modify the testbench code. This behavior is called “overriding.” Using this, a user can, from the environment or a test, replace any object which is at any level of the hierarchy with a user-defined object.

There are three basic steps to be followed when using the UVM factory:

- Registration
- Construction
- Overriding

Registration:

While defining a class, its type has to be registered with the factory. UVM has several predefined macros to facilitate this process:

```
`uvm_component_utils(class_type_name)
`uvm_component_param_utils(class_type_name #(params))
`uvm_object_utils(class_type_name)
`uvm_object_param_utils(class_type_name #(params))
```

The uvm_*_param_utils are used for parameterized classes and the other two macros are used for non-parameterized classes.

Construction:

To construct a UVM component or a UVM object, the static method create() should be used. This function constructs the appropriate object based on the overrides and returns it.

```
object_name = class_type::type_id::create("object_name",this);
```

Overriding:

There are two kinds of overriding:

- Type overriding
- Instance overriding

Type Overriding:

Type overriding means that every time a component class type is created within the testbench hierarchy, a substitute type, i.e. a derived class of the original component class, is created in its place. This applies to all of the instances of that component type. A factory-defined function is used for this purpose:

```
set_type_override_by_type(original_class_type, new_class_type,
replace);
```

where “replace” is a bit which, when set to 1, enables the overriding of an existing override. Otherwise, the existing override is maintained.

Instance Overriding:

As its name indicates, this substitutes only a particular instance of the component or a set of instances. The instance to be substituted is specified using the UVM component hierarchy. A factory-defined function is used for this purpose:

```
set_inst_override_by_type(“Instance path name”,original_class_type,
new_class_type);
```

The print() method prints the state of the uvm_factory, registered types, instance overrides and type overrides. It should be called at the end_of_elaboration phase.

We can use the factory overriding feature to develop error scenarios.

Step 9: Add the following code in the run phase of ***ram_driver_new.sv***:

```
uvm_report_info(get_full_name(), "New driver ",UVM_LOW);

forever begin
   seq_item_port.get_next_item(req);

   @(ram_vif.driver_if_mp.driver_cb)
   begin
   ram_vif.driver_if_mp.driver_cb.address <= req.address;
   ram_vif.driver_if_mp.driver_cb.data <= req.data;
   end

   seq_item_port.item_done();
end
```

Note: Make sure that **ram_driver_new.sv** is included in the file **ram_testbench_pkg.sv**.

Step 10: For type overriding, add the following line in the build phase of ***ram_test.sv***

```
set_type_override_by_type(ram_driver::get_type(),ram_driver_new::get_type(),"*");
```

Step 11: Run the simulation using the command in Step 5. Obtain the waveform image.

Step 12: For instance overriding, comment the line in Step 10 and add the following line in the build phase of ***ram_test.sv***

```
set_inst_override_by_type("env.agent.driver", ram_driver::get_type(), ram_driver_new::get_type());
```

Step 13: Run the simulation using the command in Step 5. It is not necessary to submit a waveform for this step as the same driver code is being called here as in Step 10.

Questions:

1. What change can you make in the verification environment to make the group coverage 100%? Re-run the simulation and take a screenshot of the functional coverage report as described in Step 7.
2. What difference do you observe between the waveforms of Steps 8 and 11? Explain how it affects the monitor and scoreboard.
3. Comment the line in step 12 in the file ram_test.sv and add the following line:

```
set_type_override_by_type(ram_monitor::get_type(),ram_monitor_new::get_type(),"*");
```

Now, run the simulation. You should be able to observe the error in the scoreboard comparison. How will you fix it?

What is to be turned in?

A report (pdf file) consisting of the coverage screenshot, waveform image and simulation result as described in Steps 7 and 8, the waveform image of Step 11, the coverage screen shot for Question 1, 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