EE 8350 Advanced Verification Methodologies for VLSI Systems
LAB 1

OBJECTIVE: To learn about the Layered Testbench Architecture and use SystemVerilog to construct a few of the components of the testbench.

In the past, verification engineers used a directed testbench to verify the functionality of their designs. For most designs, simple random stimulus is not sufficient to verify them fully. Purely random test vectors rarely create useful stimulus patterns that target important design functionalities. The Constrained Random Verification (CRV) approach provides the ability to create useful stimulus, but it can only do so when the stimulus-generation infrastructure is built with knowledge about the underlying design architecture.

SystemVerilog supports object-oriented data abstraction. Class objects contain properties and methods that operate on those properties. SystemVerilog randomization is also built in an object framework. Constraints are provided in SystemVerilog to constrain the randomness.

Layered Testbench Architecture:

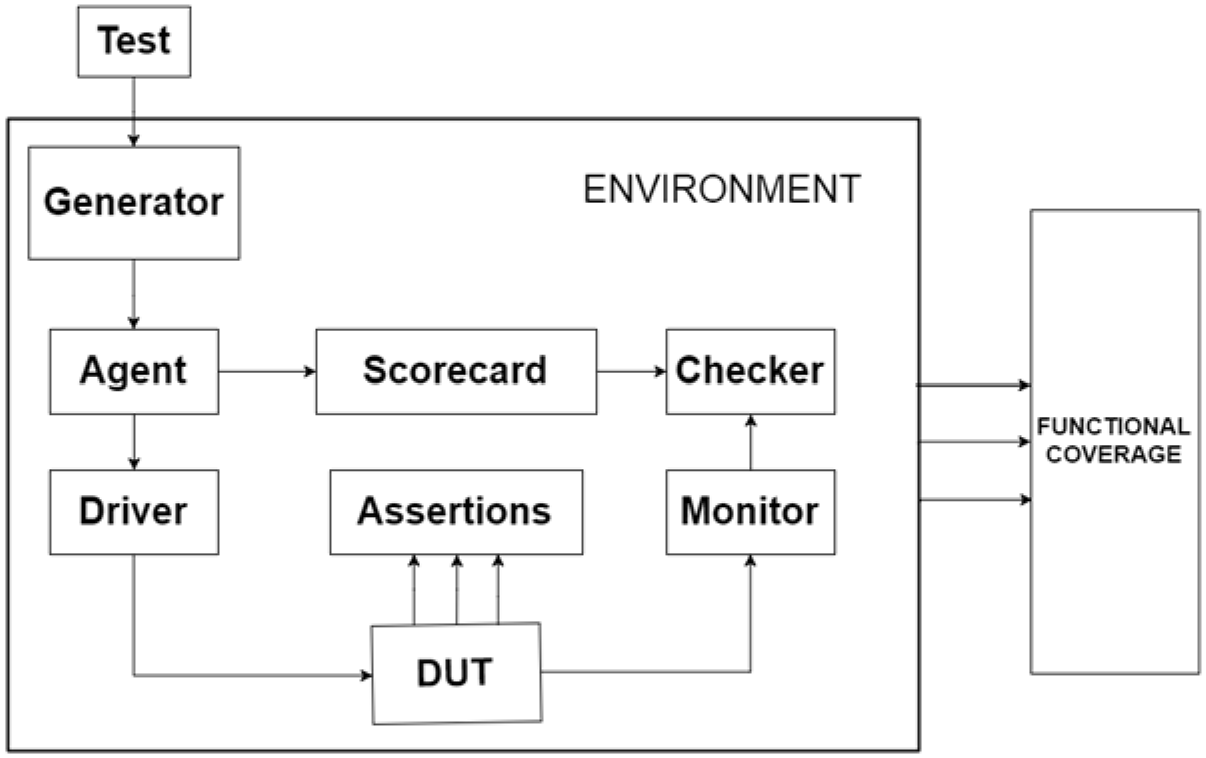


Figure 1 : Multi-layered SystemVerilog testbench.

The layered testbench is the heart of the verification environment:

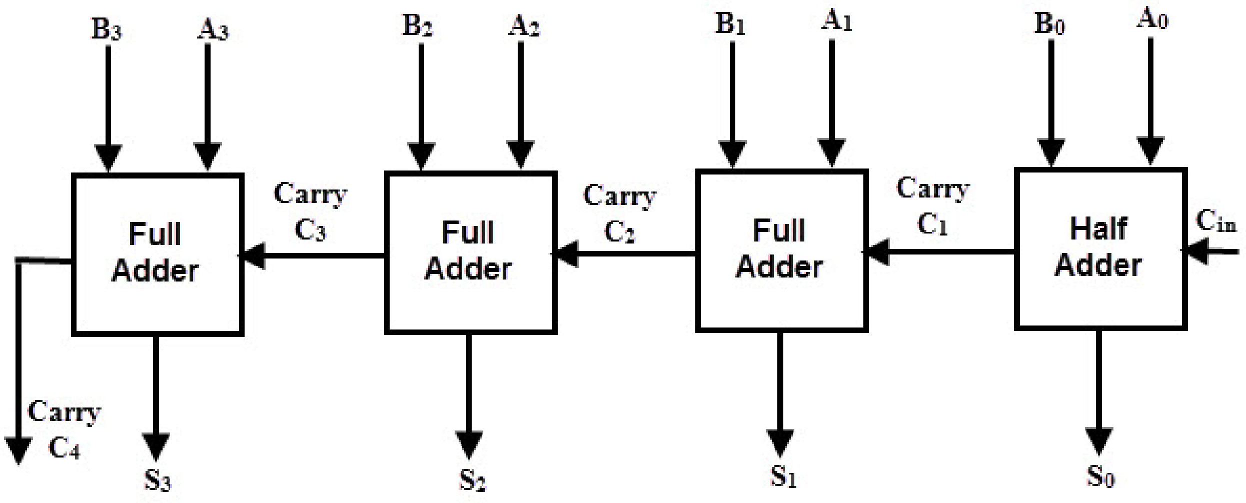
- The lowest layer is the signal layer that connects the testbench to the RTL design, which is also called the Design Under Test (DUT). It consists of interface, clocking, and modport constructs.
- The next layer contains driver and monitor components, as well as the assertions (properties) that check design intent. This layer provides a transaction-level interface to the layer above and drives the physical pins via the signal layer.
- The agent uses generators to produce streams or sequences of transactions that are applied to the driver. The generators have a set of weights, constraints or scenarios specified by the test layer. The randomness of constrained-random verification is introduced within this layer.
- The tests can define new sequences of transactions, synchronize multiple transaction streams, generate sequences, or supply directed stimulus directly to the interface layer.

The layered approach facilitates reuse in several ways. Lower layers can be removed and replaced with transaction-level models for architectural or system performance analysis. Layers can be reused between different projects, since the interaction between each layer is clearly defined. Moreover, the entire testbench is reusable across tests since only the test layer needs to be modified in order to generate new tests.

In this lab, we will examine how SystemVerilog constructs are used to build a layered testbench for an adder RTL module.

Adder RTL:

We shall be using a 4-bit ripple-carry adder for this lab.



We will construct the following verification components in this lab:

Interface: An interface is a construct in SystemVerilog created specifically to encapsulate the communication between blocks. It replaces a long list of port names by a single name. This reduces the amount of code needed to model port connections and it improves the code's maintainability and readability. Interfaces are very useful in designs where there are multiple instances of the same module. In such cases, specifying the port connections for every instantiation of the module would adversely affect the readability of the code.

Modport: A modport groups and specifies the port directions to the wires/signals declared within the interface. By specifying the port directions, a modport provides access restrictions. For example, the clock signal is generated in the testbench and driven to the DUT. This makes it an output signal from the testbench and an input signal to the DUT. Modports are used to specify this difference. In short, the interface bundles the signals and the modports specify the directions of the signals.

For our adder example, we need two modports - one for the DUT and the other for the testbench.

Step 1: Define the following modports in ***adder_interface.sv***:

```
modport dut (output c_out, output sum, input a, input b);
modport tb (input c_out, input sum, output a, output b);
```

Sequence_item: A sequence item is a class object which includes the information needed to model the communication between the DUT and the testbench. It can include variables, constraints and methods for operating on the variables. In general, a sequence item encapsulates the signals that form the external stimulus to the DUT.

For our adder example, a sequence item is defined in the file: adder_sequence_item.sv

The stimulus for the adder RTL requires two randomly generated 4-bit input values. SystemVerilog constraints can be used to restrict the inputs within a certain range of values.

Step 2: Add the following constraints in the class sequence_item. It is provided in **adder_sequence_item.sv.**

```
constraint input_constraint
{
  a inside {[4'd0:4'd15]};
  b inside {[4'd0:4'd15]};
}
```

Driver: The driver is a block whose role is to interact with the DUT. A sequence item creates inputs at a high level of abstraction. The driver converts this input into actual design inputs, as defined in the specification of the design's interface.

For our adder example, the driver drives inputs a and b onto the adder interface. A pointer to the adder interface is passed on while instantiating the driver.

The driver code is provided in ***adder_driver.sv***.

Step 3: Add the following code in the ***drive_t*** task. It drives the interface signals to send the values ***a*** and ***b*** to the adder.

```
        tb_ports.a = item.a;
        tb_ports.b = item.b;
```

Monitor: The monitor is a self-contained model that observes the communication of the DUT with the testbench. It should observe the outputs of the design and, in case of not respecting the protocol's rules, the monitor will return an error.

The monitor code is provided in ***adder_monitor.sv***

Step 4: Add the following code in the monitor_t task. It monitors the adder interface and flags an error if the ***sum*** or ***c_out*** value is not correct.

```
        if((tb_ports.sum != sum[3:0]) || (tb_ports.c_out != sum[4]))
        begin
          $display("actual sum - %d actual carry - %d expected sum - %d
expected carry - %d\n",tb_ports.sum,tb_ports.c_out,sum[3:0],sum[4]);
        end
```

Environment: The environment is a very simple class that instantiates the agent and the scoreboard and connects them together.

For our adder example, the environment is used to instantiate the driver and the monitor by passing the adder interface object to them.

The environment code is given in ***adder_env.sv***.

Step 5: Add the following code in adder_env.sv:

- Instantiate the driver, sequence items and monitor:

```
driver drv = new(tb_mp);
monitor mon = new(tb_mp);
sequence_item s0 = new();
sequence_item s1 = new();
```

- Call the monitor_t task:

```
mon.monitor_t();
```

- Randomize both sequence items and send them to the driver by calling drive_t();

```
if(s0.randomize() == 1)
 begin
   drv.drive_t(s0);
 end
 #15;

 if(s1.randomize() == 1)
 begin
   drv.drive_t(s1);
 end
```

Step 6: Run the following commands on the command line:

1. `vcs -sverilog top.sv adder_interface.sv adder.v -debug_access -debug_pp –lca`

2. `./simv +ntb_random_seed=5258623`
   (Note: Enter your student ID for the seed value.)

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

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

Step 8: The dve tool opens on running the previous command.
Select ***"adder_tb_top"*** from the ***hierarchy*** column and click the + button beside it.
Select the "***a_int"*** subtab.
Select all signals that appear under the ***variable*** column.
Press ***Ctrl + 4*** to view the signals in the form of a waveform.
Go to ***File->print***
Select ***Image***.
Select ***ok***.

The image will be stored in the same directory as your simulation run.
Attach the image in your report.

Questions:

1. Write a constraint such that the random variable ‘a’ in the adder testbench gets randomized to a value in the range [0-7] with twice the probability compared to the values [8-15].
2. Write an inline constraint such that the random variable ‘b’ in adder testbench gets randomized in the range of values [0-7].
3. What is the difference between rand and randc?
4. What is a virtual interface? How is it useful?
5. What is an abstract class? How is it useful?

What is to be turned in?

A report (pdf file) consisting of the waveform image as described in Step 8 and the answers to the above questions. Submit your work using the link provided on the class Canvas page.

References used in the preparation of this lab write-up:

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