Politecnico di Torino

Department of Electronics and Telecommunications

Multidisciplinary Workshop
SoC Verification Strategies

Preparatory Workshop on SoC Verification: Technical Report

Alfredo Paolino: s295152@studenti.polito.it
Vincenzo Petrolo: s295043@studenti.polito.it
Roman Ali: s300976@studenti.polito.it

December 3, 2022

Contents

Chapter 1

Step 1: SystemVerilog learning

1.1 Introduction

In this chapter, step 1 of *"Preparatory Workshop on SoC Verification"* will be described.
The goal of this first step is to learn how to effectively use SystemVerilog testbenches in order to test a given component. In order to do so, some basic components designed in VHDL for the *Microelectronic Systems* course have been tested.
Concerning combinational components, we selected the following:

- 2:1 Multiplexer (MUX)
- Ripple Carry Adder (RCA)
- Arithmetic-Logic Unit (ALU)

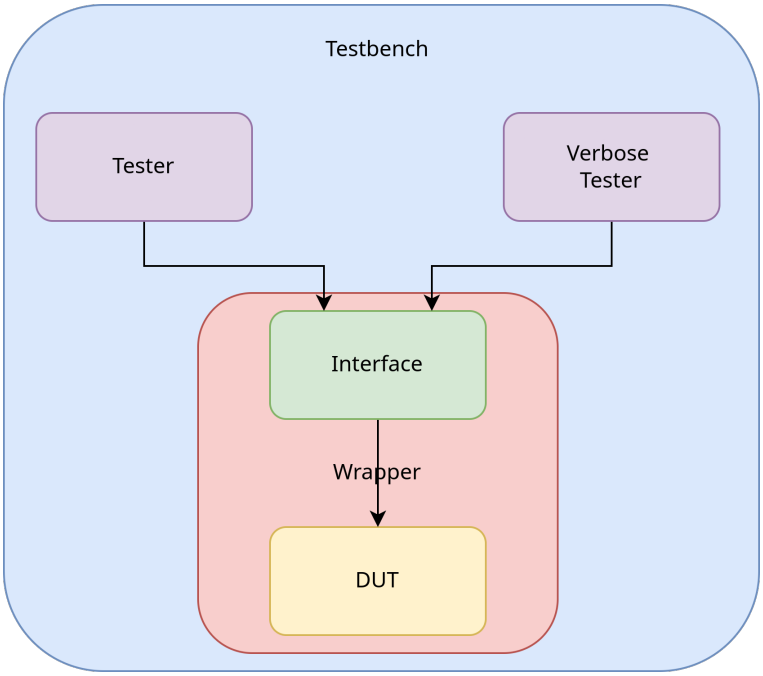
While for what concerns sequential components, we selected the following:

- Register
- Accumulator

The first step was becoming familiar with the SystemVerilog syntax and environment. In order to do so, each group member took some time to look for useful material, read the provided guide and try to catch all the essential concepts from the provided Testbench (TB) example.
When ready to test the above-mentioned components, we proceeded with the environment setting.
All the simulations have been run on the *led-x3850-2.polito.it* server where *Mentor Graphics QuestaSim 10.7C* is available. However, to be able to work in parallel and have a more efficient workspace, each group member set his own environment on VSCode connecting to the server via SSH.
The last thing to mention is how the work on Git has been organized. Even though this first step was not so complex, we still decided to adopt a branch-merging strategy in order to keep the master branch always clean and operative.

1.2 Background

The goal of this first step is to use SystemVerilog to write testbenches for the basic components. By using SystemVerilog we can write a TB composed of various blocks, creating a modular structure so that the re-usability of the TB increases. The main components are shown in the picture below.



To design our TB we followed the guidelines given by the ALU example, exploiting SystemVerilog's OOP features. Used components are:

- Interface: Defines an interface for the component, it can also implement some tasks so that it is easy to perform actions on the signals (e.g. reset sequence for a register). This module helps in writing reusable components since it hides the actual signals' implementation thus avoiding the necessity to change the whole testbench. A further feature of the interface is the **modport**, which allows the definition of different signals' directions for the modules connected to the interface.
- Wrapper: Used to hide the connections between the interface and the component in the TB. Furthermore, it allows to embed VHDL-based DUTs into the System Verilog environments.
- Assertions: Are used to define what the design should do in terms of mathematical expressions. (e.g. when the reset signal is high, the output of the register drops to 0)
- Tester: It is in charge of generating a set of randomized input signals to test the corner cases of the design. In order to do so, a set of constraints increase the probability that some set of signals appears with respect to others.
- Testbench: It includes all the previously cited components.

This new approach seemed complex at a first glance, but the provided example helped us a lot to understand every single component. With respect to classical simulation testing, carried out by checking the correctness of waveforms, the SystemVerilog approach is for sure faster, especially when it comes to testing large components with a lot of signals.

1.3 Design

In this section, we will introduce the Register-Transfer Level (RTL) design of each Device Under Test (DUT) and carefully describe the testbench we developed in SystemVerilog.

1.3.1 2:1 Multiplexer

The first component taken into account is a generic 2:1 MUX. The RTL description of a MUX is simple and straightforward. Thus, a 2:1 MUX is a perfect starting point to see and appreciate the strength of SystemVerilog.
The purpose of a MUX is to set the output equal to one of its inputs, depending on the value of a control signal.
The power of *QuestaSim* is that it accepts mixed designs, with some portions in VHDL and some others in SystemVerilog. It is important to remember that the testing of VHDL components with SystemVerilog TBs must be enabled by adding the parameter *-mixedsvvh* to the *vcom* command when compiling the component. What follows is then the VHDL implementation of the 2:1 MUX main process:

```
process (IN0, IN1, S)
    begin
        if (S = '0') then
            O <= IN0;
        else
            O <= IN1;
        end if;
    end process;
```

With the component ready we started writing the TB.
First of all, we wrote the interface. Due to the lack of a driver class (the tester itself is in charge of generating random stimuli and driving the input signals), a single modport has been created. This modport is used in the MUX wrapper in order to correctly set the direction of signals.
The second task of the interface is to generate and deliver the clock to the entire TB. In this specific case, the clock is not delivered to the DUT due to its combinational behavior.
Being the pivot point between the DUT and the TB, the interface is also used to assert the correct value of outputs coming from the DUT. What follows is the testing property with the corresponding assertion:

```
property p_result;
        @(negedge clk)
        case(s)
                0:      o == in0;
                1:      o == in1;
                default: o == 'h0;
        endcase
endproperty

a_result: assert property (p_result)
else begin
    err_num++;
    $error("%s", `PRINT_OP(in0, in1, s, o));
end
```

The outcome of the property assertion is used to increment the number of errors in the case one occurred.
Connected to the interface, we built the tester. Its task is dual: to generate and randomize input stimuli and to run the actual test after resetting the DUT.
In order to easily generate random stimuli we created a structure with the 3 inputs, and declared one instance of it as *rand*. The following code shows how the randomization is executed:

```
typedef struct packed {
    logic[NBIT-1:0] in0;
    logic[NBIT-1:0] in1;
    logic           s;
} op_t;

rand op_t mux21_op;

constraint in0_in1_c {
        mux21_op.in0 dist {
                0               :=10,
                (1<<NBIT)-1     :=10,
                1<<(NBIT-1)-1   :=10,
                [1:(1<<NBIT)-2] :=1
        };
        mux21_op.in1 dist {
                0               :=10,
                (1<<NBIT)-1     :=10,
                1<<(NBIT-1)-1   :=10,
                [1:(1<<NBIT)-2] :=1
        };
};

...

function void rand_mux21_op();
    assert (this.randomize())   // check the method's return value
    else   $error("ERROR while calling 'randomize()' method");

    taif.in0   = mux21_op.in0;
    taif.in1   = mux21_op.in1;
    taif.s     = mux21_op.s;

    acov.cov_sample();
endfunction
```

It can be seen from this piece of code that at the end of the randomization the coverage is updated; this is a useful SystemVerilog feature used to check if the TB tested the component in a comprehensive way.
In order to do so, covergroups separated by coverpoints have been set. Coverpoints for *in0* and *in1* was not strictly necessary, but they have been added to try this SystemVerilog feature. What follows is the setting of coverpoints:

```
covergroup mux21_cg;

    op_cp: coverpoint aif.s {
        bins zero       = {0};
        bins one        = {1};
    }

    in0_cp: coverpoint aif.in0 {
        bins corner[]   = {0, (1<<NBIT)-1, (1<<(NBIT-1))-1};
        bins others     = default;
    }
    in1_cp: coverpoint aif.in1 {
        bins corner[]   = {0, (1<<NBIT)-1, (1<<(NBIT-1))-1};
        bins others     = default;
    }
endgroup: mux21_cg
```

The last block of the TB is the top level entity, where all the sub-blocks are declared and instantiated. In this block, the test is launched and the outcome is shown.

1.3.2 Ripple Carry Adder

The RCA is a component that implements the addition between two operands. The description is performed in a generic and behavioral way using VHDL. For the complete explanation of TB's sub-blocks, you can refer to section 1.3.1 where a detailed description of each one of them is given. We will analyze here only the main differences and the key points.
For what concerns the TB, we re-used the one for the ALU but changed the interface, the properties, and the input signals' constraints.
In order to check the coverage of inputs for the design, we used covergroups and coverpoints. Overall, we chose 5 coverpoints of which 3 test the overflow case, and one shows the ripple of the carry from the Least Significant Bit (LSB) to the Most Significant Bit (MSB). Since a coverpoint should test only one signal at a time and we wanted to check a condition involving two signals, we merged them using the concatenation operator and defined also the corner case using the repetition operator combined with the concatenation to generate a signal that resembled the merged signal for the coverpoint:

```
/*Overflow as above but with carry in and in2*/
overflow3 : coverpoint {rif.in2, rif.c_in} {
    bins corner[] = {   {{DWIDTH-1{1'b1}},1'b1}};
    bins others = default;
}
```

It is worth mentioning that the interface makes use of a self-generated clock signal so that we have a reference to test the properties of the design. In this case, the property of the design is simply checking that the addition of the two input operands corresponds with the output provided by the RCA:

```
property p_addition_result;
    @(posedge clk) {c_out,res} == in1 + in2 + c_in;
endproperty
```

1.3.3 Arithmetic-Logic Unit

The ALU is a combinational component used to perform a wide range of operations on two input operands, depending on the value of a function signal. We described a generic ALU in VHDL with a behavioral description style.
For the complete explanation of TB's sub-blocks, refer again to section 1.3.1.
The implemented operations are:

- ADD
- SUB
- MULT
- AND
- OR
- XOR
- LSL
- LSR
- RL
- RR

In order to have a complete test, we set a coverpoint for each operation. In this way, we can ensure that the ALU always performs the correct operation and gives the correct result.
On top of that other coverpoints have been set for the two input operands too. The purpose is to have a not uniform probability distribution for the operands, in this way we can test corners and more critical cases. What follows is the implementation of coverpoints and probability distribution:

```
op_cp: coverpoint aif.alu_op iff (aif.rst_n) {
    bins add        = {add};
    bins sub        = {sub};
    bins mult       = {mult};
    bins bitand     = {bitand};
    bins bitor      = {bitor};
    bins bitxor     = {bitxor};
    bins funclsl    = {funclsl};
    bins funclsr    = {funclsr};
    bins funcrl     = {funcrl};
    bins funcrr     = {funcrr};
}

a_cp: coverpoint aif.alu_a iff (aif.rst_n) {
    bins corner[]   = {0, (1<<DWIDTH)-1, (1<<(DWIDTH-1))-1};
    bins others     = default;
}

b_cp: coverpoint aif.alu_b iff (aif.rst_n) {
    bins corner[]   = {0, (1<<DWIDTH)-1, (1<<(DWIDTH-1))-1};
    bins others     = default;
}

...
```

```
constraint ab_dist_c {
        alu_op.a dist {
                0                   :=10,
                (1<<DWIDTH)-1       :=10,
                1<<(DWIDTH-1)-1     :=10,
                [1:(1<<DWIDTH)-2]   :=1
        };
        alu_op.b dist {
                0                   :=10,
                (1<<DWIDTH)-1       :=10,
                1<<(DWIDTH-1)-1     :=10,
                [1:(1<<DWIDTH)-2]   :=1
        };
};
```

Properties are instead used to assert the correct component's behavior. In this case, the only property used checks that the output signal is equal to the corresponding operation between the two input operands. What follows is the property used for the ALU:

```
property p_result;
    logic [DWIDTH-1:0] res;
    @(negedge clk)
    case (alu_op)
        /* Arithemtic operations */
        add:        alu_res == (alu_a + alu_b);
        sub:        alu_res == (alu_a - alu_b);
        mult:       alu_res == (alu_a[MULT_WIDTH-1:0]) * (alu_b[MULT_WIDTH-1:0]);

        /* Bitwise operations */
        bitand:     alu_res == (alu_a & alu_b);
        bitor:      alu_res == (alu_a | alu_b);
        bitxor:     alu_res == (alu_a ^ alu_b);

        /* Logical shift operations */
        funclsl:    alu_res == (alu_a << alu_b);
        funclsr:    alu_res == (alu_a >> alu_b);

        /* Rotate operations */
        funcrl:     alu_res == (alu_a << alu_b[SHIFT_WIDTH-1:0])
                               | alu_a >> (DWIDTH - alu_b[SHIFT_WIDTH-1:0]);
        funcrr:     alu_res == (alu_a >> alu_b[SHIFT_WIDTH-1:0])
                               | alu_a << (DWIDTH - alu_b[SHIFT_WIDTH-1:0]);

        /* With other operations, return 0 */
        default:    alu_res == 'h0;
    endcase
endproperty
```

1.3.4 Register

We further tested a sequential module which is the register. The design is carried out using VHDL and a generic-behavioral description. The register has an asynchronous reset signal. As in the other designs, we described the interface, constraints, and properties of the signals. For the complete explanation of TB's sub-blocks, refer again to section 1.3.1.
There is only one coverpoint for the register which checks if the reset signal covers the values 0 and 1:

```
covergroup reg_cg;
   rst_cp : coverpoint rif.rst {
    bins rst_values = {0,1};
   }
endgroup : reg_cg;
```

The properties describe the behavior of the design. One is used when reset signal is 1:

```
property reset_p;
    @(posedge clk or posedge rst) rst |-> Q == '0;
endproperty
```

And another one used to check that data loaded at a given clock cycle is on the output of the register after 1 clock cycle:

```
property data_loading_p;
    @(posedge clk) disable iff(rst) ##1 Q == past(D);
endproperty
```

1.3.5 Accumulator

Finally, we tested the accumulator. This design is also carried out in VHDL with a behavioral description style. The accumulator has an asynchronous reset. For the complete explanation of TB's sub-blocks i.e. the interface, constraints, and properties of the signals, refer again to section 1.3.1.
There are 3 coverpoints for the accumulator. One for the MUX select line and the other two for the 2 input signals.

```
op_cp: coverpoint aif.acu_s iff (aif.rst_n) {
    bins zero       = {0};
    bins one        = {1};
}

a_cp: coverpoint aif.acu_a iff (aif.rst_n) {
    bins corner[]   = {0, (1<<NBIT)-1, (1<<(NBIT-1))-1};
    bins others     = default;
}

b_cp: coverpoint aif.acu_b iff (aif.rst_n) {
   bins corner[]   = {0, (1<<NBIT)-1, (1<<(NBIT-1))-1};
   bins others     = default;
```

There are two asserted properties that check the correctness of the VHDL design. The first property, p-reset, is used to check for a result equal to zero right after the reset signal's negative edge. The second property, p-results, checks for the correct result one clock cycle after the inputs.

```
property p-reset;
    @(posedge clk or negedge rst_n)
    !rst_n |-> acu_res == 'h0;
endproperty
a_reset: assert property (p_reset);


property p-result;
    @(posedge clk) disable iff (!rst_n)
        case (acu_s)
        0:        ##1 acu_res == $past(acu_a + acu_b);
        1:        ##1 acu_res == $past(acu_res + acu_a);
        default:  ##1 acu_res == 'h0;
    endcase

endproperty
```

1.4 Results

1.4.1 2:1 Multiplexer Simulation and Results

The first step needed to start the test is to declare and instantiate the interface, the wrapper, and the interface in the top level entity. With this done, the simulation can start.
At each and every clock cycle we generate a new set of inputs depending on the probability distribution of inputs we defined before. The inputs are propagated through the interface to the wrapper and finally to the DUT. After that, we sample the DUT inputs to update the overall coverage of corner cases. This is all repeated in a loop of a given number of cycles. When the test is over we get the outcome with the corresponding coverage. What follows is the piece of code that carries out the test:

```
mux21_if #(NBIT)    aif();

mux21_wrap #(NBIT)  aw(aif.mux21_port);

mux21_tester #(NBIT) tst;

...

$display("\nTEST #1 - Launching MUX21 quiet test...");
tst.run_test(num_cycles);
$display("TEST #1 - Test completed!");
$display("TEST #1 - Functional coverage: %.2f%%", tst.get_cov());
```

We can also launch more than one test, with different features. For example, we implemented another tester, called verbose tester, which prints on the screen each tested operation. The advantage is twofold: first of all, we have a more detailed test in terms of received information, and then further increase the coverage.
For the MUX it is important to make tests with the selection signal both at 0 and at 1, then some other coverpoints have been added to get hands in covergroups and distributions. By removing these coverpoints we can reach 100% coverage with very few tests.

1.4.2 Ripple Carry Adder Simulation and Results

The simulation is carried out in the same way as the MUX. Since the coverage space is quite large in this case, it is not enough to run the simulation with the default number of cycles (i.e. 10) but instead by using 100 cycles it is more likely to obtain a 100% coverage of the coverpoints. Furthermore, the simulation shows no violation of the properties described in the previous section.

1.4.3 Arithmetic-Logic Unit Simulation and Results

The simulation is similar to the RCA, but in this case we have to repeat the tests for each possible operation. This means that the coverage space is much wider, so we had to increase even more the number of tests to be able to reach 100% coverage.

1.4.4 Register Simulation and Results

The simulation for the register component is straightforward since here the coverpoint is only related to the reset signal. In fact, it is trivial to obtain 100% coverage using a small amount of simulation time. Furthermore, during the simulation all the assertions for the properties previously described are correct.

1.4.5 Accumulator Simulation and Results

Simulation is carried out in the same way as the previous modules. Since the coverage point is large for the adder inputs, a significant number of clock cycles are required to reach 100% coverage. In this design 100% coverage was reached when the TB was run for 100 clock cycles.

1.5 Conclusion

In conclusion, thanks to this first Workshop step, we learned:

- SystemVerilog features (OOP-feats, properties, assertions, coverpoints, distributions, etc...)
- How to test small components using SystemVerilog
- How easy and fast is it to test a component (even a large one) through SystemVerilog instead of analyzing waveforms
- How to work in a team and in parallel by using SSH, Git, and LaTeX

We are now ready to start with step 2.

Chapter 2

Step 2: UVM learning

2.1 Introduction

In this chapter, step 2 of *"Preparatory Workshop on SoC Verification"* will be described.
The goal of this second step is to learn the Universal Verification Methodology (UVM) and how to efficiently write Testbenches using it. Some more advanced components designed in VHDL for the *Microelectronics Systems* course have been tested using UVM. Components to be tested are again combinational as well as sequential ones.
For the combinational component, we selected the following:

- P4 Adder

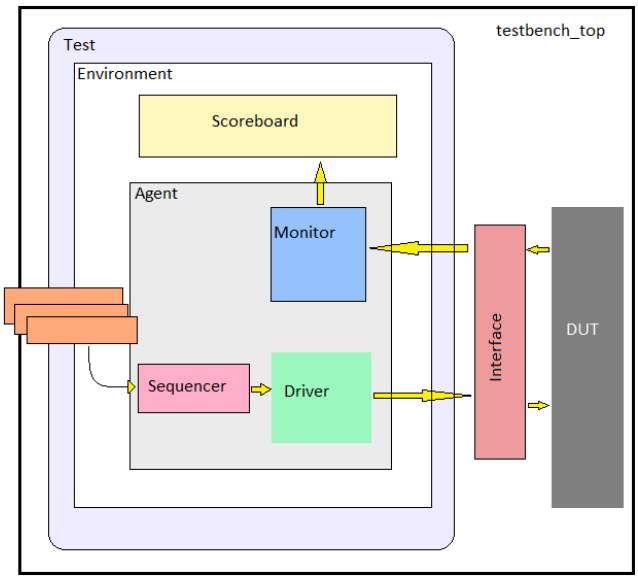
For the sequential components, we selected the following:

- Register File with Windowing
- FSM-based Control Unit

As we have already completed step 1, the platform/environment to write the TBs and test the components has already been established.
We used *led-x3850-2.polito.it* server where *Mentor Graphics QuestaSim 10.7C* is available for the simulations. Similarly, we used the same environment on VSCode connecting to the server via SSH. The last thing to mention is how the work on Git has been organized. Again a branch-merging strategy was adopted in order to keep the master branch always clean and operative.

2.2 Background

The goal of this second step is to use Universal Verification Methodology (UVM) for some components designed in VHDL. By using UVM we can write a modular TB with various blocks. Some of these blocks can be reused for other designs increasing the re-usability of the TB. UVM is a library that exploits the basic features of SystemVerilog to write TBs for more complex designs. The block diagram of the components and how they interface with each other is shown in the picture below.



Now we will discuss each component in detail:

- Testbench Top: This is the TB module and it instantiates the interface and the DUT and connects it to the verification components. On top of that, it generates the clock and reset stimuli and starts the UVM_Test by calling the static method *run_test()*;
- Test: This component creates the environment and sequence objects by calling *create()*. Then, it starts the execution of sequences by calling the *start()* method. The test defines the test scenario for the TB, in this way the same environment can be re-used for different test cases.
- Environment: This component contains reusable verification modules i.e. agents and scoreboards. It is better to have agents and scoreboards inside the environment to increase the re-usability of the TB.

- Agent: This component typically instantiates a driver, a monitor, and a sequencer. An agent can be configured either as active or passive. In the former case, all three components are instantiated and the agent generates stimuli driving the DUT, while in the latter only the monitor is instantiated and the agent simply samples DUT signals without driving them.
- Driver: Transaction-level objects are obtained from the sequencer through a TLM port and the driver drives them to the DUT via an interface handle.
- Sequencer: The sequencer controls the flow of request and response sequence items between sequences and the driver.
- Monitor: This component is a passive entity responsible for capturing signal activity from DUT virtual interface, and converts the signal level activity to the transaction level data that can be transferred to the scoreboard or other components.
- Scoreboard: This component contains checkers which verify the functionality of the DUT. It compares the DUT outputs with the expected values (golden reference values or values generated from a reference model). Monitor and scoreboard will communicate via TLM ports and exports.
- Sequence: A sequence generates a series of sequence items and sends it to the driver via the sequencer. A sequence is parameterized with the type of sequence item, this defines the type of the item sequence that will send/receive to/from the driver.
- Sequence_item: The sequence item consist of data fields required for generating the stimuli. In order to generate the stimuli, the sequence items are randomized in sequences. Therefore data properties in sequence items should generally be declared as rand and can have constraints defined.
- Interface: An interface is a bundle of signals or nets through which a TB communicates with a DUT. A virtual interface is a variable that represents an interface instance. In our case, we use an interface so that the components inside the agent can communicate with the DUT.

2.3 Design

2.3.1 P4-Adder

The P4-Adder is a special type of Carry-Lookahead adder which uses a regular tree of Propagate (P) and Propagate&Generate (PG) blocks.
The module provides a rather simple interface:

- A: First operand;
- B: Second operand;
- Cin: If 0 the module performs an addition, if 1 it performs a subtraction (A-B).

With this component ready, we started creating the backbone structure of our UVM environment.

Sequence item

The sequence item is the starting point of our UVM environment journey.
We use this component to describe the transactions, and we exploit constraints, to force the operands towards values that trigger overflow scenarios.

```
class packet #(parameter NBIT = 32, NBIT_PER_BLOCK = 4) extends uvm_sequence_item ;

    rand logic [NBIT-1:0] A;
    rand logic [NBIT-1:0] B;
    rand logic Cin;
    rand logic Cout;
    rand logic [NBIT-1:0] S;
    ...
    constraint input_distributions {
        A dist {
            1                 :=1000,
            (1<<NBIT)-1       :=1000, //input is all 1s
            (1<<(NBIT-1))-1   :=1000, //input is 0111..
            [1:(1<<NBIT)-2]   :=1 //other inputs
        };
    ...
```

Sequencer

In order to generate transactions, we need a Sequencer. We create it without further customizing it.

```
class sequencer extends uvm_sequencer #(packet);
    `uvm_component_utils(sequencer)


  function new(string name = "sequencer", uvm_component parent = null);
    super.new(name, parent);
  endfunction //new()
```

Driver

Our implementation of the Driver is simple. We use the Sequencer to get new transactions and wiggle the pin of the DUT accordingly by means of the interface. In addition, we create the coverage module which is in charge of monitoring the data and reporting the final bins coverage.

```
class driver extends uvm_driver #(packet);
    `uvm_component_utils(driver)
    virtual dut_if vif;
    protected static adder_coverage #(parameters::NBIT, parameters::NBIT_PER_BLOCK) cov;

    ...
    virtual task run_phase(uvm_phase phase);
        // Enable coverage sampling
        cov.cov_start();
        forever begin
            @(posedge vif.clk)
            super.run_phase(phase);
            //use the seq_item_port handle to get the next transaction
            seq_item_port.get_next_item(req);
            //Drive the interface
            vif.A = req.A;
            vif.B = req.B;
            vif.Cin = req.Cin;
            // Sample the data on the interface
            cov.cov_sample();

            //Next request
            seq_item_port.item_done();
        end
```

Monitor

Since we are dealing with a combinational circuit our monitor is very basic.
At each and every rising edge of the clock, it copies the value on the interface into a new transaction. After that, it writes packets to be sent to Scoreboard on its item collector port.

```
class monitor extends uvm_monitor;
    `uvm_component_utils(monitor)
    ...
   task run_phase(uvm_phase phase);
        // Wait for the positive edge of the clock
        //Maybe using the clock is not necessary
        forever begin
            wait(!vif.rst);
            @(posedge vif.clk);
            // Sample on the rising edge of the clock
            tr.A = vif.A;
            tr.B = vif.B;
            tr.Cin = vif.Cin;

            // sample the output on the rising edge of clk
            //`uvm_info(get_name(), $sformatf("Cout: %x", vif.Cout), UVM_NONE)

            tr.Cout = vif.Cout;
            tr.S = vif.S;

            // Write the sampled item to the scoreboard
            item_collected_port.write(tr);
        end
```

Agent

The role of the Agent component is simply to instantiate all the inner UVM components and then connect them.

```
class agent extends uvm_agent;
    `uvm_component_utils(agent)
    ...
        virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        sqr = sequencer::type_id::create("sqr", this);
        drv = driver::type_id::create("drv", this);
        mon = monitor::type_id::create("mon", this);
    endfunction

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);

        // Perform the connection to make monitor communicate with scoreboard
        mon.item_collected_port.connect(item_collected_port);
        // Connect the sequencer to the driver
        drv.seq_item_port.connect(sqr.seq_item_export);
    endfunction: connect_phase
```

Scoreboard

The scoreboard is the keystone in the UVM environment. In fact, it receives captured transactions from the Monitor and compares them with a golden reference to catch unexpected behaviors of the DUT. In order to do that, we described also an Oracle object which is a high-level description of an adder.
The communication between the Scoreboard and the Monitor is done by means of an *analysis port*. Packets received through this port are pushed into a packets queue in order not to lose any single packet even in case of packets overcrowding.

```
class scoreboard extends uvm_scoreboard;
  `uvm_component_utils(scoreboard);

  uvm_analysis_imp #(packet, scoreboard) item_collect_export;
  // Define a dynamic array
  packet pkts[$];
  protected static adder_oracle oracle;

  function void write(packet p);
    pkts.push_back(p);
  endfunction
    ...
  task run_phase(uvm_phase phase);
    packet p;
    forever begin
      wait (pkts.size > 0);

      if (pkts.size > 0) begin
        p = pkts.pop_front();
        oracle.validate(p.A, p.B, p.Cin, p.S, p.Cout);
      end
    end
  endtask : run_phase
```

Environment

Once again, as in the Agent, the environment is required to instantiate and connect inner components among them.

```
class env extends uvm_env;
    `uvm_component_utils(env);
    ...
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        ag = agent::type_id::create("ag",this);
        sb = scoreboard::type_id::create("sb", this);
    endfunction: build_phase

    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);

        // connect the agent to the scoreboard inside the environment
        ag.mon.item_collected_port.connect(sb.item_collect_export);
    endfunction: connect_phase
```

Test

The test is the uppermost UVM component. It encloses all the UVM components inside. Thanks to this component we can decide how many tests to run through the UVM commandline processor.

```
class test extends uvm_test;
    `uvm_component_utils(test);
    ...
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        e = env::type_id::create("e", this);
        cmdline = uvm_cmdline_processor::get_inst();
    endfunction: build_phase

    task run_phase(uvm_phase phase);
        phase.raise_objection(this);
        `uvm_info(get_name(), "<run_phase> started, objection raised.", UVM_NONE)
        s = base_seq::type_id::create("s");

        if (cmdline.get_arg_value("+n_test=", N_TESTS) == 0) begin
            // Default to 10 tests
            N_TESTS="10";
        end

        repeat(N_TESTS.atoi()) begin
          #(parameters::CLK_PERIOD/2); s.start(e.ag.sqr);
        end
```

Testbench Top

As already said, this module simply instantiates all the components needed for the test and generates CLK and RST signals to drive the interface and the DUT.
It is worth mentioning why we needed in this case an explicit handle to the uvm_root top level component. The problem we faced was that right after the simulation with *Mentor Graphic's QuestaSim*, the test was suddenly ending. This was not the desired behavior, because we still had to go through the entire process of functional and code coverage.
We found the solution by setting $top.finish\_on\_completition = 0$. By doing this, the *.do* file was correctly executed up to the exit point.

```
module tb_top;
    logic clk,rst;

    dut_if #(parameters::NBIT, parameters::NBIT_PER_BLOCK) vif(rst, clk);

    // Instantiante the dut wrapper which performs connections
    dut_wrapper #(parameters::NBIT, parameters::NBIT_PER_BLOCK) dut(vif.input_pov);

    // Initialize the clock and reset signal
    // use the reset to start sampling after the first clock period
    // otherwise it will throw errors
    initial begin
        clk = 1'b1;
        rst = 1;
        // only at the second clock period, start monitoring
        #(2.5*parameters::CLK_PERIOD);
        rst = 0;
    end

    // Generate the clock
    always #parameters::CLK_PERIOD begin
        clk = ~clk;
    end

    initial begin
        uvm_config_db#(virtual dut_if)::set(uvm_root::get(), "*", "vif", vif);
    end

    initial begin
        uvm_root top;

        top = uvm_root::get();

        top.finish_on_completion = 0;

        run_test("test");
        $stop;
    end
endmodule
```

2.3.2 Windowed Register File

The need for fast context switches in modern operating systems led to the design of a windowing mechanism for the standard register file. This component, aka Windowed Register File hereby called WRF for the sake of brevity, helps avoid the need of accessing memory after calling a subroutine.
The interface of such a component is the following:

```
entity WRF_generic is
  generic (M         : integer := 8;
           N         : integer := 8;
           F         : integer := 8;
           DATA_BIT  : integer := 32
           );
  port (CLK          : in  std_logic;
        RST          : in  std_logic;
        EN           : in  std_logic;
        RD1          : in  std_logic;
        RD2          : in  std_logic;
        WR           : in  std_logic;
        CALL         : in  std_logic;
        RET          : in  std_logic;
        ADDR_WR      : in  std_logic_vector(log2(3*N+M)-1 downto 0);
        ADDR_RD1     : in  std_logic_vector(log2(3*N+M)-1 downto 0);
        ADDR_RD2     : in  std_logic_vector(log2(3*N+M)-1 downto 0);
        DATA_IN_MEM  : in  std_logic_vector(DATA_BIT-1 downto 0);
        DATA_IN      : in  std_logic_vector(DATA_BIT-1 downto 0);
        SPILL        : out std_logic;
        FILL         : out std_logic;
        OUT_MEM      : out std_logic_vector(DATA_BIT-1 downto 0);
        OUT1         : out std_logic_vector(DATA_BIT-1 downto 0);
        OUT2         : out std_logic_vector(DATA_BIT-1 downto 0));

end WRF_generic;
```

Register Abstraction Layer endeavor

The UVM library provides the users with a framework called Register Abstraction Layer (RAL). The aim of RAL is to help in accessing the registers of a DUT.
In most nowadays ASICs, we can access registers to configure them (e.g. configuring the baud rate of a UART peripheral).
With this in mind, we wanted to perform a simple MATS algorithm to test our WRF by means of the RAL.
In order to accomplish our task, we need to build the environment shown in Figure 2.1:

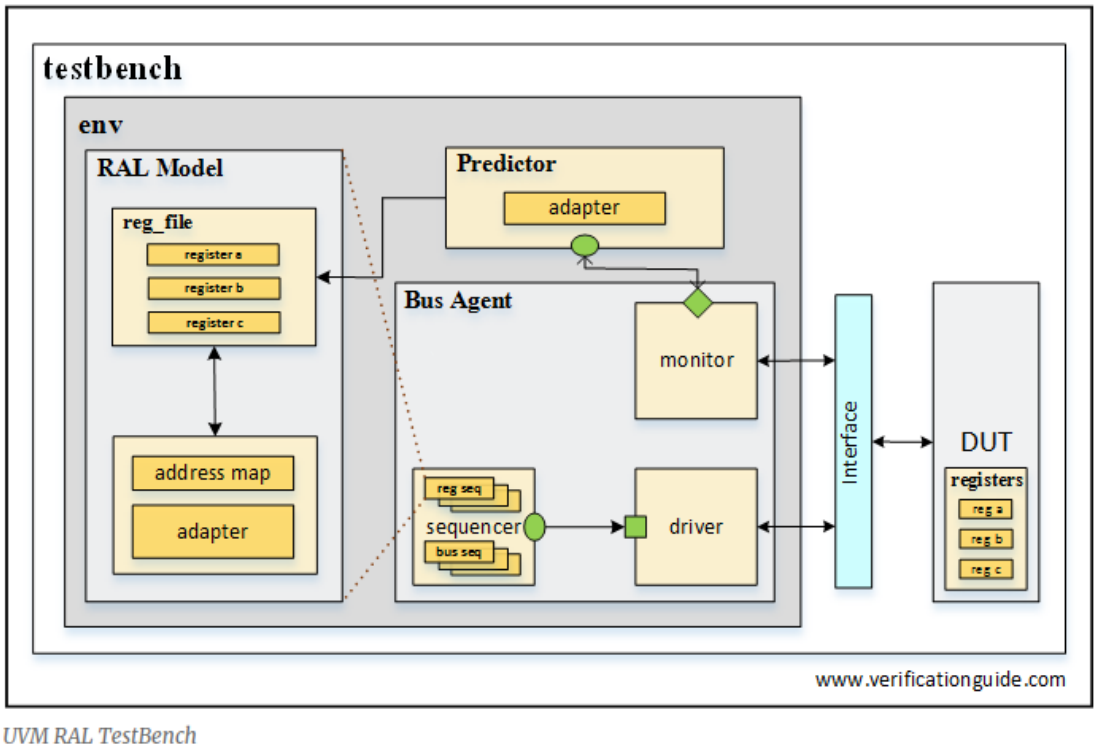


Figure 2.1: UVM RAL structure

The overall structure works in the following way:

1) Everything starts from the test class, which starts a write or a read transaction;

2) This transaction is understandable by the RAL model but not by the DUT, therefore it is converted by the Adapter module;

3) If the transaction was a read, then the DUT provides a result back which is intercepted by the Predictor module;

4) The predictor module sends this transaction to the RAL model through the Adapter;

5) In a read transaction scenario, we want to compare the result coming from the DUT and what is stored inside the RAL model and see if they match.

This procedure might seem long and tedious, but thanks to the high-level interface everything happens under the hood thus improving readability and reusability.
We started by creating the RAL model (called regblock in our code), which is a golden reference of our WRF. The register model is in turn populated with instances of the uvm component uvm_reg, all declared as read/write by means of uvm_reg_field.

```
class regblock extends uvm_reg_block;
    `uvm_object_utils(regblock)
    // Define a dynamic array of umv_reg
    rand my_reg registers[$];
    int actual_win;
    ...

       // Now for each register, create it and then map it by passing a uvm_map
       for (uvm_reg_addr_t i=0; i < parameters::N_REGS; i++) begin
            // Create and configure each register
            this.registers.push_back(my_reg::type_id::create($sformatf("reg%d", i)));

            // Build the register
            this.registers[i].build();

            // Configure the register
            this.registers[i].configure(this, null, "");

            // Map the register to each address
            this.default_map.add_reg(   this.registers[i],
                                        i,      /*Address of the register*/
                                        "RW",   /*Access mode*/
                                        0       /*Unmapped false*/
                                    );
       end
    endfunction
```

Right after that, we implemented the Adapter module:

```
class RF_adapter extends uvm_reg_adapter;
    int off_win = 0;
    `uvm_object_utils(RF_adapter)
    ...
 virtual function void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
        WRF_seq_item pkt;

        // Try to cast the bus item into WRF_seq_item
        if (! $cast(pkt, bus_item)) begin
            `uvm_fatal(get_name(), "couldn't cast!")
        end

        // Assign if R or W
        rw.kind = (pkt.WR && !(pkt.CALL || pkt.RET)) ? UVM_WRITE : UVM_READ;
        // Assign address
        rw.addr = pkt.WR ? pkt.ADDR_WR : pkt.ADDR_RD1;
        // Assign data
        rw.data = pkt.WR ? pkt.DATA_IN : pkt.DATA_OUT1;
        // Transaction is ok
        rw.status = UVM_IS_OK;

    endfunction
    virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
        WRF_seq_item pkt = WRF_seq_item#(8,8,8,32)::type_id::create("pkt");
        if (rw.kind == UVM_WRITE) begin
            pkt.WR = 1;
            pkt.EN = 1;
            pkt.RD1 = 0;
            pkt.RD2 = 0;
            pkt.ADDR_WR = rw.addr;
            pkt.DATA_IN = rw.data;
        end else begin
            pkt.WR = 0;
            pkt.EN = 1;
            pkt.RD1 = 1;
            pkt.RD2 = 0;

            pkt.ADDR_RD1 = rw.addr;
            pkt.OUT1 = rw.data;
        end


        return pkt;

    endfunction
```

The predictor can be left as a simple uvm_predictor and doesn't need a custom implementation. Therefore we skip implementing it and proceed directly to the Test class where we implemented the MATS algorithm.

```
// MATS
// Foreach datapattern
foreach (data_patterns[j]) begin
    // Write the datapattern and test if the written value is read correctly back
    for (int i=0; i<parameters::REGS_PER_WIN; i++) begin
        `uvm_info(get_name(), $sformatf("i=%d\n",i ), UVM_NONE)

        ral_WR_test(i, data_patterns[j], passed);
        // Check if passed the test
        if (passed) begin
            `print_success(get_name(), "Mirrored value match with desired value")
        end else begin
            `print_failure(get_name(), ...
        end
    end
end
```

Unluckily, our version of the WRF provides read data 1 clock cycle after the read command. On the other hand, the RAL model expects to receive the read data on the same clock cycle. Eventually, due to the deadline getting closer we resorted to the standard approach which mimics what is done in RAL.

Sequence item

The sequence item of this DUT was slightly modified with an added field that classifies if a packet is a response (i.e. coming from the DUT) or not.

```
class WRF_seq_item#(parameter M=8, N=8, F=8, DATA_BIT=32) extends uvm_sequence_item;

    bit is_response;
    //  Group: Variables
    rand bit EN;
    rand bit RD1;
    ...
```

Monitor

The monitor should sample both the transactions going to and from the DUT. We faced an issue in sampling the responses since our DUT gives the read data 1 clock cycle later. In fact, the monitor should wait for the responses but doing so, should not block it from detecting new transactions going to the DUT.
We achieve this task by means of the fork/join_none construct.

```
forever begin
    // Wait to get out reset phase
    wait(!vif.RST);

    fork
        // Thread for capturing inputs
        begin
            @(posedge vif.CLK);
            #1; // Needed to consume time

            if (vif.RD1 == 1 || vif.RD2 == 1) begin
                fork
                    // Don't wait threads to complete, since I don't want monitor to wait
                    // Thread for capturing the outputs after a Read
                    begin
                        @(posedge vif.CLK);
                        #1; //Needed to consume time

                        response_item.is_response = 1;  // It's a response transaction
                        response_item.OUT1 = vif.OUT1;
                        response_item.OUT2 = vif.OUT2;

                        item_collect_port.write(response_item);
                    end

                join_none
            end
            // Now sample the data
            mon_item.is_response = 0;   // Not a response transaction
            mon_item.EN = vif.EN;
            mon_item.RD1 = vif.RD1;
```

Scoreboard

The scoreboard receives transactions from the monitor and uses a golden reference of the WRF to compare the read transactions.

```
// If I get a response
if (sb_item.is_response == 1) begin
    // Join the two transactions into a single one
    join_transactions(past_read, sb_item);

    // Check on read 1 port
    if (past_read.RD1 == 1) begin
        // Now check the oracle
        read_data = oracle.read(past_read.ADDR_RD1);
        // Finally compare the two values
        if (read_data == past_read.OUT1) begin
            // Success
            `print_success(get_name(), "Success")
        end else begin
            // Failure
            `print_failure(get_name(), ...
        end
    end
```

Test

In the test class, we implemented the MATS algorithm along with other tests that helps us in reaching high coverage of our properties.

```
// MATS procedure over all the windows
// Foreach datapattern
for (int k=0; k<parameters::WINDOWS_NO; k++) begin
    // For each window perform MATS
    foreach (data_patterns[j]) begin
        // Write the datapattern to all the window
        for (int i=0; i<3*parameters::STD_REGS_NO+parameters::GLOBAL_REGS_NO; i++) begin
            write_procedure(i, data_patterns[j]);
        end
        // Read the datapattern from the window using the two ports concurrently
        // In this way it will take less clock cycles to read back the value
        // We test also the reading behaviour on the same address of the regfile
        for (int i=0; i<(3*parameters::STD_REGS_NO+parameters::GLOBAL_REGS_NO)/2; i++) begin
            doubled_read(i);
        end
    end

    // I reached the end of the test, each window is verified
    // Therefore I don't need keep reading anymore
    if (k == parameters::WINDOWS_NO-1) begin
        // Don't do anything
        continue;
    end
    // Now go to the next window
    go_next_win();  // issues a call
    // Put every command to 0
    issue_wait();
    // If i'm on the last window, then a SPILL is going to be fired
    // Therefore wait here until it goes back to 0 bc the RF
    // is idle during that period
    if (k == parameters::WINDOWS_NO-2) begin
        wait_for_SPILL();
    end
    `uvm_info(get_name(), ...
```

2.3.3 Control Unit (FSM)

Each complex component endowed with a datapath needs precisely temporized control signals to correctly bring out results.
A Control Unit (CU) is in charge of generating these correct temporized signals.
A CU can be implemented in different ways:

- Hardwired, by means of a small LUT
- Miro-programmed, by means of a micro-code memory
- As a Finite State Machine (FSM)

In the following section, we are going through the full UVM test of a FSM CU.
Having already discussed in detail the complete P4 adder test, we are going to analyze for the CU only those components with substantial differences.
The interface for the CU is the following:

```
entity CU_FSM is
generic (
    OPCODE_SIZE : integer := 6;       -- Op Code Size
    FUNC_SIZE : integer := 11     -- Func Field Size for R-Type Ops
    );
port (
    OPCODE : in std_logic_vector(OPCODE_SIZE-1 downto 0);
    FUNC : in std_logic_vector(FUNC_SIZE-1 downto 0);
    CLK, RST : in std_logic;
    -- FIRST PIPE STAGE OUTPUTS
    EN1     : out std_logic; -- enables the register file and the pipeline registers
    RF1     : out std_logic; -- enables the read port 1 of the register file
    RF2     : out std_logic; -- enables the read port 2 of the register file
    WF1     : out std_logic; -- enables the write port of the register file
    -- SECOND PIPE STAGE OUTPUTS
    EN2     : out std_logic; -- enables the pipe registers
    S1      : out std_logic; -- input selection of the first multiplexer
    S2      : out std_logic; -- input selection of the second multiplexer
    ALU1    : out std_logic; -- alu control bit
    ALU2    : out std_logic; -- alu control bit
    -- THIRD PIPE STAGE OUTPUTS
    EN3     : out std_logic; -- enables the memory and the pipeline registers
    RM      : out std_logic; -- enables the read-out of the memory
    WM      : out std_logic; -- enables the write-in of the memory
    S3      : out std_logic -- input selection of the multiplexer
    );
end CU_FSM;
```

Sequence item

For what concerns the sequence item object, is a basic wrapper for all the data fields required for the test. What is worth mentioning is that we added a new data field (*FSM_STATE*), not collected from the interface but updated in the Monitor and needed by the Scoreboard to know in which state the CU is and act consequently.
The *OPCODE* distribution has been inserted to test only the significant inputs (and also because the DUT is not meant to manage unexpected *OPCODE* inputs). Input *6'b000000* has a higher probability because it should be tested with 4 different combinations of *FUNC* input.

```
class CU_sequence_item #(parameter OPCODE_SIZE = 6, parameter FUNC_SIZE = 11)
↪   extends uvm_sequence_item ;

    rand bit [OPCODE_SIZE - 1 : 0] OPCODE;
    rand bit [FUNC_SIZE - 1 : 0] FUNC;

    bit EN1; //enables the register file and the pipeline registers
    ...
    bit S3; // input selection of the multiplexer

    bit [1:0] FSM_STATE;     // FSM state indicator, updated by monitor not by
↪   interface


    constraint OPCODE_dist {   OPCODE dist  {   6'b000000 := 4,
                                                6'b010000 := 1,
                                                6'b010001 := 1,
                                                6'b010010 := 1,
                                                6'b010011 := 1,
                                                6'b100000 := 1,
                                                6'b100001 := 1,
                                                6'b100010 := 1,
                                                6'b100011 := 1,
                                                6'b110000 := 1,
                                                6'b101000 := 1,
                                                6'b010100 := 1,
                                                6'b100100 := 1
                                            };
    }
```

Driver

The driver is basically the same w.r.t. the P4 one, with the only difference being that the sampling is done not on the rising edge of the clock but asynchronously, as soon as the inputs change.
This has been done because of some timing problems. In fact, by sampling exactly on the edge, the expected results were related to old inputs, causing some issues in the Scoreboard. Probably this could be solved by adding a clocking block to the interface, but we decided to proceed with a simpler implementation.

Monitor

Again very similar to the P4 one. In this case, there is also the update of *FSM_STATE* data field in order to follow the CU state flow.

```
if (tr.FSM_STATE == 3) begin
    tr.FSM_STATE += 2; //skip S_rst and directly go to S_oper
end
else begin
    tr.FSM_STATE += 1;
end
```

Scoreboard

Again, the Scoreboard is where the magic happens. Differently from the P4 Scoreboard, this time there is no oracle, but assertions to verify the design have been directly implemented in the Scoreboard itself.
The test is divided into three parts and driven by the value of *FSM_STATE*. Depending on its value, the valuable bits for that particular FSM cycle are tested.

```
task run_phase(uvm_phase phase);
  CU_sequence_item p;
    forever begin
      wait (pkts.size > 0);
      if (pkts.size > 0) begin
        p = pkts.pop_front();

        if (p.FSM_STATE == 1) begin
          assert (p.EN1 == 1 &&
                  p.EN2 == 0 &&
                  p.EN3 == 0 &&
                  p.RF1 == 1 &&
                  p.RF2 == 1 &&
                  p.WF1 == 0 &&
                  p.S1 == 0 &&
                  p.S2 == 0 &&
                  p.ALU1 == 0 &&
                  p.ALU2 == 0 &&
                  p.RM == 0 &&
                  p.WM == 0 &&
                  p.S3 == 0
          ) else begin
            `uvm_error(get_name(), "Error! Checker failed in clock cycle 1")
          end
        end

        if (p.FSM_STATE == 2) begin
          case (p.OPCODE)
            6'b000000 : case(p.FUNC[1:0])
              2'b00 : assert (p.S1 == 0 && p.S2 == 0 && p.ALU1 == 0 && p.ALU2 == 0)
                      else
                      `uvm_error(get_name(), "Error! Checker failed in clock cycle 2")

              2'b01 : assert (p.S1 == 0 && p.S2 == 0 && p.ALU1 == 0 && p.ALU2 == 1)
                      else
                      `uvm_error(get_name(), "Error! Checker failed in clock cycle 2")

              2'b10 : assert (p.S1 == 0 && p.S2 == 0 && p.ALU1 == 1 && p.ALU2 == 0)
                      else
                      `uvm_error(get_name(), "Error! Checker failed in clock cycle 2")

              2'b11 : assert (p.S1 == 0 && p.S2 == 0 && p.ALU1 == 1 && p.ALU2 == 1)
                      else
                      `uvm_error(get_name(), "Error! Checker failed in clock cycle 2")
                        endcase
```

```
            6'b010000 : assert (p.S1 == 1 && p.S2 == 0 && p.ALU1 == 0 && p.ALU2 == 0)
                else
                `uvm_error(get_name(), "Error! Checker failed in clock cycle 2")

            ...

          endcase
        end

        if (p.FSM_STATE == 3) begin
            case(p.OPCODE)
              6'b010000 : assert (p.WF1 == 1 && p.WM == 0 && p.RM == 0 && p.S3 == 0)
                  else
                  `uvm_error(get_name(), "Error! Checker failed in clock cycle 3")

             ...

            endcase
        end
      end
    end

endtask
```

2.4 Results

In order to check whether the three components behaved correctly during the test, we implemented different testing methods for all of them, including:

- User defined UVM macros
- SystemVerilog Assertions (SVA)
- Code coverage
- Functional coverage

User defined UVM macros

This solution is basically an extension of the basic UVM macros *'uvm_info*, *'uvm_warning* and *'uvm_error*. We defined our macros in order to highlight the console messages with different colors and have a better understanding of what is happening.

```
typedef enum {BLACK, BROWN, GREEN, YELLOW, BLUE, MAGENTA, CYAN, GRAY, RED,
↪  WHITE} color_t;

string font_format[color_t] = '{BLACK:"\033[30m%s\033[0m",
↪  BROWN:"\033[31m%s\033[0m", GREEN:"\033[32m%s\033[0m",
↪  YELLOW:"\033[33m%s\033[0m", BLUE:"\033[34m%s\033[0m",
↪  MAGENTA:"\033[35m%s\033[0m", CYAN:"\033[36m%s\033[0m",
↪  GRAY:"\033[37m%s\033[0m", RED:"\033[38m%s\033[0m"};


// use this macro in the following way
↪  uvm_print_success(ssformat(...blabla...))
`define print_success(NAME, STRING_TO_PRINT) \
    `uvm_info(NAME, $sformatf(font_format[GREEN], STRING_TO_PRINT), UVM_LOW)
`define print_failure(NAME, STRING_TO_PRINT) \
    `uvm_info(NAME, $sformatf(font_format[RED], STRING_TO_PRINT), UVM_LOW)\
    `uvm_error(NAME, "Error!")
`define print_warning(NAME, STRING_TO_PRINT) \
    `uvm_info(NAME, $sformatf(font_format[YELLOW], STRING_TO_PRINT), UVM_LOW)
```

SVA

In order to test for some temporal properties of the WRF, we used properties and SystemVerilog Assertions.
The tested properties are symmetric both for SPILL/FILL signals:

- A SPILL/FILL should activate after n (design parameter) CALL/RET;
- The SPILL/FILL signals should stay high as many clock cycles as the number of IN+LOCALS registers.

```
always @(posedge SPILL) spilled_wins++;
always @(posedge FILL) spilled_wins--;

// Property for the SPILL activation after WINDOWS_NO-1 call are issued
property spill_activation;
    @(posedge CLK)

    // If CALL is activated and i'm exceding the number of windows
    // Spill activates on the next CC
    if (CALL && counter >= parameters::WINDOWS_NO-2) ##1 SPILL;
endproperty
...
property spill_activation_time;
    // When SPILL procedure ends
    @(negedge SPILL)
    if (!RET) spill_time == 2*parameters::STD_REGS_NO - 1;
endproperty
```

Code coverage

Code coverage was used to check how many lines of code of our DUT were tested, and if all the branches were taken at least once. In order to enable this feature we simply added a line of code at the end of the simulation script:

```
coverage report -output code_coverage.txt -lines
```

By doing this, we obtained a report file containing all the required information.

Functional coverage

Functional coverage was used to check that all the combinations of inputs were correctly tested. For this purpose, we declared a covergroup for each component, with a coverpoint for each input in turn composed of multiple bins. Through the function *cov_ sample()* used in the Driver, we were able to sample the inputs and check to which bin they belonged.
In order to enable we simply added some lines of code at the end of the simulation script:

```
quietly regexp {20[1-3][0-9]\.[0-9]{1,2}} [vsim -version] ver
quietly if {$ver > 2020} { set src_opt "-srcfile=*" } else { set src_opt "-byfile" }
coverage report -output func_cover.txt $src_opt -details -all -directive -cvg
```

By doing this, we obtained a report containing the amount of covered/missed bins.

2.4.1 P4 Adder

We report here the results for both code and functional coverage. Regarding the branch coverage, we obtain always 100% over the conditions, except the one reported below.

```
================================Condition Details================================

Condition Coverage for file ../src/CLA.vhd --

----------------Focused Condition View-------------------
Line       44 Item    1  (((2 ** (temp_log + 1)) <= value) and (temp_log < 31))
Condition totals: 1 of 2 input terms covered = 50.00%

                       Input Term   Covered  Reason for no coverage   Hint
                      -----------  --------  ----------------------  --------------
  ((2 ** (temp_log + 1)) <= value)         Y
                  (temp_log < 31)          N  '_0' not hit             Hit '_0'

     Rows:       Hits  FEC Target                           Non-masking condition(s)
 ---------  ---------  --------------------                 -------------------------
  Row   1:          1  ((2 ** (temp_log + 1)) <= value)_0  -
  Row   2:          1  ((2 ** (temp_log + 1)) <= value)_1  (temp_log < 31)
  Row   3:    ***0***  (temp_log < 31)_0                    ((2 ** (temp_log + 1)) <= value)
  Row   4:          1  (temp_log < 31)_1                    ((2 ** (temp_log + 1)) <= value)
  ...
  Total Coverage By File (code coverage only, filtered view): 94.44%


  COVERGROUP COVERAGE:
--------------------------------------------------------------------------------
Covergroup                                              Metric       Goal    Status

--------------------------------------------------------------------------------
 TYPE /top_sv_unit/adder_coverage/adder_coverage__1/cg_adder
                                                        79.16%        100    Uncovered
    covered/total bins:                                     11         14
    missing/total bins:                                      3         14
    % Hit:                                              78.57%        100
    Coverpoint cg_adder::A_cg                           75.00%        100    Uncovered
        covered/total bins:                                  3          4
        missing/total bins:                                  1          4
        % Hit:                                          75.00%        100
        bin corner[0]                                        5          1    Covered
        bin corner[1]                                        5          1    Covered
        bin corner[64]                                       0          1    ZERO
        bin corner[255]                                      3          1    Covered
    Coverpoint cg_adder::B_cg                          100.00%        100    Covered
        covered/total bins:                                  4          4
        missing/total bins:                                  0          4
        % Hit:                                         100.00%        100
```

```
        bin corner[0]                                3          1    Covered
        bin corner[1]                                1          1    Covered
        bin corner[64]                               4          1    Covered
        bin corner[255]                              3          1    Covered
    Coverpoint cg_adder::Cin_cg                100.00%        100    Covered
        covered/total bins:                          2          2
        missing/total bins:                          0          2
        % Hit:                                 100.00%        100
        bin corner[0]                              519          1    Covered
        bin corner[1]                              481          1    Covered
    Coverpoint cg_adder::overflow1               0.00%        100    ZERO
        covered/total bins:                          0          2
        missing/total bins:                          2          2
        % Hit:                                   0.00%        100
        bin corner[511]                              0          1    ZERO
        bin corner[65281]                            0          1    ZERO
    Coverpoint cg_adder::overflow2             100.00%        100    Covered
        covered/total bins:                          1          1
        missing/total bins:                          0          1
        % Hit:                                 100.00%        100
        bin corner[255]                              2          1    Covered
    Coverpoint cg_adder::overflow3             100.00%        100    Covered
        covered/total bins:                          1          1
        missing/total bins:                          0          1
        % Hit:                                 100.00%        100
        bin corner[255]                              1          1    Covered

TOTAL COVERGROUP COVERAGE: 79.16%  COVERGROUP TYPES: 1
```

2.4.2 Windowed Register File

We report the results obtained with code and functional coverage. Again, the branch coverage is 100% in most of the branches except the one reported below:

```
------------------------------------IF Branch------------------------------------
    41                                    1146     Count coming in to IF
    41             1                      1145           if (RST = '0') then
    57             1                         1           elsif(RST = '1') then
                                          ***0***     All False Count
Branch totals: 2 hits of 3 branches = 66.66%

================================Condition Details================================

Condition Coverage for file ../src/functions.vhd --

----------------Focused Condition View-------------------
Line       18 Item    1  (((2 ** temp_log) < value) and (temp_log < 31))
Condition totals: 1 of 2 input terms covered = 50.00%

                Input Term   Covered  Reason for no coverage   Hint
               -----------  --------  ----------------------  --------------
  ((2 ** temp_log) < value)         Y
           (temp_log < 31)         N  '_0' not hit             Hit '_0'

     Rows:       Hits  FEC Target                 Non-masking condition(s)
 ---------  ---------  --------------------       -------------------------
  Row   1:          1  ((2 ** temp_log) < value)_0  -
  Row   2:          1  ((2 ** temp_log) < value)_1  (temp_log < 31)
  Row   3:     ***0***  (temp_log < 31)_0            ((2 ** temp_log) < value)
  Row   4:          1  (temp_log < 31)_1            ((2 ** temp_log) < value)



================================================================================

Total Coverage By File (code coverage only, filtered view): 90.85%

COVERGROUP COVERAGE:
--------------------------------------------------------------------------------
Covergroup                                              Metric       Goal    Status

--------------------------------------------------------------------------------
 TYPE /WRF_tb_sv_unit/coverage/cg                      100.00%        100    Covered
    covered/total bins:                                    107        107
    missing/total bins:                                      0        107
    % Hit:                                             100.00%        100
    Coverpoint cg::WR_cg                               100.00%        100    Covered
        covered/total bins:                                  2          2
        missing/total bins:                                  0          2
```

% Hit:	100.00%	100	
bin corner[0]	567	1	Covered
bin corner[1]	544	1	Covered
Coverpoint cg::RD1_cg	100.00%	100	Covered
covered/total bins:	2	2	
missing/total bins:	0	2	
% Hit:	100.00%	100	
bin corner[0]	839	1	Covered
bin corner[1]	272	1	Covered
Coverpoint cg::RD2_cg	100.00%	100	Covered
covered/total bins:	2	2	
missing/total bins:	0	2	
% Hit:	100.00%	100	
bin corner[0]	840	1	Covered
bin corner[1]	271	1	Covered
Coverpoint cg::CALL_cg	100.00%	100	Covered
covered/total bins:	2	2	
missing/total bins:	0	2	
% Hit:	100.00%	100	
bin corner[0]	1103	1	Covered
bin corner[1]	8	1	Covered
Coverpoint cg::RET_cg	100.00%	100	Covered
covered/total bins:	2	2	
missing/total bins:	0	2	
% Hit:	100.00%	100	
bin corner[0]	1103	1	Covered
bin corner[1]	8	1	Covered
Coverpoint cg::double_read_cg	100.00%	100	Covered
covered/total bins:	1	1	
missing/total bins:	0	1	
% Hit:	100.00%	100	
bin corner[1]	271	1	Covered
Coverpoint cg::write_addr_cg	100.00%	100	Covered
covered/total bins:	32	32	
missing/total bins:	0	32	
% Hit:	100.00%	100	
bin corner[0]	35	1	Covered
bin corner[1]	33	1	Covered
bin corner[2]	42	1	Covered
bin corner[3]	33	1	Covered
bin corner[4]	35	1	Covered
bin corner[5]	37	1	Covered
bin corner[6]	40	1	Covered
bin corner[7]	31	1	Covered
...			
bin corner[28]	37	1	Covered
bin corner[29]	33	1	Covered
bin corner[30]	37	1	Covered
bin corner[31]	29	1	Covered

```
    Coverpoint cg::rd_addr1_cg                          100.00%        100    Covered
        covered/total bins:                                  32         32
        missing/total bins:                                   0         32
        % Hit:                                           100.00%        100
        bin corner[0]                                        36          1    Covered
        bin corner[1]                                        36          1    Covered
        bin corner[2]                                        40          1    Covered
        bin corner[3]                                        30          1    Covered
        bin corner[4]                                        39          1    Covered
        ...
        bin corner[29]                                       29          1    Covered
        bin corner[30]                                       34          1    Covered
        bin corner[31]                                       37          1    Covered
    Coverpoint cg::rd_addr2_cg                          100.00%        100    Covered
        covered/total bins:                                  32         32
        missing/total bins:                                   0         32
        % Hit:                                           100.00%        100
        bin corner[0]                                        34          1    Covered
        bin corner[1]                                        37          1    Covered
        bin corner[2]                                        37          1    Covered
        bin corner[3]                                        36          1    Covered
        bin corner[4]                                        33          1    Covered
        bin corner[5]                                        31          1    Covered
        bin corner[6]                                        31          1    Covered
        bin corner[7]                                        26          1    Covered
        ...
        bin corner[31]                                       34          1    Covered

TOTAL COVERGROUP COVERAGE: 100.00%  COVERGROUP TYPES: 1
```

2.4.3 Control Unit

We report here the results obtained with code and functional coverage for the CU:

```
================================================================================
Statement Coverage:
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Stmts                           61        61         0    100.00

================================================================================
Branch Coverage:
    Enabled Coverage            Active      Hits    Misses % Covered
    ----------------            ------      ----    ------ ---------
    Branches                         8         8         0    100.00

================================================================================
COVERGROUP COVERAGE:
--------------------------------------------------------------------------------
Covergroup                                              Metric       Goal    Status
                                                      100.00%        100    Covered
    covered/total bins:                                    16         16
    missing/total bins:                                     0         16
    % Hit:                                            100.00%        100
    Coverpoint CU_cg::OPCODE_I_type_cp                100.00%        100    Covered
        covered/total bins:                                12         12
        missing/total bins:                                 0         12
        % Hit:                                        100.00%        100
        bin corner[16]                                     8          1    Covered
        bin corner[17]                                     7          1    Covered
        bin corner[18]                                     5          1    Covered
        bin corner[19]                                     7          1    Covered
        bin corner[20]                                     8          1    Covered
        bin corner[32]                                     5          1    Covered
        bin corner[33]                                     3          1    Covered
        bin corner[34]                                     6          1    Covered
        bin corner[35]                                     5          1    Covered
        bin corner[36]                                     6          1    Covered
        bin corner[40]                                     5          1    Covered
        bin corner[48]                                     5          1    Covered
    Coverpoint CU_cg::OPCODE_R_type_FUNC_cp           100.00%        100    Covered
        covered/total bins:                                4          4
        missing/total bins:                                0          4
        % Hit:                                        100.00%        100
        bin corner[0]                                      7          1    Covered
        bin corner[1]                                      6          1    Covered
        bin corner[2]                                     10          1    Covered
        bin corner[3]                                      7          1    Covered
```

2.5 Conclusion

In conclusion, thanks to this second Workshop step, we learnt:

- UVM features (Testbench hierarchy, UVM components, RAL, implementation details, etc...)
- How to test complex components using SystemVerilog and UVM
- How easy and fast is it to test a component (even a large one) through SystemVerilog and UVM instead of analyzing waveforms.
- How to reuse some UVM components for different tests
- How to find useful material even when the topic is peculiar and sources begin to run low
- How to work in a team and in parallel by using SSH, Git, and LaTeX

Acronyms

ALU Arithmetic-Logic Unit

DUT Device Under Test

RTL Register-Transfer Level

UVM Universal Verification Methodology

MUX Multiplexer

RCA Ripple Carry Adder

TB Testbench

FSM Finite State Machine

CU Control Unit