EE 8350 Advanced Verification Methodologies for VLSI Systems
LAB 4

OBJECTIVE: To learn about the uvm_config_db and the associated set and get functions.

UVM_CONFIG_DB:
We use a hierarchical arrangement of components to easily share configurations and other parameters across different testbench components. To support this, UVM provides a database of objects and variables that can be created and accessed using strings. The database is a lookup table, which uses a string as a key, having the capabilities to add and retrieve entries. This is done using the **uvm _config_db**. With **uvm_config_db**, objects can share handles to their data members with other objects. These other testbench components can then have access to the object without knowing where it is in the hierarchy. It is similar to making some class variables global or public. In the database, the handles are identified by an assigned 'type' and 'name'.

We have already learned about interfaces and modports in Lab 1. An interface is a collection of common signals between two entities, and the signal directions are specified by the modports. A virtual interface is a handle pointing to an interface instance. A virtual interface acts as a mechanism to connect the DUT and the testbench together. The testbench accesses the DUT signals via the virtual interface, and vice versa.

In this lab, we will discuss how ***uvm_config_db*** is applied to facilitate the use of a virtual interface in a UVM testbench architecture. The ***uvm_config_db::set*** and ***uvm_config_db::get*** methods are used to store and retrieve information from the database, respectively.

set method:

void uvm_config_db#(type T)::set(uvm_component cntxt, string inst_name, string field_name, T value);

where:
• **T** is the type of the element being configured. The type can be scalar objects, class handles, queues, lists or virtual interfaces.

• **cntxt** is the hierarchical starting point of where the database entry is accessible. ***"this"*** would be used as the context if the call to set() is within a class. To set the virtual interface correctly in the database, the code has to be placed inside of a module, so there would be no class context. Normally, the virtual interface will be defined and set to the database in the top module. Hence the context would be ***"null"***.

• **inst_name** is a hierarchical path that limits accessibility of the database entry. Example:

top.env.agent.monitor	- only the monitor can get the component stored in the database.
top.*	- all of the scopes whose top-level component is top can get the component stored in the database.
top.env.*.monitor	- all of the scopes in env that end in monitor can get the component stored in the database.

• **field_name** is the label used as a lookup for the database entry.
• **value** is the value to be stored in the database.

get method:

bit uvm_config_db#(type T)::get(uvm_component cntxt, string inst_name, string field_name, ref T value);

• The scope provided by cntxt and inst_name is compared with the scope configured with the set command. If it matches, the object is allowed to be taken from the config_db.
• **value** is the variable to which the value is to be given from the database.

(The other fields are same as for the set method.)

The method **returns 1 if it is successful** and 0 if there is no such resource of this type in database.

Step 1: Open the file named ***adder_interface.sv*** and define the interface for a 4-bit adder.

```
logic [3:0] sum;
logic c_out;
logic [3:0] a;
logic [3:0] b;
```

Step 2:

- Open the file ***top.sv***
- In module top, import the UVM and adder package files.

```
import uvm_pkg::*;
import adder_testbench_pkg::*;
```

- Instantiate the adder interface.

```
adder_interface dut_if1();
```

- Instantiate the adder DUT and connect it to the interface.

```
adder dut1(dut_if1);
```

- In an initial block, invoke the UVM configuration database set method to pass the interface object.

```
            uvm_config_db#(virtual adder_interface)::set(null, "*",
"dut_vif", dut_if1);
```

Note the arguments that are used:
null - Since it is a top module and not a class.
*   - Pass down to all objects.
dut_vif - the label used as a lookup for the virtual interface.
Dut_if1 – The actual object name.

- Call ***run_test().***

```
      run_test();
```

- Add a $monitor statement to display the adder inputs, sum and carry.

```
      $monitor("a - %d b - %d sum - %d carry -
 %d",dut_if1.a,dut_if1.b,dut_if1.sum,dut_if1.c_out);
```

Step 3:

- Open the file ***adder_test.sv***
- Register the test using **uvm_component_utils**

```
      `uvm_component_utils(adder_test)
```

- Define an object of type ***adder_driver***

```
      adder_driver drv;
```

- Call the super.new function.

```
       super.new(name, parent);
```

- In the build function, instantiate the driver.

```
      drv = adder_driver::type_id::create("drv", this);
```

- In the run task, add the following line to raise an objection:

```
      phase.raise_objection(this);
```

- In the same task, add the following line to drop the objection:

```
      phase.drop_objection(this);
```

Step 4:

- Open the file ***adder_driver.sv***
- Register the driver using **uvm_component_utils**

```
`uvm_component_utils(adder_driver)
```

- Define an object for the virtual interface as follows:

```
virtual adder_interface dut_vif;
```

- Call the super.new function.

```
super.new(name, parent);
```

- In the build function, get the virtual interface using the **uvm_config_db::get()** function.

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

Note the arguments that are used:

this - The context of the driver object where we are obtaining the virtual interface.
“”   - Since the object is being used only in the driver.
dut_vif - The label used as a lookup for the virtual interface.
dut_vif – The actual object name.

Here, the scope used in the get method is: “uvm_test_top.driver”
this in the contxt field implies: uvm_test_top.driver

The scope can also be provided by setting the contxt field to uvm_test_top and the inst_name field to driver.

- Drive the pins of interface. Add the following code in the run_phase task:

```
for (int i = 0; i < 8; i++) begin
  dut_vif.a = $urandom_range(0,15);
  dut_vif.b = $urandom_range(0,15);
  #15;
end
```

Step 5:

- Open a new file ***adder_testbench_pkg.sv***
- Add the following code into that:

```
`include "adder_driver.sv"
`include "adder_test.sv"
```

Step 6: Run the following commands on the command line. Take a screenshot of the result.

```
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
adder_interface.sv  adder.v -l compile.log +vcs+dumpvars+verilog.vpd -
debug_all

./simv +UVM_TESTNAME=adder_test +ntb_random_seed=5258623
```

(Note: Enter your student ID for the random seed)

Step 7: Add the following highlighted code in the run_phase task of the driver (**adder_driver.sv**) code.

```
phase.raise_objection(this);

      // Wiggle pins of DUT
      for (int i = 0; i < 8; i++) begin
             dut_vif.a = $urandom_range(0,15);
             dut_vif.b = $urandom_range(0,15);
      #15;
      end

phase.drop_objection(this);
```

Run the following commands on the command line. Take a screenshot of the result.

```
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
adder_interface.sv  adder.v -l compile.log +vcs+dumpvars+verilog.vpd -
debug_all

./simv +UVM_TESTNAME=adder_test +ntb_random_seed=5258623
```

(Note: Enter your student ID for the random seed)

Observe the difference between the simulation results of Step 6 and Step 7.

Questions:

1. Modify the set command in Step 2 such that only the driver can get the interface.
2. What are static member functions of a class? How are they accessed?
3. Note that type parameterization is used in uvm_config_db. What is the default parameter value for uvm_config_db?
4. What is the function of the runtime option **+UVM_CONFIG_DB_TRACE*?*** (Check the simulation results by providing this runtime option. No need to turn in the results.)
5. What is the use of raise_objection? Explain the difference in the simulation results of Step 6 and Step 7.

What is to be turned in?

A report (pdf file) consisting of the screenshots in Steps 6 and 7, 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