Politecnico di Torino

III Facoltà di Ingegneria

Exercises and Homeworks for the course Integrated Systems Architecture

Master degree in Electrical Engineering

Authors: Group 20

Alessandro Lovesio, Lorenzo Poloni, Simone Pont

April 10, 2021

Many thanks to Prof. Mariagrazia Graziano for providing us with this template.

Contents

Link to our group repository:
`https://github.com/AleLovesio/Laboratory_ISA_Group20.20`
Direct link to the folder of Lab 4 in the same repository:
`https://github.com/AleLovesio/Laboratory_ISA_Group20.20/tree/main/Lab4`

0.1 Introduction

In this last laboratory we are tasked to perform a more formal verification process. In the past laboratories we verified manually the correctness of the results. This was still an acceptable method considering the relatively limited size of the employed test combinations. However, for a more reliable validation, an automated and formal process is needed. In particular, the obtained results are usually compared with a reference model (or golden model) that emulates the circuit behavior at a higher level of abstraction, which allows to try different options at a faster pace. The used verification framework is the UVM that is nowadays a de facto standard for building modular testbenches. It is a free framework written in SystemVerilog by verification experts throughout the years.

0.2 UVM

The Universal Verification Methodology (UVM) is a standardized methodology to build a modular and reusable verification environment in a layered, object-oriented approach. In this way, it is possible to perform multiple tests on the same architecture but also apply the same set of stimuli to different architectures in order to compare them. The UVM class library brings much automation to the SystemVerilog language such as sequences and data automation features (packing, copy, compare).
The general architecture is split into two different parts:

- A verification environment (TB), based on SystemVerilog (SV) classes that rely on Transaction Level Modeling (TLM) protocol.
- The Hardware Description Language (HDL) containing the DUT source files and its interface.

Drivers and Monitors are used for the communication between these two parts and translate pin signals into transactions and vice-versa. In our case the same TB was used for all the considered DUTs with only some small changes.
The verification of correct behavior of the DUT is obtained by comparing its results with the ones obtained with a high level reference model. The DUT module is based on a Finite State Machine (FSM) to manage the valid-ready signals and to obtain the synchronization between the circuit and the reference model.
The test-bench is made up of different modules, the most important are:

- The Refmod: used to define the high level reference model used to obtain the correct results.
- The Comparator: used to compare the results coming from the DUT with the correct ones from the reference model.
- The Monitor: used to display the results of the DUT, the Refmod and the Comparator.

During the laboratory all the different modules are managed and directly included using the top.sv file, making possible to just compile the top entity to start the simulation.

0.3 Testing an adder

The first circuit to be considered was a simple adder that was tested in different configurations. In particular, both the parallelism and the constraints on random input generation were modified. The first run was performed sticking to the default settings of the given files. In this way we could understand how the UVM works and the parameters that we could modify to get a different configuration. The obtained results are shown in the following picture:

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  106
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Comparator Match]   101
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [env]     1
```

After that we modified the *packet_in.sv* and *packet_out.sv* to consider a different data parallelism and verify again the correct behavior of the adder. We used a new input parallelism of 64-bit and we obtained again a complete match, as reported in the following picture:

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  106
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Comparator Match]   101
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [env]     1
```

Then we moved on to the insertion of some constraints on the generation of the random inputs. In particular, we decided to check two different constraints:

- Operand A in a range between 1 and 1000 and B always higher than 1000000.
- Operand A in a range between 100 and 1000 and B always lower than 10 times A.

To obtain these constraints the *inside* operator was used together with other equations, the code for them is reported in the following picture:

```
//two possible constraints, use only one at time (comment the unused)
constraint range {  A inside {[1:1000]};
                    B > 1000000; }

constraint max { A inside { [ 100:1000 ] };
                 B < 10*A; }
```

Also in these two configurations, we obtained the complete match of all the results:

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  106
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Comparator Match]   101
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [env]     1
```

(a) Constraint "range"

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  106
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Comparator Match]   101
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [env]     1
```

(b) Constraint "max"

Finally, to also consider a situation were the DUT results do not match the ones provided by the reference model, we modified the *refmod.sv* file. In that file, the executed operation was changed into a subtraction so that the results are always wrong. After having re-run the verification process, the UVM returned warnings related to all the errors that were present. In the following picture the obtained results are reported:

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  207
# UVM_WARNING :  101
# UVM_ERROR :    1
# UVM_FATAL :    0
# ** Report counts by id
# [Comparator Mismatch]   101
# [MISCMP]   202
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [env]     2
```

We can see how in this case all the 101 operations are wrong and we have a complete mismatch of the results.

0.4 Testing the MBE Dadda multiplier

After the initial tests on the simple adder we moved on to the analysis of a more complex circuit: the MBE Dadda-tree multiplier, implemented during the second laboratory. For this purpose we slightly modified some of the different files used during the simulation. In particular:

- In the *dut_if.sv*, *packet_in.sv* and *packet_out.sv* files we modified the data parallelism to correctly interface the new DUT.
- In the *refmod.sv* file we changed the reference model implementing a multiplication.
- In the *DUT.sv* file we modified the instantiated DUT and the related signals.

In addition, considering that the multiplier works with unsigned numbers, the constraints were also changed to generate only positive values. In this way, we could ensure the validity of all inputs. As we already did with the adder, we ran the simulation and the verification with 101 random inputs and all the obtained results matched the reference model:

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  106
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Comparator Match]   101
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [env]     1
```

We can notice that, in this case, the number of errors and warnings related to the UVM is zero. This means that all the DUT's results were correctly checked and they correspond to the ones obtained with the reference model.

0.5 Testing the complete floating point multiplier

Having checked the correct behavior of the MBE Dadda-tree multiplier, we concentrated on the verification of the complete floating point multiplier from the second laboratory. In this case, we are no longer considering a fully combinational circuit and so more changes in the UVM files are needed. Additional modifications are required as now the input and output data are in the 32-bit floating point format.

0.5.1 Changes related to the pipeline stages

Most of the changes related to the presence of pipeline stages inside the DUT were applied in the *DUT.sv* file. To correctly manage the latency between the arrival of a new input to the DUT and the generation of the related output we decided to modify the state machine present inside the file. In particular, we added two new states used to achieve synchronization.
The original states before the modifications were the following three:

- INITIAL: state used after the reset to initialize the variables before entering in WAIT.
- WAIT: state used to wait until new inputs are present and then display the result of the DUT. The presence of a new output is also reported.
- SEND: state used to enable the generation of new inputs after having correctly received the output.

The INITIAL state is used only after the reset, while in runtime the machine continuously alternates the WAIT and SEND states.
We decided to split the WAIT state into two different states WAIT_0 and WAIT_1 and to add a completely new state called ALIGN to obtain the synchronization between the DUT and the reference model. The final result is the following:

- INITIAL: state used after the reset to initialize the variables before entering in WAIT_0.
- WAIT_0: state used to wait until new inputs are present at the DUT and save them in local variables.
- ALIGN: state used to maintain fixed the inputs until a new output is ready. The time to wait can be controlled using a variable that represents the number of internal pipeline stages of the DUT.

- WAIT_1: state used to report the presence of a new output and to display the result of the DUT.
- SEND: state used to enable the generation of new inputs after having correctly received the output.

The INITIAL state is again used only once after the reset but now during runtime, the three other states loop following the sequence: WAIT_0 - ALIGN - WAIT_1 - SEND - WAIT_0.
In this way the comparison between the DUT and the reference model results is done with the correct timing even if the simulation time increases. However, the penalty on simulation time is not problematic.

0.5.2 Changes related to the 32-bit floating point format

To correctly manage the presence of floating point data we needed to act on different files. First of all, we found a way to correctly generate the 32-bit random inputs. This is because, due to QS restrictions, we cannot directly generate random floating point numbers. Thus, we decided to generate two 32-bit vectors with random values and then convert them to floating point numbers using the *$bitstoshortreal()* directive. In this way, we could also apply constraints more easily on the generated values.
The *$bitstoshortreal()* directive, together with its complementary one *$shortrealtobits()*, were also used to correctly implement the reference model and to print the data in the floating point format. In particular, the *refmod.sv* file was modified to implement the high level floating point multiplication in this way:

```
//calculation of the correct result
tr_out.data = $shortrealtobits($bitstoshortreal(tr_in.A) * $bitstoshortreal(tr_in.B));
```

The contents of *$display()* directive on both the *DUT.sv* and the *refmod.sv* files were also modified to print the data using the floating point exponential format. In the following picture, an example related to the printing of the reference model results is reported:

```
//printing the data on screen
$display("refmod: input A = %e, input B = %e, output OUT = %e",$bitstoshortreal(tr_in.A), $bitstoshortreal(tr_in.B), $bitstoshortreal(tr_out.data));
$display("refmod: input A = %b, input B = %b, output OUT = %b",tr_in.A, tr_in.B, tr_out.data);
```

0.5.3 Final results

After the two described modifications we tested our circuit and we obtained the following result:

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  108
# UVM_WARNING :    2
# UVM_ERROR :    1
# UVM_FATAL :    0
# ** Report counts by id
# [Comparator Match]    99
# [Comparator Mismatch]     2
# [MISCMP]     4
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [env]     2
```

We can notice the presence of two mismatches and with a deeper analysis we found out that they are related to the denormal numbers. We saw that the DUT did not deal with denormals while the reference model did. For this reason, when the result of the multiplication is a denormal number the comparison fails and a mismatch arises:

```
Mismatch #1
# refmod: input A = 1.040164e-12, input B = 1.156272e-32, output OUT = 1.261169e-44
# refmod: input A = 00101011100100100110001111011001, input B = 00001010011100000010010111111101, output OUT = 00000000000000000000000000001001
# FP mul: input A = 1.040164e-12, input B = 1.156272e-32, output OUT = 0.000000e+00
# FP mul: input A = 00101011100100100110001111011001, input B = 00001010011100000010010111111101, output OUT = 00000000000000000000000000000000

Mismatch #2
# refmod: input A = -7.537926e-31, input B = -1.476553e-10, output OUT = 1.113009e-40
# refmod: input A = 10001101011101001001111010100100, input B = 10101111001000100101100101001001, output OUT = 00000000000000010011011001000011
# FP mul: input A = -7.537926e-31, input B = -1.476553e-10, output OUT = 0.000000e+00
# FP mul: input A = 10001101011101001001111010100100, input B = 10101111001000100101100101001001, output OUT = 00000000000000000000000000000000
```

To fix this issue we decided to slightly change the behavior of the reference model such that it considers denormal numbers equal to zero. To do this, we simply modified the *refmod.sv* file adding some controls on the value of the result:

```
//calculation of the correct result without considering denormal numbers.
tmp = $bitstoshortreal(tr_in.A) * $bitstoshortreal(tr_in.B);
tmp_min_pos = $bitstoshortreal(32'b00000000100000000000000000000000);
tmp_min_neg = $bitstoshortreal(32'b10000000100000000000000000000000);
if (tmp > tmp_min_neg && tmp < tmp_min_pos) begin
    if (tmp >= +0) begin
        tr_out.data = 32'b00000000000000000000000000000000;
    end
    else begin
        tr_out.data = 32'b10000000000000000000000000000000;
    end
end
else begin
    tr_out.data = $shortrealtobits(tmp);
end
```

The *tmp_min _pos* and *tmp_min _neg* variables contain the minimum positive and negative numbers that can be represented without considering denormals. These two values are used as comparison to detect when a result has to be forced to zero or not. This is done by the *if-else* statement. After these modifications we simulated again the complete floating point multiplier and in that case we obtained a complete match for all results:

```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :  106
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Comparator Match]   101
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [env]     1
```

0.6 Conclusion

Implementing a formal and precise verification of our circuits is an important part of the design. To obtain reliable results in a limited amount of time, the automation of this process is necessary and it can be easily achieved using the UVM framework. During this last laboratory we learned about how this verification methodology works and about its flexibility. We noticed also how simple it is to check for the behavior of a device in a number of random situations. With only some small modifications we can easily verify different circuits, compare them with a high level reference model that can be fully customizable.
The power of UVM and the capability to understand and use it will be for sure a great help during the testing phases of our future designs.