Politecnico di Torino

Integrated Systems Architecture

A.A. 2020/2021

Prof. Guido Masera
Prof. Maurizio Martina

Multiplier verification with UVM

15 Apr 2021

Antona Gaspare	275336
Barrera Alessandro	275337
Liu Huicai	273375

Link to GitHub: `https://github.com/Dragosk97/ISA-laboratories.git`

Contents

1 Introduction

The Universal Verification Methodology (UVM) has become the standard for verification of integrated circuits design. In this laboratory it is exploited in order to verify the behaviour of the MBE-Dadda tree multiplier implemented before. Then the same methodology is used to verify the whole floating point multiplier.

Firstly, a simple adder is tested in order to gain familiarity with the UVM. And then the test are repeated for both the MBE-Dadda tree and the floating point multipliers.

The UVM testbench in figure 1 shows how the Design Under Test, **DUT**, interacts with the testbench to verify its functionality.

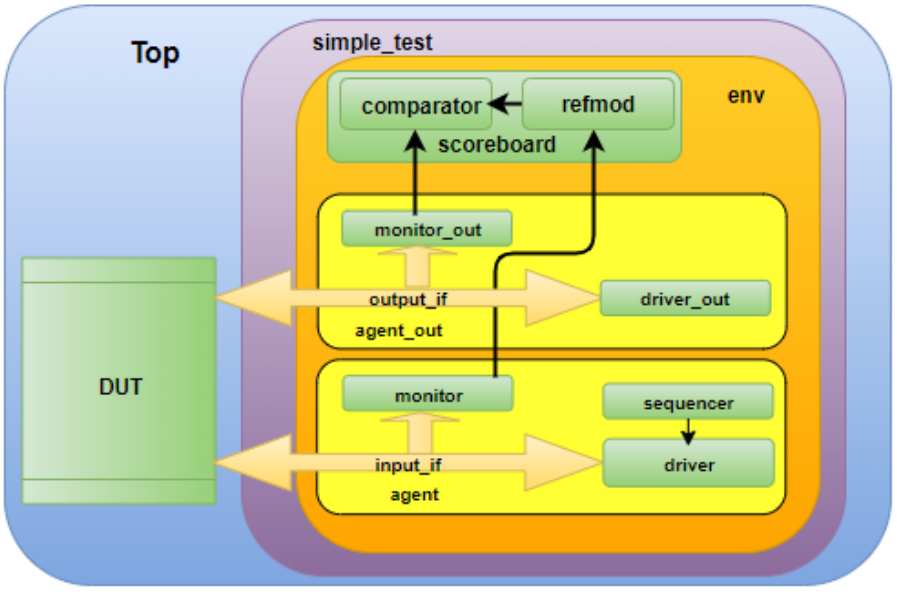

Figure 1: UVM testbench

The top-level *top.sv* include all the modules and it is used to generate the clock and reset stimuli. It also has the *simple_test.sv*, that is a test class inside the top, responsible for performing the useful tests, creating the environment *env.sv* and connecting the sequence to the sequencer. This class is formed by the scoreboard and one or more agents.
The first one checks whether the transactions produced by the DUT are correct or not. It is formed by a comparator, *comparator.sv*, a class that compares data between the reference model and the DUT. The *refmod.sv* is an executable specification, a golden model, that predicts the correct results from the provided stimulus. It simulates the DUT at high level of abstraction.

Furthermore, the active agent, *agent.sv* is a class that contains three components: a sequencer, a driver and a monitor. The agent contains functions for the build phase to create hierarchies and for the connect phase to connect the components.

Inside this agent, there is the sequencer, *sequencer.sv*, the class used to stimulate the DUT, generating the sequences of data to be elaborated. Since the sequencer sends transactions (packets of data in a high level of abstraction) and the DUT only understands the data coming from the interface, a class called driver, *driver.sv*, is responsible for the translation of the sequence to signal activity, making the device under test interface able to understand these data. In addition, the monitors, *monitor.sv*, perform the inverse function of drivers, retrieving the transactions to be sent to the *refmod*.

Moreover, a passive agent, composed by only the driver and monitor, is exploited to read the DUT output and provide it to the scoreboard in order to perform the comparison.

2 Testing an adder

To gain confidence with the UVM framework, the given adder is tested, verifying that the DUT and refmod results matched. Then, some modifications were applied.

It is possible to see that the *top.sv* file in the *tb* folder includes all the required testbench, interfaces and DUT files. Therefore, inside the *sim* folder, the tool **QuestaSim** is launched by using the commands *source /software/scripts/init_questa10.7c* and *vsim*. Then, as usual, the *work* folder is created. The source code is compiled by using *vlog -sv ../tb/top.sv* and then the simulation is run with the commands *vsim top* and *run* 4 *us*. As the simulation proceeds, the UVM framework logs its results in the transcript and summarises the obtained results at the end. In this way, it is possible to see if the DUT and the *refmod* results matched, in figure 2.

```
--- 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
```

Figure 2: UVM report summary

At this point, it is interesting to apply some modifications, such as: use a different word length, apply some constraints to the randomly generated data and modify the *refmod* to obtain some mismatches.

Word length modification. The first attempt was to have a different data word length. In order to do so, two files of the testbench must be modified, *packet_in.sv* and *packet_out.sv*, by using for instance **longint** type instead of **integer**. Also the DUT must be modified accordingly, supporting 64-bit operands.

Constraints of the randomly generated numbers. The second modification consisted into applying some constraints to the randomly generated numbers within the *packet_in* class to define a particular condition. This is obtained by using the **constraint** statement. Here, it is possible to use the relational operators `<`, `<=`, `>`, `>=` and `==` to compose the expression of the constraint to a variable. More than one expression can be part of the same constraint as long as only one operator is present in each expression.

Moreover, the **inside** operator can be exploited to express the upper and lower bounds of the range in which the variable must be included. For instance, this was

used in the example of figure 3, in which the operand **A** is restricted to the range between 100 and 1000 and **B** must be lower than 10 times **A**.

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

Figure 3: Adder with constraints

Using the directive `$display()`, it is possible to see what happens to **A** and **B**, and to verify that the applied constraints are actually observed.

Mismatching refmod. To cause some mismatches, a simple modification could be to change the operation implemented within the *refmod*, for instance by subtracting the operands instead of adding them. In this way, repeating the simulation, the UVM report summary will show the warnings and comparator mismatches in figure 4.

```
tr_out.data = tr_in.A - tr_in.B;
```

```
--- 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
```

Figure 4: UVM report summary

3 Testing the MBE-Dadda tree multiplier

In order to test the MBE-Dadda tree multiplier with the same UVM framework, some opportune modifications should be done:

Device under test. At first, the file *DUT.sv* is changed with the purpose to instantiate the combinational multiplier as a module in the DUT:

```
MBE_mult dadda_under_test(.a(in_inter.A),.b(in_inter.B),.p(out_inter.data));
```

Interface of the DUT. Then, the file *dut_if.sv* is changed in order to accommodate the multiplier data width. In fact, in the MBE-Dadda implementation the inputs are on 24 bits while the output is on 48 bits.

```
logic [23:0] A, B;
logic [47:0] data;
logic valid, ready;
```

Golden model. After that, the operation performed in *refmod.sv* is modified to perform the multiplication instead of the addition.

```
tr_out.data = tr_in.A * tr_in.B;
```

Inputs and output. Finally, in the file *packet_in.sv*, two random integer numbers belonging to the range $[0\colon 2^{24}-1]$ are generated as the inputs of the testbench. On the other hand, for the output in the file *packet_out.sv*, a **longint** type is needed due to the length of the result of the multiplication.

After all these modifications, the tool **QuestaSim** has been launched and, then, it was possible to appreciate in the UVM report summary that the DUT and the *refmod* results matched.

An interesting thing to be noticed is that the UVM framework performs one comparison every 3 clock cycles for a total of 101 couples of inputs. This is due to the state machine present in the file *DUT.sv*. Every time that the DUT is ready to receive a new pair of inputs, the signal **in_inter.ready** goes to one and the machine remains in the state **WAIT** until the inputs become available, namely when the signal **in_inter.valid** is equal to 1. Then the operation is performed. Finally when the output of the DUT is ready, the signal **out_inter.valid** is asserted and the machine goes in the state **SEND** where the signal **out_inter.valid** is cleared and the **in_inter.valid** is asserted, the comparison between the DUT and the reference model is performed and a new cycle begins.

4 Testing the whole floating point multiplier

Finally, the floating point multiplier is tested, although it is more difficult to manage as this is not a fully combinational block due to pipeline stages.

In this case, when a new pair of inputs is sampled, the output will not be available in the same clock cycle. Therefore, in order to exploit the correct timing, the FSM in *DUT_generic.sv* must be modified. In particular, the input data must propagate through the pipeline of the multiplier before comparing its result with the *refmod* output.

The idea is to have a state machine that asserts the **out_inter.valid** signal every time the output of the floating point multiplier is about to be available. A new couple of inputs is taken only after the comparison. To reach this aim, some idle states may be inserted between the states **WAIT** and **SEND**. The number of required idle states corresponds to the number of pipeline stages minus one. In this way, it is possible to test any sequential circuit.

In the presented work, the FSM remains in a single state named **PIPEFILL** as long as necessary. The parameter **num_pipe** is defined to be equal to the number of pipeline stages within the DUT, and the machine remains in the **PIPEFILL** state until a variable **cnt** is less than **num_pipe - 2**. Whenever this condition is not verified anymore, the signal **out_inter.valid** is asserted. At this point, the FSM moves to the **SEND** state where the DUT computation is shown by the `$display()` statement and compared to the expected value from the *refmod*. It is worth noting that two variables, **A_pipe** and **B_pipe**, are used to keep track of the input corresponding to that specific output in order to correctly display the performed operation.

The class defined in the file *refmod_generic.sv* exploits the `$shortrealtobits()` and `$bitstoshortreal()` directives to manage the floating point numbers in order to correctly evaluate the multiplication. In order to show the floating point operators, the general format floating point specifier `%g`, is used within the `$display()` directive.

The simulation is done using the script *run_sim.do*, where all the files needed are compiled and *top.sv* is tested. The total simulation time has increased from 4 μs to 7 μs because, in this case, the number of clock cycles used to perform the multiplication of each pair of inputs is greater than 3, so more time is needed to complete the simulation of 101 input samples.

The simulation is run in the usual way obtaining a successful outcome, as shown in figure 5. Here it is possible to see the 99 comparator matches and 2 mismatches, which are due to the denormalization format of data.

```
--- 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
```

Figure 5: UVM report summary

4.1 Further modifications

The state machine described before compares an output every 6 clock cycles. An interesting thing to do is to try to elaborate two data at the same time within the pipeline.

This is achieved by modifying the previous FSM within the DUT module removing the counter in the **PIPEFILL** state, which is now used only as an intermediate state between the **WAIT** and **SEND** states. In this way, a single data which pass through the whole pipeline will experience the FSM to pass from the **WAIT** state to **PIPEFILL**, then to the **SEND** state, where the previous result is provided. Then again to the **WAIT** state, where a new data is loaded, then **PIPEFILL** and finally **SEND** state again, where its result is presented in output. It is worth noting that two variables, **A_pipe** and **B_pipe**, are used to keep track of the input corresponding to that specific output in order to correctly display the performed operation.

Moreover, it is also necessary that the *refmod* operation is delayed. This is achieved by exploiting the variables **var_A** and **var_B** in the *refmod.sv* that save the input from *packet_in* and elaborate them only when a new data is requested.

The results obtained in the UVM report summary in figure 6 shows 98 comparator matches and 3 comparator mismatches, two of them due to the denormalization of data, while the first comparison is wrong since the FPU multiplier output is not valid yet.

```
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :  109
UVM_WARNING :    3
UVM_ERROR :    1
UVM_FATAL :    0
** Report counts by id
[Comparator Match]    98
[Comparator Mismatch]     3
[MISCMP]     6
[Questa UVM]     2
[RNTST]     1
[TEST_DONE]     1
[env]     2
```

Figure 6: UVM report summary