Integrated Systems Architectures

Verification and UVM Description

Many thanks to Eng. Michele Caon for providing us with the guidelines for SystemVerilog verification.

The aim of this lab is to verify a floating point unit and to start practicing with the Universal Verification Methodology (UVM).

1 Verification

SystemVerilog enables powerful verification of complex design by also easing randomization. Follows a simple SystemVerilog testbench to verify a sequential ALU. For additional information about the SystemVerilog language and its features you can refer to `https://www.chipverify.com/systemverilog/systemverilog-tutorial`.

The example structure is:

- **src** contains the ALU source files, namely:
  - **alu_pkg.sv** This file contains the definition of the data type listing the operation supported by the ALU.
  - **alu.sv** A module containing the behavioural description of the ALU.
- **tb** contains the ALU testbench files
  - **alu_tb.sv** This is the top-level module of the SystemVerilog testbench. The ALU, the interface, and a tester object are instantiated here, and the `run_test()` task from the tester object is launched.
  - **alu_tester.svh** This class contains all the properties and methods that are necessary to test the ALU.
  - **alu_verbose_tester.svh** This class is extended from *alu_tester.svh* and contains the methods required to print additional information about te test being performed.
  - **alu_op_verbose_tester.svh** This class extends *alu_verbose_tester.svh* with a constraint to only generate one type of ALU operations (e.g., ADD, BITOR, etc.).

  - **alu_if.sv** An interface to connect the testbench to the DUT (the ALU). The interface also implements the self-checking code by means of the SystemVerilog Assertions defined in *alu_if_sva.svh*.
  - **alu_if_sva.svh** Assertions to verify that the output of the ALU is consistent with the expected one at each clock cycle. The assertion at [line 47] verifies that the ALU output is correctly reset, while the one at [line 87] (with the correspondent 'property') verifies the correctness of the output according to the current ALU operation.
  - **alu_wrap.sv** A simple wrapper to use the interface defined in *alu_if.sv* instead of direct port mapping. This component also allows to connect the interface to a VHDL DUT (compiled separately) if necessary.
  - **alu_cov.svh** A class to track the functional coverage of the issued ALU operations and operands. Since a static instance of this class is instantiated in *alu_tester.svh*, the total coverage is increased with every test instance which `run_test()` task is executed.

- **sim** Simulation macros
  - **compile.f** The list of the files to be compiled before the simulation is launched. The compilation order guarantees that all the dependencies are resolved.
  - **sim.do** Tcl macro containing the `vsim` command to launch the simulation (`run -all`).
  - **report-cov.do** Command to write the functional coverage report to a file.

1.1 Simulation

1.1.1 with GUI

1. Launch QuestaSim (`vsim`)
2. Create a new library (`vlib work`)
3. Compile the ALU source files, namely select all the files in *src*.

4. Compile the testbench source, namely select all the files in *tb* (except the .svh ones).
5. Start the simulation.
6. If you wish, you can add the ALU signals to the waveform viewer.
7. Run the simulation (`run -all`). The simulation will automatically stop when both tests are executed. Look at the output log to see the messages printed by the testbench during the simulation.

1.2 Without GUI

1. Move into the simulation directory (`cd sim`).
2. Delete any existing library (if necessary) and create a new library named 'work' (`vdel -all`, `vlib work`)
3. Compile all the modules and packages: `vlog -F compile.f`
4. Run the simulation: `vsim -c -sv_seed random -do sim.do alu_tb`. The simulator will print the testbench messages directly on the host terminal. Use the `-sv_seed random` option of `vsim`' to randomly initialize the SystemVerilog pseudorandom generator and obtain different test vectors (i.e., the 'rand' signals) in each run. To increase the number of test cycles (that is 10 by default), you can use the `+n<NUM>` runtime option. For example, to issue 100 ALU operations, the `vsim` command-line will be: `vsim -c -sv_seed random -do sim.do alu_tb +n100`. To write the coverage report to a file (*sim/func_cover.txt* by default), use the `report-cov.do` macro: `vsim -c -sv_seed random -do sim.do alu_tb -do report-cov.do`

2 UVM

2.1 Introduction and background

Industries have always struggled to effectively validate their products in order to decrease the time-to-market and their products

reliability. They jointly developed many reusable verification frameworks, the UVM being the latest evolution. It is nowadays a de facto standard for building modular testbenches, resorting to SystemVerilog (SV) unique capabilities. Indeed, SV is not only an extension of Verilog HDL (as it adds some features to ease hardware description) but also a powerful verification language as it includes many non-synthesizable constructs specifically thought for verification purposes. UVM is a free framework written in SV by verification experts through the years.

2.2 UVM phases

Each element of an UVM-based testbench is a component derived from an existing UVM class. Each class follows simulation phases, which are ordered execution steps implemented as methods. The main UVM phases are:

- the *build_phase*, that is responsible for the creation and configuration of the testbench structure, constructing the components of the hierarchy
- the *connect_phase*, that is used to connect different sub-components in a class
- the *run_phase*, which is the main phase, where the simulation is executed
- the *report_phase*, that can be used to display results from the simulation

The complete 12-phases sequence is shown in Figure 1.

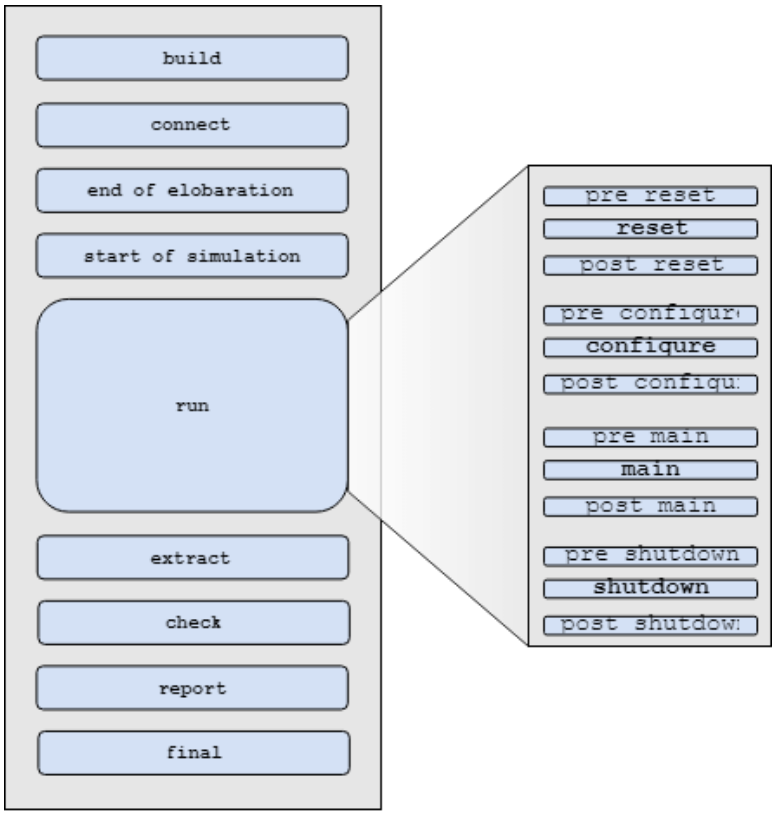

Figura 1: UVM phases

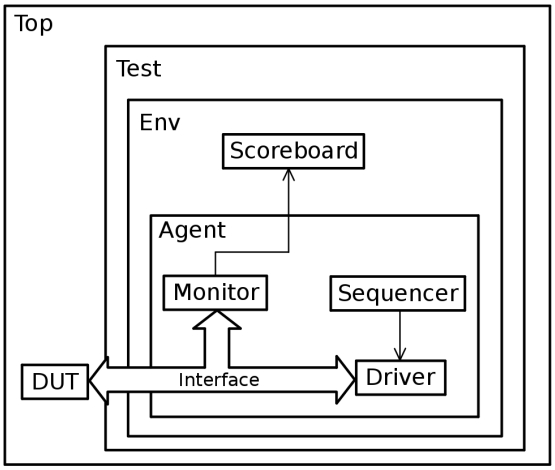

Figura 2: UVM-based tb typical structure

2.3 UVM testbench elements

An UVM-based testbench is structured as in Figure 2. It is comprised of one or more of each of these elements:

- the *DUT*, (Device Under Test), which is the HDL design you are testing
- the *Sequencer*, which generates the signals to be sent to the DUT
- the *Driver*, that sends the signals generated by the sequencer to the DUT through a SV interface
- the *Monitor*, that passively observes the interface transactions and collects them (both input and output transactions), very useful to check also that signals are compliant to specifications
- the *Scoreboard*, that compares the results with the ones produced reference model, signaling eventual discrepancies

Usually the combination of Monitor, Sequencer and Driver is called an *Agent*, the combination of the Agent(s) and the Scoreboard(s) is called *Environment* (Env) and the *Test* encompasses the Env(s). Eventually, a *Top* module encapsulates the Test and the DUT.

3 Testing an adder

If you open the `top.sv` file in the `tb` folder you will see that all the required files, also interfaces and DUT, are directly included. This makes possible to just compile the top entity to start the simulation. The given source code should run just by using `vlog -sv ../tb/top.sv` followed by `vsim top` and then `run 4 us`. You should see in the simulator log that the UVM framework is giving you information about the simulation, in particular that your DUT and the refmod results are matched.

This design presents two agents, one that monitors the input interface and the other that handles the output interface transactions. The input agent sends data to the refmod while the output one observes the results and send them to the comparator in the scoreboard.

In order to get used to the UVM framework, try to modify the testbench (in particular *packet_in.sv* and *packet_out.sv*), the interface and the DUT to accomodate the new word length. The given refmod will follow packet_in and packet_out automatically.

After testing the modified testbench you can change the refmod behavior so that its results wlll not match the ones given by the DUT. An easy way is to change the `+` sign in the addition to a minus sign to make it a subtractor. By recompiling and re-running the top module you will see that UVM will give you warnings, counting and displaying the total number of UVM_INFOs, UVM_WARNINGs, UVM_ERRORs and so on in the UVM report summary at the end of the simulation.

4 Testing the MBE-Dadda tree multiplier

Modify the testbench to test the MBE-Dadda tree multiplier from lab 2. You will also need to modify the interface to accomodate the multiplier data width and the refmod so that it will perform the multiplication instead of the addition.
In this way you can verify that your MBE-Dadda tree multiplier works as expected using the UVM framework.

5 Testing the whole floating point multiplier

Once you verified that your integer multiplier is working you can try testing the full floating point multiplier from lab 2. It is more difficult to manage it than the MBE-Dadda as this is not a fully combinational block due to the pipeline stages. Moreover, integrating a 16 bit floating point reference model is not obvious. We suggest to exploit the program you already used in lab2 to generate random (or nearly random) 16 bit arrays for the inputs, by writing them on a file and reading them from your testbench.

Alternatively, you can try to import the *flexfloat*[1] library in SystemVerilog. **This solution has not been tested yet.** This operation can be done by using the SystemVerilog Direct Programming Interface (DPI)[2]. However, the DPI has some limitations on C function return values and argument data types, including agument passing. Thus, the library must be first modified by introducing some new C functions complying with DPI. Then, the library can be compiled. Preliminary experiments have shown that you can compile it on CentOs without using cmake, but simply using `gcc`. The command line is[3]:

```
gcc -shared -Bsymbolic -fPIC -o flexfloat.so -lutil flexfloat.c
```

Then, you can *import* the functions in a SystemVerilog module by using the `import "DPI-C" function` statement. Finally, you have to call QuestaSim (`vsim`) with the option:

[1] `https://github.com/oprecomp/flexfloat`

[2] `https://www.doulos.com/knowhow/systemverilog/systemverilog-tutorials/systemverilog-dpi-tutorial`

[3] `https://github.com/esl-epfl/x-heep/blob/main/scripts/sim/compile_uart_dpi.sh`

```
-sv_lib <path-to-the-library>/flexfloat
```

You can find an example of DPI use for a UART peripheralin the X-HEEP project:

https://github.com/esl-epfl/x-heep/tree/main/hw/vendor/lowrisc_opentitan/hw/dv/dpi/uartdpi