Politecnico di Torino

Integrated System Architecture
UVM

Fourth Lab Report

`https://github.com/aledolme19/ISA_LAB.git`

Francesco Babbaro Pierfrancesco Boccardi Alessandra Dolmeta

2021-2022

1 Introduction

UVM (**Universal Verification Methodology**) is a standardized methodology for verifying integrated circuit designs. Nowadays, it is a *de facto* standard for building modular testbenches, resorting to **SystemVerilog** (SV) unique capabilities, in particular its Object-Oriented nature. SV is an extension of Verilog HDL, adding some features to ease hardware description and many non-synthesizable constructs specifically thought for verification purposes.
In order to have a consistent testbench flow, UVM introduces *phases* to synchronize major functional steps a simulation runs through. These steps are sequential in nature which are executed in the following order:

- **build phase**: where the testbench is constructed, connected and configured.
- **connect phase**: is used to connect different sub-components in a class.
- **run-time phase**: stimulus generation and time consuming simulation takes place here.
- **clean up phase**: where the test results are collected and reported.

The UVM testbench is illustrated in Figure 1. The DUT is the hardware implementation that interacts with the testbench in order to verify its functionality.

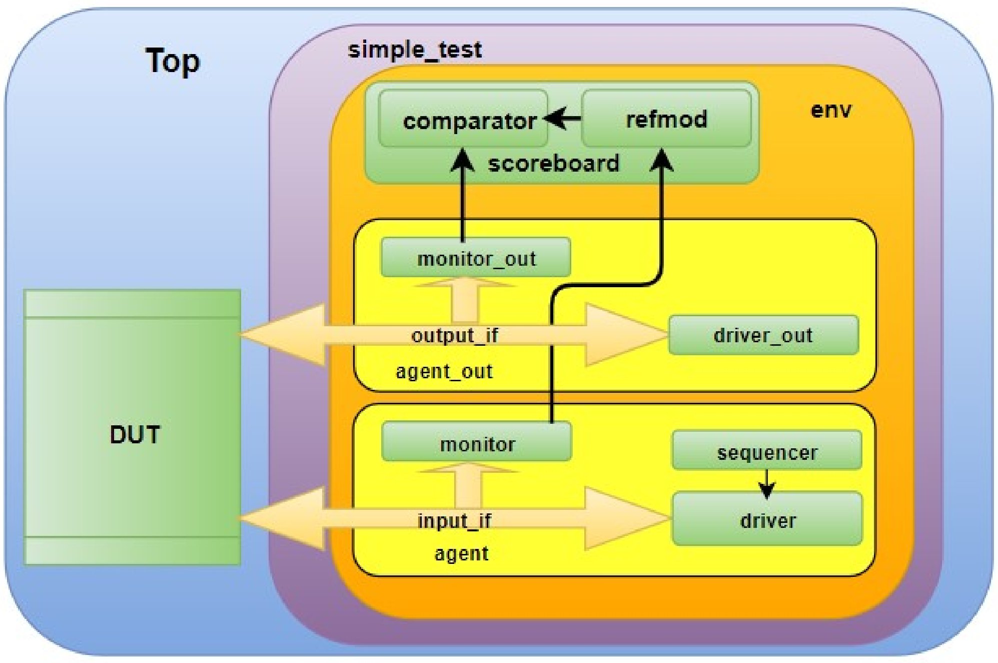


Figure 1: UVM testbench

To stimulate the DUT, a class named `sequencer` generates sequences of data to be transmitted to the DUT. Since the sequencer sends transactions (packets of data in a high level of abstraction) and the DUT only understands the logica signals coming from the interface, a class called `driver` converts packets of data to signals feeding the DUT.
The data crossing the interface should be captured for a later validation of the stimuli. Since **the driver only converts transactions to signals**, another block performs the inverse function of the driver. The `monitor` is a component that reads the communication between the driver and the DUT and retrieves the transaction. The class monitor reads the data on the interface and converts it into transaction to be compared with the reference model.
The reference model (`refmod`) is a model conceived in an early phase, before the RTL implementation, assumed as ideal. It simulates the DUT at a high level of abstraction. 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.
An **agent** is a class that typically contains three components: a **sequencer**, a **driver** and a **monitor**. There are two types of agents: Active Agent, which contains all the three components and the Passive Agent, which only contains the monitor and the driver. The agent contains functions for the build phase to create hierarchies and for the connect phase to connect the components.
A **comparator** is a **class that compares data between the reference model and the DUT**. The reference model and the comparator form the scoreboard that checks whether the transactions produced by the DUT are correct or not. One or more agents and the scoreboard form the Environment class `env`.
The test class is responsible for performing the tests creating the environment and connecting the sequence to the sequencer. And finally, the top module instantiates the DUT and the testbench.

UVM Hierarchy

The simplest UVM hierarchy is composed by:

- UVM Testbench;
- UVM Test;
- UVM Environment.

Testbench

Testbench instantiates the Device Under Test (DUT) and Test class and configures the connection between them.
We will perform a **transaction-based verification**. This allows simulation and debug of the design at the transaction level. It is the most basic testbench, using a UVM agent that is made of:

- **sequence**: the stimulus generator, which creates transaction-level traffic to send them to the driver;
- **sequencer**: the *arbiter*, which controls the flows of request and response sequence item between sequences and driver;
- **driver**: takes these transactions from the sequencer and then converts them into pin signal-level activity, and drives the DUT;
- **monitor**: snoops the signal-level activity and converts them back transactions that are sent to a scoreboard;
- **scoreboard**: gets the monitored transactions from the monitor compares them with expected transactions (response transactions).

Test

In the UVM testbench, the UVM-test is the top level UVM component. It is a class that **encapsulates test-specific instructions written by the test writer**. The UVM test instantiate the top-level environment, configure the environment and apply stimulus by invoking UVM sequences through the environment.

Environment

The UVM environment is a component that groups together other verification components that are interrelated. The components that are usually instantiated inside the UVM environment are **UVM agents** and **UVM scoreboards**.
UVM agents drives the signal-level interface of the DUT.
UVM scoreboard check the response of the DUT against expected response.[1]

[1]Theory reference: `https://sistenix.com/basic_uvm.html`

2 Testing an Adder

In this section, we are going to verify the correct behaviour of a simple adder.
The top-level **top.sv** is a SV module that instantiates the HDL top module, with DUT and its interface.
The `top` module is firstly initializing `clk` and `rst` stimuli. SystemVerilog introduces several new data types, many of these familiar to C environment. The idea is that algorithms modelled in C can more easily be converted to SystemVerilog if the two languages have the same data types.
`clk` and `rst` are **logic** variables, which are are 4-state data types that can be 0, 1, X (unknown) or Z (high-impedance). Moreover, `clk` is then defined with an *always* procedural block.
Then, it sets the virtual interface handling to the UVM configuration database and it finally starts UVM testbench by calling the static method `run_test()`. This method constructs the **root node of the testbench hierarchy**, which defines the test case to be executed by specifying the configuration of the testbench components and the stimuli.

2.1 Code Analysis

We thank our colleague *Imane El Fentis* MSc thesis[1] for this explanation. Part of his work has been here reported (since it has been used the same SystemVerilog code), in order to briefly explain the aim of each .sv files.

The interface is `dut_if`. **Modport** groups and specifies the port directions to the wires/signals declared within the interface: the first one is used used to describe the inputs going to the DUT and second one is used for the interface of signals coming out from DUT.

```
interface dut_if(input clk, rst);
    logic [31:0] A, B;
    logic [31:0] data;
    logic valid, ready;

    modport port_in (input clk, rst, A, B, valid, output ready);
    modport port_out (input clk, rst, output valid, data, ready);
endinterface
```

`packet_in` and `packet_out` are transactions sent to the DUT and received from it respectively. Both are extended from `uvm_sequence_item` and registered in the factory, and then built using new() method, each of the packets has its own properties.

Sequence class `sequence_in` is derived from `uvm_sequence` base class and it is parametrized with #(`packet_in`). It is also registered in the factory and built. A task is used, in order to exploit `start_item`. This tells the sequencer that the sequence is available to be arbitrated. Then, it randomizes the returned value (`assert(tx.randomize());`) and exploits `finish_item` to send the randomized sequence_item to the driver.

The `sequencer` class is derived from `uvm_sequencer` parametrized with `packet_in`. It is registered in the factory using `'uvm_component_utils` not `'uvm_object_utils` and it will arbitrate the randomized transactions and then send them to the driver.

The `driver` class is derived from `uvm_driver` always parametrized with `packet_in`. In this class is defined the **virtual interface**. SystemVerilog classes can reference signals within an interface via a virtual interface handle: this allows a class to either assign or sample values of signals inside an interface, or to call tasks or functions within an interface.
In `task run_phase`, it uses fork-join to run in parallel three tasks: `reset_signals()`, `get_and_drive(phase)` and `record_tr()`.
The **virtual protected task** is used when we don't want the methods and members be accessible from outside only by the child inherited. In this task, the inputs are reset when reset=1.
In the `get_and_drive` task, it waits until reset gets activated low and when the clock is positive, it loops forever (`forever begin`) to get multiple transactions from sequencer using `seq_item_-port.get(req)`.

In the `drive_transfer` task, the transactions are transferred to the virtual variables. It waits for one clock cycle to generate and record then. It ends record after another clock cycle that is used for hold time, the task is ended.

The `driver_out` is extended from `uvm_driver`, while `monitor` class is defined from `uvm_monitor`. This receives data from the DUT and sends them to scoreboard.
Like the driver, the `monitor_out` uses virtual variables to get data from DUT.

The `agent` instantiates the components: **sequencer**, **driver** and **monitor**. It consists of the handles of these components, after registration in the factory and constructing it, it uses `uvm_-analysis_port` to get transactions then create the handle of sequencer, driver and monitor. Then, it connects monitor to `item_collected_port` and the driver with `seq_item_export`. The `agent_out` connects `driver_out` and `monitor_out`.

The `reference model` is extended from `uvm_component`, it is registered in the factory and constructed. This is one of file we will modify the most.
It has two handles of `packet_in` and `packet_out`. In run phase, it calls two functions: “sum” and “sum1”. These functions are imported in the top of the file using `import \DPI -C" context function`. This gets the transaction inputs and then pass them as arguments to these functions.

```
class refmod extends uvm_component;
    `uvm_component_utils(refmod)

    packet_in tr_in;
    packet_out tr_out;
    uvm_get_port #(packet_in) in;
    uvm_put_port #(packet_out) out;

    function new(string name = "refmod", uvm_component parent);
        super.new(name, parent);
        in = new("in", this);
        out = new("out", this);
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        tr_out = packet_out::type_id::create("tr_out", this);
    endfunction: build_phase

    virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);

        forever begin
            in.get(tr_in);
            tr_out.data = tr_in.A + tr_in.B;
            $display("refmod: input A = %d, input B = %d, output OUT = %d",tr_in.A, tr_in.B,
                ↪ tr_out.data);
            $display("refmod: input A = %b, input B = %b, output OUT = %b",tr_in.A, tr_in.B,
                ↪ tr_out.data);
            out.put(tr_out);
        end
    endtask: run_phase
endclass: refmod
```

The `comparator` is extended from `uvm_scoreboard`. It defines some string that are parametrized to accept a data object of type T. In run phase, it raises the objection and drops it in order to coordinate status information between the components. The `put()` is used in refmod to get the expected result. Since **the comparator is the receiver with respect to reference model**, it must define the put task, which is a blocking task.
The `try_put()` is a non blocking function, and it will attempt to perform a put operation. Therefore, it will return true if it succeeds, and false if does not, while the `can_put()` function is just a test to see if a non-blocking put operation would succeed without actually performing the operation.
The comparator is in charge of comparing the result of the DUT with the one obtained during the verification test, and reporting if the two results are matching or not.

The `environment` class connects and instantiates the component that are in bottom layer such as agent, `agent_out`, `refmod` and `comparator`.
`simple_test` is where the container environment and transactions are instantiated.

Finally, `top` is the top level where DUT is instantiated. All included files can grouped and packed in one package then import it. This will be the only SystemVerilog file to be compiled during Modelsim simulation.

2.2 QuestaSim Simulations

2.2.1 First Simulation

The aim of this section is simply to simulate the files as they were supplied.
The files that must be compiled in QuestaSim environment are: `adder.sv`, `DUT.sv`, `dut_if.sv` and `top.sv`. In `adder.sv` there is simply the behavioral description of the adder. In `DUT.sv` there is a Finite State Machine (FSM) which controls the sequence of states of our simulation, while in `dut_if` there is the interface of our component.
`top.sv` includes all the different SystemVerilog files that have been commented in subsection 2.1.

2.2.2 Second Simulation: changing word length

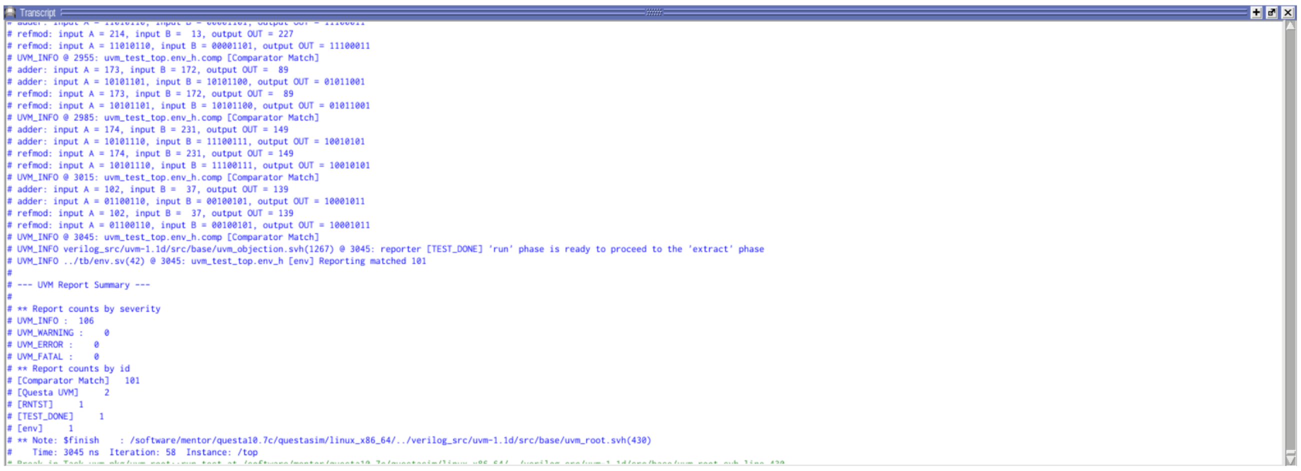
In order to get used to the UVM framework, we try to modify the work length of the `adder`, modifying the testbench files accordingly.
Therefore, we need to modify the following files: `packet_in`, `packet_out`, `DUT_if` and `adder`.
This is done in order to **accommodate the new word length**. The given `refmod` will follow `packet_in` and `packet_out` automatically.
From:

```
    rand integer A;
    rand integer B;
```

The two inputs has been modified as followed:

```
    rand bit[7:0] A;
    rand bit[7:0] B;
```

An integer is a number without a fractional part. It is intrinsically on 32 bits.
In SystemVerilog, a new data type is introduced. One of the most important 2-state data type that is added is the `bit`. This is quite used in testbenches. A variable of type `bit` can be either 0 or 1. A range from MSB to LSB can be provided, and it will represent and store multiple bits. In our example, the word length has been changed from 32 bit to 8 bit.
Moreover, variables can be declared random using `rand`: their value will be uniformly distributed over their range.
If this variable is randomized without any constrains, then any value in the range of interest will be assigned with an equal probability.



```
# refmod: input A = 214, input B =  13, output OUT = 227
# refmod: input A = 11010110, input B = 00001101, output OUT = 11100011
# UVM_INFO @ 2955: uvm_test_top.env_h.comp [Comparator Match]
# adder: input A = 173, input B = 172, output OUT =  89
# adder: input A = 10101101, input B = 10101100, output OUT = 01011001
# refmod: input A = 173, input B = 172, output OUT =  89
# refmod: input A = 10101101, input B = 10101100, output OUT = 01011001
# UVM_INFO @ 2985: uvm_test_top.env_h.comp [Comparator Match]
# adder: input A = 174, input B = 231, output OUT = 149
# adder: input A = 10101110, input B = 11100111, output OUT = 10010101
# refmod: input A = 174, input B = 231, output OUT = 149
# refmod: input A = 10101110, input B = 11100111, output OUT = 10010101
# UVM_INFO @ 3015: uvm_test_top.env_h.comp [Comparator Match]
# adder: input A = 102, input B =  37, output OUT = 139
# adder: input A = 01100110, input B = 00100101, output OUT = 10001011
# refmod: input A = 102, input B =  37, output OUT = 139
# refmod: input A = 01100110, input B = 00100101, output OUT = 10001011
# UVM_INFO @ 3045: uvm_test_top.env_h.comp [Comparator Match]
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 3045: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ../tb/env.sv(42) @ 3045: uvm_test_top.env_h [env] Reporting matched 101
#
# --- 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
# ** Note: $finish    : /software/mentor/questa10.7c/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 3045 ns  Iteration: 58  Instance: /top
```

Figure 2: UVM testbench: 8 bit word length

As previously stated, the comparator is in charge of comparing the result of the DUT with the one obtained during the verification test, and reporting if the two results are matching or not.

As shown in Figure 2, the verification has been performed correctly. In fact, among the 101 test that has been done, 101 results matched.

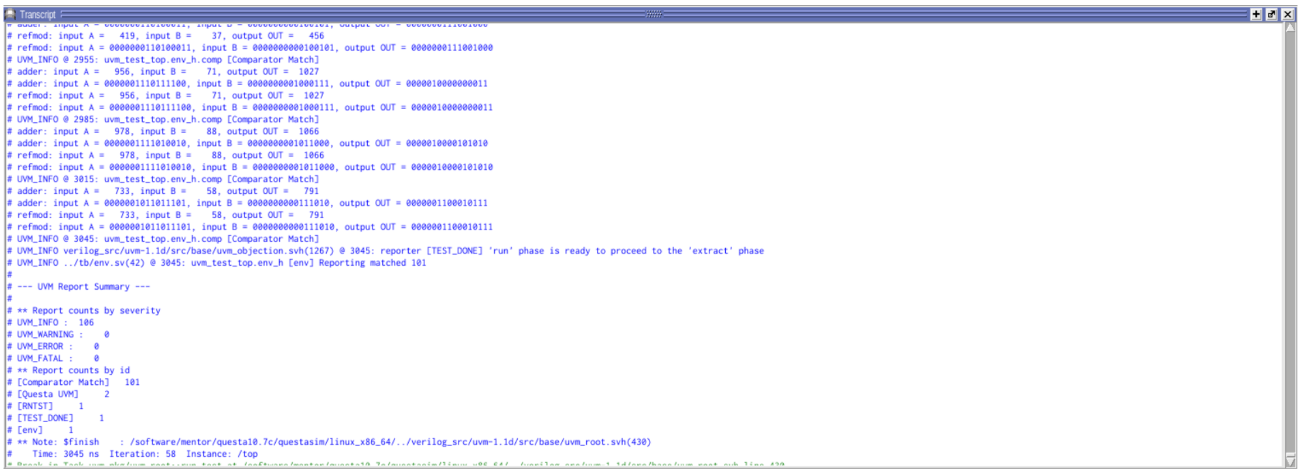
2.2.3 Third Simulation: constraints

Now we try to add some constraints to the random number generation in the packet in module. SV rand constraints are very flexible and powerful, they can accommodate many verification needs.

```
    rand bit [15:0] A;
    rand bit [15:0] B;

    constraint my_range_A { A > 100; A < 1000; }
    constraint my_range_B { B < A/10; }
```

As shown above, the word length has been changed again. The `constraint` command is used in order to fix a particular range of values in which our input will be distributed.
In this example, since no other constraints are given, SystemVerilog gathers all the values and chooses between the values with equal probability.
Moreover, we can find also a dependency between the two constraints, since B is directly proportional to the value assigned to A.



```
# refmod: input A =   419, input B =    37, output OUT =   456
# refmod: input A = 0000000110100011, input B = 0000000000100101, output OUT = 0000000111001000
# UVM_INFO @ 2955: uvm_test_top.env_h.comp [Comparator Match]
# adder: input A =   956, input B =    71, output OUT =  1027
# adder: input A = 0000001110111100, input B = 0000000001000111, output OUT = 0000010000000011
# refmod: input A =   956, input B =    71, output OUT =  1027
# refmod: input A = 0000001110111100, input B = 0000000001000111, output OUT = 0000010000000011
# UVM_INFO @ 2985: uvm_test_top.env_h.comp [Comparator Match]
# adder: input A =   978, input B =    88, output OUT =  1066
# adder: input A = 0000001111010010, input B = 0000000001011000, output OUT = 0000010000101010
# refmod: input A =   978, input B =    88, output OUT =  1066
# refmod: input A = 0000001111010010, input B = 0000000001011000, output OUT = 0000010000101010
# UVM_INFO @ 3015: uvm_test_top.env_h.comp [Comparator Match]
# adder: input A =   733, input B =    58, output OUT =   791
# adder: input A = 0000001011011101, input B = 0000000000111010, output OUT = 0000001100010111
# refmod: input A =   733, input B =    58, output OUT =   791
# refmod: input A = 0000001011011101, input B = 0000000000111010, output OUT = 0000001100010111
# UVM_INFO @ 3045: uvm_test_top.env_h.comp [Comparator Match]
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 3045: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ../tb/env.sv(42) @ 3045: uvm_test_top.env_h [env] Reporting matched 101
#
# --- 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
# ** Note: $finish    : /software/mentor/questa10.7c/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 3045 ns  Iteration: 58  Instance: /top
```

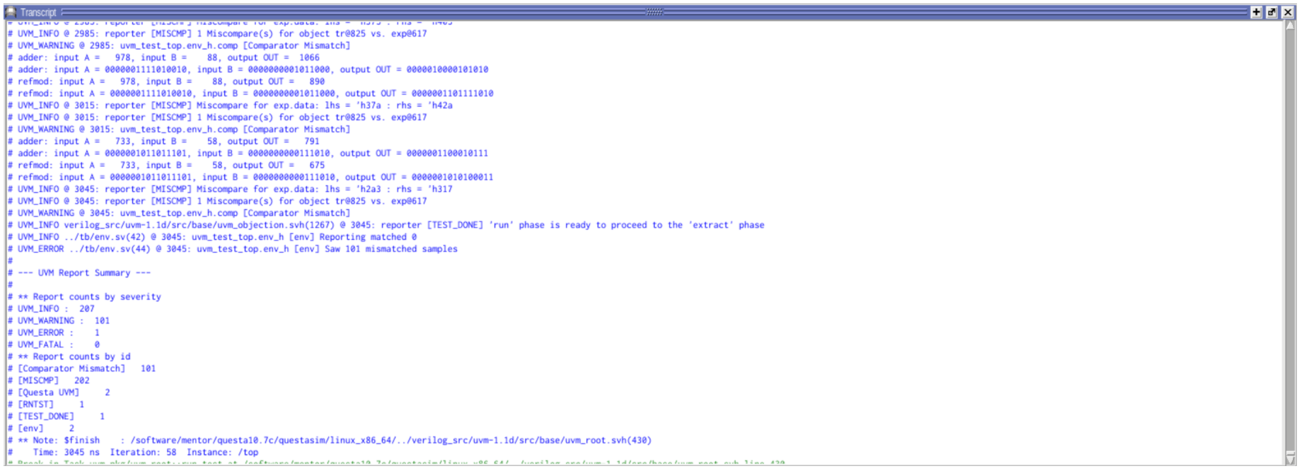
Figure 3: UVM testbench: 16 bit word length with constraints

2.2.4 Fourth Simulation: changing `refmod`

Now we try to modify **`refmod`**, by changing the operator used. We substitute the addition with a subtraction, in order to verify if the UVM testbench is working properly.

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

In fact, since we are testing an adder but comparing it with a subtractor, it should report a number of mismatches equal to all the couples of inputs examined for which `B`$\neq$`0`. In fact, the only case in which we get the same results for the sum and for the subtraction between two numbers is when the second operand is 0.



```
# UVM_INFO @ 2985: reporter [MISCMP] 1 Miscompare(s) for object tr@825 vs. exp@617
# UVM_WARNING @ 2985: uvm_test_top.env_h.comp [Comparator Mismatch]
# adder: input A =   978, input B =    88, output OUT =  1066
# adder: input A = 0000001111010010, input B = 0000000001011000, output OUT = 0000010000101010
# refmod: input A =   978, input B =    88, output OUT =   890
# refmod: input A = 0000001111010010, input B = 0000000001011000, output OUT = 0000001101111010
# UVM_INFO @ 3015: reporter [MISCMP] Miscompare for exp.data: lhs = 'h37a : rhs = 'h42a
# UVM_INFO @ 3015: reporter [MISCMP] 1 Miscompare(s) for object tr@825 vs. exp@617
# UVM_WARNING @ 3015: uvm_test_top.env_h.comp [Comparator Mismatch]
# adder: input A =   733, input B =    58, output OUT =   791
# adder: input A = 0000001011011101, input B = 0000000000111010, output OUT = 0000001100010111
# refmod: input A =   733, input B =    58, output OUT =   675
# refmod: input A = 0000001011011101, input B = 0000000000111010, output OUT = 0000001010100011
# UVM_INFO @ 3045: reporter [MISCMP] Miscompare for exp.data: lhs = 'h2a3 : rhs = 'h317
# UVM_INFO @ 3045: reporter [MISCMP] 1 Miscompare(s) for object tr@825 vs. exp@617
# UVM_WARNING @ 3045: uvm_test_top.env_h.comp [Comparator Mismatch]
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 3045: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ../tb/env.sv(42) @ 3045: uvm_test_top.env_h [env] Reporting matched 0
# UVM_ERROR ../tb/env.sv(44) @ 3045: uvm_test_top.env_h [env] Saw 101 mismatched samples
#
# --- 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
# ** Note: $finish    : /software/mentor/questa10.7c/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 3045 ns  Iteration: 58  Instance: /top
```

Figure 4: UVM testbench: 16 bit word length with mismatch

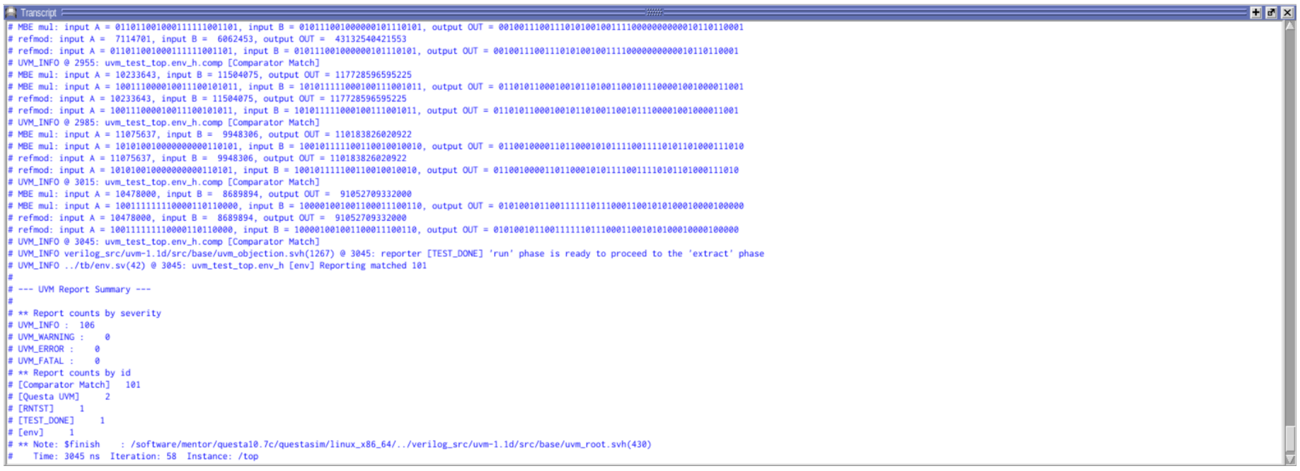
As shown in Figure 4, the verification has been performed correctly. In fact, among the 101 test that has been done, 101 results mismatched, sign that there is no couple for which `B=0`. This is the result that we were expecting, considering that the DUT is performing an addition while the **`refmod`** file is performing a subtraction.

3 Testing the MBE-Dadda tree multiplier

The aim of this section is simply to simulate and test the files related to the MBE-Dadda tree multiplier that we have implemented in the second laboratory.
We need to modify the following files: `packet_in`, `packet_out`, `DUT`, `dut_if`.
`packet_in` word length of A and B have been changed to 24bit, instead the result needs 48 bits.
Compared to the previous simulations, the DUT must be replaced with the **MBE multiplier** to test. Transaction objects (`packet_in`, `packet_out`), in fact, were changed in order to be compliant with the new DUT.
`refmod` must be changed too. In particular, in this case we are interested in performing a multiplication, instead of addition. Therefore, the code will be as shown below.

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

No constraints were set to inputs, because we are interested to test the multiplier reliability over the whole input range.



```
# MBE mul: input A = 011011001000111111001101, input B = 010111001000000101110101, output OUT = 001001110011101010010011110000000000010110110001
# refmod: input A =  7114701, input B =  6062453, output OUT =  43132540421553
# refmod: input A = 011011001000111111001101, input B = 010111001000000101110101, output OUT = 001001110011101010010011110000000000010110110001
# UVM_INFO @ 2955: uvm_test_top.env_h.comp [Comparator Match]
# MBE mul: input A = 10233643, input B = 11504075, output OUT = 117728596595225
# MBE mul: input A = 100111000010011100101011, input B = 101011111000100111001011, output OUT = 011010110001001011010011001011100001001000011001
# refmod: input A = 10233643, input B = 11504075, output OUT = 117728596595225
# refmod: input A = 100111000010011100101011, input B = 101011111000100111001011, output OUT = 011010110001001011010011001011100001001000011001
# UVM_INFO @ 2985: uvm_test_top.env_h.comp [Comparator Match]
# MBE mul: input A = 11075637, input B =  9948306, output OUT = 110183826020922
# MBE mul: input A = 101010010000000000110101, input B = 100101111100110010010010, output OUT = 011001000011011000101011110011110101101000111010
# refmod: input A = 11075637, input B =  9948306, output OUT = 110183826020922
# refmod: input A = 101010010000000000110101, input B = 100101111100110010010010, output OUT = 011001000011011000101011110011110101101000111010
# UVM_INFO @ 3015: uvm_test_top.env_h.comp [Comparator Match]
# MBE mul: input A = 10478000, input B =  8689894, output OUT =  91052709332000
# MBE mul: input A = 100111111110000110110000, input B = 100001001001100011100110, output OUT = 010100101100111111011100011001010100010000100000
# refmod: input A = 10478000, input B =  8689894, output OUT =  91052709332000
# refmod: input A = 100111111110000110110000, input B = 100001001001100011100110, output OUT = 010100101100111111011100011001010100010000100000
# UVM_INFO @ 3045: uvm_test_top.env_h.comp [Comparator Match]
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 3045: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ../tb/env.sv(42) @ 3045: uvm_test_top.env_h [env] Reporting matched 101
#
# --- 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
# ** Note: $finish    : /software/mentor/questa10.7c/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 3045 ns  Iteration: 58  Instance: /top
```

Figure 5: UVM testbench: 24 bit MBE-Dadda tree Multiplier

Again, the comparator is in charge of comparing the result of the DUT with the one obtained during the verification test, and reporting if the two results are matching or not.
As shown in Figure 5, the verification has been performed correctly. In fact, among the 101 test that has been done, 101 results matched.

4 Testing the whole floating point multiplier

The aim of this section is simply to simulate and test the files related to the whole floating point multiplier that we have implemented in the second laboratory.
We need to modify the following files: packet_in, packet_out, DUT, dut_if.
packet_in word length of A and B have been changed to 32bit, as for the result.
Compared to the previous simulations, the DUT must be replaced with the **floating point multiplier** to test. Transaction objects (packet_in, packet_out), in fact, were changed in order to be compliant with the new DUT.
refmod must be changed too.
Its code is shown below.

```
class refmod extends uvm_component;
    `uvm_component_utils(refmod)

    packet_in tr_in;
    packet_out tr_out;
    uvm_get_port #(packet_in) in;
    uvm_put_port #(packet_out) out;

    int count = 0;
    bit [2:0][31:0] data_out;

    function new(string name = "refmod", uvm_component parent);
        super.new(name, parent);
        in = new("in", this);
        out = new("out", this);
        for(int i = 0; i < 3; i++) begin
            data_out[i] = 'x;
        end
    endfunction

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        tr_out = packet_out::type_id::create("tr_out", this);
    endfunction: build_phase

    virtual task run_phase(uvm_phase phase);
        super.run_phase(phase);

        forever begin
        in.get(tr_in);
        for(int i = 2; i > 0; i--) begin
            data_out[i] = data_out[i-1];
        end

        data_out[0] = $shortrealtobits($bitstoshortreal(tr_in.A) * $bitstoshortreal(tr_in.B));

        $display("----------Simulation %d----------", count);
        $display("refmod: input A=%f, input B=%f, output OUT = %f", $bitstoshortreal(tr_in.A),
            ↪ $bitstoshortreal(tr_in.B), $bitstoshortreal(data_out[0]));
        $display("refmod: input A = %b, input B = %b, output OUT = %b", tr_in.A, tr_in.B,
            ↪ data_out[0]);
        count++;
        tr_out.data = data_out[2];
        out.put(tr_out);
        end
    endtask: run_phase
endclass: refmod
```

Since the floating point multiplier under test is pipelined, we lose the synchronisation between DUT and reference model. As a consequence, it leads to wrong comparisons. This problem was overcame storing the products computed by the reference model into a vector.
In the constructor this vector was initialised using unknown value `X`, in this way the first comparisons, until the pipe queue is not full, do not generate **mismatch errors**.
This **vector acts like a right shift register**. So at each computed multiplication all the elements are shifted one position towards right, the product is stored into the first location (`index 0`), instead the value sent to the comparator is the last one (`index 2`).
Actually, the floating point multiplier has 6 pipeline registers. Nevertheless, the vector has lower memory locations. This is because inputs remain constant for more than one clock cycle. Therefore, in the meantime data go forward through pipeline registers.

Without any other modification, there would be some mismatches between the floating-point multiplier result and the results obtained by the verification process. This mismatches are due to the multiplier itself.
In fact, the floating point multiplier has not been implemented to work with denormal number. In the `UnpackFP` component has been implemented a logic function able to recognize if a number is denormal or not. However, the flag `isDN` is left `OPEN` when the `UnpackFP` components are instantiated in the first stage of the multiplier.
In the following stages of the multiplier, in the case the result obtained after the multiplication is a denormal number, the final result is approximated with zero.
Our reference model is not doing such an approximation, therefore, for each of the test performed in which the result is a denormal number, there would be a mismatch between the two results.
In order to solve the problem, without modifying the code of the floating-point multiplier that was supplied to us, we have modified the `refmod` file.
In particular, after computing the multiplication and saved the result in `data_out[0]`, we add the following control statement:

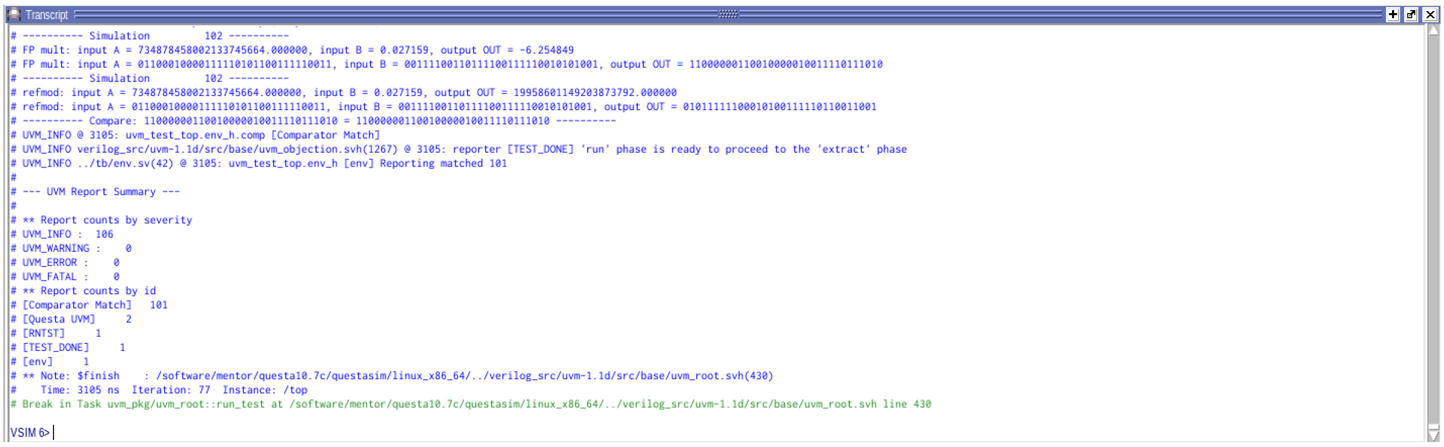
```
if(($bitstoshortreal(data_out[0])>0 && $bitstoshortreal(data_out[0])<$bitstoshortreal(32'
    ↪ h00800000)) || ($bitstoshortreal(data_out[0])<0 && $bitstoshortreal(data_out[0])>
    ↪ $bitstoshortreal(32'h80800000)))
    data_out[0] = 32'h00000000;
```

By doing so, we have that any denormal result obtained is approximated as a zero also in the reference model. In particular:

- if the result is positive `AND` it is lower than 1.17549435082e-38 (00000000100000000000000000000000), the output given by the reference model `data_out[0]` is set to 0;
- if the result is negative `AND` it is greater than -1.17549435082e-38 (10000000100000000000000000000000), the output given by the reference model `data_out[0]` is set to 0;

No constraints were set to inputs in order to test the FP-multiplier architecture on the whole input span.

The comparator is in charge of comparing the result of the DUT with the one obtained during the verification test, and reporting if the two results are matching or not.



```
# ---------- Simulation         102 ----------
# FP mult: input A = 7348784580021337456640.000000, input B = 0.027159, output OUT = -6.254849
# FP mult: input A = 01100010000111110101100111110011, input B = 00111100110111100111110010101001, output OUT = 11000000110010000010011110111010
# ---------- Simulation         102 ----------
# refmod: input A = 7348784580021337456640.000000, input B = 0.027159, output OUT = 199586011492038737920.000000
# refmod: input A = 01100010000111110101100111110011, input B = 00111100110111100111110010101001, output OUT = 01011111100010100111110110011001
# ---------- Compare: 11000000110010000010011110111010 = 11000000110010000010011110111010 ----------
# UVM_INFO @ 3105: uvm_test_top.env_h.comp [Comparator Match]
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 3105: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ../tb/env.sv(42) @ 3105: uvm_test_top.env_h [env] Reporting matched 101
#
# --- 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
# ** Note: $finish    : /software/mentor/questa10.7c/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 3105 ns  Iteration: 77  Instance: /top
# Break in Task uvm_pkg/uvm_root::run_test at /software/mentor/questa10.7c/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh line 430

VSIM 6>
```

Figure 6: UVM testbench: Floating-Point Multiplier

As shown in Figure 6, now the verification is performed correctly. In fact, among the 101 test that has been done, 101 results matched.

References

[1] Imane El Fentis. "Methodologies for SOC verification". MA thesis. 2020.