MIPS architecture verification

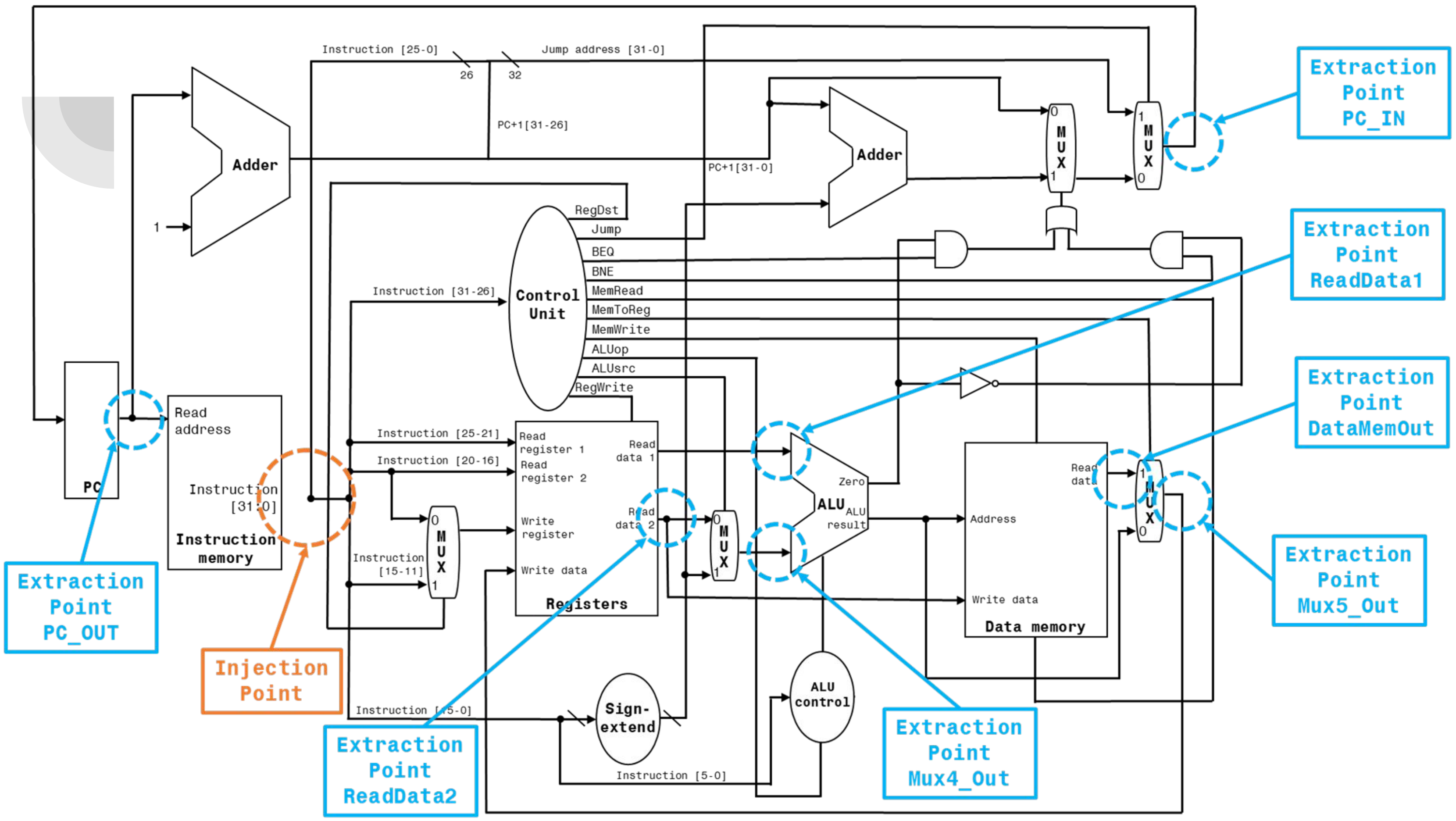
Pre-silicon Verification Spring 2022

Josue Isaias Gomez Cosme
Bruno Hernández López
Jose Alberto Gomez Diaz
Christian Aaron Ortega Blanco

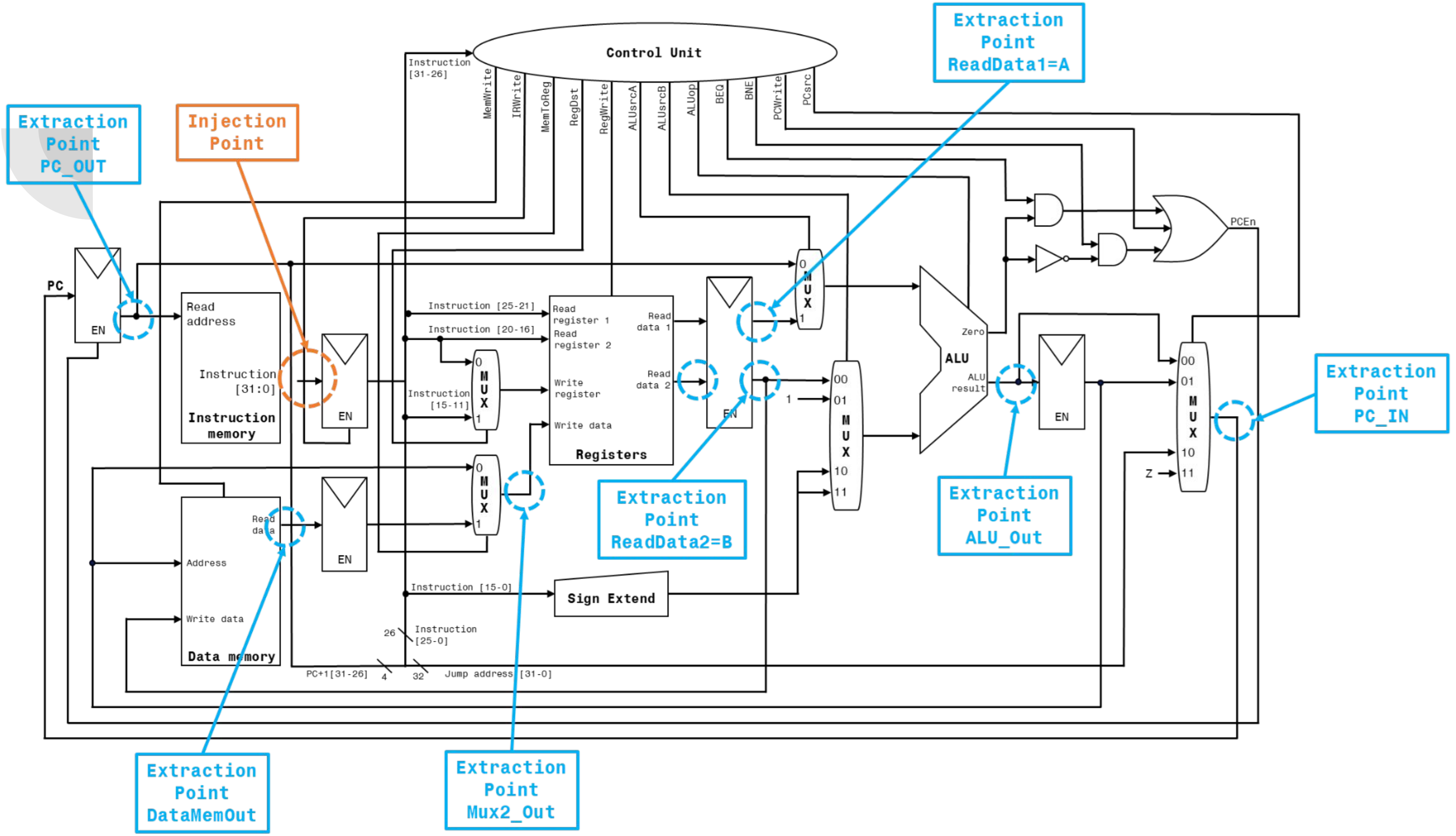
Index

Abstract

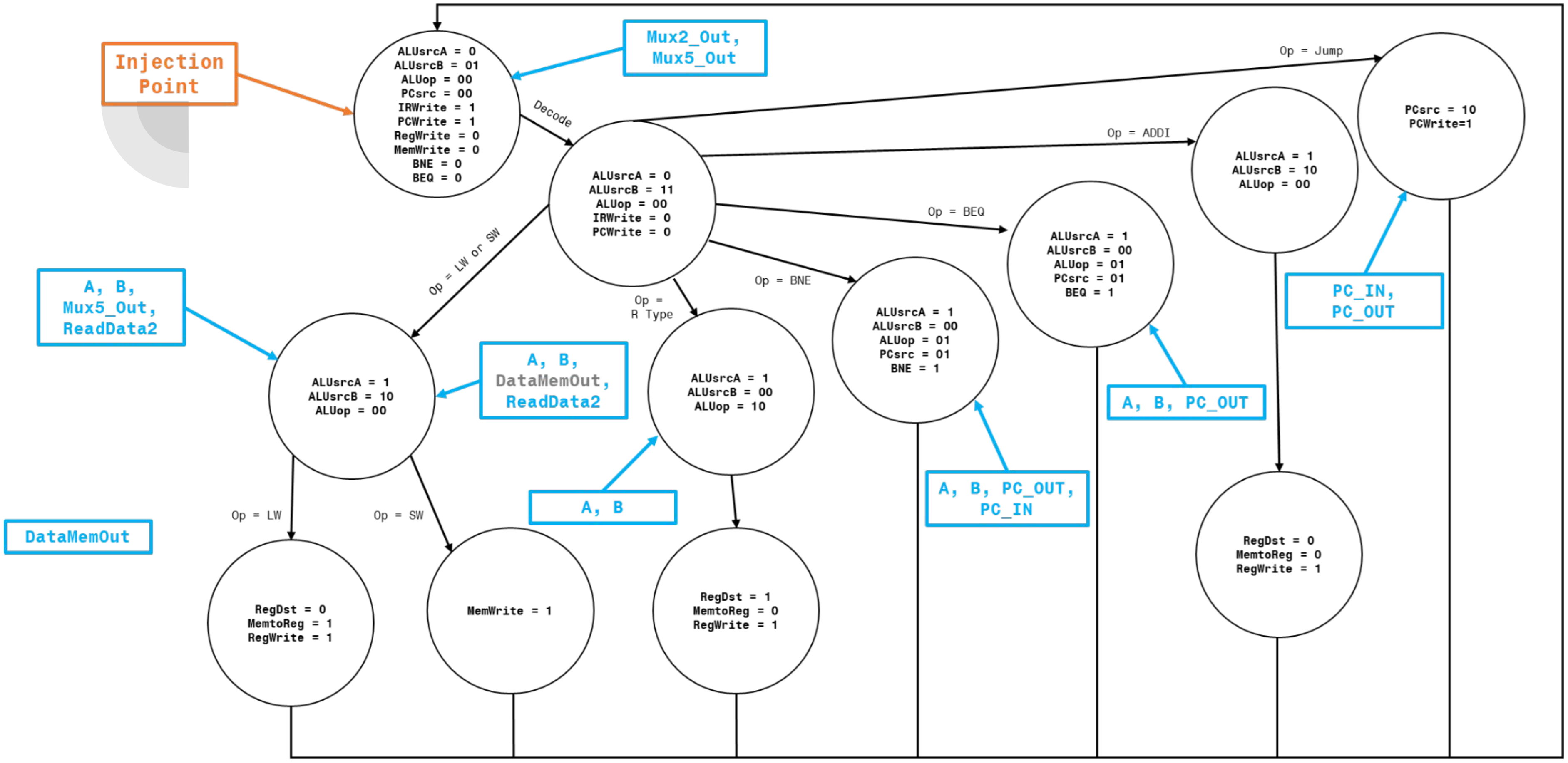
This project has been carried out to understand and analyze the operation and usefulness of verification, and how it can improve the speed at which errors are found within a supposedly finished module.



Monocycle MIPS injection / extraction points

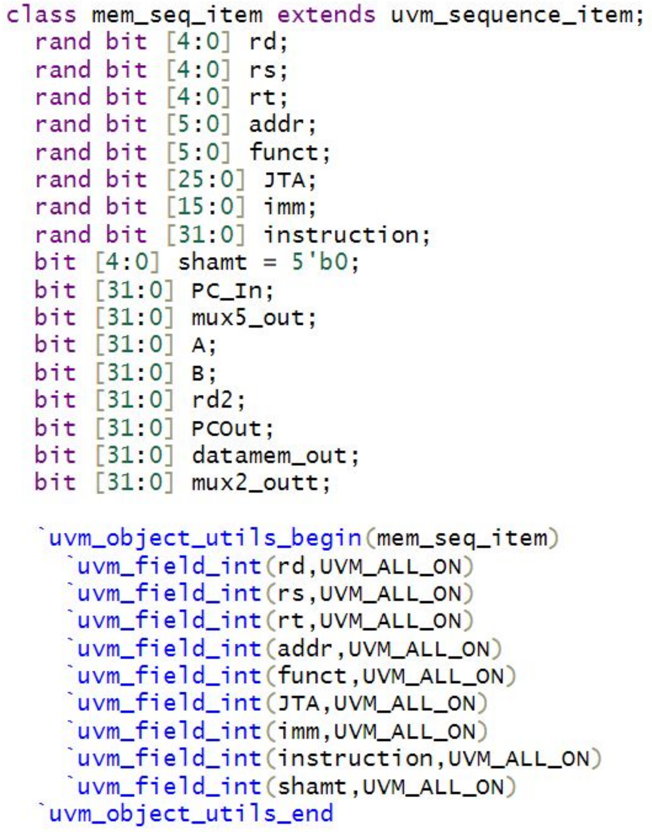


Multicycle MIPS injection / extraction points

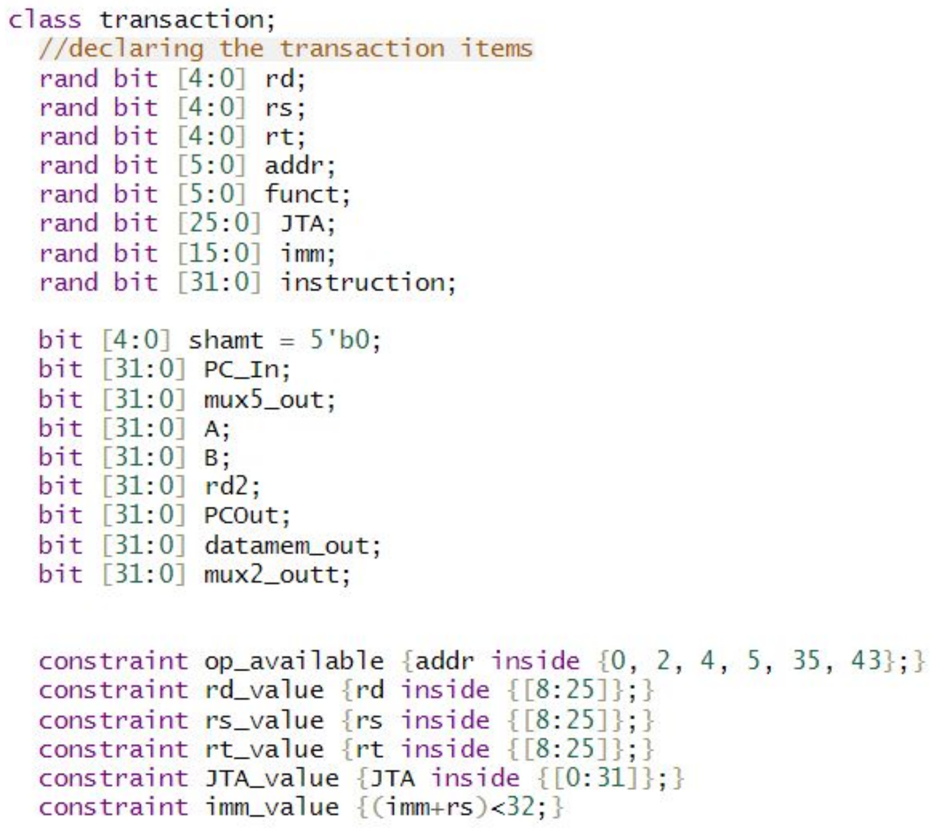


State-machine diagram injection / extraction points

Sequence Item UVM/Transaction SV

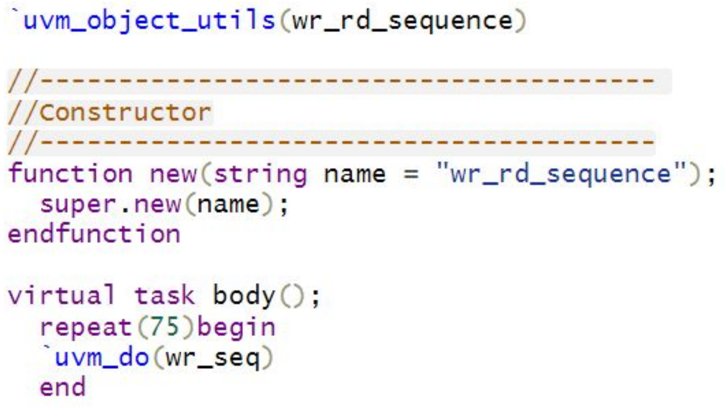


```
class mem_seq_item extends uvm_sequence_item;
  rand bit [4:0] rd;
  rand bit [4:0] rs;
  rand bit [4:0] rt;
  rand bit [5:0] addr;
  rand bit [5:0] funct;
  rand bit [25:0] JTA;
  rand bit [15:0] imm;
  rand bit [31:0] instruction;
  bit [4:0] shamt = 5'b0;
  bit [31:0] PC_In;
  bit [31:0] mux5_out;
  bit [31:0] A;
  bit [31:0] B;
  bit [31:0] rd2;
  bit [31:0] PCOut;
  bit [31:0] datamem_out;
  bit [31:0] mux2_outt;

  `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
```

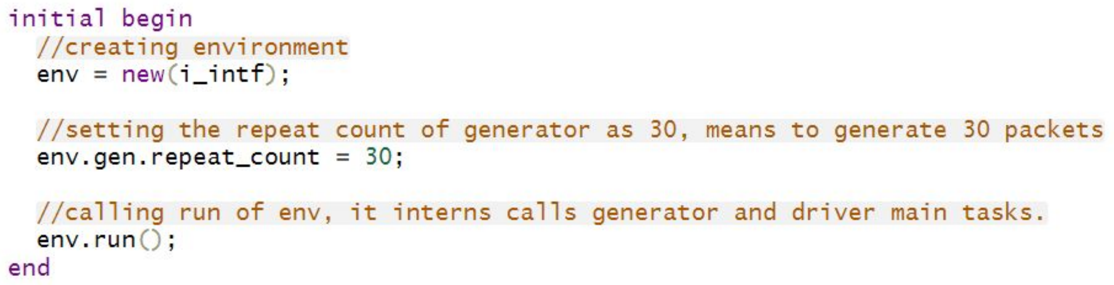


```
class transaction;
  //declaring the transaction items
  rand bit [4:0] rd;
  rand bit [4:0] rs;
  rand bit [4:0] rt;
  rand bit [5:0] addr;
  rand bit [5:0] funct;
  rand bit [25:0] JTA;
  rand bit [15:0] imm;
  rand bit [31:0] instruction;

  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;
  bit [31:0] mux2_outt;


  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;}
```

Generate multiple instructions

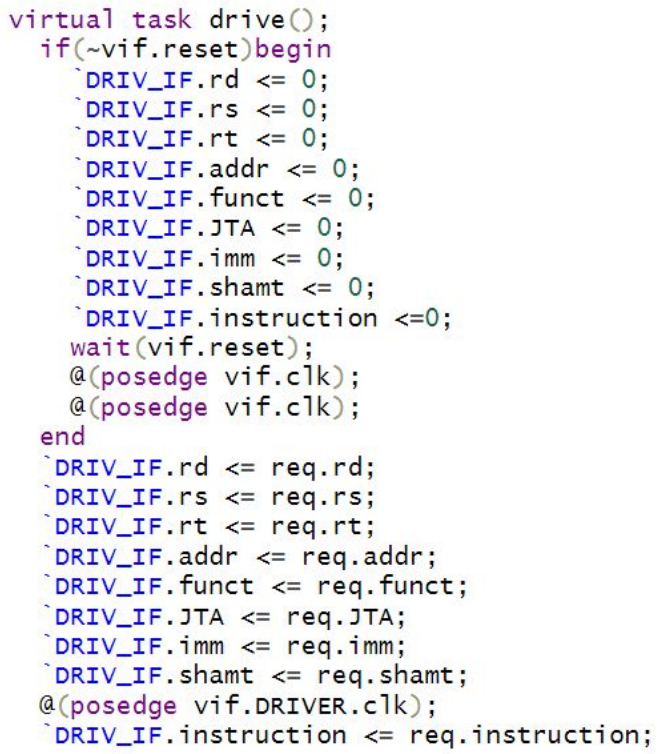


```
`uvm_object_utils(wr_rd_sequence)

//---------------------------------------
//Constructor
//---------------------------------------
function new(string name = "wr_rd_sequence");
  super.new(name);
endfunction

virtual task body();
  repeat(75)begin
   `uvm_do(wr_seq)
  end
```

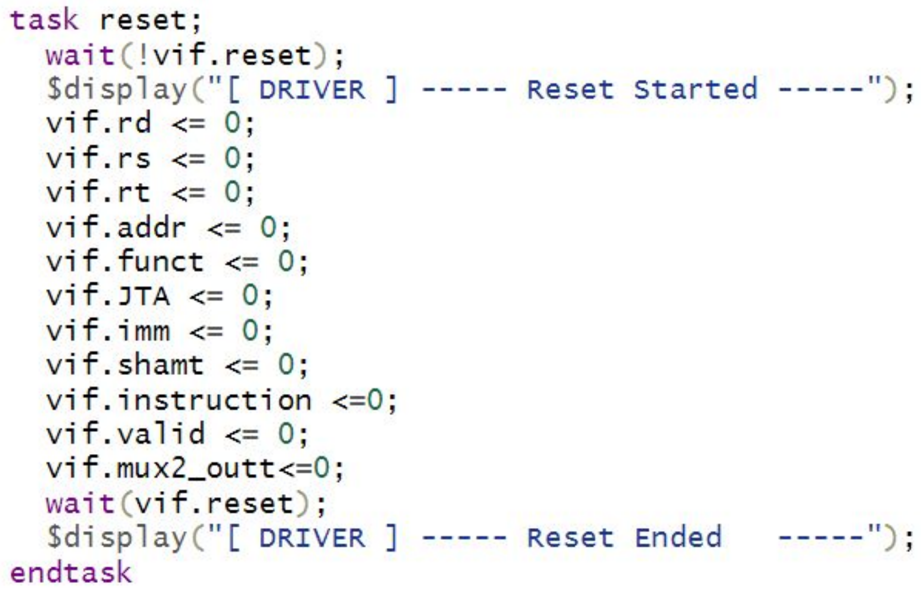


```
initial begin
  //creating environment
  env = new(i_intf);

  //setting the repeat count of generator as 30, means to generate 30 packets
  env.gen.repeat_count = 30;

  //calling run of env, it interns calls generator and driver main tasks.
  env.run();
end
```

Reset

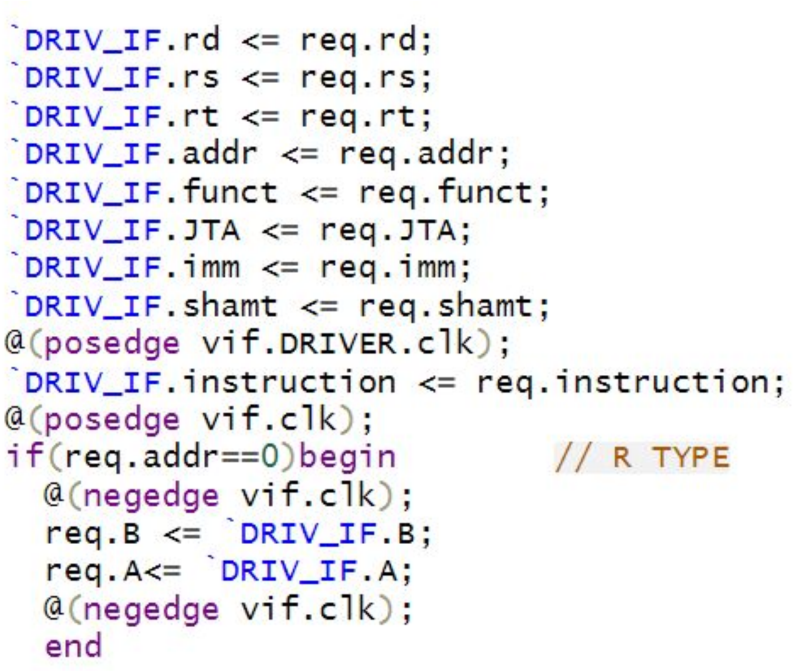


```
virtual task drive();
  if(~vif.reset)begin
    `DRIV_IF.rd <= 0;
    `DRIV_IF.rs <= 0;
    `DRIV_IF.rt <= 0;
    `DRIV_IF.addr <= 0;
    `DRIV_IF.funct <= 0;
    `DRIV_IF.JTA <= 0;
    `DRIV_IF.imm <= 0;
    `DRIV_IF.shamt <= 0;
    `DRIV_IF.instruction <=0;
    wait(vif.reset);
    @(posedge vif.clk);
    @(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;
```

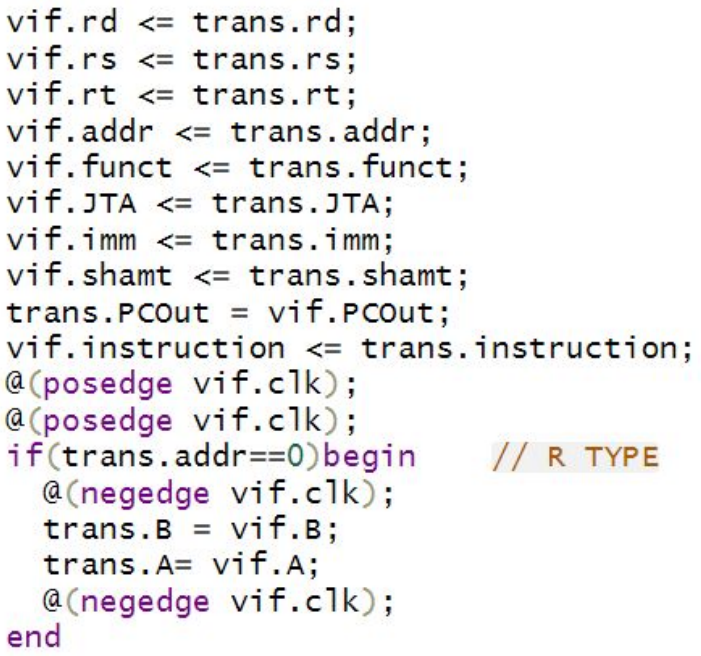


```
task reset;
  wait(!vif.reset);
  $display("[ DRIVER ] ----- Reset Started -----");
  vif.rd <= 0;
  vif.rs <= 0;
  vif.rt <= 0;
  vif.addr <= 0;
  vif.funct <= 0;
  vif.JTA <= 0;
  vif.imm <= 0;
  vif.shamt <= 0;
  vif.instruction <=0;
  vif.valid <= 0;
  vif.mux2_outt<=0;
  wait(vif.reset);
  $display("[ DRIVER ] ----- Reset Ended   -----");
endtask
```

Injection



```
`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
```



```
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
```

Monitor

Class declaration

Virtual Interface

Transactions

Constructors

Tasks

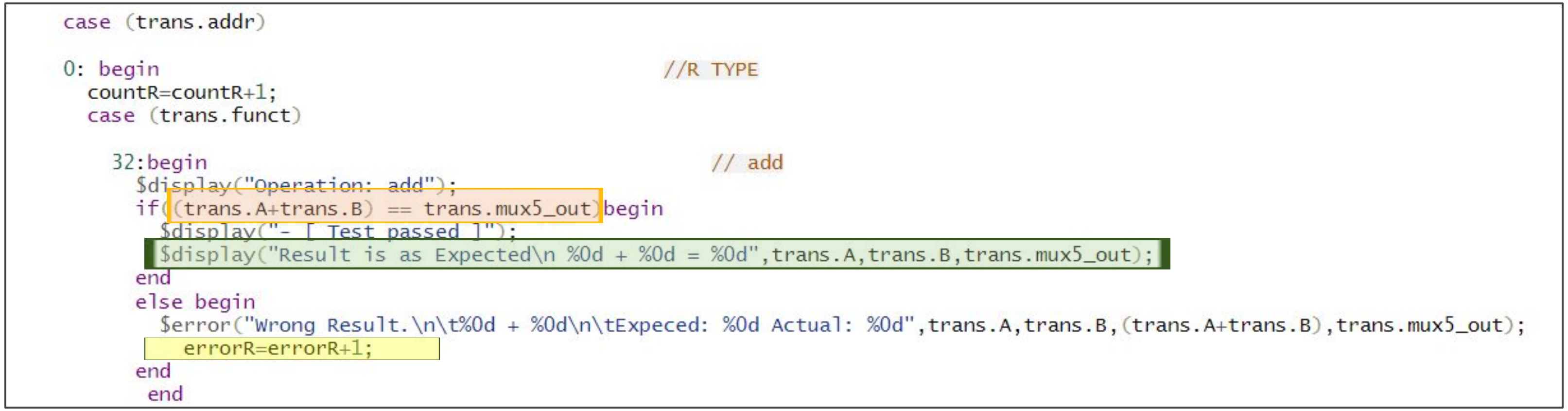
```
class monitor;
  virtual intf vif;
  mailbox mon2scb;

  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
```

SystemVerilog multicycle 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
      @(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
```

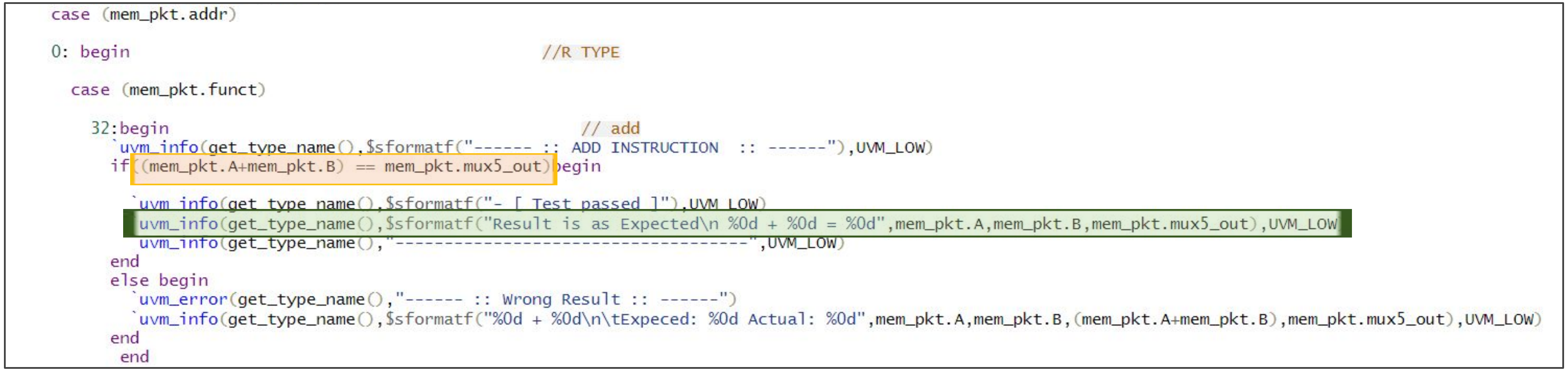
UVM monocycle Monitor

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
```

SystemVerilog scoreboard



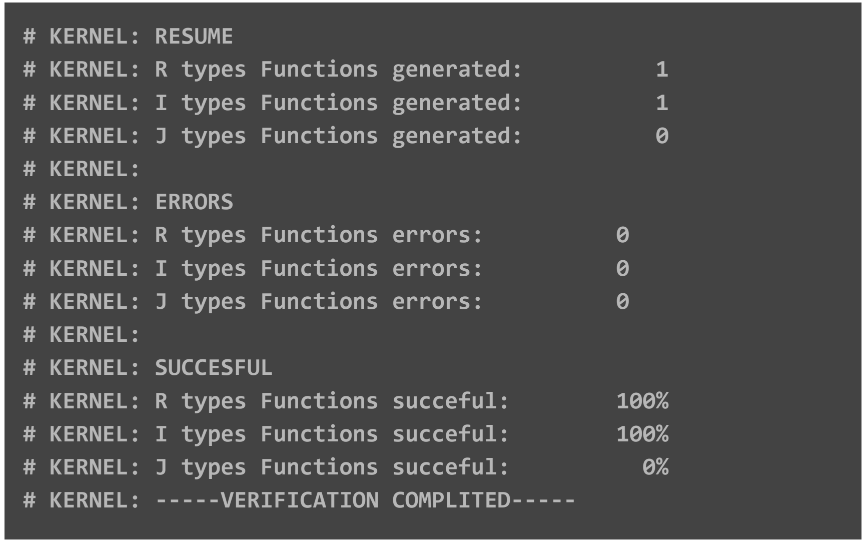
```
case (mem_pkt.addr)

0: begin                                        //R TYPE

  case (mem_pkt.funct)

    32:begin                                       // add
      `uvm_info(get_type_name(),$sformatf("------ :: ADD INSTRUCTION  :: ------"),UVM_LOW)
      if((mem_pkt.A+mem_pkt.B) == mem_pkt.mux5_out)begin

        `uvm_info(get_type_name(),$sformatf("- [ Test passed ]"),UVM_LOW)
        `uvm_info(get_type_name(),$sformatf("Result is as Expected\n %0d + %0d = %0d",mem_pkt.A,mem_pkt.B,mem_pkt.mux5_out),UVM_LOW)
        `uvm_info(get_type_name(),"------------------------------------",UVM_LOW)
      end
      else begin
        `uvm_error(get_type_name(),"------ :: Wrong Result :: ------")
        `uvm_info(get_type_name(),$sformatf("%0d + %0d\n\tExpeced: %0d Actual: %0d",mem_pkt.A,mem_pkt.B,(mem_pkt.A+mem_pkt.B),mem_pkt.mux5_out),UVM_LOW)
      end
       end
```

UVM scoreboard

Results:

monocycle & multicycle verification visualization (System Verilog)

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.

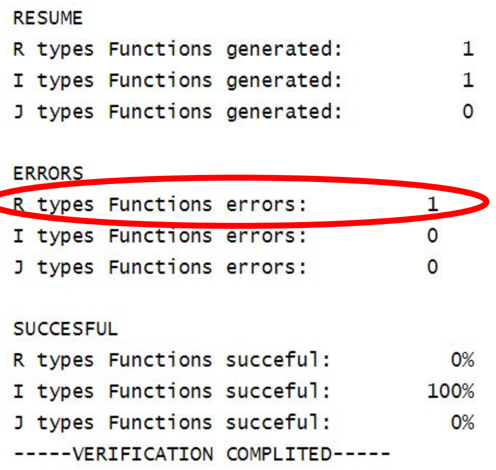


```
# KERNEL: RESUME
# KERNEL: R types Functions generated:          1
# KERNEL: I types Functions generated:          1
# KERNEL: J types Functions generated:          0
# KERNEL:
# KERNEL: ERRORS
# KERNEL: R types Functions errors:          0
# KERNEL: I types Functions errors:          0
# KERNEL: 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:          0%
# KERNEL: -----VERIFICATION COMPLITED-----
```

Results:

monocycle & multicycle error visualization (System Verilog)

If there's an instruction that could not execute successfully or the result is invalid, it's because an error on the DUT module, in order to find which is the origin of the error just look the summary of the verification result and the instruction specifications.

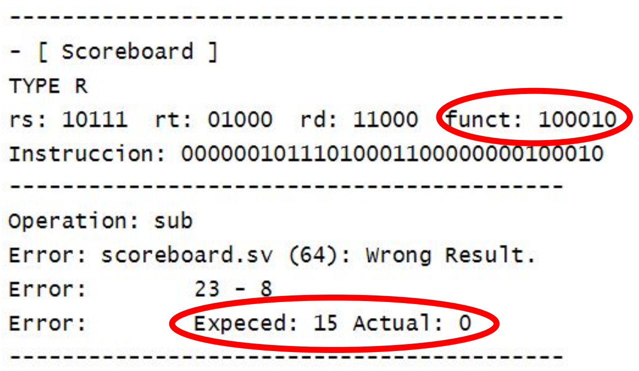


```
RESUME
R types Functions generated:          1
I types Functions generated:          1
J types Functions generated:          0

ERRORS
R types Functions errors:          1
I types Functions errors:          0
J types Functions errors:          0

SUCCESFUL
R types Functions succeful:          0%
I types Functions succeful:        100%
J types Functions succeful:          0%
-----VERIFICATION COMPLITED-----
```

Results:

monocycle & multicycle error visualization (System Verilog)

Now it's easier to see that the problem is located on the execution of a R-type instruction, and as part of the results we can see the specific function that induce the error on the validation



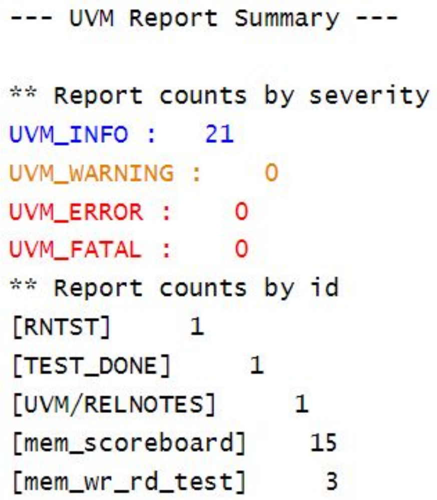
```
------------------------------------------
- [ Scoreboard ]
TYPE R
rs: 10111  rt: 01000  rd: 11000  funct: 100010
Instruccion: 00000010111010001100000000100010
------------------------------------------
Operation: sub
Error: scoreboard.sv (64): Wrong Result.
Error:        23 - 8
Error:        Expeced: 15 Actual: 0
------------------------------------------
```

in this example the error is induced by the subtract function, and seems like the operation is not performed.

Results:

monocycle & multicycle verification visualization (UVM)

UVM has a different testbench architecture, pretty much the same for some components. the main difference on the results display is that now we have a lot more information of the evaluated modules and test bench elements.

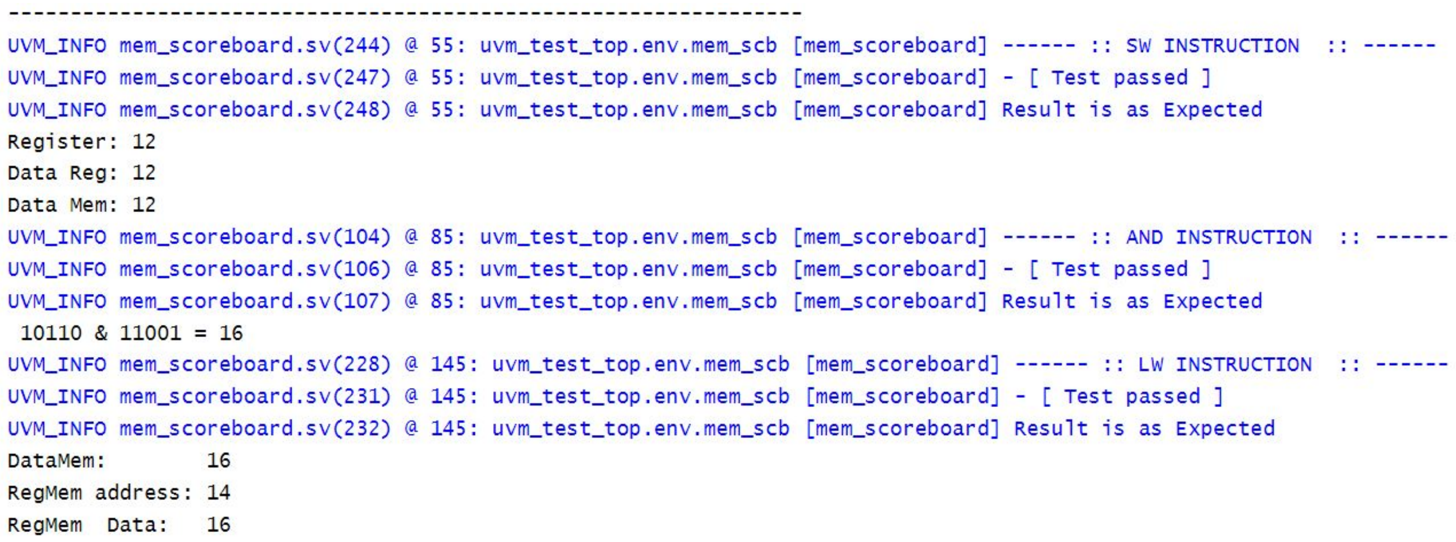


```
--- UVM Report Summary ---

** 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
```

Results:

monocycle & multicycle verification visualization (UVM)

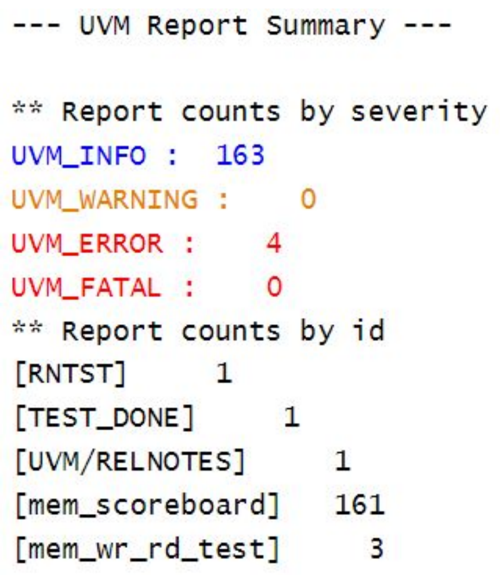
Alike the previous verification, we also obtain detailed information of each instruction given by the Sequencer element.



```
----------------------------------------------------------------
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
```

Results:
monocycle & multicycle error visualization (UVM)

Errors and warning report helps to verify if there's a critical malfunction of the architecture, but it doesn't show where the problems are.

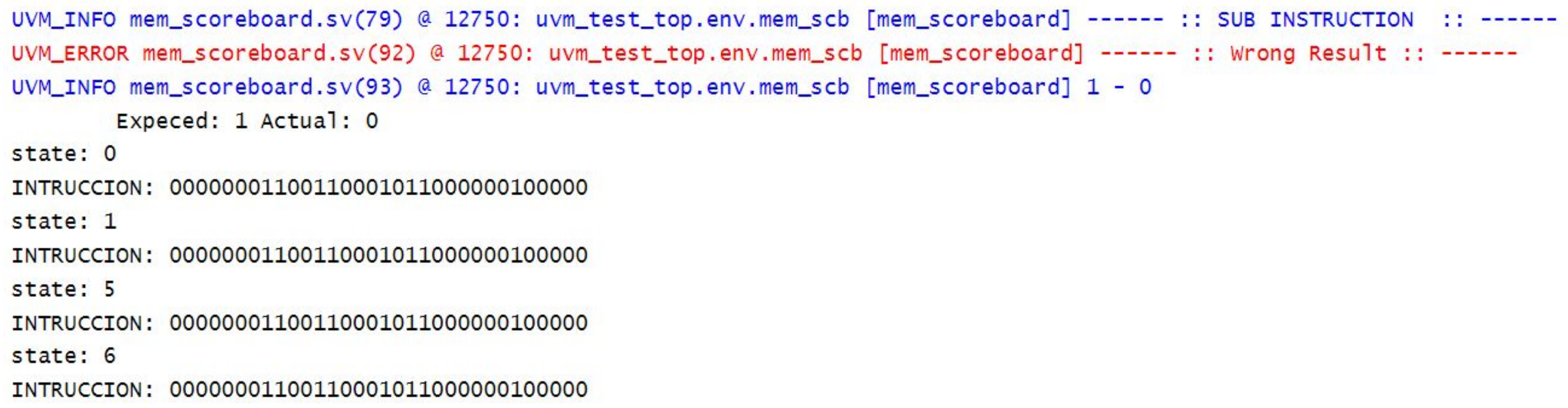


```
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :  163
UVM_WARNING :    0
UVM_ERROR :    4
UVM_FATAL :    0
** Report counts by id
[RNTST]     1
[TEST_DONE]     1
[UVM/RELNOTES]     1
[mem_scoreboard]   161
[mem_wr_rd_test]     3
```

Results:

monocycle & multicycle error visualization (UVM)

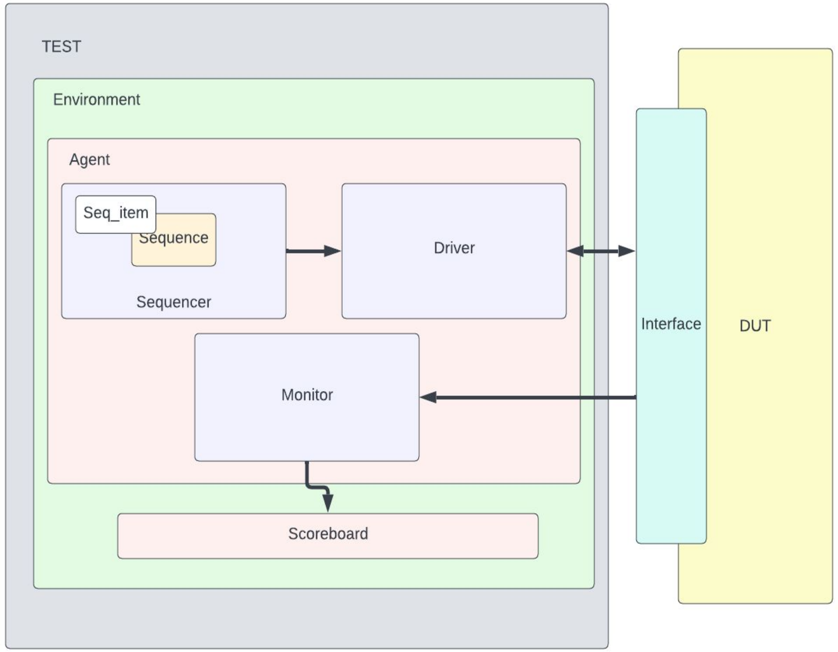
The missfuntion could be attached to the datapath of the instructions that hasn't pass the test.



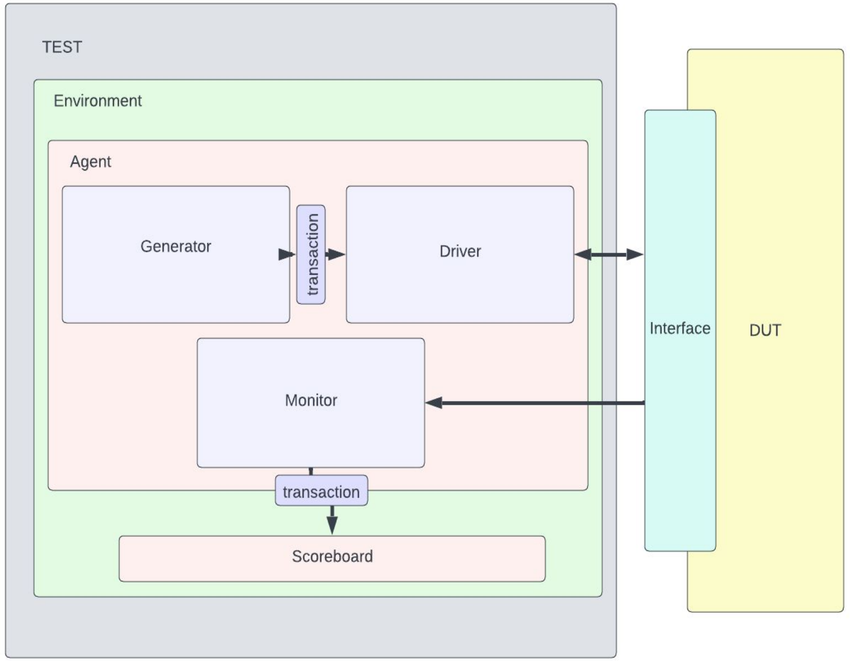
```
UVM_INFO mem_scoreboard.sv(79) @ 12750: uvm_test_top.env.mem_scb [mem_scoreboard] ------ :: SUB INSTRUCTION  :: ------
UVM_ERROR mem_scoreboard.sv(92) @ 12750: uvm_test_top.env.mem_scb [mem_scoreboard] ------ :: Wrong Result :: ------
UVM_INFO mem_scoreboard.sv(93) @ 12750: uvm_test_top.env.mem_scb [mem_scoreboard] 1 - 0
        Expeced: 1 Actual: 0
state: 0
INTRUCCION: 00000001100110001011000000100000
state: 1
INTRUCCION: 00000001100110001011000000100000
state: 5
INTRUCCION: 00000001100110001011000000100000
state: 6
INTRUCCION: 00000001100110001011000000100000
```

*this example has a deliberate missfuntion on the subtract operation in the ALU module.

Differences between Systemverilog and UVM



a) UVM



b) System Verilog

Differences between Systemverilog and UVM

Transaction class (system verilog):

fields required to generate the stimulus signals, here are declared the randomization parameter for each signal.

Sequence Item (UVM):

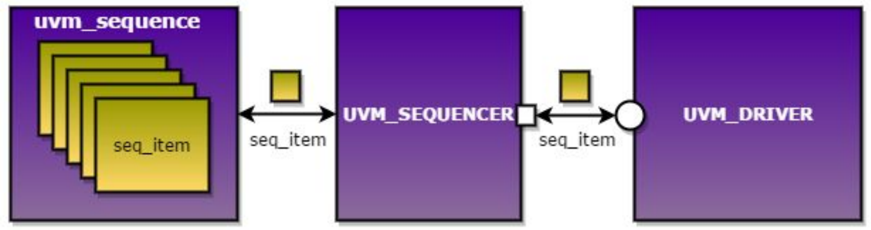
data fields (macros) required to generate the stimulus signals, here are declared the randomization parameter for each signal.

Differences between Systemverilog and UVM

Generator (system verilog):
Generate the stimulus by randomizing the transaction class and sends the randomized class to driver.

Sequence (UVM):
Generates a series of sequence_item's

Sequencer (UVM):
Controls the flow of request and response items between sequences and the driver



UVM Sequence

Recommendations

- Always keep in mind the purpose of the verification
- Before coding, determine the injection and extraction points, as well as the time where you need to extract them
- Be aware of the advantages and disadvantages while using UVM or SystemVerilog.
- Make it easier to understand for others

Conclusions

We determined the injection and extraction points from both MIPS in order to perform the verification process.

We compared the differences between SystemVerilog Verification and UVM

By introducing errors into the MIPS design, we were able to interpret instruction errors and prepare reports with the information of the problem.

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/

Any questions?

Josue Isaias Gomez Cosme - josue.gomez@inaoe.mx

Bruno Hernández López - bruno.hernandez@inaoe.mx

Jose Alberto Gomez Diaz - alberto.gomez@inaoe.mx

Christian Aaron Ortega Blanco - christian.ortega@inaoe.mx