Pre-silicon Verification Spring 2022

MIPS UVM / SystemVerilog Verification

Team 2

Bruno Hernandez Lopez
José Alberto Gómez Diaz
Josue isaias Gomez Cosme
Christian Aaron Ortega Blanco

July 07, 2022

Content

Abstract

This project has been carried out to understand and analyze the operation and usefulness of verification, and how it can improve the speed at which errors are found within a supposedly finished module.

Verification is a process that tries to confirm that the design works correctly and how it was expected.

To achieve proper verification it is necessary to generate stimulus for the input signals and monitor the output signals by comparing the output of the device with the output expected to be received. If these last two match then the device will have passed the test, otherwise the device is considered not to be working properly.

The process of verification relies in 10 important aspects, and they are:

+Design

+Interface

+Transaction

+Generator

+Driver

+Monitor

+Scoreboard

+Environment

+Test program

+Testbench

Structure

SystemVerilog for verification

The methodology used for this project is based on the Testbench or Verification Environment is used to test the functional correctness of the Design Under Test (DUT) by generating and driving a predefined input sequence to a design, capturing the design output and comparing it with respect to the expected output.

SystemVerilog Components

The SystemVerilog language is a combination of concepts from several languages. The components of the SystemVerilog language are;

- Verilog HDL concepts
- Vera-based testbench constructs
- OpenVera assertions
- Synopsys VCS DirectC simulation interface for C and C++
- A coverage application programming interface providing links to coverage metrics

System Verilog phases

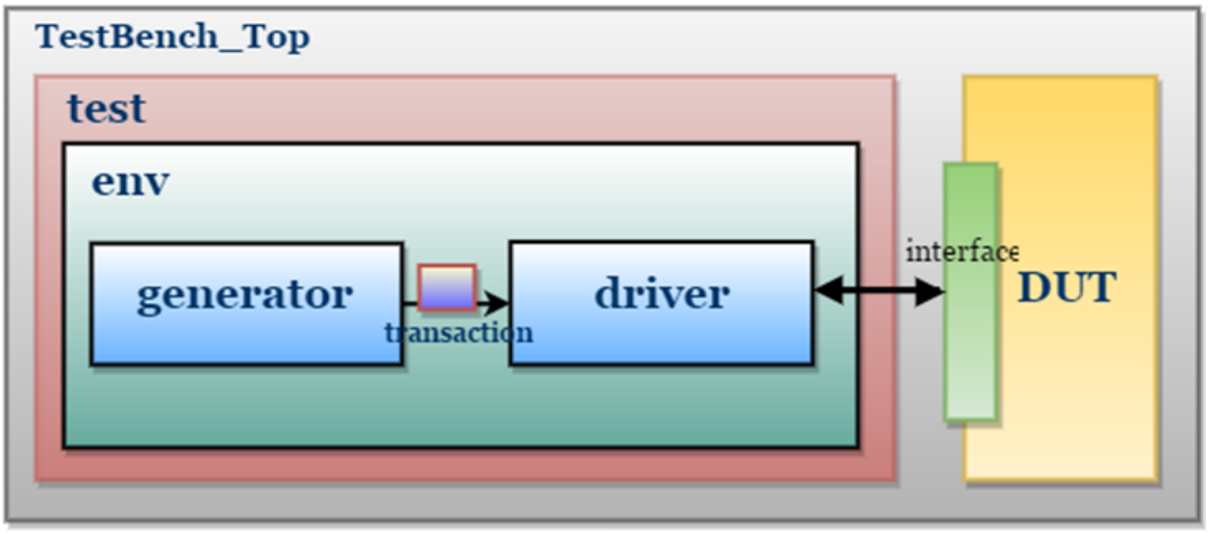
The TestBench or verification environment is a group of classes or components. where each component is performing a specific operation. i.e. generating stimuli, driving, monitoring, etc. and those classes will be named based on the operation. Each of these parts is important to the verification process and are listed in the following table (1);

Name	Type	Description
Transaction	class	Defines the pin level activity generated by agent (to drive to DUT through the driver) or the activity has to be observed

		by agent (Placeholder for the activity monitored by the monitor on DUT signals).
Generator	class	Generates the stimulus (create and randomize the transaction class) and send it to the Driver. Sends each one to the driver through some synchronization element, typically a mailbox.
Driver	class	Receives the stimulus (transaction) from a generator and drives the packet level data inside the transaction into pin level (to DUT). Provides a reset task.
Monitor	class	Observes pin level activity on interface signals and converts into packet level which is sent to the components such as scoreboard.
Interface	class	An interface is a container class, which groups the class's (generator, driver, and monitor) specific to an interface or protocol.
Scoreboard	class	Receives data items from monitors and compares them with expected values. Expected values can be either golden reference values or generated from the reference model
Environment	class	The environment is a container class for grouping higher level components like agent's and scoreboard.
Test Program	Program	The test is responsible for, Configuring the testbench

		• Initiate the testbench components construction process. • Initiate the stimulus driving. • It creates a scope that encapsulates program-wide data.
Testbench_top	class	This is the topmost file, which connects the DUT and TestBench. It consists of DUT, Test and interface instances, the interface connects the DUT and TestBench.

For the design of the program architecture methodology to pass the Testbench, the following image taken from the [Ref. 1] page was used.



TestBench Architecture

We will start from the inside out, and the first class to take into account is;

Transaction Class

Here are generated the necessary fields to generate the stimuli, it is also used to mark the position of the signals for the DUT monitor. The first step was to declare the transaction fields, these fields contain the signals that will be used to verify the module. Then random stimuli with "rand" were used in the fields, and read and write operations were performed.

Generator Class

The generator class is responsible for generating the stimulus by randomizing the transaction class and sending the random class to the controller. First the transaction class identifier was declared, and random stimuli were obtained with "rand". A mailbox was used for the random transaction to the driver, where the mailbox and the environment class identifier were declared, because the mailbox will be shared between the generator and the controller. Variables were added to control the number of packages to be created and an event was added for the completion of the generation process.

Interface Class

The interface will group the program signals, specify the address to use (Modport) and synchronize the monitor signals with the clock (Clocking Block). All signals to be used in the other classes were declared.

Driver class

The driver class is responsible for receiving the stimulus generated from the generator and driving DUT by assigning transaction class values to the interface signals. The interface and mailbox were declared through the constructor, a reset task is added, which initializes the interface signals so that they can be accessed. A task is added to the unit to drive the packet to the interface signal. In addition to adding a local variable to track the number of packets handled and increment the variable in the unit task, until both packets are equal to terminate the simulation.

Environment class

The Environment class contains Mailbox, Generator and Driver. It creates the mailbox, generator and driver share the mailbox identifier in the generator and driver. After declaring the driver and handles of the mailbox, a new mailbox, generator and driver are constructed for the interface identifier with the "new" method. For optimal accessibility, the driver and generator are split into 3 methods;

- pre_test() - Method to call initialization, i.e. restart method.
- test() - Method to call Stimulus Generation and Stimulus Driving.

- post_test() - Method to wait for completion of generation and driving.

We added a task for the execution of all the above mentioned, and it ends with "$finish" for the simulation.

Test Class

The test code is written with the program block. The test is responsible for creating the environment, setting up the testbench, i.e., setting the type and number of transactions to be generated and initializing the stimulus pipeline. First an environment was declared, the number of transactions to be generated was configured and a stimulus was created for its conduction.

Testbench Top

This is the top file, which connects the DUT and TestBench, it consists of DUT, Test and Interface instances. The interface connects the DUT and TestBench. The clock is declared and generated, the interface instance is created, the design instances are created and the interface signals are connected. A test instance is created and passed to the interface identifier, logic is added to generate the tests in verilog, (dump).

Having the main points of verification in System Verilog, the need arose to have a standard methodology that provides a guide for the application of various verification techniques, so as to achieve the maximum possible efficiency and effectiveness in the development of an electronic design.

UVM for verification

The UVM (Universal Verification Methodology) verification system is an Accellera Systems Initiative standard that was developed through the cooperative work of the main manufacturers and consumers of EDA (Electronic Design Automation) tools. All this was possible thanks to the solid base already existing in OVM (Open Verification Methodology) and to the contributions included from VMM (Verification Methodology Manual).

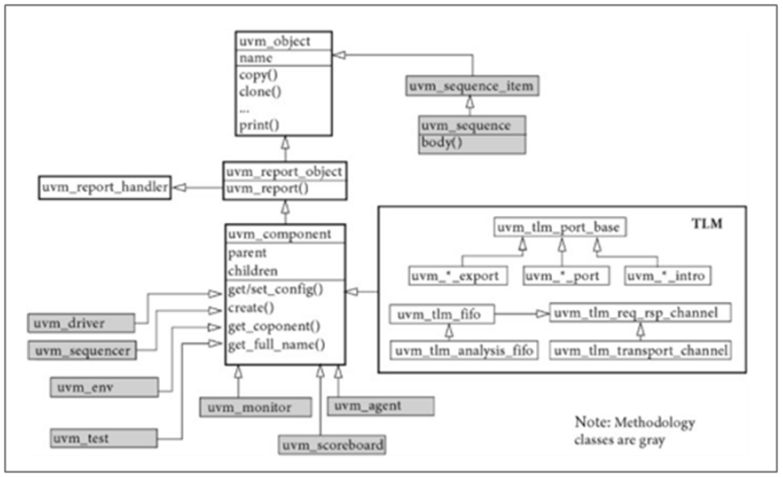
UVM Components

The UVM (Universal Verification Methodology) verification system is an Accellera Systems Initiative standard that was developed through the cooperative work of the main manufacturers and consumers of EDA (Electronic Design Automation) tools. All this was possible thanks to the solid base already existing in OVM (Open Verification Methodology) and to the contributions included from VMM (Verification Methodology Manual).

From a general point of view, UVM consists of a library of base classes, utilities and macros described in SystemVerilog that facilitates the creation of structured verification environments. These components can be encapsulated and instantiated hierarchically, resulting in a test bench, and are controlled through a set of phases to initialize, execute and complete each test.

Among the different classes included in the UVM package, three fundamental groups can be distinguished, which are described below;

- uvm_object. This is the base or parent class in the UVM class hierarchy, from which the rest of the classes inherit. It defines and automates a series of methods for the implementation of functionalities common to all classes, as well as object identification attributes.
- uvm_component. It is the parent class of all the UVC to be integrated in a UVM verification environment. It is a very important class, since it includes all the functionalities of the methodology common to the UVC (Factory, phases, etc.).
- uvm_sequence_item and uvm_sequence. These are the classes used for stimulus generation and for grouping the DUV responses to be verified.



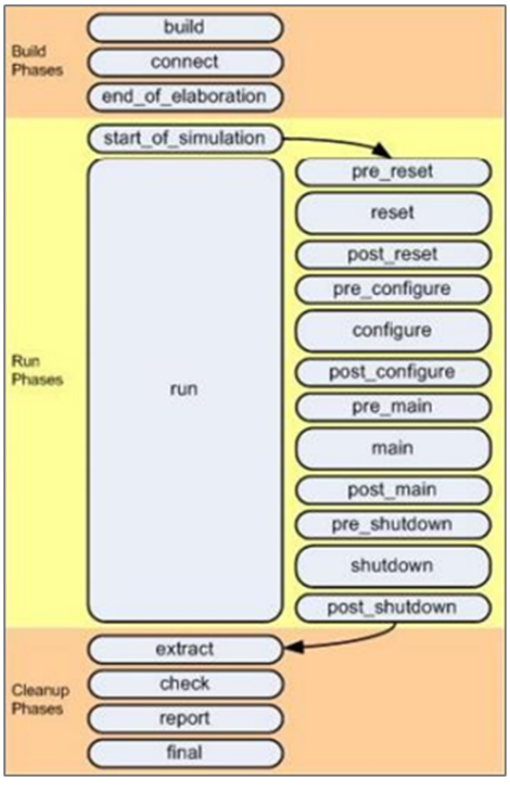
Partial hierarchy of the UVM class library

The objects belonging to the subclasses of uvm_component are static and are part of the test bench hierarchy during the whole simulation. The instantiated objects of the uvm_sequence_item and uvm_sequence classes, on the other hand, are dynamic and are related to the DUV input and output data, so they are created, used and discarded at run time.

As specified in the previous image, the classes of the methodology to which UVM gives access are those highlighted in gray (which will be studied in detail later), so that each of them already includes the basic functionalities of each component or element.

UVM phases

A simulation in a UVM verification environment consists of the sequential execution of a series of phases. In SystemVerilog, classes are referenced during the simulation, so it is necessary to make sure that the environment has been created in its entirety before starting the execution of a test, or that the different elements have been built before trying to connect them. For this purpose, UVM defines the phases shown in the following figure, which are divided into three main groups and are executed sequentially following the specified order.



UVM phases and order of execution

The build phases are executed at the beginning of the UVM simulation and their main purpose is to build, configure and connect the component hierarchy of the verification environment. All these phases are functions that are executed at zero simulation time, i.e. they are not time consuming.
In a common verification environment, the most important and used phases are the following:

build_phase

In this phase the component hierarchy of the verification environment is built following a top-down philosophy, from the highest level of the hierarchy to the lowest ones.

connect_phase

In this phase, the components included in the current hierarchy level are connected. In this case, a bottom-up philosophy is followed, from the components of the lowest levels of the hierarchy to the highest level.

run_phase

This phase describes the behavior of each of the components that make up the verification environment. All components implement this phase in parallel in time. As can be seen in the previous image, this phase is composed of multiple sub-phases, although for a test of low or medium complexity these sub-phases are not useful, so the corresponding code is written directly under the name run_phase. As this is the only phase that consumes execution time, it is necessary to determine when it has finished, for which the objections mechanism is used. This mechanism counts the number of components and sequences that are still participating in the test bench by defining three methods:

raise_objection

It signals the moment in which the execution of the run_phase is initiated in a component.

drop_objection

It signals the moment when the run_phase ends in a component. Once this method is invoked on all components that have previously used the raise_objection method, the simulation is terminated and the execution of the processing phases is prepared.
set_drain_time
Sets a time interval to wait for the end of the simulation once all components have called the drop_objection method.

end_of_elaboration_phase or start_of_simulation_phase

In these phases certain aspects can be displayed on the screen after the test bench creation phases, such as information about the final topology of the environment hierarchy or configuration information.

report_phase

This phase is used to display the simulation results on the console or write them to an external file.

UVM provides a flexible configuration mechanism through a database in order to allow the configuration of certain features of a component without the need to do it statically, or to use the Factory as an intermediary. This gives reusability to verification components or UVCs, as generic components can be configured for a specific mode of operation at runtime.

Important parts

Monocycle verification Systemverilog

Transaction

```
class transaction;
  //declaring the transaction items
  rand bit [4:0] rd;
  rand bit [4:0] rs;
  rand bit [4:0] rt;
  rand bit [5:0] addr;
```

```
  rand bit [5:0] funct;
  rand bit [25:0] JTA;
  rand bit [15:0] imm;
  rand bit [31:0] instruction;
  bit [4:0] shamt = 5'b0;
  bit [31:0] PC_In;
  bit [31:0] mux5_out;
  bit [31:0] A;
  bit [31:0] B;
  bit [31:0] rd2;
  bit [31:0] PCOut;
  bit [31:0] datamem_out;
  constraint op_available {addr inside {0, 2, 4, 5, 35, 43};}
  constraint rd_value {rd inside {[8:25]};}
  constraint rs_value {rs inside {[8:25]};}
  constraint rt_value {rt inside {[8:25]};}
  constraint JTA_value {JTA inside {[0:31]};}
  //constraint imm_value {(imm)<16;}
  constraint imm_value {(imm+rs)<32;}
  constraint funct_range{
    (addr == 0) -> funct inside {32,34,36,37,42};
  }
  typedef enum {r, i, j} AddrType;
  rand AddrType atype;
  constraint optype_range{
    (addr == 0) -> instruction [31:0] == {addr,rs,rt,rd,shamt,funct};
    (addr inside {4,5,35,43}) -> instruction [31:0] == {addr,rs,rt,imm};
    (addr == 2) -> instruction [31:0] == {addr,JTA};}
  constraint addr_range
  { (atype == r) -> addr == 0;
    (atype == i) -> addr inside {4,5,35,43};
    (atype == j) -> addr == 2;}
  function void display(string name);
    $display("-----------------------------------------");
    $display("- %s ",name);
    if(addr==4||addr==5||addr==35||addr==43)begin //tipo I
      $display("TYPE I");
      $display("rs: %b  rt: %b  Imm: %b",rs,rt,imm);
    end
    else if(addr==0)begin //tipo R
      $display("TYPE R");
      $display("rs: %b  rt: %b  rd: %b  funct: %b",rs,rt,rd,funct);
    end
    else if(addr==2)begin //tipo J
      $display("TYPE J");
      $display("JTA: %b",{JTA});
    end
       $display("Instruccion: %b",instruction);
    $display("-----------------------------------------");
  endfunction
endclass
```

Driver

```
`define DRIV_IF vif.PC_In
class driver;
  //used to count the number of transactions
  int no_transactions;
  //creating virtual interface handle
  virtual intf vif;
  //creating mailbox handle
  mailbox gen2driv;
  //constructor
  function new(virtual intf vif,mailbox gen2driv);
    //getting the interface
    this.vif = vif;
    //getting the mailbox handles from  environment
    this.gen2driv = gen2driv;
  endfunction
  //Reset task, Reset the Interface signals to default/initial values
  task reset;
    $display("[ DRIVER ] ----- Reset Started -----");
    vif.rd <= 0;
    vif.rs <= 0;
    vif.rt <= 0;
    vif.addr <= 0;
    vif.funct <= 0;
    vif.JTA <= 0;
    vif.imm <= 0;
    vif.shamt <= 0;
    vif.instruction <=0;
    vif.valid <= 0;
    wait(vif.reset);
    $display("[ DRIVER ] ----- Reset Ended   -----");
  endtask
  //drivers the transaction items to interface signals
  task main;
    forever begin
      transaction trans;
      gen2driv.get(trans);
      @(posedge vif.clk);
      vif.valid <= 1;
      vif.rd <= trans.rd;
      vif.rs <= trans.rs;
      vif.rt <= trans.rt;
      vif.addr <= trans.addr;
      vif.funct <= trans.funct;
      vif.JTA <= trans.JTA;
      vif.imm <= trans.imm;
      vif.shamt <= trans.shamt;
      vif.instruction <= trans.instruction;
      @(negedge vif.clk);
      trans.B = vif.B;
      trans.A= vif.A;
      trans.PC_In = vif.PC_In;
      trans.PCOut = vif.PCOut;
      trans.mux5_out = vif.mux5_out;
      @(posedge vif.clk);
```

```
      trans.rd2 = vif.rd2;
      trans.datamem_out = vif.datamem_out;
      vif.valid <= 0;
      @(posedge vif.clk);
      trans.display("[ Driver ]");
      no_transactions++;
    end
  endtask
endclass
```

Monitor

```
class monitor;//creating virtual interface handle
  virtual intf vif;
  //creating mailbox handle
  mailbox mon2scb;
  //constructor
  function new(virtual intf vif,mailbox mon2scb);
    //getting the interface
    this.vif = vif;
    //getting the mailbox handles from  environment
    this.mon2scb = mon2scb;
  endfunction
 //Samples the interface signal and send the sample packet to scoreboard
  task main;
    forever begin
      transaction trans;
      trans = new();
      @(posedge vif.clk);
      wait(vif.valid);
      trans.rd    = vif.rd;
      trans.rs    = vif.rs;
      trans.rt    = vif.rt;
      trans.addr    = vif.addr;
      trans.funct    = vif.funct;
      trans.JTA    = vif.JTA;
      trans.imm    = vif.imm;
      trans.shamt    = vif.shamt;
      trans.instruction    = vif.instruction;
      @(negedge vif.clk);
      trans.B = vif.B;
      trans.A = vif.A;
      trans.PC_In = vif.PC_In;
      trans.PCOut = vif.PCOut;
      trans.mux5_out = vif.mux5_out;
      @(posedge vif.clk);
      trans.datamem_out = vif.datamem_out;
      trans.rd2 = vif.rd2;
      mon2scb.put(trans);
```

```
      trans.display("[ Monitor ]");
    end
  endtask

endclass
```

Scoreboard

```
class scoreboard;
  int countR,countI,countJ,errorR,errorI,errorJ,succR,succI,succJ;
  mailbox mon2scb;
    //used to count the number of transactions
  int no_transactions;
  //constructor
  function new(mailbox mon2scb);
    //getting the mailbox handles from  environment
    this.mon2scb = mon2scb;
  endfunction
  task main;
    transaction trans;
    forever begin
      mon2scb.get(trans);
      trans.display("[ Scoreboard ]");
      case (trans.addr)
      0: begin
//R TYPE
        countR=countR+1;
        case (trans.funct)
          32:begin               // add
            $display("Operation: add");
            if((trans.A+trans.B) == trans.mux5_out)begin
              $display("- [ Test passed ]");
              $display("Result is as Expected\n %0d + %0d = %0d",trans.A,trans.B,trans.mux5_out);
            end
            else begin
              $error("Wrong Result.\n\t%0d + %0d\n\tExpeced: %0d Actual:
%0d",trans.A,trans.B,(trans.A+trans.B),trans.mux5_out);
                  errorR=errorR+1;
            end
             end
          34:begin                        // sub
            $display("Operation: sub");
            if((trans.A-trans.B) == trans.mux5_out)begin
              if(trans.A>trans.B)begin
                $display("- [ Test passed ]");
                    $display("Result is as Expected\n %0d - %0d =
%0d",trans.A,trans.B,trans.mux5_out);
              end
              else begin
                $display("- [ Test passed ]");
                $display("Result is as Expected\n%b - \n%b = \n%b",trans.A,trans.B,trans.mux5_out);
              end
            end
            else
              if(trans.A>trans.B)begin
              $error("Wrong Result.\n\t%0d - %0d\n\tExpeced: %0d Actual:
%0d",trans.A,trans.B,(trans.A-trans.B),trans.mux5_out);
                errorR=errorR+1;
              end
```

```
            else begin
              $error("Wrong Result.\n\t%32b - %32b\n\tExpeced: %32b Actual:
%32b",trans.A,trans.B,(trans.A-trans.B),trans.mux5_out);
              errorR=errorR+1;
            end
             end

          36:begin                        // and
            $display("Operation: and");
            if((trans.A&trans.B) == trans.mux5_out)begin
              $display("- [ Test passed ]");
              $display("Result is as Expected %32b &\n\t\t\t%32b
=\n\t\t\t%32b",trans.A,trans.B,trans.mux5_out);
            end
            else
              begin
              $error("Wrong Result.\n\t%32b & %32b\n\tExpeced: %0b Actual:
%0b",trans.A,trans.B,(trans.A&trans.B),trans.mux5_out);
            errorR=errorR+1;
              end
             end
          37:begin                        // or
            $display("Operation: or");
            if((trans.A|trans.B) == trans.mux5_out)begin
              $display("- [ Test passed ]");
              $display("Result is as Expected %32b |\n\t\t\t%32b
=\n\t\t\t%32b",trans.A,trans.B,trans.mux5_out);
            end
            else begin
              $error("Wrong Result.\n\t%32b | %32b\n\tExpeced: %0b Actual:
%0b",trans.A,trans.B,(trans.A|trans.B),trans.mux5_out);
              errorR=errorR+1;
            end
             end
          42:begin               // set less than
            $display("Operation: slt");
            if((trans.A<trans.B) == trans.mux5_out)begin
              $display("- [ Test passed ]");
              $display("Result is as Expected %0d < %0d = %0b",trans.A,trans.B,trans.mux5_out);
            end
            else begin
              $error("Wrong Result.\n\t%0d < %0d\n\tExpeced: %0d Actual:
%0d",trans.A,trans.B,(trans.A<trans.B),trans.mux5_out);
                   errorR=errorR+1;
            end
             end
          default: begin
            $error("invalid Rtype option");
            errorR=errorR+1;
          end
        endcase
        no_transactions++;
    end
    2:begin
      $display("JUMP INSTRUCTION");              //J TYPE
      countJ=countJ+1;
      if (trans.PC_In==trans.JTA)begin
        $display("- [ Test passed ]");
        $display("Result is as Expected\nJTA = %0d PC = %0d",trans.JTA,trans.PC_In);
      end
      else begin
        $error("Wrong Result.\n\t PC Expeced: %0d PC Actual: %0d",trans.JTA,trans.PC_In);
        errorJ=errorJ+1;
```

```
      end
        no_transactions++;
    end
    4:begin                      //BEQ
      $display("BEQ INSTRUCTION");
      countI=countI+1;
      if(trans.A==trans.B)begin
        $display("%h = %h",trans.A,trans.B);
        if(trans.PC_In==trans.imm+trans.PCOut+1)begin
          $display("- [ Test passed ]");
          $display("Result is as Expected \nPCIn:    %h\nPCOut:   %h\nimm:
%h",trans.PC_In,trans.PCOut,trans.imm);
        end
        else
          begin
                $error("Wrong Result.\nPCIn:    %0d\nPCOut:   %0d\nimm:     %0d\n\tPC Expeced: %0d
PC Actual: %0d",trans.PC_In,trans.PCOut,trans.imm,(trans.imm+trans.PCOut+1),trans.PC_In);
        errorI=errorI+1;
          end
      end
      else if (trans.PC_In==trans.PCOut+1)begin
        $display("%h != %h",trans.A,trans.B);
        $display("- [ Test passed ]");
        $display("Result is as Expected \nPC In:   %h\nPC Out:  %h",trans.PC_In,trans.PCOut);
      end
      else begin
        $display("%h != %h",trans.A,trans.B);
        $error("Wrong Result.\nPCIn:    %0d\nPCOut:   %0d\n\tPC Expeced: %0d PC Actual:
%0d",trans.PC_In,trans.PCOut,(trans.PCOut+1),trans.PC_In);
        errorI=errorI+1;
      end
      no_transactions++;
    end
    5:begin              //BNE
      $display("BNE INSTRUCTION");
      countI=countI+1;
      if(trans.A!=trans.B)begin
        $display("%h != %h",trans.A,trans.B);
        if(trans.PC_In==trans.imm+trans.PCOut+1)begin
          $display("- [ Test passed ]");
          $display("Result is as Expected \nPCIn:    %h\nPCOut:   %h\nimm:
%h",trans.PC_In,trans.PCOut,trans.imm);
        end
        else begin
          $error("Wrong Result.\nPCIn:    %0d\nPCOut:   %0d\nimm:     %0d\n\tPC Expeced: %0d PC
Actual: %0d",trans.PC_In,trans.PCOut,trans.imm,(trans.imm+trans.PCOut+1),trans.PC_In);
        errorI=errorI+1;
        end
      end
      else if (trans.PC_In==trans.PCOut+1)begin
        $display("%h == %h",trans.A,trans.B);
        $display("- [ Test passed ]");
        $display("Result is as Expected \nPC In:   %h\nPC Out:  %h",trans.PC_In,trans.PCOut);
      end
      else begin
        $error("Wrong Result.\nPCIn:    %0d\nPCOut:   %0d\n\tPC Expeced: %0d PC Actual:
%0d",trans.PC_In,trans.PCOut,(trans.PCOut+1),trans.PC_In);
        errorI=errorI+1;
      end
      no_transactions++;
    end
    35:begin                     //LW
      $display("LW INSTRUCTION");
```

```
      countI=countI+1;
      if(trans.rd2==trans.mux5_out)begin
        $display("- [ Test passed ]");
        $display("Result is as Expected \nDataMem:        %0d  \nRegMem address: %0d  \nRegMem
Data:   %0d ",trans.mux5_out,trans.rt,trans.rd2);
      end
      else begin
        $error("Wrong Result.\n\tAddress: %b\n\tExpeced: %0d Actual:
%0d",trans.rt,trans.mux5_out,trans.rd2);
        errorI=errorI+1;
      end
      no_transactions++;
    end
    43:begin                              //SW
      $display("SW INSTRUCTION");
      countI=countI+1;
      if(trans.datamem_out==trans.rd2)begin
        $display("- [ Test passed ]");
        $display("Result is as Expected \nRegister: %0d \nData Reg: %0d  \nData Mem:
%0d",trans.rt,trans.rd2,trans.datamem_out);
      end
      else begin
        $error("Wrong Result.\n\tAddress: %b\n\tExpeced: %0d \n\tActual:
%0d",trans.rt,trans.rd2,trans.datamem_out);
        errorI=errorI+1;
      end
    end
    default: $error("invalid option");
    endcase
      succR=(((countR-errorR)*100)/countR);
      succI=(((countI-errorI)*100)/countI);
      succJ=(((countJ-errorJ)*100)/countJ);
    end
  endtask
  task count;
    $display ("RESUME");
    $display("R types Functions generated:",countR);
    $display("I types Functions generated:",countI);
    $display("J types Functions generated:",countJ);
    $display ("\nERRORS");
    $display("R types Functions errors:",errorR);
    $display("I types Functions errors:",errorI);
    $display("J types Functions errors:",errorJ);
    $display ("\nSUCCESFUL");
    $display("R types Functions succeful:",succR,"%%");
    $display("I types Functions succeful:",succI,"%%");
    $display("J types Functions succeful:",succJ,"%%");
    $display ("-----VERIFICATION COMPLITED-----");
   endtask
endclass
```

Multicycle verification Systemverilog

Driver

```
`define DRIV_IF vif.PC_In
class driver;
  int no_transactions;
  virtual intf vif;
```

```
mailbox gen2driv;
function new(virtual intf vif,mailbox gen2driv);
  this.vif = vif;
  this.gen2driv = gen2driv;
endfunction
task reset;
  wait(!vif.reset);
  $display("[ DRIVER ] ----- Reset Started -----");
  vif.rd <= 0;
  vif.rs <= 0;
  vif.rt <= 0;
  vif.addr <= 0;
  vif.funct <= 0;
  vif.JTA <= 0;
  vif.imm <= 0;
  vif.shamt <= 0;
  vif.instruction <=0;
  vif.valid <= 0;
  vif.mux2_outt<=0;
  wait(vif.reset);
  $display("[ DRIVER ] ----- Reset Ended   -----");
endtask
task main;
  begin
    @(posedge vif.clk);
    @(posedge vif.clk);
    @(posedge vif.clk);
  end
  forever begin
    transaction trans;
    gen2driv.get(trans);
    vif.valid <= 1;
    vif.rd <= trans.rd;
    vif.rs <= trans.rs;
    vif.rt <= trans.rt;
    vif.addr <= trans.addr;
    vif.funct <= trans.funct;
    vif.JTA <= trans.JTA;
    vif.imm <= trans.imm;
    vif.shamt <= trans.shamt;
    trans.PCOut = vif.PCOut;
    vif.instruction <= trans.instruction;
    @(posedge vif.clk);
    @(posedge vif.clk);
    if(trans.addr==0)begin // R TYPE
      @(negedge vif.clk);
      trans.B = vif.B;
     trans.A= vif.A;
      @(negedge vif.clk);
    end
    else if(trans.addr==4)begin//Beq
      @(negedge vif.clk);
      trans.B = vif.B;
     trans.A= vif.A;
      trans.PCOut = vif.PCOut;
    end
```

```
      else if(trans.addr==5)begin//Bne
        @(negedge vif.clk);
        trans.B = vif.B;
       trans.A= vif.A;
        trans.PCOut = vif.PCOut;
        trans.PC_In = vif.PC_In;
      end
      else if(trans.addr==35)begin//Load Word
        @(negedge vif.clk);
        trans.B = vif.B;
       trans.A= vif.A;
        trans.mux5_out=vif.mux5_out;
        trans.rd2=vif.rd2;
        @(negedge vif.clk);
        trans.datamem_out  = vif.datamem_out;
      end
      else if(trans.addr==43)begin//Store Word
        @(negedge vif.clk);
        trans.B = vif.B;
       trans.A= vif.A;
        trans.datamem_out  = vif.datamem_out;
        trans.rd2=vif.rd2;
        @(negedge vif.clk);
      end
      else if(trans.addr==2)begin//J-Type
        @(negedge vif.clk);
        trans.PC_In = vif.PC_In;
        trans.PCOut = vif.PCOut;
      end
      @(posedge vif.clk);
      trans.mux5_out = vif.mux5_out;
      trans.mux2_outt=vif.mux2_outt;
      vif.valid <= 0;
      trans.display("[ Driver ]");
      no_transactions++;
    end
  endtask
endclass
```

Monitor

```
class monitor;
  virtual intf vif;
  mailbox mon2scb;
  function new(virtual intf vif,mailbox mon2scb);
    this.vif = vif;
    this.mon2scb = mon2scb;
  endfunction
  task main;
    forever begin
      transaction trans;
      trans = new();
      wait(vif.instruction);
```

```
@(posedge vif.clk);
trans.rd   = vif.rd;
trans.rs   = vif.rs;
trans.rt   = vif.rt;
trans.addr   = vif.addr;
trans.funct   = vif.funct;
trans.JTA   = vif.JTA;
trans.imm   = vif.imm;
trans.shamt   = vif.shamt;
@(posedge vif.clk);
if(trans.addr==0)begin// R TYPE
  @(negedge vif.clk);
  trans.B = vif.B;
 trans.A= vif.A;
  @(negedge vif.clk);
        trans.mux5_out = vif.mux5_out;
end
else if(trans.addr==4)begin//Beq

  @(negedge vif.clk);
  trans.B = vif.B;
 trans.A= vif.A;
  trans.PCOut = vif.PCOut;
end
else if(trans.addr==5)begin//Bne
  @(negedge vif.clk);
  trans.B = vif.B;
 trans.A= vif.A;
  trans.PCOut = vif.PCOut;
  trans.PC_In = vif.PC_In;
end
else if(trans.addr==35)begin//Load Word
  @(negedge vif.clk);
  trans.B = vif.B;
 trans.A= vif.A;
  trans.mux5_out=vif.mux5_out;
  trans.rd2=vif.rd2;
  trans.datamem_out  = vif.datamem_out;
  @(negedge vif.clk);
end
else if(trans.addr==43)begin//Store Word
  @(negedge vif.clk);
  trans.B = vif.B;
 trans.A= vif.A;
  trans.datamem_out  = vif.datamem_out;
  trans.rd2=vif.rd2;
  @(negedge vif.clk);
end
else if(trans.addr==2)begin    //J-Type
  @(negedge vif.clk);
  trans.PC_In = vif.PC_In;
  trans.PCOut = vif.PCOut;
end
@(posedge vif.clk);
if(trans.addr==5||trans.addr==4)
 trans.PC_In = vif.PC_In;
```

```
      trans.datamem_out = vif.datamem_out;
      trans.mux2_outt=vif.mux2_outt;
      trans.rd2 = vif.rd2;
      mon2scb.put(trans);
      trans.display("[ Monitor ]");
      trans.instruction   = vif.instruction;
    end
  endtask
endclass
```

Monocycle verification UVM

Sequence item

```
class mem_seq_item extends uvm_sequence_item;
  rand bit [4:0] rd;
  rand bit [4:0] rs;
  rand bit [4:0] rt;
  rand bit [5:0] addr;
  rand bit [5:0] funct;
  rand bit [25:0] JTA;
  rand bit [15:0] imm;
  rand bit [31:0] instruction;
  bit [4:0] shamt = 5'b0;
  bit [31:0] PC_In;
  bit [31:0] mux5_out;
  bit [31:0] A;
  bit [31:0] B;
  bit [31:0] rd2;
  bit [31:0] PCOut;
  bit [31:0] datamem_out;
  //---------------------------------------
  //Utility and Field macros
  //---------------------------------------
  `uvm_object_utils_begin(mem_seq_item)
    `uvm_field_int(rd,UVM_ALL_ON)
    `uvm_field_int(rs,UVM_ALL_ON)
    `uvm_field_int(rt,UVM_ALL_ON)
    `uvm_field_int(addr,UVM_ALL_ON)
    `uvm_field_int(funct,UVM_ALL_ON)
    `uvm_field_int(JTA,UVM_ALL_ON)
    `uvm_field_int(imm,UVM_ALL_ON)
    `uvm_field_int(instruction,UVM_ALL_ON)
    `uvm_field_int(shamt,UVM_ALL_ON)
    `uvm_object_utils_end
  function new(string name = "mem_seq_item");
    super.new(name);
  endfunction
  //constaints
  constraint op_available {addr inside {0, 2, 4, 5, 35, 43};}
  constraint rd_value {rd inside {[8:25]};}
  constraint rs_value {rs inside {[8:25]};}
  constraint rt_value {rt inside {[8:25]};}
```

```
  constraint JTA_value {JTA inside {[0:31]};}
  constraint imm_value {(imm+rs)<32;}
  constraint funct_range{
    (addr == 0) -> funct inside {32,34,36,37,42};  }
  typedef enum {r, i, j} AddrType;
  rand AddrType atype;
  constraint optype_range{
    (addr == 0) -> instruction [31:0] == {addr,rs,rt,rd,shamt,funct};
    (addr inside {4,5,35,43}) -> instruction [31:0] == {addr,rs,rt,imm};
    (addr == 2) -> instruction [31:0] == {addr,JTA}; }
  constraint addr_range
  { (atype == r) -> addr == 0;
    (atype == i) -> addr inside {4,5,35,43};
    (atype == j) -> addr == 2;}
endclass
```

Driver

```
`define DRIV_IF vif.DRIVER.driver_cb
class mem_driver extends uvm_driver #(mem_seq_item);
  virtual mem_if vif;
  `uvm_component_utils(mem_driver)
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new
  // build phase
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
     if(!uvm_config_db#(virtual mem_if)::get(this, "", "vif", vif))
       `uvm_fatal("NO_VIF",{"virtual interface must be set for:
",get_full_name(),".vif"});
  endfunction: build_phase
  // run phase
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      drive();
      seq_item_port.item_done();
    end
  endtask : run_phase
  // drives the value's from seq_item to interface signals
  virtual task drive();
    `DRIV_IF.rd <= 0;
    `DRIV_IF.rs <= 0;
    `DRIV_IF.rt <= 0;
    `DRIV_IF.addr <= 0;
    `DRIV_IF.funct <= 0;
    `DRIV_IF.JTA <= 0;
    `DRIV_IF.imm <= 0;
    `DRIV_IF.shamt <= 0;
    `DRIV_IF.instruction <=0;
    @(posedge vif.DRIVER.clk);
      `DRIV_IF.rd <= req.rd;
```

```
      `DRIV_IF.rs <= req.rs;
      `DRIV_IF.rt <= req.rt;
      `DRIV_IF.addr <= req.addr;
      `DRIV_IF.funct <= req.funct;
      `DRIV_IF.JTA <= req.JTA;
      `DRIV_IF.imm <= req.imm;
      `DRIV_IF.shamt <= req.shamt;
      `DRIV_IF.instruction <= req.instruction;
      @(posedge vif.DRIVER.clk);
    `DRIV_IF.instruction <= req.instruction;
    `DRIV_IF.InstMem_Out <= req.instruction;
      req.B <= `DRIV_IF.B;
      req.A<= `DRIV_IF.A;
      req.PC_In<= `DRIV_IF.PC_In;
      req.PCOut <= `DRIV_IF.PCOut;
      req.mux5_out<= `DRIV_IF.mux5_out;
      @(posedge vif.DRIVER.clk);
      req.rd2 <= `DRIV_IF.rd2;
      req.datamem_out <= `DRIV_IF.datamem_out;
      @(posedge vif.DRIVER.clk);
  endtask : drive
endclass : mem_driver
```

Monitor

```
class mem_monitor extends uvm_monitor;
  virtual mem_if vif;
  uvm_analysis_port #(mem_seq_item) item_collected_port;
  mem_seq_item trans_collected;
  `uvm_component_utils(mem_monitor)
  function new (string name, uvm_component parent);
    super.new(name, parent);
    trans_collected = new();
    item_collected_port = new("item_collected_port", this);
  endfunction : new
  // build_phase - getting the interface handle
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual mem_if)::get(this, "", "vif", vif))
       `uvm_fatal("NOVIF",{"virtual interface must be set for:
",get_full_name(),".vif"});
  endfunction: build_phase
  virtual task run_phase(uvm_phase phase);
    forever begin
      @(posedge vif.MONITOR.clk);
      trans_collected.rd    = vif.monitor_cb.rd;
      trans_collected.rs    = vif.monitor_cb.rs;
      trans_collected.rt    = vif.monitor_cb.rt;
      trans_collected.addr    = vif.monitor_cb.addr;
      trans_collected.funct    = vif.monitor_cb.funct;
      trans_collected.JTA    = vif.monitor_cb.JTA;
      trans_collected.imm    = vif.monitor_cb.imm;
      trans_collected.shamt    = vif.monitor_cb.shamt;
```

```
      trans_collected.instruction   = vif.monitor_cb.instruction;
      @(posedge vif.MONITOR.clk);
      trans_collected.B = vif.monitor_cb.B;
      trans_collected.A = vif.monitor_cb.A;
      trans_collected.PC_In = vif.monitor_cb.PC_In;
      trans_collected.PCOut = vif.monitor_cb.PCOut;
      trans_collected.mux5_out = vif.monitor_cb.mux5_out;
      @(posedge vif.MONITOR.clk);
      trans_collected.datamem_out = vif.monitor_cb.datamem_out;
      trans_collected.rd2 = vif.monitor_cb.rd2;
         item_collected_port.write(trans_collected);
      end
  endtask : run_phase
endclass : mem_monitor
```

Multicycle verification UVM

Driver

```
`define DRIV_IF vif//.DRIVER.driver_cb
class mem_driver extends uvm_driver #(mem_seq_item);
  virtual mem_if vif;
  `uvm_component_utils(mem_driver)
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction : new
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
     if(!uvm_config_db#(virtual mem_if)::get(this, "", "vif", vif))
       `uvm_fatal("NO_VIF",{"virtual interface must be set for:
",get_full_name(),".vif"});
  endfunction: build_phase
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      drive();
      seq_item_port.item_done();
    end
  endtask : run_phase
  virtual task drive();
    if(~vif.reset)begin
      `DRIV_IF.rd <= 0;
      `DRIV_IF.rs <= 0;
      `DRIV_IF.rt <= 0;
      `DRIV_IF.addr <= 0;
      `DRIV_IF.funct <= 0;
      `DRIV_IF.JTA <= 0;
      `DRIV_IF.imm <= 0;
      `DRIV_IF.shamt <= 0;
      `DRIV_IF.instruction <=0;
```

```
    wait(vif.reset);
    @(posedge vif.clk);
    @(posedge vif.clk);
  end
  `DRIV_IF.rd <= req.rd;
  `DRIV_IF.rs <= req.rs;
  `DRIV_IF.rt <= req.rt;
  `DRIV_IF.addr <= req.addr;
  `DRIV_IF.funct <= req.funct;
  `DRIV_IF.JTA <= req.JTA;
  `DRIV_IF.imm <= req.imm;
  `DRIV_IF.shamt <= req.shamt;
  @(posedge vif.DRIVER.clk);
  `DRIV_IF.instruction <= req.instruction;
  @(posedge vif.clk);
  if(req.addr==0)begin              // R TYPE
    @(negedge vif.clk);
    req.B <= `DRIV_IF.B;
    req.A<= `DRIV_IF.A;
    @(negedge vif.clk);
  end
    else if(req.addr==4)begin//Beq
      @(negedge vif.clk);
     req.B <= `DRIV_IF.B;
     req.A<= `DRIV_IF.A;
     req.PCOut <= `DRIV_IF.PCOut;
    end
    else if(req.addr==5)begin//Bne
       req.B <= `DRIV_IF.B;
     req.A<= `DRIV_IF.A;
     req.PCOut <= `DRIV_IF.PCOut;
      req.PC_In<= `DRIV_IF.PC_In;
    end
  else if(req.addr==35)begin//Load Word
      @(negedge vif.clk);
     req.B <= `DRIV_IF.B;
     req.A<= `DRIV_IF.A;
     req.mux5_out<= `DRIV_IF.mux5_out;
     req.rd2 <= `DRIV_IF.rd2;
     @(negedge vif.clk);
     req.datamem_out <= `DRIV_IF.datamem_out;
  end
  else if(req.addr==43)begin//Store Word
     @(negedge vif.clk);
     req.B <= `DRIV_IF.B;
     req.A<= `DRIV_IF.A;
     req.datamem_out <= `DRIV_IF.datamem_out;
     req.rd2 <= `DRIV_IF.rd2;
     @(negedge vif.clk);
  end
    else if(req.addr==2)begin//J-Type
      @(negedge vif.clk);
      req.PCOut <= `DRIV_IF.PCOut;
      req.PC_In<= `DRIV_IF.PC_In;
    end
  @(posedge vif.clk);
```

```
        req.mux5_out<= `DRIV_IF.mux5_out;
    req.mux2_outt<= `DRIV_IF.mux2_outt;
  endtask : drive
endclass : mem_driver
```

Monitor

```
class mem_monitor extends uvm_monitor;
  virtual mem_if vif;
  uvm_analysis_port #(mem_seq_item) item_collected_port;
  mem_seq_item trans_collected;
  `uvm_component_utils(mem_monitor)
  function new (string name, uvm_component parent);
    super.new(name, parent);
    trans_collected = new();
    item_collected_port = new("item_collected_port", this);
  endfunction : new
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if(!uvm_config_db#(virtual mem_if)::get(this, "", "vif", vif))
       `uvm_fatal("NOVIF",{"virtual interface must be set for:
",get_full_name(),".vif"});
  endfunction: build_phase
  virtual task run_phase(uvm_phase phase);
    forever begin
      wait(vif.monitor_cb.instruction);
      @(posedge vif.clk);
      trans_collected.rd   = vif.rd;
      trans_collected.rs   = vif.rs;
      trans_collected.rt   = vif.rt;
      trans_collected.addr   = vif.addr;
      trans_collected.funct   = vif.funct;
      trans_collected.JTA   = vif.JTA;
      trans_collected.imm   = vif.imm;
      trans_collected.shamt   = vif.shamt;
      if(trans_collected.addr==0)begin// R TYPE
        @(posedge vif.clk);
        @(negedge vif.clk);
        trans_collected.B = vif.B;
        trans_collected.A = vif.A;
        @(negedge vif.clk);
        trans_collected.mux5_out = vif.mux5_out;
      end
      else if(trans_collected.addr==4)begin//Beq
        trans_collected.PCOut = vif.PCOut;
        @(posedge vif.clk);
        @(negedge vif.clk);
        trans_collected.B = vif.B;
        trans_collected.A = vif.A;
      end
      else if(trans_collected.addr==5)begin//Bne
        trans_collected.PCOut = vif.PCOut;
        @(posedge vif.clk);
```

```
        @(negedge vif.clk);
        trans_collected.B = vif.B;
        trans_collected.A = vif.A;
        trans_collected.PC_In = vif.PC_In;
      end
      else if(trans_collected.addr==35)begin//Load Word
        @(posedge vif.clk);
        @(negedge vif.clk);
        trans_collected.B = vif.B;
        trans_collected.A = vif.A;
        trans_collected.mux5_out = vif.mux5_out;
        trans_collected.rd2 = vif.rd2;
        trans_collected.datamem_out = vif.datamem_out;
        @(negedge vif.clk);
      end
      else if(trans_collected.addr==43)begin//Store Word
        @(posedge vif.clk);
        @(negedge vif.clk);
        trans_collected.B = vif.B;
        trans_collected.A = vif.A;
        trans_collected.rd2 = vif.rd2;
        @(negedge vif.clk);
      end
      else if(trans_collected.addr==2)begin //J-Type
        @(posedge vif.clk);
        trans_collected.PCOut = vif.PCOut;
        trans_collected.PC_In = vif.PC_In;
        @(negedge vif.clk);
      end
      @(posedge vif.clk);
      if(trans_collected.addr==4)
         trans_collected.PC_In = vif.PC_In;
         trans_collected.datamem_out = vif.datamem_out;
         trans_collected.mux2_outt = vif.mux2_outt;
       item_collected_port.write(trans_collected);
      end
  endtask : run_phase
endclass : mem_monitor
```

Verification Results

Verilog Codes

Monocycle verification Systemverilog:

https://www.edaplayground.com/x/EyaS

Multicycle verification Systemverilog:

https://www.edaplayground.com/x/MBku

Monocycle verification UVM:

https://www.edaplayground.com/x/K5kY

Multicycle verification UVM:

https://www.edaplayground.com/x/d2tG

Results

While running the verification, the environment is going to send random instructions from the generator and after the execution of each instruction, the scoreboard is going to determine if the architecture is working properly, during this the terminar will be displaying the resulting instructions from the generator and a summary with the scoreboard results, as the following instruction type verification.

Monocycle and Multicycle vérification (Systemverilog):

After the execution of the verification, the first thing on display is the details of every test by hierarchy in their classes. for example, if there was an error on the

architecture after the execution of the verification, we can see in detail where the error could be.

```
# KERNEL: - [ Test passed ]
# KERNEL: Result is as Expected
# KERNEL: Register: 15
# KERNEL: Data Reg: 15
# KERNEL: Data Mem: 15
# KERNEL: ----------------------------------------
# KERNEL: - [ Driver ]
# KERNEL: TYPE I
# KERNEL: rs: 01011  rt: 01111  Imm: 0000000000000101
# KERNEL: Instruccion: 10101101011011110000000000000101
# KERNEL: ----------------------------------------
# KERNEL: ----------------------------------------
# KERNEL: - [ Monitor ]
# KERNEL: TYPE J
# KERNEL: JTA: 00000000000000000000010010
# KERNEL: Instruccion: 00001000000000000000000000010010
# KERNEL: ----------------------------------------
# KERNEL: ----------------------------------------
# KERNEL: - [ Scoreboard ]
# KERNEL: TYPE J
# KERNEL: JTA: 00000000000000000000010010
# KERNEL: Instruccion: 00001000000000000000000000010010
# KERNEL: ----------------------------------------
# KERNEL: JUMP INSTRUCTION
# KERNEL: - [ Test passed ]
# KERNEL: Result is as Expected
# KERNEL: JTA = 18 PC = 18
# KERNEL: ----------------------------------------
# KERNEL: - [ Driver ]
# KERNEL: TYPE J
# KERNEL: JTA: 00000000000000000000010010
# KERNEL: Instruccion: 00001000000000000000000000010010
```

The summary results are located at the bottom of the verification report, this is helpful if the user only wants to know if the architecture is working properly. the result summary is split on tree parts, the Resume displays the type and number of instructions generated for the test, the Errors count the number of errors found for each instruction type, the las list display the percentage of functionality for each instruction type based on the number of successfully tested instructions.

```
# KERNEL: RESUME
# KERNEL: R types Functions generated:          2
# KERNEL: I types Functions generated:          6
# KERNEL: J types Functions generated:          2
# KERNEL:
# KERNEL: ERRORS
# KERNEL: R types Functions errors:          0
# KERNEL: I types Functions errors:          0
# KERNEL: J types Functions errors:          0
# KERNEL:
# KERNEL: SUCCESFUL
# KERNEL: R types Functions succeful:        100%
# KERNEL: I types Functions succeful:        100%
# KERNEL: J types Functions succeful:        100%
# KERNEL: -----VERIFICATION COMPLITED-----
```

Monocycle and Multicycle vérification (UVM):

The UVM methodology it's pretty similar to the system verilog verification. At first the display shows every instruction under test, the expected result and the actual result, an error will be detected if those values were unequal.

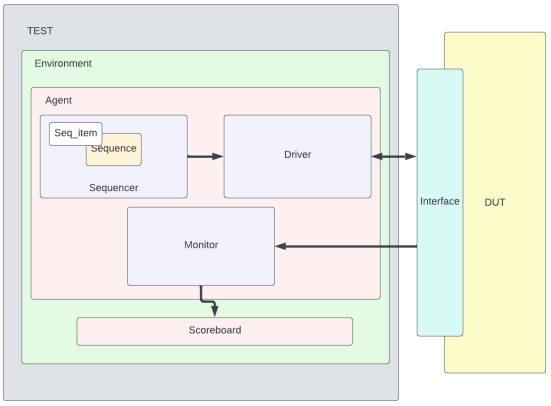
```
UVM_INFO mem_scoreboard.sv(244) @ 55: uvm_test_top.env.mem_scb [mem_scoreboard] ------ :: SW INSTRUCTION  :: ------
UVM_INFO mem_scoreboard.sv(247) @ 55: uvm_test_top.env.mem_scb [mem_scoreboard] - [ Test passed ]
UVM_INFO mem_scoreboard.sv(248) @ 55: uvm_test_top.env.mem_scb [mem_scoreboard] Result is as Expected
Register: 12
Data Reg: 12
Data Mem: 12
UVM_INFO mem_scoreboard.sv(104) @ 85: uvm_test_top.env.mem_scb [mem_scoreboard] ------ :: AND INSTRUCTION  :: ------
UVM_INFO mem_scoreboard.sv(106) @ 85: uvm_test_top.env.mem_scb [mem_scoreboard] - [ Test passed ]
UVM_INFO mem_scoreboard.sv(107) @ 85: uvm_test_top.env.mem_scb [mem_scoreboard] Result is as Expected
 10110 & 11001 = 16
UVM_INFO mem_scoreboard.sv(228) @ 145: uvm_test_top.env.mem_scb [mem_scoreboard] ------ :: LW INSTRUCTION  :: ------
UVM_INFO mem_scoreboard.sv(231) @ 145: uvm_test_top.env.mem_scb [mem_scoreboard] - [ Test passed ]
UVM_INFO mem_scoreboard.sv(232) @ 145: uvm_test_top.env.mem_scb [mem_scoreboard] Result is as Expected
DataMem:        16
RegMem address: 14
RegMem  Data:   16
UVM_INFO mem_scoreboard.sv(199) @ 175: uvm_test_top.env.mem_scb [mem_scoreboard] ------ :: BNE INSTRUCTION  :: ------
UVM_INFO mem_scoreboard.sv(202) @ 175: uvm_test_top.env.mem_scb [mem_scoreboard] 00000017 !=  00000015
UVM_INFO mem_scoreboard.sv(204) @ 175: uvm_test_top.env.mem_scb [mem_scoreboard] - [ Test passed ]
UVM_INFO mem_scoreboard.sv(205) @ 175: uvm_test_top.env.mem_scb [mem_scoreboard] Result is as Expected
PCIn:    00000004
PCOut:   00000000
imm:     0003
JUMP INSTRUCTION
UVM_INFO mem_scoreboard.sv(155) @ 205: uvm_test_top.env.mem_scb [mem_scoreboard] - [ Test passed ]
Result is as Expected
JTA = 1 PC = 1
JUMP INSTRUCTION
UVM_INFO mem_scoreboard.sv(155) @ 235: uvm_test_top.env.mem_scb [mem_scoreboard] - [ Test passed ]
Result is as Expected
JTA = 1 PC = 1
```

With UVM the summary report only shows there was an error detected and actually doesn't specify the type of instruction that makes the error. another difference is that now we get more information about the verification result that could helpful to debug errors from the architecture.

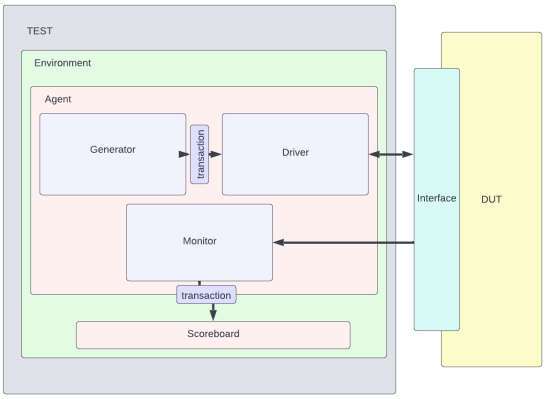
```
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :   21
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[TEST_DONE]     1
[UVM/RELNOTES]     1
[mem_scoreboard]    15
[mem_wr_rd_test]     3
```

Differences between Verilog/UVM

There are some slight differences between system Verilog verification and UVM. in both cases they bring a layer of abstraction where every component in the verification environment has a specific role, here is where we can find the main differences. For example the UVM model applies macros to provide generic utilities like transaction level modeling and component hierarchy that allows automation features like copy, print and compare. Like system verilog for verification but instead using macros we need to declare all those components.



a) UVM



b) System Verilog

Another significant difference between both verification methods is the signal generation. UVM applies a sequencer that drives an instruction from the sequence module that at the same time contains the result of the macros to generate the stimulus signals. System verilog verification is slightly different, it generates the stimulus by randomizing the transaction class, then it is sended to the driver.

In system Verilog Validation we obtain as result a list of instructions that were tested, the instruction type and the percentage of functionality given by the number of errors during the test, as well it returns every instruction tested and an error message on the scoreboard display if an error was found.

```
# KERNEL: ----------------------------------------
# KERNEL: RESUME
# KERNEL: R types Functions generated:          3
# KERNEL: I types Functions generated:         24
# KERNEL: J types Functions generated:          3
# KERNEL:
# KERNEL: ERRORS
# KERNEL: R types Functions errors:          0
# KERNEL: I types Functions errors:          0
# KERNEL: J types Functions errors:          0
# KERNEL:
# KERNEL: SUCCESFUL
# KERNEL: R types Functions succeful:        100%
# KERNEL: I types Functions succeful:        100%
# KERNEL: J types Functions succeful:        100%
# KERNEL: -----VERIFICATION COMPLITED-----
```

a) Result resume (SV)

```
----------------------------------------
- [ Scoreboard ]
TYPE I
rs: 10100  rt: 10001  Imm: 0000000000000000
Instruccion: 00010010100100010000000000000000
----------------------------------------
BEQ INSTRUCTION
00000000 != 00000011
Error: scoreboard.sv (153): Wrong Result.
Error: PCIn:    0
Error: PCOut:   0
Error:        PC Expeced: 1 PC Actual: 0
----------------------------------------
```

b) Scoreboard error display (SV)

```
----------------------------------------
- [ Driver ]
TYPE R
rs: 10111  rt: 01000  rd: 11000  funct: 100010
Instruccion: 00000010111010001100000000100010
----------------------------------------
----------------------------------------
- [ Monitor ]
TYPE R
rs: 10111  rt: 01000  rd: 11000  funct: 100010
Instruccion: 00000000000000000000000000000000
----------------------------------------
----------------------------------------
- [ Scoreboard ]
TYPE R
rs: 10111  rt: 01000  rd: 11000  funct: 100010
Instruccion: 00000010111010001100000000100010
----------------------------------------
Operation: sub
- [ Test passed ]
Result is as Expected
 23 - 8 = 15
----------------------------------------
----------------------------------------
```

a) Test details (SV)

```
UVM_INFO mem_scoreboard.sv(244) @ 55: uvm_test_top.env.mem_scb [mem_scoreboard] ------ :: SW INSTRUCTION  :: ------
UVM_INFO mem_scoreboard.sv(247) @ 55: uvm_test_top.env.mem_scb [mem_scoreboard] - [ Test passed ]
UVM_INFO mem_scoreboard.sv(248) @ 55: uvm_test_top.env.mem_scb [mem_scoreboard] Result is as Expected
Register: 12
Data Reg: 12
Data Mem: 12
```

b) Test details (UVM)

The results are more explicit with UVM, as a report we can easily see if there was a problem during a test, it also displays every test step and even a class report that provides information about every class hierarchy.

```
----------------------------------------------------------------
Name                    Type                    Size  Value
----------------------------------------------------------------
uvm_test_top            mem_wr_rd_test          -     @341
  env                   mem_model_env           -     @354
    mem_agnt            mem_agent               -     @369
      driver            mem_driver              -     @411
        rsp_port        uvm_analysis_port       -     @430
        seq_item_port   uvm_seq_item_pull_port  -     @420
      monitor           mem_monitor             -     @388
        item_collected_port uvm_analysis_port   -     @401
      sequencer         mem_sequencer           -     @440
        rsp_export      uvm_analysis_export     -     @449
        seq_item_export uvm_seq_item_pull_imp   -     @567
        arbitration_queue array                 0     -
        lock_queue      array                   0     -
        num_last_reqs   integral                32    'd1
        num_last_rsps   integral                32    'd1
    mem_scb             mem_scoreboard          -     @378
      item_collected_export uvm_analysis_imp    -     @584
----------------------------------------------------------------
```

a) Class hierarchy (UVM)

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

** Report counts by severity
UVM_INFO :   21
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[TEST_DONE]     1
[UVM/RELNOTES]     1
[mem_scoreboard]    15
[mem_wr_rd_test]     3
```

b) Report summary (UVM)

Recommendations

Each of the methods have a common goal, and that is verification. For this reason some of the advice to take into account lies in the needs of each individual, since in some ways Systemverilog may be simpler and more intuitive than UVM, but both are equally important depending on the needs. The essential thing is to understand the two verification processes, and you will realize that the only difference between the two processes is the compatibility and availability for understanding between various users of the verification system.

Conclusions

The verification process, whether System Verilog or UVM, has been very helpful during the course, since it was the most important stage, where essential knowledge for the understanding of the processes was obtained.

During the development of these projects we realized that, although verification in Systemverilog is simpler, the use of UVM can facilitate the verification of more complex systems that require more random instructions for their verification.

Bruno Hernandez Lopez

We managed to complete both monocycle and multicycle MIPS verifications using SystemVerilog and UVM, while doing that, we learned the differences and similarities, as well as the capabilities that they have to perform the verification not only for MIPS, but with any smaller or larger designs. Understanding all of the verification classes helped to develop the tests needed and provided a better approach of the importance of having a good verification test.

José Alberto Gómez Diaz

When performing a verification, it is very important to ensure that the signals are injected at the right time, otherwise the device under test will not behave as expected, which results in a failed test. If the information is injected and extracted properly, verification is very useful to ensure that the device under test is working correctly.

Josue isaias Gomez Cosme

Thanks to this project, I was able to understand the importance of verification in the creation process. In addition to the 2 very important paths such as SystemVerilog and UVM. Each of them is useful in different branches and fields, and they have their own qualities, but their goal is the same. Understanding how they both work opens the door to an easier way to verify the correct functioning of any program.

Christian Aaron Ortega Blanco

As a concept, the verification process is very simple to understand. I interpreted it as a method to guarantee the full functionality of something, or in this case to guarantee the right instruction functionality from a MIPS architecture. Took time realice what we need to do with the given tools, but at the end we start making progress solving one problem at a time. I'm impressed by how much we did in a couple of weeks.

References

1. José Alberto, Josué Isaías, Bruno Hernandez, Christian Ortega. (2022). Verificación multiciclo (UVM). 05/07/2022, de INAOE Sitio web: https://www.edaplayground.com/x/d2tG
2. José Alberto, Josué Isaías, Bruno Hernandez, Christian Ortega. (2022). Verificación Multiciclo (SV). 05/07/2022, de INAOE Sitio web: https://www.edaplayground.com/x/MBku
3. José Alberto, Josué Isaías, Bruno Hernandez, Christian Ortega. (2022). Verificación Monociclo (UVM). 05/07/2022, de INAOE Sitio web: https://www.edaplayground.com/x/K5kY
4. José Alberto, Josué Isaías, Bruno Hernandez, Christian Ortega. (2022). Verificación Monociclo (SV). 05/07/2022, de INAOE Sitio web: https://www.edaplayground.com/x/EyaS
5. Verification Guide. (2020). UVM tutorial for beginners. 05/07/2022, de Verification Guide Sitio web: https://verificationguide.com/uvm/uvm-tutorial/
6. Verification Guide. (2020). Memory Model TestBench Without Monitor, Agent, and Scoreboard. 05/07/2022, de Verification Guide Sitio web: https://verificationguide.com/systemverilog-examples/systemverilog-testbench-example-01/