EE 8350 Advanced Verification Methodologies for VLSI Systems
LAB 2

NOTE: Before starting this lab, you must go through the steps of LAB 0, which can be found in the UVM Information section of the course page.

OBJECTIVE: This lab provides an introduction to UVM and the UVM phases.

UVM: The Universal Verification Methodology (UVM) consists of class libraries needed for the development of a well-constructed, reusable SystemVerilog-based verification environment. In essence, UVM consists of set of useful base classes having pre-defined methods. A desired verification environment can be developed by extending these base classes.

UVM Architecture:

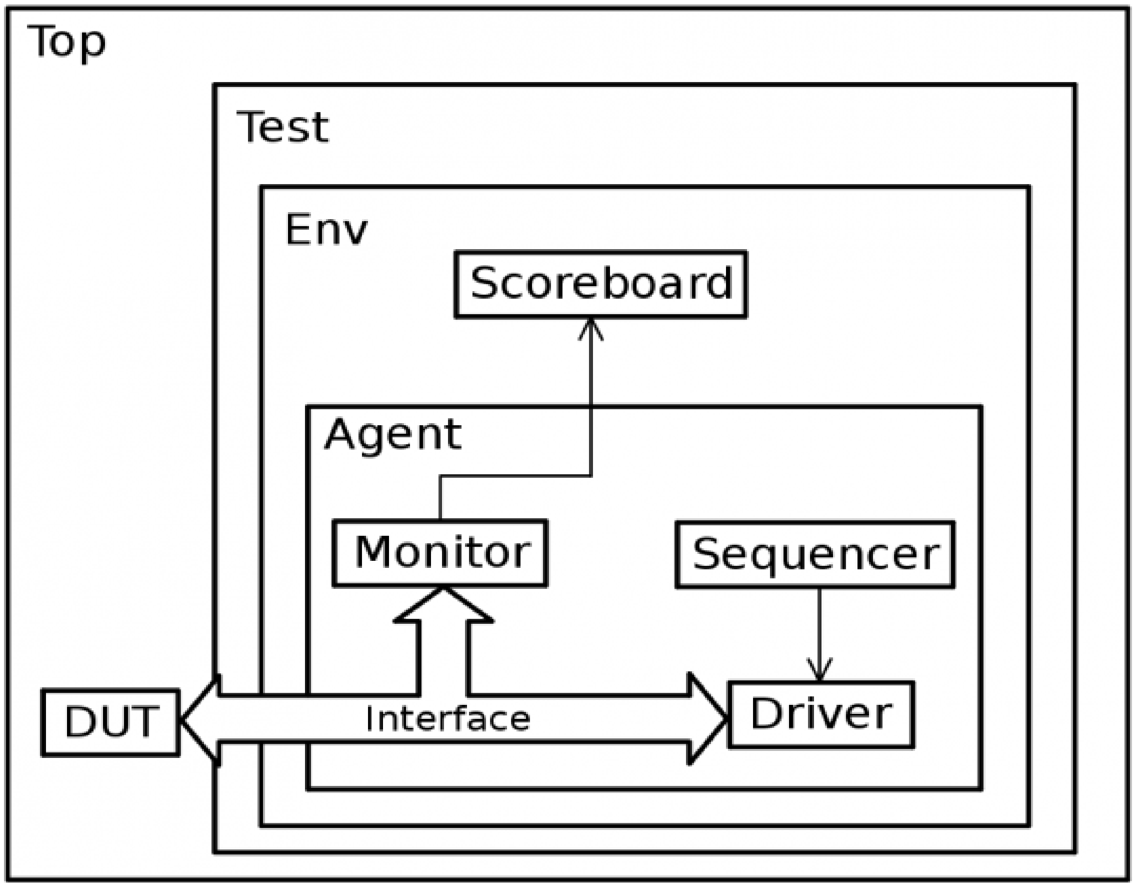

Fig 1: UVM testbench architecture (Ref: colorlesscube.com)

We have discussed the concepts of Interface, Driver and Monitor in Lab 1. We will dive deeper into other UVM components in the later labs. In this lab, we will learn how to run a UVM test case and understand the phases in which the simulation is carried out.

Running a UVM test case:

To start a UVM testbench, the run_test() method has to be called from the static part of the testbench. It is usually called from within an initial block in the top-level module of the testbench.

Calling run_test() constructs the UVM environment root component and then initiates the UVM phasing. The run_test() method can be passed a string argument to define the default type name of a uvm_component derived class which is used as the root node of the testbench hierarchy.

The run_test() method also checks for a command line plusarg called UVM_TESTNAME and uses that plusarg string to lookup a factory registered uvm_component, overriding any default type name. The root node defines the test case to be executed by specifying the configuration of the testbench components and the stimulus to be executed by them. By convention, the root node will be derived from a uvm_test component, but it can be derived from any uvm_component.

To run a test by passing testname through ***UVM_TESTNAME*** plusarg:

Step 1: Run the following commands on the command line

```
       vcs -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 -l
compile.log +vcs+dumpvars+verilog.vpd -debug_all

       ./simv +UVM_TESTNAME=uvm_test_1
```

Step 2: Take a screenshot of the result.

To run a test by passing string to ***run_test()*** method:

Step 3: Pass the string “uvm_test_1” as a parameter to the ***run_test*** function call in ***top.sv***

```
      run_test("uvm_test_1");
```

Step 4: Run the following commands on command line

```
      vcs -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 -l
compile.log +vcs+dumpvars+verilog.vpd -debug_all

      ./simv
```

Step 5: Take a screenshot of the result.

UVM Phases: All these classes have various simulation phases. Phases are ordered steps of execution that are implemented as methods. When we derive a new class, the simulation of our testbench will go through these different steps in order to construct, configure and connect the testbench component hierarchy.

UVM phases provide a mechanism to synchronize the major functional steps that a simulation runs through. UVM groups the phases into the following three types:

- Build Phases – Here, the testbench is constructed, connected and configured.
- Run-time Phases - Here, stimulus generation and time-consuming simulation occur.
- Clean up Phases - Here, the test results are collected and reported.

Build Phases: All the build phase methods are functions and therefore execute in zero simulation time.

- **build:**
  Once the UVM testbench root node component is constructed, the build phase starts to execute. It constructs the testbench component hierarchy.
- **connect:**
  The connect phase is used to make TLM connections between components or to assign handles to testbench resources. It has to occur after the build method has put the testbench component hierarchy together.
- **end_of_elaboration:**
  The end_of_elaboration phase is used to make any final adjustments to the structure, configuration or connectivity of the testbench before simulation starts. Its implementation can assume that the testbench component hierarchy and inter-connectivity is in place.

Run-time Phases: The testbench stimulus is generated and executed during the run-time phases which follow the build phases. After the start_of_simulation phase, the UVM executes the run phase. The run phase consists of the individual tasks pre_reset through post_shutdown. The run-time phases are as follows:

- start_of_simulation
- run:
  - pre_reset
  - reset
  - post_reset
  - pre_configure
  - configure
  - post_configure
  - pre_main
  - main
  - post_main
  - pre_shutdown
  - shutdown
  - post_shutdown

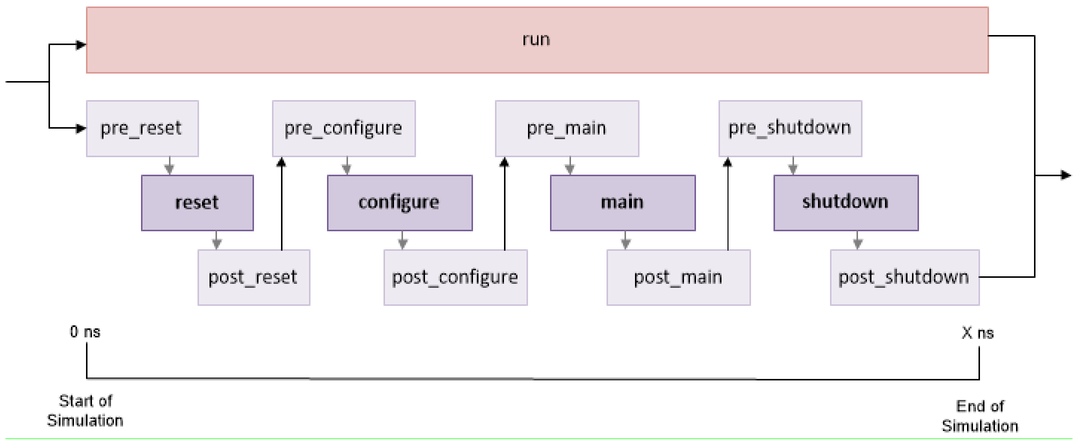

Fig 2: Run Phase Execution Order (Ref: chipverify.com)

Clean Up Phases:
The clean up phases are used to extract information from scoreboards and functional coverage monitors to determine whether the test case has passed and/or reached its coverage goals. The clean up phases are implemented as functions and therefore take zero time to execute.

- **extract:**
  The extract phase is used to retrieve and process information from scoreboards and functional coverage monitors.
- **check**
  The check phase is used to check that the DUT behaved correctly and to identify any errors that may have occurred during the execution of the testbench.
- **report**
  The report phase is used to display the results of the simulation or to write the results to a file.
- **final**
  The final phase is used to complete any other outstanding actions that the testbench has not already completed.

Let us try to understand the order of execution of the UVM phases.

Step 6: Uncomment the following lines from ***uvm_test_1.sv***

```
uvm_cmpt_1 cmpt_1;
cmpt_1 = uvm_cmpt_1::type_id::create("cmpt_1", this);
```

Step 7: Run the `vcs` and `simv` commands.
Observe how each of the phases are executed.

Step 8: Uncomment the following lines from ***uvm_cmpt_1.sv***

```
uvm_cmpt_2 cmpt_2;
cmpt_2 = uvm_cmpt_2::type_id::create("cmpt_2", this);
```

Step 9: Run the **`vcs`** and **`simv`** commands and take a screenshot of the results.
Observe how each of the phases are executed.

Step 10: Comment the lines in Step 8. Uncomment the following lines from ***uvm_test_1.sv***

```
uvm_cmpt_2 cmpt_2;
cmpt_2 = uvm_cmpt_2::type_id::create("cmpt_2", this);
```

Step 11: Run the **`vcs`** and **`simv`** commands and a take screenshot of the results.
Observe how each of the phases are executed.

Step 9 shows the phase execution for a hierarchy of components and Step 11 shows the phase execution for concurrent components.

Let us now see how run phases get implemented in a UVM simulation.

Step 12: Comment the lines in Step 10.

Step 13: Uncomment the following lines in ***uvm_cmpt_1.sv***

```
/*
      task pre_reset_phase(uvm_phase phase);
        phase.raise_objection(this);
            #1 uvm_report_info(get_full_name(), "Class uvm_cmpt_1 - pre_reset
phase",UVM_LOW);
        phase.drop_objection(this);
      endtask

      task reset_phase(uvm_phase phase);
        phase.raise_objection(this);
            #1  uvm_report_info(get_full_name(),  "Class  uvm_cmpt_1  -  reset
phase",UVM_LOW);
        phase.drop_objection(this);
      endtask

      task post_reset_phase(uvm_phase phase);
        phase.raise_objection(this);
            #1   uvm_report_info(get_full_name(),   "Class   uvm_cmpt_1   -
post_reset phase",UVM_LOW);
        phase.drop_objection(this);
      endtask
*/
```

Step 14: Run the **`vcs`** and **`simv`** commands and take a screenshot of the results.

You should be able to see display statements from pre_reset, reset and post_reset tasks. Observe the order in which the run phases are printed. Also, observe the time in which the statements are printed and relate them with the delay values given in the code.

Step 15: Uncomment the same lines as in Step 13 in ***uvm_cmpt_2.sv.*** Uncomment the lines as in Step 10 in ***uvm_test_1.sv***.

Step 16: Run the **`vcs`** and **`simv`** commands and take a screenshot of the results.

Observe the order in which run phases are printed. Observe how the run phases are being executed between two concurrent UVM components.

Questions:

1. Fill in the following table with the execution order (i.e., top-down or bottom-up) for each function listed. Execution order refers to the testbench hierarchy order in which the function is being called during simulation.

   Top-down – The component at the top of test bench hierarchy gets implemented before the component at the bottom of test bench hierarchy. In this lab, top-down order would be the function of uvm_cmpt_1 being called before the function of uvm_cmpt_2.

   Bottom–up – The component at the bottom of test bench hierarchy gets implemented before the component at the top of test bench hierarchy. In this lab, bottom-up order would be the function of uvm_cmpt_2 being called before the function of uvm_cmpt_1.

Phase	**Execution order (top-down or bottom-up)**
Build	
Connect	
end of elaboration	
Extract	
Check	
Report	
Final	

2. Give the order of execution of the run-time phases.

3. Fill in the following table with the time of execution of uvm_report_info in each task of cmpt1 and cmpt_2 based on the following code:

uvm_cmpt_1.sv:

```
      task pre_reset_phase(uvm_phase phase);
        phase.raise_objection(this);
            #1 uvm_report_info(get_full_name(), "Class uvm_cmpt_1 - pre_reset
phase",UVM_LOW);
        phase.drop_objection(this);
      endtask

      task reset_phase(uvm_phase phase);
        phase.raise_objection(this);
            #1  uvm_report_info(get_full_name(),  "Class  uvm_cmpt_1  -  reset
phase",UVM_LOW);
        phase.drop_objection(this);
      endtask

      task post_reset_phase(uvm_phase phase);
        phase.raise_objection(this);
            #1    uvm_report_info(get_full_name(),    "Class    uvm_cmpt_1    -
post_reset phase",UVM_LOW);
        phase.drop_objection(this);
      endtask
```

uvm_cmpt2_sv:

```
task pre_reset_phase(uvm_phase phase);
          phase.raise_objection(this);
             #2    uvm_report_info(get_full_name(),    "Class    uvm_cmpt_2    -
pre_reset phase",UVM_LOW);
          phase.drop_objection(this);
        endtask

        task reset_phase(uvm_phase phase);
          phase.raise_objection(this);
             #2  uvm_report_info(get_full_name(),  "Class  uvm_cmpt_2  -  reset
phase",UVM_LOW);
          phase.drop_objection(this);
        endtask

        task post_reset_phase(uvm_phase phase);
          phase.raise_objection(this);
```

```
            #2   uvm_report_info(get_full_name(),   "Class   uvm_cmpt_2   -
post_reset phase",UVM_LOW);
          phase.drop_objection(this);
        endtask
```

Tasks	Time of execution of uvm_report_info	
	cmpt_1	cmpt_2
run		
pre_reset		
reset		
post_reset		

What is to be turned in?

A report (pdf file) consisting of the answers to the above questions as well screenshots for the following results:

1. UVM test run using a command line argument.
2. UVM test run using a string passed to the run_test() method.
3. UVM phase execution order for 2 UVM components defined in a hierarchical manner.
4. UVM phase execution order for 2 UVM components defined in a concurrent manner.
5. UVM run phase execution order within a UVM component.
6. UVM run phase execution order for 2 UVM components defined in a concurrent manner.

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