Politecnico di Torino
III Facoltà di Ingegneria

Integrated systems architecture

Laboratory 4

Laurea Magistrale in Ingegneria Elettronica
Orientamento: Sistemi Elettronici

Group n. 9

Authors:

Favero Simone S270686
Micelli Federico S270456
Spanna Francesca S278040

Repository Github: github.com/FrSp13/ISA_2020_Group_09_Lab_4

Index

Introduction to UVM

The aim of this laboratory is to verify several architectures relying on the Universal Verification Methodology (UVM).
The basic idea of this process is to test the architecture with a series of input data, generated with proper constraints, and compare the output results with values provided by a reference model.

The developed UVM testbench resorts on the use of HDL languages, such as VHDL or Verilog, for the Device Under Test (DUT) description and System Verilog for the verification environment. It is important to highlight the Object-Oriented nature of System Verilog. For this reason, the testbench can be organized in a hierarchical and flexible way: the definition of several classes allows the reutilization of the test framework in different scenarios.

The following figure reports a schematic view of the main blocks included in the verification environment, representing the main flow for a test sequence.

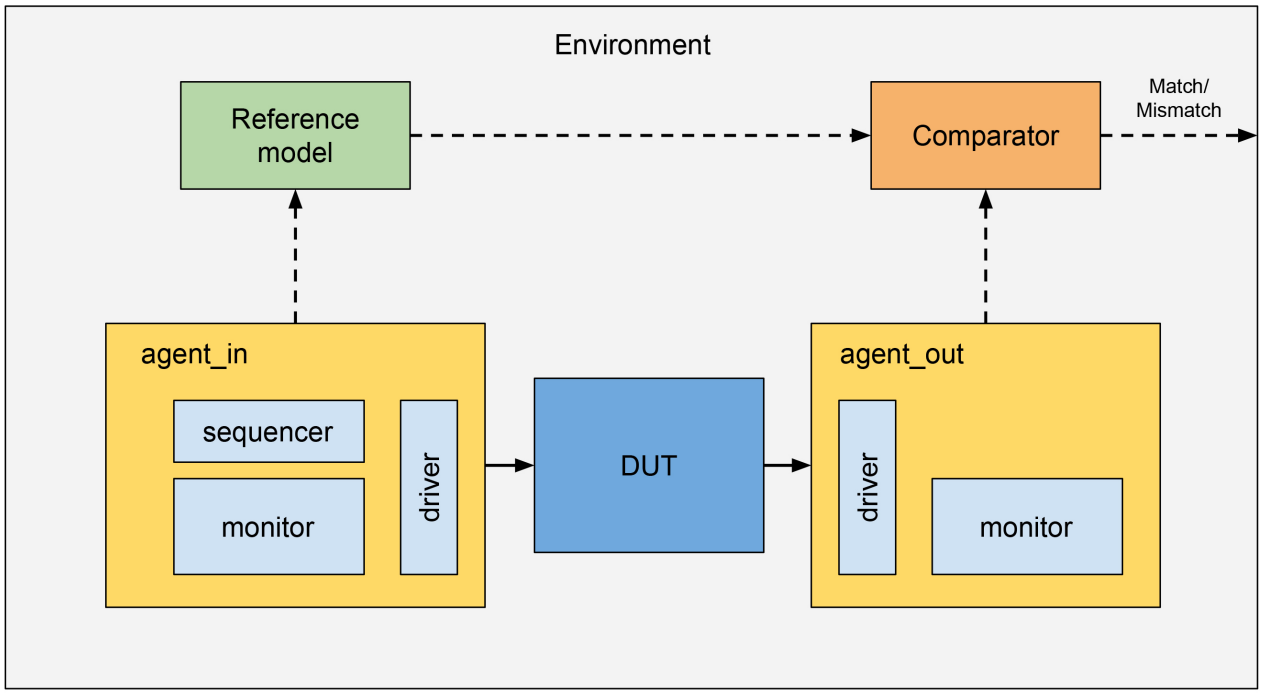

Figure 1.1 - Test environment scheme

As visible in the figure, the device under test interacts with two agents. The input one, *agent_in*, is composed by a sequencer that manages the generation of input sequences, which are converted into electrical digital signals by the driver; in this way they can be correctly interpreted by the DUT.

The results generated by the DUT are handled by an output agent, *agent_out*, which is also including a specific driver to accomplish to opposite conversion.
Moreover, units called monitors are present in both agents. They are able to provide data to external blocks.

The reference model is meant to execute the computation on the same data generated by the sequencer, so that the comparison can be performed between the expected result and the actual one provided by the DUT: a comparator block is employed for this task.

It is important to remark that the communication between blocks is handled using an handshake protocol, so that operations are correctly synchronized.

Adder

A generic System Verilog adder description has been provided and included in the DUT definition, with the purpose of testing its functionality.

QuestaSim simulator has been employed to run the developed testbench from the top-level entity. All the useful results are detailed in the log transcript file: it reports data from both the DUT and the reference model, keeping track of the amount of Match and Mismatch cases.

From the transcript file related to this first analysis, named *transcript_adder_01_original*, a correct behavior has been noticed since no mismatches occured. Indeed, it is possible to observe that, for each test sequence, the output of the reference model is equal to the result provided by the adder under test. The report summary is shown below.

```
# ---- 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
```

2.1 Parallelism variation

The next request consists in changing the adder parallelism: it has been decided to halve the original 32-bits parallelism, obtaining a 16-bits wide architecture. Thanks to the flexibility of the test environment, it is possible to accomplish the verification of the new adder model by simply applying modifications to the generic data parallelism involved in the testbench.

In particular, the DUT interface has been modified to accomodate the new parallelism. Moreover, both the input and output packets, involved in the sequences definition, have been adapted too. These simple changes listed below are able to ensure a full compatibility of the testbench with the new model under test.

```
// duf_if.sv modifications
logic [31:0] A, B;        -->     logic [15:0] A, B;
logic [31:0] data;        -->     logic [15:0] data;

// packet_in.sv modifications
rand integer A;           -->     rand bit [15:0] A;
rand integer B;           -->     rand bit [15:0] B;

// packet_out.sv modifications
integer data;    -->      bit [15:0] data;
```

Observing the new generated transcript file, named *transcript_adder_02_changing_parallelism*, no mismatches are reported. The report summary is shown below.

```
# ---- 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
```

2.2 Inputs constraints

The use of System Verilog allows to add some constraints on the generated input sequences, specifying them in the *packet_in* definition.
As a first approach, two specific functions have been declared with the purpose to generate the adder inputs within certain ranges, as described below.

$$100 < A < 1000$$
$$B < 10 \cdot A$$

In particular, the first condition has been obtained using the *inside* operator, associated to the required range.

```
constraint A_distribution {
                A inside {[100:1000]};
        }

constraint B_distribution {
                B < 10*A;
        }
```

The testbench has been launched again and the correct behavior has been verified. Moreover, from the generated transcript file, it is possible to observe how the inputs are included in the desired ranges.

For the sake of simplicity, only a couple of examples are reported.

```
# adder: input A =   935, input B =  5449, output OUT =  6384
# adder: input A = 0000001110100111, input B = 0001010101001001, \
                  output OUT = 0001100011110000
# refmod: input A =   935, input B =  5449, output OUT =  6384
# refmod: input A = 0000001110100111, input B = 0001010101001001, \
                  output OUT = 0001100011110000
# UVM_INFO @ 375: uvm_test_top.env_h.comp [Comparator Match]

# adder: input A =   405, input B =    35, output OUT =   440
# adder: input A = 0000000110010101, input B = 0000000000100011, \
                  output OUT = 0000000110111000
# refmod: input A =   405, input B =    35, output OUT =   440
# refmod: input A = 0000000110010101, input B = 0000000000100011, \
                  output OUT = 0000000110111000
# UVM_INFO @ 405: uvm_test_top.env_h.comp [Comparator Match]
```

2.3 Wrong reference model

To analyze the testbench behavior in case of wrong comparisons, the reference model has been modified. It has been forced to perform a different operation with respect to the DUT: in particular, the addition between the two input operands has been replaced by a subtraction.

```
tr_out.data = tr_in.A + tr_in.B   -->   tr_out.data = tr_in.A - tr_in.B
```

For this reason, only mismatch conditions are expected.

```
# --- 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
```

On the summary reported above, it is possible to notice how all the tested sequences lead to a mismatch condition.
In the transcript file, a warning occurs for each encountered mismatch. In addition to that, an error is reported in the summary, meaning that the verification procedure failed, since at least one mismatch has been found.

2.4 Additional verification: overflow condition

A further analysis has been performed on the given adder, in order to verify the correct behavior in case of overflow condition.

To test the described scenario, several constraints have been applied to the provided inputs. In particular, since they are expressed on 32 bits, the following boundaries have been imposed.

```
constraint A_distribution {
        A > 2**31;
}

constraint B_distribution {
        B > 2**31;
}
```

The adder manages correctly the tested condition, as shown in the couple of examples and transcript summary reported below.

```
// Adder overflow examples
# adder: input A = 2682128720, input B = 3715728878, \
        output OUT = 2102890302
# adder: input A = 10011111110111100000100101010000, \
        input B = 11011101011110011000010111101110, \
        output OUT = 01111101010101111000111100111110
# refmod: input A = 2682128720, input B = 3715728878, \
        output OUT = 2102890302
# refmod: input A = 10011111110111100000100101010000, \
        input B = 11011101011110011000010111101110, \
        output OUT = 01111101010101111000111100111110
# UVM_INFO @ 165: uvm_test_top.env_h.comp [Comparator Match]
#
#
# adder: input A = 2287896343, input B = 3259091957, \
        output OUT = 1252021004
# adder: input A = 10001000010111101000011100010111, \
        input B = 11000010010000011100101111110101, \
        output OUT = 01001010101000000101001100001100
# refmod: input A = 2287896343, input B = 3259091957, \
        output OUT = 1252021004
# refmod: input A = 10001000010111101000011100010111, \
        input B = 11000010010000011100101111110101, \
        output OUT = 01001010101000000101001100001100
# UVM_INFO @ 195: uvm_test_top.env_h.comp [Comparator Match]


// Summary
# ---- 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
```

MBE integer multiplier

It is possible to rely on the testbench flexibility to adapt the environment to a different device under test. In particular, the MBE multiplier developed in laboratory 2 has been considered for this purpose.

In order to properly set the environment, several modifications have been applied. First of all, as shown in the following code, it has been necessary to create an instance of the MBE multiplier inside the *DUT.sv* file, which also manages the interface with other components through specific signals.

```
Significand_Multiplier \
MBE_under_test(.A(in_inter.A),.B(in_inter.B),.Z(out_inter.data));
```

Afterwards, the sequence generation and the DUT interfaces have been adapted to guarantee the correct parallelism of 32 bits in input and 64 bits in output, as required by the MBE.
Finally, the reference model has been modified to perform multiplication between the two input operands.

Since the MBE multiplier is described with several VHDL files, it cannot be included directly inside the System Verilog testbench. Therefore, all the files related to the VHDL description have been compiled first, followed by the testbench ones.

The developed multiplier presents a correct behavior, as reported in the given transcript file. The related summary is shown below.

```
# --- 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
```

3.1 Additional verification: large input values

The behavior of the MBE multiplier has been tested in presence of large input values, considering the parallelism of 32 bits. The following constraints have been applied to the two input operands to reach the wanted condition.

```
constraint A_distribution {
        A > 2**31;
}

constraint B_distribution {
        B > 2**31;
}
```

Also in this case, no mismatches are reported in the summary included in the transcript file. Moreover, it is possible to notice how the whole 64 bits of the results are exploited to correctly represent the output data.

```
// Result example
# MBE: input A = 2856251920, input B = 3622420489, \
        output Z = 10346545476753588880
# MBE: input A = 10101010001111101111001000010000, \
        input B = 11010111111010011100000000001001, \
        output Z = 1000111110010110010100000100101011110010010100101000001010010000
# refmod: input A = 2856251920, input B = 3622420489, \
        output Z = 10346545476753588880
# refmod: input A = 10101010001111101111001000010000, \
        input B = 11010111111010011100000000001001, \
        output Z = 1000111110010110010100000100101011110010010100101000001010010000
# UVM_INFO @ 2925: uvm_test_top.env_h.comp [Comparator Match]


// Summary
# ---- 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
```

Single precision floating point multiplier

The MBE multiplier tested in the previous section is included in a floating point architecture to multiply the significand fields of the two input data. It is requested to verify the behavior of this single precision FP multiplier.

Several modifications have been applied to the test environment: first of all, the involved parallelism for both inputs and outputs has been set to a fixed value of 32 bits, following the same approach described in the previous sections.

Since the multiplier works with a floating point representation, also the reference model is supposed to work in the same way. Therefore, its System Verilog description has been adapted by adding specific conversion directives from bits to short real number and viceversa.

```
// binary multiplication
tr_out.data = tr_in.A * tr_in.B;

// FP multiplication
tr_out.data = $shortrealtobits($bitstoshortreal(tr_in.A)*$bitstoshortreal(tr_in.B));
```

Another adjustement is related to the display directive: it has been set to express the output results, as well as the input data, in a floating point format. Therefore, the *%d* specifier has been replaced with *%.10f*.

An instance of the floating point multiplier has been declared in the *DUT.sv* file. In this specific case, it is important to highlight the pipeline organization of the architecture, meaning that the result is not provided in a fully combinational way as it was for the previously tested architectures.

As a consequence, it is necessary to define a clock signal for the multiplier and consider the additional latency required to fully process a sample.
The following figure summarizes the floating point organization of the multiplier.

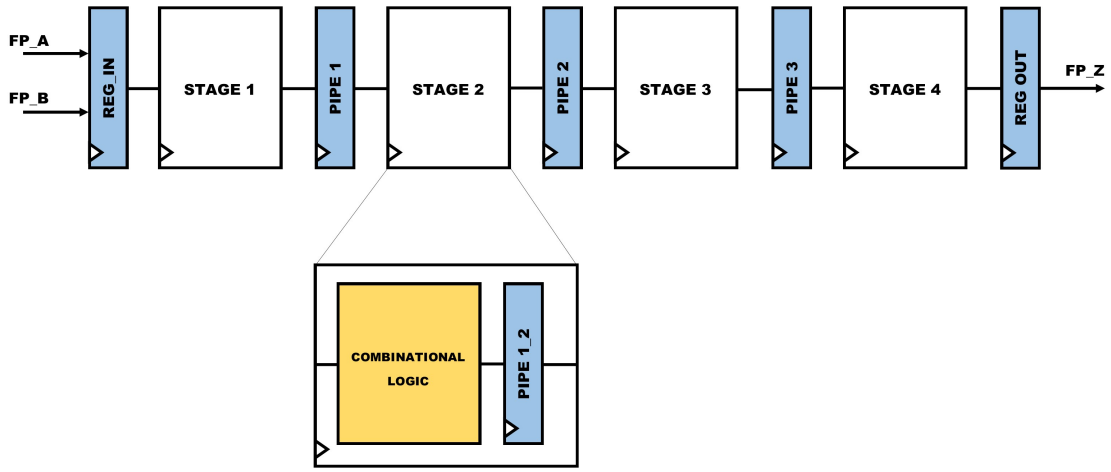

Figure 4.1 - Floating point multiplier pipeline

As visible from the figure above, it is necessary to wait for a latency equal to six clock cycles to ensure the output result validity.

In particular, the pipeline behavior has been emulated in the WAIT state of the *DUT.sv* file: a cascade of dummy signals has been implemented to ensure the correct synchronization between the tested architecture and the reference model results for the comparison.

In addition to that, in order to guarantee the coherence between the input and the output values reported in the transcript file, the former are delayed thanks to several declared signals.

It has been chosen to test each sample considering the full latency of the FP multiplier.
The dummy pipeline propagation in the WAIT state is shown below: it is remarkable how the output results validity is ensured in the sixth stage of pipe.

```
WAIT: begin
        if(in_inter.valid) begin
                in_inter.ready <= 0;
                pipe_0 <= 0;

                pipe_1 <= pipe_0;
                A1 <= in_inter.A;
                B1 <= in_inter.B;

                pipe_2 <= pipe_1;
                A2 <= A1;
                B2 <= B1;

                pipe_3 <= pipe_2;
                A3 <= A2;
                B3 <= B2;

                pipe_4 <= pipe_3;
                A4 <= A3;
                B4 <= B3;

                pipe_5 <= pipe_4;
                A5 <= A4;
                B5 <= B4;

                pipe_6 <= pipe_5;
                A6 <= A5;
                B6 <= B5;

                if(pipe_5) begin
                        $display("adder: input FP_A = %.10f, \
                        input FP_B = %.10f, output FP_Z = %.10f",$bitstoshortreal(A6), \
                        $bitstoshortreal(B6),$bitstoshortreal(out_inter.data));
                        $display("adder: input FP_A = %b, \
                        input FP_B = %b, output FP_Z = %b",A6,B6,out_inter.data);

                        state <= SEND;
                        out_inter.valid <= 1;
                end

        end
end


SEND: begin
          if(out_inter.ready) begin
                 out_inter.valid <= 0;
                 in_inter.ready <= 1;
                  state <= WAIT;
                 pipe_0 <= 1;
           end
end
```

Considering the case of random inputs generation, some mismatches are present in the transcript file at the end of the verification process. It is possible to notice that those mismatches are related to a denormal form representation of some floating point numbers.

In order to overcome this issue, it is possible to set constraints on the input generation with the purpose of guaranteeing the use of normal form numbers for both inputs and outputs. The following code has been included in the *packet_in.sv* file.

```
rand bit [31:0] A;
rand bit [31:0] B;

        constraint A_exp_distribution {
                A[30:23] inside {[64:190]};
        }

        constraint B_exp_distribution {
                B[30:23] inside {[64:190]};
        }


        constraint A_mantissa_distribution {
                A[22:0] > 2**22;
        }

        constraint B_mantissa_distribution {
                B[22:0] > 2**22;
        }
```

As visible in the reported code, it is necessary to consider in a separate way the different fields of a floating point number: sign, significand and exponent.
In order to ensure a normal form for the inputs, it is necessary to apply the following conditions:

$$0 < \exp_A < 255$$
$$0 < \exp_B < 255$$

The same boundaries need to be applied on the output exponent.

$$\exp_{out} = \exp_A + \exp_B - 127$$
$$0 < \exp_{out} < 255$$

Considering all the conditions listed above, the following ranges have been obtained.

$$64 < \exp_A, \exp_B < 190$$

Once again the testbench has been launched and no mismatches occured, as shown in the following report, which includes both a summary and a couple of results provided in the transcript file.

```
// Sequences examples
# refmod: input A = -14.3836116791, \
  input B = -7.3881936073, output OUT = 106.2689056396
# refmod: input A = 11000001011001100010001101000110, \
  input B = 11000000111011000110110000010101, \
  output OUT = 01000010110101001000100110101110
# adder: input FP_A = -14.3836116791, input \
  FP_B = -7.3881936073, output FP_Z = 106.2689056396
# adder: input FP_A = 11000001011001100010001101000110, \
  input FP_B = 11000000111011000110110000010101, output \
  FP_Z = 01000010110101001000100110101110
# UVM_INFO @ 5695: uvm_test_top.env_h.comp [Comparator Match]
#
#
# refmod: input A = 7.8595175743, \
  input B = 3.1247549057, output OUT = 24.5590667725
# refmod: input A = 01000000111110111000000100101011, \
  input B = 01000000010001111111101111111100, \
  output OUT = 01000001110001000111100011111000
# adder: input FP_A = 7.8595175743, \
  input FP_B = 3.1247549057, output FP_Z = 24.5590667725
# adder: input FP_A = 01000000111110111000000100101011, \
  input FP_B = 01000000010001111111101111111100, output \
  FP_Z = 01000001110001000111100011111000
# UVM_INFO @ 5755: uvm_test_top.env_h.comp [Comparator Match]


// Summary
# ---- 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
```

Moreover, as visible from the transcript file, denormal form numbers are not present as expected.