**Abdelrahman Mohamed Ali**

**UVM FIFO**

12 May, 2024

|  |  |
| --- | --- |
| **Gmail:** | [abdelrahmansalby23@gmail.com](mailto:abdelrahmansalby23@gmail.com) |
| **LinkedIn:** | [Abdelrahman Mohamed](https://www.linkedin.com/in/abdelrahman-mohamed-814a9022a/) |
| **GitHub:** | [Abdelrahman1810](https://github.com/Abdelrahman1810?tab=repositories) |

**Bug report**

|  |  |  |  |
| --- | --- | --- | --- |
| Label | Where in code | Explain | How FIX |
| invalid\_opcode | Line 23 | opcode\_reg[3]  is out of range | Replace opcode\_reg[3]  With opcode\_reg[1] |
| Blinking Leds | Line42 | Bypass is lead signal,  If Bypass then we ignore invalid Cases | The condition of (if):  Bug: invalid  FIX: invalid and not bupass |
| Checking invaled for out - bypass | Line 56 🡪 58 | Bypass is lead signal,  If Bypass then we ignore invalid Cases | Checking for bypass first |
| Case to check opcode | Line 65 | Case should check register opcode not input opcode | case(opcode\_reg), instead of:  case(opcode) |
| opcode == 3’h0 | Line 68 🡪 74 | When opcode==0,  Should OR not AND | Use (|) instead of (&) |
| opcode == 3’h1 | Line 78 🡪 84 | When opcode==1,  Should XOR not OR | Use (^) instead of (|) |

|  |
| --- |
| assign invalid\_opcode = opcode\_reg[2] & opcode\_reg[3]; // Bug  assign invalid\_opcode = opcode\_reg[2] & opcode\_reg[0]; // Fix |
| if (invalid) // Bug      if (invalid && !(bypass\_A\_reg || bypass\_B\_reg)) // FIX |
| if (invalid)  // Bug          out <= 0; // Bug      else if (bypass\_A\_reg && bypass\_B\_reg)        out <= (INPUT\_PRIORITY == "A")? A\_reg: B\_reg;      else if (bypass\_A\_reg)        out <= A\_reg;      else if (bypass\_B\_reg)        out <= B\_reg;      if (bypass\_A\_reg && bypass\_B\_reg)        out <= (INPUT\_PRIORITY == "A")? A\_reg: B\_reg;      else if (bypass\_A\_reg)        out <= A\_reg;      else if (bypass\_B\_reg)        out <= B\_reg;      else if (invalid)  // FIX          out <= 0; // FIX |
| case (opcode) // Bug          case (opcode\_reg) // FIX |
| &A\_reg: &B\_reg; // Bug  out <= &A\_reg; // Bug   out <= &B\_reg; // Bug  out <= A\_reg & B\_reg; // Bug  ---------------  |A\_reg: |B\_reg; // FIX  out <= |A\_reg; // FIX  out <= |B\_reg; // FIX  out <= A\_reg | B\_reg; // FIX  --------------- |
| |A\_reg: |B\_reg; // Bug  out <= |A\_reg; // Bug  out <= |B\_reg; // Bug  out <= A\_reg | B\_reg; // Bug  ^A\_reg: ^B\_reg; // Fix  out <= ^A\_reg; // Fix  out <= ^B\_reg; // Fix  out <= A\_reg ^ B\_reg; // Fix |

**/// Do file**

vlib work

vlog -coveropt 3 +cover +acc {shared\_pkg\shared\_pkg.sv}

###

vlog -coveropt 3 +cover +acc {Interface\interface.sv}

vlog -coveropt 3 +cover +acc {Interface\ALSU\_ref.sv}

###

vlog -coveropt 3 +cover +acc {DUT\ALSU.sv}

vlog -coveropt 3 +cover +acc {DUT\assertion.sv}

###

vlog -coveropt 3 +cover +acc {objects\configration.sv}

vlog -coveropt 3 +cover +acc {objects\sequence\_Items\sequenceItem.sv}

vlog -coveropt 3 +cover +acc {objects\sequence\_Items\sequenceItem\_Valid.sv}

vlog -coveropt 3 +cover +acc {objects\ALSU\_sequence\ALSU\_reset\_sequence.sv}

vlog -coveropt 3 +cover +acc {objects\ALSU\_sequence\ALSU\_main\_sequence.sv}

###

vlog -coveropt 3 +cover +acc {UVM\Test\env\agent\driver\driver.sv}

vlog -coveropt 3 +cover +acc {UVM\Test\env\agent\monitor\monitor.sv}

vlog -coveropt 3 +cover +acc {UVM\Test\env\agent\sequencer\sequencer.sv}

vlog -coveropt 3 +cover +acc {UVM\Test\env\agent\agent.sv}

###

vlog -coveropt 3 +cover +acc {UVM\Test\env\coverage\_collector\coverage\_collector.sv}

vlog -coveropt 3 +cover +acc {UVM\Test\env\scoreboard\scoreboard.sv}

vlog -coveropt 3 +cover +acc {UVM\Test\env\env.sv}

###

vlog -coveropt 3 +cover +acc {UVM\Test\test.sv}

vlog -coveropt 3 +cover +acc {UVM\top.sv}

#vsim -voptargs=+acc work.top -classdebug -uvmcontrol=all

vsim +UVM\_VERBOSITY=UVM\_LOW -voptargs=+acc work.top -classdebug -uvmcontrol=all -cover

add wave -position insertpoint sim:/top/intf/\*

add wave -radix hexadecimal sim:/top/intf/out

add wave -radix hexadecimal sim:/top/intf/out\_ref

add wave -color Orchid -radix hexadecimal sim:/top/intf/leds

add wave -color gold   -radix hexadecimal sim:/top/intf/leds\_ref

run -all

#quit -sim

**/// shared\_pkg**

`timescale 1ps/1ps

package shared\_pkg;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

parameter INPUT\_PRIORITY = "A";

parameter FULL\_ADDER = "ON";

typedef enum reg [2:0] {OR, XOR, ADD, MULT, SHIFT, ROTATE, INVALID\_6, INVALID\_7} opcode\_e;

parameter MAXPOS = 3;

parameter ZERO = 0;

parameter MAXNEG = -4;

parameter HOBBIT\_LOOP = 1000;

parameter DWARF\_LOOP = 10\_000;

parameter HUMAN\_LOOP = 20\_000;

parameter GIANT\_LOOP = 30\_000;

bit alwaysValid = 0;

int RESET\_DIST = 5;

endpackage

**/// interface**

`timescale 1ps/1ps

import shared\_pkg::\*;

interface ALSU\_interface(input bit clk);

logic rst, cin, red\_op\_A, red\_op\_B, bypass\_A, bypass\_B, direction, serial\_in;

opcode\_e opcode;

logic signed [2:0] A, B;

logic [15:0] leds;

logic [5:0] out;

logic [15:0] leds\_ref;

logic [5:0] out\_ref;

bit inputs\_are\_driven;

endinterface //ALSU\_interface

**/// ALSU\_ref**

import shared\_pkg::\*;

module ALSU\_ref (A, B, cin, serial\_in, red\_op\_A, red\_op\_B, opcode, bypass\_A, bypass\_B, clk, rst, direction, leds, out);

input clk, rst, cin, red\_op\_A, red\_op\_B, bypass\_A, bypass\_B, direction, serial\_in;

//input opcode\_e opcode;

input [2:0] opcode;

input signed [2:0] A, B;

output reg [15:0] leds;

output reg [5:0] out;

reg cin\_reg, red\_op\_A\_reg, red\_op\_B\_reg, bypass\_A\_reg, bypass\_B\_reg, direction\_reg, serial\_in\_reg;

reg [2:0] A\_reg, B\_reg;

reg [2:0] opcode\_reg;

wire invalid, invalid\_c1, invalid\_c2;

assign invalid\_c1 = (opcode\_reg == 6 || opcode\_reg == 7);

assign invalid\_c2 = (red\_op\_A\_reg || red\_op\_B\_reg) && (opcode\_reg != 0 && opcode\_reg != 1);

assign invalid = invalid\_c1 || invalid\_c2;

always @(posedge clk or posedge rst) begin

    if (rst) begin

        out <= 0;

        leds <= 0;

        cin\_reg <= 0;

        red\_op\_A\_reg <= 0;

        red\_op\_B\_reg <= 0;

        bypass\_A\_reg <= 0;

        bypass\_B\_reg <= 0;

        direction\_reg <= 0;

        serial\_in\_reg <= 0;

        opcode\_reg <= 0;

        A\_reg <= 0;

        B\_reg <= 0;

    end

    else begin

        if (invalid&&!(bypass\_A\_reg || bypass\_B\_reg)) leds <= ~leds;

        else leds <= 0;

        cin\_reg <= cin;

        red\_op\_A\_reg <= red\_op\_A;

        red\_op\_B\_reg <= red\_op\_B;

        bypass\_A\_reg <= bypass\_A;

        bypass\_B\_reg <= bypass\_B;

        direction\_reg <= direction;

        serial\_in\_reg <= serial\_in;

        opcode\_reg <= opcode;

        A\_reg <= A;

        B\_reg <= B;

        if (bypass\_A\_reg && bypass\_B\_reg)

            out <= (INPUT\_PRIORITY=="A")? A\_reg:B\_reg;

        else if (bypass\_A\_reg)

            out <= A\_reg;

        else if (bypass\_B\_reg)

            out <= B\_reg;

        else if (invalid)

            out <= 0;

        else if (opcode\_reg == 0) begin

            if (red\_op\_A\_reg && red\_op\_B\_reg)

                out <= (INPUT\_PRIORITY=="A")? (|A\_reg):(|B\_reg);

            else if (red\_op\_A\_reg)

                out <= |A\_reg;

            else if (red\_op\_B\_reg)

                out <= |B\_reg;

            else

                out <= A\_reg|B\_reg;

        end

        else if (opcode\_reg == 1) begin

            if (red\_op\_A\_reg && red\_op\_B\_reg)

                out <= (INPUT\_PRIORITY=="A")? (^A\_reg):(^B\_reg);

            else

                out <= (red\_op\_A\_reg)? (^A\_reg): (red\_op\_B\_reg)? (^B\_reg):(A\_reg^B\_reg);

        end

        else if (opcode\_reg == 2)

            out <= (FULL\_ADDER=="ON")? (A\_reg + B\_reg + cin\_reg):(A\_reg + B\_reg);

        else if (opcode\_reg == 3)

            out <= A\_reg \* B\_reg;

        else if (opcode\_reg == 4) begin

            if (direction\_reg)

                out <= {out[4:0], serial\_in\_reg};

            else

                out <= {serial\_in\_reg, out[5:1]};

        end

        else if (opcode\_reg == 5) begin

            if (direction\_reg)

                out <= {out[4:0],out[5]};

            else

                out <= {out[0],out[5:1]};

        end

    end

end

endmodule

**/// ALSU after FIX**

////////////////////////////////////////////////////////////////////////////////

// Author: Kareem Waseem

// Course: Digital Verification using SV & UVM

//

// Description: ALSU Design

//

////////////////////////////////////////////////////////////////////////////////

import shared\_pkg::\*;

module ALSU(A, B, cin, serial\_in, red\_op\_A, red\_op\_B, opcode, bypass\_A, bypass\_B, clk, rst, direction, leds, out);

input clk, rst, cin, red\_op\_A, red\_op\_B, bypass\_A, bypass\_B, direction, serial\_in;

input [2:0] opcode;

input signed [2:0] A, B;

output reg [15:0] leds;

output reg [5:0] out;

reg cin\_reg, red\_op\_A\_reg, red\_op\_B\_reg, bypass\_A\_reg, bypass\_B\_reg, direction\_reg, serial\_in\_reg;

reg [2:0] opcode\_reg, A\_reg, B\_reg;

wire invalid\_red\_op, invalid\_opcode, invalid;

assign invalid\_red\_op = (red\_op\_A\_reg | red\_op\_B\_reg) & (opcode\_reg[1] | opcode\_reg[2]);

assign invalid\_opcode = opcode\_reg[1] & opcode\_reg[2]; // Fix

assign invalid = invalid\_red\_op | invalid\_opcode;

always @(posedge clk or posedge rst) begin

  if(rst) begin

    leds <= 0;

    out <= 0;

    cin\_reg <= 0;

    red\_op\_B\_reg <= 0;

    red\_op\_A\_reg <= 0;

    bypass\_B\_reg <= 0;

    bypass\_A\_reg <= 0;

    direction\_reg <= 0;

    serial\_in\_reg <= 0;

    opcode\_reg <= 0;

    A\_reg <= 0;

    B\_reg <= 0;

  end

  else begin

    if (invalid && !(bypass\_A\_reg || bypass\_B\_reg)) // FIX

      leds <= ~leds;

    else

      leds <= 0;

    cin\_reg <= cin;

    red\_op\_B\_reg <= red\_op\_B;

    red\_op\_A\_reg <= red\_op\_A;

    bypass\_B\_reg <= bypass\_B;

    bypass\_A\_reg <= bypass\_A;

    direction\_reg <= direction;

    serial\_in\_reg <= serial\_in;

    opcode\_reg <= opcode;

    A\_reg <= A;

    B\_reg <= B;

    if (bypass\_A\_reg && bypass\_B\_reg)

      out <= (INPUT\_PRIORITY == "A")? A\_reg: B\_reg;

    else if (bypass\_A\_reg)

      out <= A\_reg;

    else if (bypass\_B\_reg)

      out <= B\_reg;

    else if (invalid)  // FIX

        out <= 0; // FIX

    else begin

        case (opcode\_reg) // FIX

          3'h0: begin

            if (red\_op\_A\_reg && red\_op\_B\_reg)

              out = (INPUT\_PRIORITY == "A")? |A\_reg: |B\_reg; // FIX

            else if (red\_op\_A\_reg)

              out <= |A\_reg; // FIX

            else if (red\_op\_B\_reg)

              out <= |B\_reg; // FIX

            else

              out <= A\_reg | B\_reg; // FIX

          end

          3'h1: begin

            if (red\_op\_A\_reg && red\_op\_B\_reg)

              out <= (INPUT\_PRIORITY == "A")? ^A\_reg: ^B\_reg; // Fix

            else if (red\_op\_A\_reg)

              out <= ^A\_reg; // Fix

            else if (red\_op\_B\_reg)

              out <= ^B\_reg; // Fix

            else

              out <= A\_reg ^ B\_reg; // Fix

          end

          3'h2: begin

            if (FULL\_ADDER == "ON")

              out <= A\_reg + B\_reg + cin\_reg;

            else

              out <= A\_reg + B\_reg;

          end

          3'h3: out <= A\_reg \* B\_reg;

          3'h4: begin

            if (direction\_reg)

              out <= {out[4:0], serial\_in\_reg};

            else

              out <= {serial\_in\_reg, out[5:1]};

          end

          3'h5: begin

            if (direction\_reg)

              out <= {out[4:0], out[5]};

            else

              out <= {out[0], out[5:1]};

          end

        endcase

    end

  end

end

endmodule

**/// assertion**

import shared\_pkg::\*;

`define reset disable iff(rst)

module ALSU\_sva(A, B, cin, serial\_in, red\_op\_A, red\_op\_B, opcode, bypass\_A, bypass\_B, clk, rst, direction, leds, out);

input bit clk;

input logic rst, cin, red\_op\_A, red\_op\_B, bypass\_A, bypass\_B, direction, serial\_in;

input [2:0] opcode;

input logic signed [2:0] A, B;

input logic [15:0] leds;

input logic [5:0] out;

////////////////////////////////////////////////////////

// this variable to make writing assertion more easer //

////////////////////////////////////////////////////////

bit isBypass, InvalidOP, bitwise;

assign isBypass = bypass\_A | bypass\_B;

assign InvalidOP = opcode==6 | opcode==7;

assign bitwise = opcode==0 | opcode==1;

assign isRed = red\_op\_B | red\_op\_A;

///////////////////////////////////

// assertion of bypass operation //

///////////////////////////////////

bypass\_A\_assert: assert property (@(posedge clk) `reset bypass\_A |-> ##2 out==$past(A,2));

bypass\_B\_assert: assert property (@(posedge clk) `reset (bypass\_B && !bypass\_A) |-> ##2 out==$past(B,2));

//////////////////////////////////////

// assertion of reduction operation //

//////////////////////////////////////

sequence redValid(red, op,red2);

    (red && !isBypass && opcode==op && !red2);

endsequence

redA\_OR\_assert: assert property (@(posedge clk) `reset redValid(red\_op\_A,0,0)        |-> ##2 out==|($past(A,2)));

redB\_OR\_assert: assert property (@(posedge clk) `reset redValid(red\_op\_B,0,red\_op\_A) |-> ##2 out==|($past(B,2)));

redA\_XR\_assert: assert property (@(posedge clk) `reset redValid(red\_op\_A,1,0)        |-> ##2 out==^($past(A,2)));

redB\_XR\_assert: assert property (@(posedge clk) `reset redValid(red\_op\_B,1,red\_op\_A) |-> ##2 out==^($past(B,2)));

////////////////////////////////

// assertion of invalid cases //

////////////////////////////////

sequence Invalid\_seq;

    ((isRed && !isBypass && !bitwise) || (InvalidOP && !isBypass));

endsequence

Invalid\_assert: assert property (@(posedge clk) `reset Invalid\_seq |-> ##[0:2](out==0 && leds=='hffff));

////////////////////////////////////////

// assertion when opcode always valid //

////////////////////////////////////////

always\_comb begin : assert\_alwaysValid

  if (alwaysValid) begin

    alwValid\_opcode\_assert:  assert final (!InvalidOP);

    alwZero\_bypass\_A\_assert: assert final (!bypass\_A);

    alwZero\_bypass\_B\_assert: assert final (!bypass\_B);

    alwZero\_red\_op\_A\_assert: assert final (!red\_op\_A);

    alwZero\_red\_op\_B\_assert: assert final (!red\_op\_B);

  end

end

/////////////////////

// reset assertion //

/////////////////////

always\_comb begin : reset\_assertion

    if (rst) begin

        rst\_out\_assert: assert final (out==0);

        rst\_leds\_assert: assert final (leds==0);

    end

end

//////////////////////////////////////////////////////////////////

// assertion for opcode Valid cases without bypass or reduction //

//////////////////////////////////////////////////////////////////

sequence sh\_ro(op, dir);

    (opcode==op && dir && !isRed && !isBypass);

endsequence

sequence op\_assert(op);

  (!isBypass && !isRed && opcode==op);

endsequence

OR\_assert:   assert property (@(posedge clk) `reset op\_assert(0) |-> ##2  out== $past(A,2) | $past(B,2) );

XOR\_assert:  assert property (@(posedge clk) `reset op\_assert(1) |-> ##2  out==($past(A,2)^$past(B,2)) );

add\_assert:  assert property (@(posedge clk) `reset op\_assert(2) |-> ##2  out==($past(A,2)+$past(B,2)+$past(cin,2)) );

mult\_assert: assert property (@(posedge clk) `reset op\_assert(3) |-> ##2  out==($past(A,2)\*$past(B,2)) );

shiftL\_assert: assert property (@(posedge clk) `reset sh\_ro(4, direction) |-> ##2  out=={$past(out[4:0]), $past(serial\_in,2)} );

shiftR\_assert: assert property (@(posedge clk) `reset sh\_ro(4, !direction) |-> ##2  out=={$past(serial\_in,2), $past(out[5:1])} );

rotateL\_assert: assert property (@(posedge clk) `reset sh\_ro(5, direction) |-> ##2  out=={$past(out[4:0]), $past(out[5])} );

rotateR\_assert: assert property (@(posedge clk) `reset sh\_ro(5, !direction) |-> ##2  out=={$past(out[0]), $past(out[5:1])} );

**/// Cover directives**

/////////////////////////////////////////////////////////////////////////////

//////////////////////////          cover          //////////////////////////

/////////////////////////////////////////////////////////////////////////////

///////////////////////////////

// cover of bypass operation //

///////////////////////////////

bypass\_A\_cover: cover property (@(posedge clk) `reset bypass\_A |-> ##2 out==$past(A,2));

bypass\_B\_cover: cover property (@(posedge clk) `reset (bypass\_B && !bypass\_A) |-> ##2 out==$past(B,2));

//////////////////////////////////

// cover of reduction operation //

//////////////////////////////////

redA\_OR\_cover: cover property (@(posedge clk) `reset redValid(red\_op\_A,0,0)        |-> ##2 out==|($past(A,2)));

redB\_OR\_cover: cover property (@(posedge clk) `reset redValid(red\_op\_B,0,red\_op\_A) |-> ##2 out==|($past(B,2)));

redA\_XR\_cover: cover property (@(posedge clk) `reset redValid(red\_op\_A,1,0)        |-> ##2 out==^($past(A,2)));

redB\_XR\_cover: cover property (@(posedge clk) `reset redValid(red\_op\_B,1,red\_op\_A) |-> ##2 out==^($past(B,2)));

////////////////////////////

// cover of invalid cases //

////////////////////////////

Invalid\_cover: cover property (@(posedge clk) `reset Invalid\_seq |-> ##[0:2](out==0 && leds=='hffff));

////////////////////////////////////

// cover when opcode always valid //

////////////////////////////////////

always\_comb begin : cover\_alwaysValid

  if (alwaysValid) begin

    alwValid\_opcode\_cover:  cover final (!InvalidOP);

    alwZero\_bypass\_A\_cover: cover final (!bypass\_A);

    alwZero\_bypass\_B\_cover: cover final (!bypass\_B);

    alwZero\_red\_op\_A\_cover: cover final (!red\_op\_A);

    alwZero\_red\_op\_B\_cover: cover final (!red\_op\_B);

  end

end

/////////////////

// reset cover //

/////////////////

always\_comb begin : reset\_cover

    if (rst) begin

        rst\_out\_cover: cover final (out==0);

        rst\_leds\_cover: cover final (leds==0);

    end

end

//////////////////////////////////////////////////////////////

// cover for opcode Valid cases without bypass or reduction //

//////////////////////////////////////////////////////////////

OR\_cover:   cover property (@(posedge clk) `reset op\_assert(0) |-> ##2  out== $past(A,2) | $past(B,2) );

XOR\_cover:  cover property (@(posedge clk) `reset op\_assert(1) |-> ##2  out==($past(A,2) ^ $past(B,2)) );

add\_cover:  cover property (@(posedge clk) `reset op\_assert(2) |-> ##2  out==($past(A,2) + $past(B,2) + $past(cin,2)) );

mult\_cover: cover property (@(posedge clk) `reset op\_assert(3) |-> ##2  out==($past(A,2) \* $past(B,2)) );

shiftL\_cover: cover property (@(posedge clk) `reset sh\_ro(4, direction) |-> ##2  out=={$past(out[4:0]), $past(serial\_in,2)} );

shiftR\_cover: cover property (@(posedge clk) `reset sh\_ro(4, !direction) |-> ##2  out=={$past(serial\_in,2), $past(out[5:1])} );

rotateL\_cover: cover property (@(posedge clk) `reset sh\_ro(5, direction) |-> ##2  out=={$past(out[4:0]), $past(out[5])} );

rotateR\_cover: cover property (@(posedge clk) `reset sh\_ro(5, !direction) |-> ##2  out=={$past(out[0]), $past(out[5:1])} );

endmodule

**/// configuration**

`timescale 1ps/1ps

package config\_pkg;

import shared\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

// Configration class

class ALSU\_config extends uvm\_object;

    `uvm\_object\_utils(ALSU\_config)

    virtual ALSU\_interface v\_if;

    function new(string name = "ALSU\_config");

        super.new(name);

    endfunction //new()

endclass //ALSU\_config

endpackage

**/// sequenceItem (Valid and Invalid)**

package sequenceItem\_pkg;

import shared\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name, this);

// Sequence Item class Valid and Invalid

class ALSU\_sequenceItem extends uvm\_sequence\_item;

    `uvm\_object\_utils(ALSU\_sequenceItem)

    logic clk;

    rand logic rst, cin, red\_op\_A, red\_op\_B, bypass\_A, bypass\_B, direction, serial\_in;

    rand opcode\_e opcode;

    rand logic signed [2:0] A, B;

    logic [15:0] leds;

    logic [5:0] out;

    logic [15:0] leds\_ref;

    logic [5:0] out\_ref;

    function new(string name = "ALSU\_sequenceItem");

        super.new(name);

    endfunction //new()

    // built in function

    function string convert2string();

        return $sformatf("%s, rst = %0d, A = %0d, B = %0d cin = %0d, red\_op\_A = %0d, red\_op\_B = %0d, bypass\_A = %0d, bypass\_B = %0d,

        direction = %0d, serial\_in = %0d \n out = %0d, out\_ref = 0%0d ",

        super.convert2string(), rst, A, B, cin, red\_op\_A, red\_op\_B, bypass\_A, bypass\_B, direction, serial\_in, out, out\_ref

        );

    endfunction

    // my function to print ALSU inputs and outputs

    function string convert2string\_stim();

        return $sformatf("rst = %0d, opcode = %0s, cin = %0d, red\_op\_A = %0d, red\_op\_B = %0d, bypass\_A = %0d, bypass\_B = %0d,

        direction = %0d, serial\_in = %0d \n out = %0d",

        rst, opcode, cin, red\_op\_A, red\_op\_B, bypass\_A, bypass\_B, direction, serial\_in, out

        );

    endfunction

    /////// ////// ////// ////// //////

    //       Constraint block        //

    ////// /////// ///// /////// //////

    constraint rules1\_7 {

        //rst constraint

            rst dist {0:=100-RESET\_DIST, 1:=RESET\_DIST};

        // Invalid cases constraint

            opcode dist {INVALID\_6:=5, INVALID\_7:=5, [0:5]:/90};

        // A & B constraint when opcode is ADD or MULT

            (opcode == MULT || opcode == ADD) -> A dist {MAXPOS:=20, ZERO:=10, MAXNEG:=20, [MAXNEG+1:MAXPOS-1]:/50};

            (opcode == MULT || opcode == ADD) -> B dist {MAXPOS:=20, ZERO:=10, MAXNEG:=20, [MAXNEG+1:MAXPOS-1]:/50};

        // A & B constraint when opcode is OR or XOR and red\_op\_A is high

            ((opcode==XOR || opcode==OR) && red\_op\_A) -> A dist {3'b001:=30, 3'b010:=30, 3'b100:=30, [MAXNEG+1:MAXPOS-1]:/10};

            ((opcode==XOR || opcode==OR) && red\_op\_A) -> B == 0;

        // A & B constraint when opcode is OR or XOR and red\_op\_B is high

            ((opcode==XOR || opcode==OR) && red\_op\_B) -> A == 0;

            ((opcode==XOR || opcode==OR) && red\_op\_B) -> B dist {3'b001:=30, 3'b010:=30, 3'b100:=30, [MAXNEG+1:MAXPOS-1]:/10};

        //  Do not constraint the inputs A or B when the operation is shift or rotate

        // it's achieved by default after the 2,3 and 4 Constraint achieved

        // bypass constraint

            bypass\_A dist {0:=90, 1:=10};

            bypass\_B dist {0:=90, 1:=10};

        // red\_op constraint

            red\_op\_A dist {0:=80, 1:=20};

            red\_op\_B dist {0:=80, 1:=20};

        // A & B constraint when opcode is OR or XOR and red\_op is low

            ((opcode==XOR || opcode==OR) && ~red\_op\_A && ~red\_op\_B) -> A == ~B;

    }

    rand opcode\_e arr [6];

    constraint rules\_8 {

        foreach(arr[i])

            arr[i] inside {OR, XOR, ADD, MULT, SHIFT, ROTATE};

        unique {arr};

        //foreach(arr[i])

    }

    constraint MAKE\_bypass\_red\_rst\_ZERO {

        rst == 0;

        bypass\_A == 0; bypass\_B == 0;

        red\_op\_A == 0; red\_op\_B == 0;

    }

    constraint INPUT\_PRIORITY\_CONS {

        red\_op\_A == red\_op\_B;

        bypass\_A == bypass\_B;

    }

endclass //ALSU\_sequenceItem extends uvm\_sequence\_item

**/// sequenceItem (Valid Only)**

package sequenceItem\_valid\_pkg;

import sequenceItem\_pkg::\*;

import shared\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name, this);

// Sequence Item class Valid Cases Only

class ALSU\_sequenceItem\_valid extends ALSU\_sequenceItem;

    `uvm\_object\_utils(ALSU\_sequenceItem\_valid)

    function new(string name = "ALSU\_sequenceItem\_valid");

        super.new(name);

    endfunction //new()

    /////// ////// ////// ////// //////

    //       Constraint block        //

    ////// /////// ///// /////// //////

    constraint rules1\_7 {

    //rst constraint

        rst dist {0:=100-RESET\_DIST, 1:=RESET\_DIST};

    // make opcode always valid

        //opcode dist {INVALID\_6:=5, INVALID\_7:=5, [0:5]:/90};

        opcode inside {OR, XOR, ADD, MULT, SHIFT, ROTATE};

        // to avoid invalid cases

        if (opcode==OR||opcode==XOR)

            red\_op\_A dist {0:=90, 1:=10};

        else

            red\_op\_A == 0;

        if (opcode==OR||opcode==XOR)

            red\_op\_B dist {0:=90, 1:=10};

        else

            red\_op\_B == 0;

    // A & B constraint when opcode is ADD or MULT

        (opcode == MULT || opcode == ADD) -> A dist {MAXPOS:=20, ZERO:=10, MAXNEG:=20, [MAXNEG+1:MAXPOS-1]:/50};

        (opcode == MULT || opcode == ADD) -> B dist {MAXPOS:=20, ZERO:=10, MAXNEG:=20, [MAXNEG+1:MAXPOS-1]:/50};

    // A & B constraint when opcode is OR or XOR and red\_op\_A is high

        ((opcode==XOR || opcode==OR) && red\_op\_A) -> A dist {3'b001:=30, 3'b010:=30, 3'b100:=30, [MAXNEG+1:MAXPOS-1]:/10};

        ((opcode==XOR || opcode==OR) && red\_op\_A) -> B == 0;

    // A & B constraint when opcode is OR or XOR and red\_op\_B is high

        ((opcode==XOR || opcode==OR) && red\_op\_B) -> A == 0;

        ((opcode==XOR || opcode==OR) && red\_op\_B) -> B dist {3'b001:=30, 3'b010:=30, 3'b100:=30, [MAXNEG+1:MAXPOS-1]:/10};

    //  Do not constraint the inputs A or B when the operation is shift or rotate

    // it's achieved by default after the 2,3 and 4 Constraint achieved

    // bypass constraint

        bypass\_A dist {0:=90, 1:=10};

        bypass\_B dist {0:=90, 1:=10};

    // A & B constraint when opcode is OR or XOR and red\_op is low

        ((opcode==XOR || opcode==OR) && ~red\_op\_A && ~red\_op\_B) -> A == ~B;

    }

endclass //ALSU\_sequenceItem\_valid extends uvm\_sequence\_item

endpackage

**/// ALSU\_reset\_sequence**

`timescale 1ps/1ps

package rst\_sequence\_pkg;

import shared\_pkg::\*;

import sequenceItem\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name);

class ALSU\_reset\_sequence extends uvm\_sequence #(ALSU\_sequenceItem);

    `uvm\_object\_utils(ALSU\_reset\_sequence)

    function new(string name = "ALSU\_reset\_sequence");

        super.new(name);

    endfunction //new()

    ALSU\_sequenceItem item;

    // Main task

    task body;

        // Creat seq\_item

        item = `create\_obj(ALSU\_sequenceItem, "item")

        item.A = 0; item.B = 0; item.opcode = OR;

        item.cin = 0; item.serial\_in = 0; item.direction = 0;

        item.bypass\_A = 0; item.bypass\_B = 0;

        item.red\_op\_A = 0; item.red\_op\_B = 0;

        start\_item(item);

        item.rst = 1;

        finish\_item(item);

    endtask

endclass //ALSU\_reset\_sequence extends uvm\_sequence #(ALSU\_sequenceItem)

endpackage

**/// ALSU\_main\_sequence**

`timescale 1ps/1ps

package main\_sequence\_pkg;

import shared\_pkg::\*;

import uvm\_pkg::\*;

import sequenceItem\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name);

class ALSU\_main\_sequence extends uvm\_sequence #(ALSU\_sequenceItem);

    `uvm\_object\_utils(ALSU\_main\_sequence)

    function new(string name = "ALSU\_main\_sequence");

        super.new(name);

    endfunction //new()

    ALSU\_sequenceItem item;

    // Main task

    task body();

        repeat(GIANT\_LOOP) begin

            item = `create\_obj(ALSU\_sequenceItem, "item")  // Creat seq\_item

            // edit constraint mode

            item.constraint\_mode(0);

            item.rules1\_7.constraint\_mode(1);

            start\_item(item);

            assert (item.randomize());

            finish\_item(item);

        end

        repeat(HOBBIT\_LOOP) begin

            item = `create\_obj(ALSU\_sequenceItem, "item")  // Creat seq\_item

            // edit constraint mode

            item.constraint\_mode(0);

            item.INPUT\_PRIORITY\_CONS.constraint\_mode(1);

            start\_item(item);

            assert (item.randomize());

            finish\_item(item);

        end

        // make opcode onley valid

        alwaysValid = 1;

        repeat(DWARF\_LOOP) begin

            item = `create\_obj(ALSU\_sequenceItem, "item")  // Creat seq\_item

        // edit constraint mode

            item.constraint\_mode(0);

            item.rules\_8.constraint\_mode(1);

            item.MAKE\_bypass\_red\_rst\_ZERO.constraint\_mode(1);

        // randomization

            assert (item.randomize());

        // loop for opcode

            foreach(item.arr[i]) begin

                start\_item(item);

                item.opcode = item.arr[i];

                finish\_item(item);

            end

        end

    endtask

endclass //ALSU\_main\_sequence extends uvm\_sequence #(ALSU\_sequenceItem)

endpackage

**/// driver**

`timescale 1ps/1ps

package driver\_pkg;

import shared\_pkg::\*;

import sequenceItem\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name);

// driver class

class ALSU\_driver extends uvm\_driver #(ALSU\_sequenceItem);

    `uvm\_component\_utils(ALSU\_driver)

    virtual ALSU\_interface v\_if;

    ALSU\_sequenceItem stim\_seq\_item;

    function new(string name = "ALSU\_driver", uvm\_component parent = null);

        super.new(name, parent);

    endfunction //new()

    task run\_phase(uvm\_phase phase);

        super.run\_phase(phase);

        forever begin

            //v\_if.inputs\_are\_driven = 0;

            stim\_seq\_item = `create\_obj(ALSU\_sequenceItem, "stim\_seq\_item")

            seq\_item\_port.get\_next\_item(stim\_seq\_item);

        // assigned inputs to interface

            v\_if.A = stim\_seq\_item.A;

            v\_if.B = stim\_seq\_item.B;

            v\_if.cin = stim\_seq\_item.cin;

            v\_if.serial\_in = stim\_seq\_item.serial\_in;

            v\_if.red\_op\_A = stim\_seq\_item.red\_op\_A;

            v\_if.red\_op\_B = stim\_seq\_item.red\_op\_B;

            v\_if.bypass\_A = stim\_seq\_item.bypass\_A;

            v\_if.bypass\_B = stim\_seq\_item.bypass\_B;

            v\_if.clk = stim\_seq\_item.clk;

            v\_if.rst = stim\_seq\_item.rst;

            v\_if.direction = stim\_seq\_item.direction;

        // assigned opcode tp interface

            v\_if.opcode = stim\_seq\_item.opcode;

            v\_if.inputs\_are\_driven = 0;

            @(negedge v\_if.clk);

        // assigned outputs to driver

            stim\_seq\_item.leds = v\_if.leds;

            stim\_seq\_item.out = v\_if.out;

        // assigned refrence to driver

            stim\_seq\_item.leds\_ref = v\_if.leds\_ref;

            stim\_seq\_item.out\_ref = v\_if.out\_ref;

            seq\_item\_port.item\_done();

            v\_if.inputs\_are\_driven = 1;

            `uvm\_info("run\_phase\_driver", stim\_seq\_item.convert2string\_stim(), UVM\_HIGH)

        end

    endtask //run\_phase

endclass //ALSU\_driver extends uvm\_driver

endpackage

**/// sequencer**

package sequencer\_pkg;

import uvm\_pkg::\*;

import sequenceItem\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name, this);

// sequencer class

class sequencer extends uvm\_sequencer #(ALSU\_sequenceItem);

    `uvm\_component\_utils(sequencer)

    function new(string name = "sequencer", uvm\_component parent = null);

        super.new(name, parent);

    endfunction //new()

endclass //sequencer extends uvm\_sequencer #(ALSU\_sequenceItem)

endpackage

**/// monitor**

package monitor\_pkg;

import shared\_pkg::\*;

import sequenceItem\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name);

// monitor

class ALSU\_monitor extends uvm\_monitor;

    `uvm\_component\_utils(ALSU\_monitor)

    virtual ALSU\_interface v\_if;

    ALSU\_sequenceItem mon\_seq\_item;

    uvm\_analysis\_port #(ALSU\_sequenceItem) mon\_port; // monitor is a port

    function new(string name = "ALSU\_monitor", uvm\_component parent = null);

        super.new(name, parent);

    endfunction //new()

    function void build\_phase(uvm\_phase phase);

        super.build\_phase(phase);

        mon\_port = new("mon\_port", this);

    endfunction

    task run\_phase(uvm\_phase phase);

      super.run\_phase(phase);

      forever begin

        mon\_seq\_item = `create\_obj(ALSU\_sequenceItem, "mon\_seq\_item")

        @(negedge v\_if.clk);

        // assigned inputs to monitor

            mon\_seq\_item.A = v\_if.A;

            mon\_seq\_item.B = v\_if.B;

            mon\_seq\_item.cin = v\_if.cin;

            mon\_seq\_item.serial\_in = v\_if.serial\_in;

            mon\_seq\_item.red\_op\_A = v\_if.red\_op\_A;

            mon\_seq\_item.red\_op\_B = v\_if.red\_op\_B;

            mon\_seq\_item.bypass\_A = v\_if.bypass\_A;

            mon\_seq\_item.bypass\_B = v\_if.bypass\_B;

            mon\_seq\_item.clk = v\_if.clk;

            mon\_seq\_item.rst = v\_if.rst;

            mon\_seq\_item.direction = v\_if.direction;

        // assigned outputs to monitor

            mon\_seq\_item.leds = v\_if.leds;

            mon\_seq\_item.out = v\_if.out;

        // assigned outputs to monitor

            mon\_seq\_item.leds\_ref = v\_if.leds\_ref;

            mon\_seq\_item.out\_ref = v\_if.out\_ref;

        // assigned opcode tp monitor

            mon\_seq\_item.opcode = opcode\_e'(v\_if.opcode);

        mon\_port.write(mon\_seq\_item); // that's mean that monitor will send the data

        `uvm\_info("run\_phase\_monitor", mon\_seq\_item.convert2string\_stim(), UVM\_FULL)

      end

    endtask //run\_pha

endclass //ALSU\_monitor extends uvm\_monitor

endpackage

**/// agent**

package agent\_pkg;

import shared\_pkg::\*;

import config\_pkg::\*;

import driver\_pkg::\*;

import sequencer\_pkg::\*;

import sequenceItem\_pkg::\*;

import monitor\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name, this);

// agent class

class ALSU\_agent extends uvm\_agent;

    `uvm\_component\_utils(ALSU\_agent)

    sequencer sqr; // mange data transfer

    ALSU\_driver drv; // inside agent

    ALSU\_monitor mon; // inside agent

    ALSU\_config cfg; // get the data of interface

    uvm\_analysis\_port #(ALSU\_sequenceItem) agt\_port; // agent is a port

    function new(string name = "ALSU\_agent", uvm\_component parent = null);

        super.new(name, parent);

    endfunction //new()

    function void build\_phase(uvm\_phase phase);

        super.build\_phase(phase);

        if (!uvm\_config\_db#(ALSU\_config)::get(this, "", "CFG", cfg))

            `uvm\_fatal("build\_phase", "DRIVER - Unable to get config");

        sqr = `create\_obj(sequencer, "sqr")

        drv = `create\_obj(ALSU\_driver, "drv")

        mon = `create\_obj(ALSU\_monitor, "mon")

        agt\_port = new("agt\_port", this);

    endfunction

    function void connect\_phase(uvm\_phase phase);

        drv.v\_if = cfg.v\_if;

        mon.v\_if = cfg.v\_if;

        drv.seq\_item\_port.connect(sqr.seq\_item\_export);

        mon.mon\_port.connect(agt\_port);

    endfunction //connect\_phase

endclass //ALSU\_agent extends uvm\_agent

endpackage

**/// coverage\_collector**

package coverage\_collector\_pkg;

import agent\_pkg::\*;

import shared\_pkg::\*;

import sequencer\_pkg::\*;

import sequenceItem\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

class ALSU\_coverage extends uvm\_component;

    `uvm\_component\_utils(ALSU\_coverage)

    uvm\_analysis\_export #(ALSU\_sequenceItem) cov\_export; // coverage export

    uvm\_tlm\_analysis\_fifo #(ALSU\_sequenceItem) cov\_fifo; // coverage fifo

    ALSU\_sequenceItem cov\_seq\_item;

    //////////////////////////////////

    //      begin Coverage Group    //

    //////////////////////////////////

    covergroup cvr\_gp;

        // input A bins

            A\_cp: coverpoint cov\_seq\_item.A {

                bins A\_data\_0 = {ZERO};

                bins A\_data\_max = {MAXPOS};

                bins A\_data\_min = {MAXNEG};

                bins A\_data\_default = default;

                bins A\_data\_[] = {001, 010, 100};

            }

        // input B bins

            B\_cp: coverpoint cov\_seq\_item.B {

                bins B\_data\_0 = {ZERO};

                bins B\_data\_max = {MAXPOS};

                bins B\_data\_min = {MAXNEG};

                bins B\_data\_default = default;

                bins B\_data\_[] = {001, 010, 100};

            }

        // cover point for reduction operation red\_op

            op\_A\_cp: coverpoint cov\_seq\_item.red\_op\_A {

                bins one = {1};

                bins zero = {0};

            }

            op\_B\_cp: coverpoint cov\_seq\_item.red\_op\_B {

                bins one = {1};

                bins zero = {0};

            }

        // Crossing to satsfied the data\_walkingones of A and B

            A\_walk: cross A\_cp, op\_A\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins A\_data\_walkingones = binsof(A\_cp.A\_data\_) && binsof(op\_A\_cp.one);

            }

            B\_walk: cross B\_cp, op\_A\_cp, op\_B\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins B\_data\_walkingones = (binsof(B\_cp.B\_data\_)

                                          && binsof(op\_A\_cp.zero)

                                          && binsof(op\_B\_cp.one));

            }

        // cover point for opcode (ALU)

            ALU\_cp: coverpoint cov\_seq\_item.opcode {

                bins Bins\_shift[] = {SHIFT, ROTATE};

                bins Bins\_arith[] = {ADD, MULT};

                bins Bins\_bitwise[] = {OR, XOR};//

                bins Bins\_invalid = {INVALID\_6, INVALID\_7};

                bins Bins\_trans = (0 => 1 => 2 => 3 => 4 => 5);

            }

        // cover point for c\_in

            cin\_cp: coverpoint cov\_seq\_item.cin;

        // cover point for serial\_in

            serial\_cp: coverpoint cov\_seq\_item.serial\_in;

        // cover point for direction

            direction\_cp: coverpoint cov\_seq\_item.direction;

        // Cross coverage between ALU\_cp and A and B

            ALU\_A: cross ALU\_cp, A\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins arith\_permutations = binsof(ALU\_cp.Bins\_arith) && binsof(A\_cp) intersect{ZERO, MAXPOS, MAXNEG};

            }

            ALU\_B: cross ALU\_cp, B\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins arith\_permutations = binsof(ALU\_cp.Bins\_arith) && binsof(B\_cp) intersect{ZERO, MAXPOS, MAXNEG};

            }

        // Cross coverage between ALU\_cp and cin\_cp

            ALU\_cin: cross ALU\_cp, cin\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins add\_cin = binsof(ALU\_cp) intersect{ADD};

            }

        // Cross coverage between ALU\_cp and serial\_cp

            ALU\_serial: cross ALU\_cp, serial\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins shift\_serial = binsof(ALU\_cp) intersect{SHIFT};

            }

        // Cross coverage between ALU\_cp and direction\_cp

            ALU\_direction: cross ALU\_cp, direction\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins sh\_ro\_direction = binsof(ALU\_cp.Bins\_shift);

            }

        // Cross coverage ALU = {OR,XOR}, red\_op\_A = 1, A = data\_walk, B = 0

            A\_data\_walk\_OR\_XOR: cross ALU\_cp, A\_walk, op\_A\_cp, B\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins A\_walk\_OR\_XOR = (binsof(ALU\_cp.Bins\_bitwise)

                                   && binsof(A\_walk.A\_data\_walkingones)

                                   && binsof(op\_A\_cp.one)

                                   && binsof(B\_cp.B\_data\_0));

            }

        // Cross coverage ALU = {OR,XOR}, red\_op\_A = 1, A = data\_walk, B = 0

            B\_data\_walk\_OR\_XOR: cross ALU\_cp, B\_walk, op\_B\_cp, A\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins B\_walk\_OR\_XOR = (binsof(ALU\_cp.Bins\_bitwise)

                                   && binsof(B\_walk.B\_data\_walkingones)

                                   && binsof(op\_B\_cp.one)

                                   && binsof(A\_cp.A\_data\_0));

            }

        // Cross coverage Invalid case 2 red\_op

            Invalid\_red\_op: cross ALU\_cp, op\_A\_cp, op\_B\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins Invalid\_reduction = (binsof(ALU\_cp.Bins\_bitwise)

                                         && (binsof(op\_A\_cp.one) || binsof(op\_B\_cp.one)));

            }

        // Invalid case with reduction operation

            reduction\_invalid: cross ALU\_cp, op\_A\_cp, op\_B\_cp {

                option.cross\_auto\_bin\_max = 0;

                bins invalid\_red\_op = (binsof(ALU\_cp) intersect{!OR, !XOR} && (binsof(op\_A\_cp.one)||binsof(op\_B\_cp.one)));

            }

        // cover point rst

            rst\_cp: coverpoint cov\_seq\_item.rst;

        // Cross coverage red\_op\_A and red\_op\_B

            red\_op\_High\_cross: cross op\_A\_cp, op\_B\_cp;

    endgroup

    ///////////////////////////////////

    //      finish Coverage Group    //

    ///////////////////////////////////

    function new(string name = "ALSU\_coverage", uvm\_component parent = null);

        super.new(name, parent);

        cvr\_gp = new();

    endfunction //new()

    function void build\_phase(uvm\_phase phase);

        super.build\_phase(phase);

        cov\_export = new("cov\_export", this);

        cov\_fifo = new("cov\_fifo", this);

    endfunction

    function void connect\_phase(uvm\_phase phase);

        super.connect\_phase(phase);

        cov\_export.connect(cov\_fifo.analysis\_export);

    endfunction

    task run\_phase(uvm\_phase phase);

        super.run\_phase(phase);

        forever begin

            cov\_fifo.get(cov\_seq\_item);

            cvr\_gp.sample();

        end

    endtask

endclass //ALSU\_coverage extends uvm\_component

endpackage

**/// scoreboard**

package scoreboard\_pkg;

import agent\_pkg::\*;

import shared\_pkg::\*;

import sequenceItem\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name, this);

class ALSU\_scoreboard extends uvm\_scoreboard;

    `uvm\_component\_utils(ALSU\_scoreboard)

    uvm\_analysis\_export #(ALSU\_sequenceItem) sb\_export; // scoreboard export

    uvm\_tlm\_analysis\_fifo #(ALSU\_sequenceItem) sb\_fifo; // scoreboard fifo

    ALSU\_sequenceItem sb\_seq\_item;

    // refrence output

    logic [15:0] leds\_ref;

    logic [5:0] out\_ref;

    // error and correct counter

    int correct\_counter = 0;

    int error\_counter = 0;

    function new(string name = "ALSU\_scoreboard", uvm\_component parent = null);

        super.new(name, parent);

    endfunction //new()

    function void build\_phase(uvm\_phase phase);

        super.build\_phase(phase);

        sb\_export = new("sb\_export", this);

        sb\_fifo = new("sb\_fifo", this);

    endfunction

    function void connect\_phase(uvm\_phase phase);

        super.connect\_phase(phase);

        sb\_export.connect(sb\_fifo.analysis\_export);

    endfunction

    task run\_phase(uvm\_phase phase);

        super.run\_phase(phase);

        forever begin

            sb\_fifo.get(sb\_seq\_item);

            //// Checking //////

            Checking\_task(sb\_seq\_item);

        end

    endtask

    task Checking\_task(ALSU\_sequenceItem chk\_item);

        if (chk\_item.out != chk\_item.out\_ref || chk\_item.leds != chk\_item.leds\_ref) begin

            error\_counter++;

            `uvm\_error("scoreboard",$sformatf("%0s\nout\_ref = %0d,",chk\_item.convert2string(), chk\_item.out\_ref))

        end else begin

            correct\_counter++;

        end

    endtask //Checking\_task

    function void report\_phase(uvm\_phase phase);

        super.report\_phase(phase);

        `uvm\_info("report\_phase", $sformatf("Total correct transaction: %0d", correct\_counter), UVM\_LOW)

        `uvm\_info("report\_phase", $sformatf("Total faild transaction: %0d", error\_counter), UVM\_LOW)

    endfunction

endclass //ALSU\_scoreboard extends uvm\_scoreboard

endpackage

**/// env**

`timescale 1ps/1ps

package env\_pkg;

import shared\_pkg::\*;

import scoreboard\_pkg::\*;

import coverage\_collector\_pkg::\*;

import agent\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name, this);

// Environment class

class ALSU\_env extends uvm\_env;

    `uvm\_component\_utils(ALSU\_env)

    ALSU\_scoreboard sb;

    ALSU\_coverage cov;

    ALSU\_agent agt;

    // declare new() function of parent uvm\_env

    function new(string name = "ALSU\_env", uvm\_component parent = null);

        super.new(name, parent);

    endfunction //new()

    // build phase function and send the parameter phase to parent uvm\_env

    function void build\_phase(uvm\_phase phase);

        super.build\_phase(phase);

        // create scoreboard, coverage and agent

        // if we have more than one I should change name (first parameter)

        agt = `create\_obj(ALSU\_agent, "agt")

        sb = `create\_obj(ALSU\_scoreboard, "sb")

        cov = `create\_obj(ALSU\_coverage, "cov")

        // Override the default instantiation of ALSU\_sequenceItem objects with ALSU\_sequenceItem\_valid objects

        //uvm\_factory::set\_default\_create\_ "ALSU\_sequenceItem", () => ALSU\_sequenceItem\_valid::type\_id::create();

    endfunction

    function void connect\_phase(uvm\_phase phase);

        agt.agt\_port.connect(sb.sb\_export);   // why we connect with export directly why not fifo first

        agt.agt\_port.connect(cov.cov\_export); // why we connect with export directly why not fifo first

    endfunction

endclass //ALSU\_env extends uvm\_env

endpackage

**/// test**

package test\_pkg;

import shared\_pkg::\*;

import env\_pkg::\*;

import config\_pkg::\*;

import main\_sequence\_pkg::\*;

import rst\_sequence\_pkg::\*;

import sequenceItem\_pkg::\*;

import sequenceItem\_valid\_pkg::\*;

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

`define create\_obj(type, name) type::type\_id::create(name, this);

class ALSU\_test extends uvm\_test;

    `uvm\_component\_utils(ALSU\_test)

    ALSU\_env env;

    ALSU\_config cfg;

    ALSU\_reset\_sequence rst\_seq;

    ALSU\_main\_sequence main\_seq;

    // declare new() function of parent uvm\_test

    function new(string name = "ALSU\_test", uvm\_component parent = null);

        super.new(name, parent);

    endfunction //new()

    // build phase function and send the parameter phase to parent uvm\_test

        function void build\_phase(uvm\_phase phase);

            uvm\_factory factory = uvm\_factory::get();

            super.build\_phase(phase);

            set\_type\_override\_by\_type(ALSU\_sequenceItem::get\_type(), ALSU\_sequenceItem\_valid::get\_type());

            factory.print();

            env = `create\_obj(ALSU\_env, "env");

            cfg = `create\_obj(ALSU\_config, "cfg");

            rst\_seq = `create\_obj(ALSU\_reset\_sequence, "rst\_seq")

            main\_seq = `create\_obj(ALSU\_main\_sequence, "main\_seq")

             if (!uvm\_config\_db#(virtual ALSU\_interface)::get(this, "", "INTERFACE", cfg.v\_if))

                    `uvm\_fatal("build\_phase", "TEST - Unable to get config");

            uvm\_config\_db#(ALSU\_config)::set(this, "\*", "CFG", cfg);

        endfunction

    // run phase function to create UVM env

        task run\_phase(uvm\_phase phase);

            super.run\_phase(phase);

            // raise and drop to start and finish of ALSU\_test

            phase.raise\_objection(this);

            // ONLY SHOW IF +UVM\_VERBOSITY=UVM\_DEBUG

            #1; `uvm\_info("run\_phase", "Inside the slaby test DEBUG", UVM\_DEBUG)

            ////////////////////

            //    stimulus    //

            ////////////////////

            // rst seq

            `uvm\_info("run\_phase", "ALSU reset seq start", UVM\_MEDIUM)

            rst\_seq.start(env.agt.sqr);

            `uvm\_info("run\_phase", "ALSU reset seq finish", UVM\_MEDIUM)

            // Main seq

            `uvm\_info("run\_phase", "ALSU main seq start", UVM\_MEDIUM)

            main\_seq.start(env.agt.sqr);

            `uvm\_info("run\_phase", "ALSU reset seq finish", UVM\_MEDIUM)

            phase.drop\_objection(this);

        endtask

endclass //ALSU\_test extends uvm\_test

endpackage

**/// top**

`timescale 1ps/1ps

import uvm\_pkg::\*;

`include "uvm\_macros.svh"

import test\_pkg::\*;

module top ();

    bit clk;

    initial begin

        forever #1 clk = ~clk;

    end

    ALSU\_interface intf (clk);

    ALSU DUT (

        intf.A, intf.B, intf.cin, intf.serial\_in, intf.red\_op\_A,

        intf.red\_op\_B, intf.opcode, intf.bypass\_A, intf.bypass\_B,

        intf.clk, intf.rst, intf.direction, intf.leds, intf.out

    );

    ALSU\_ref GLD (

        intf.A, intf.B, intf.cin, intf.serial\_in, intf.red\_op\_A,

        intf.red\_op\_B, intf.opcode, intf.bypass\_A, intf.bypass\_B,

        intf.clk, intf.rst, intf.direction, intf.leds\_ref, intf.out\_ref

    );

    bind ALSU ALSU\_sva ALSU\_sva\_inst(

        A, B, cin, serial\_in, red\_op\_A, red\_op\_B,

        opcode, bypass\_A, bypass\_B, clk, rst, direction, leds, out

    );

    initial begin

        uvm\_config\_db#(virtual ALSU\_interface)::set(null, "uvm\_test\_top", "INTERFACE", intf);

        run\_test("ALSU\_test");

    end

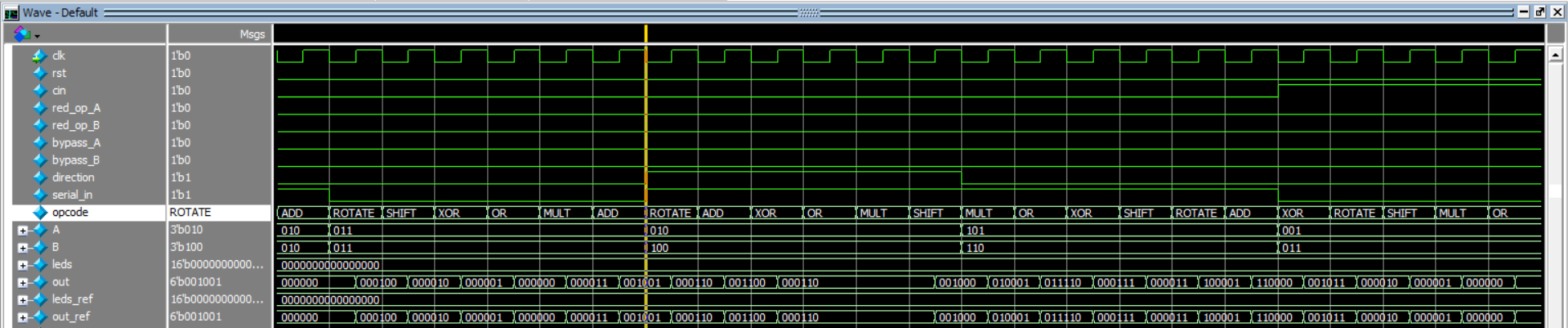
endmodule

2. Simulate the top module. Take a screenshot of the UVM report to use it in your PDF.

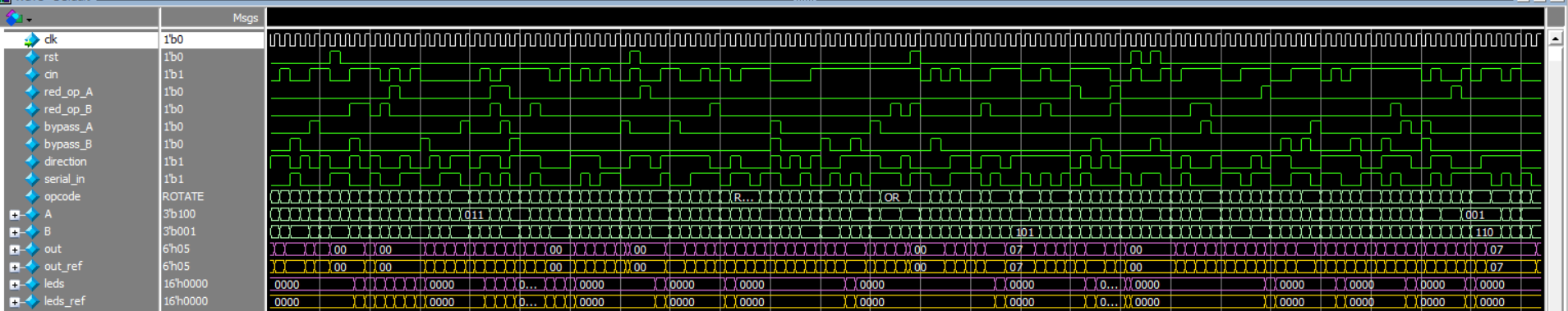
* UVM Report



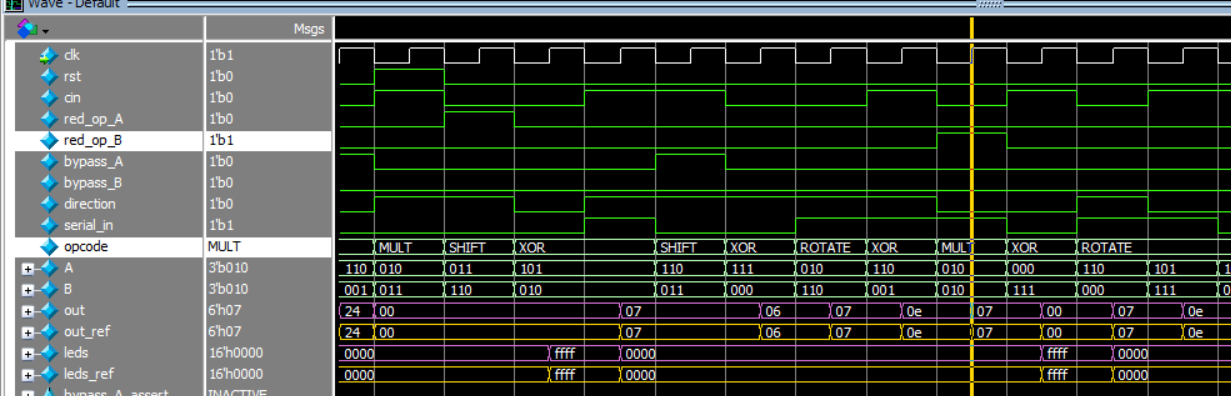
* Constraint of rule 8 🡺 make opcode only valid and unique values



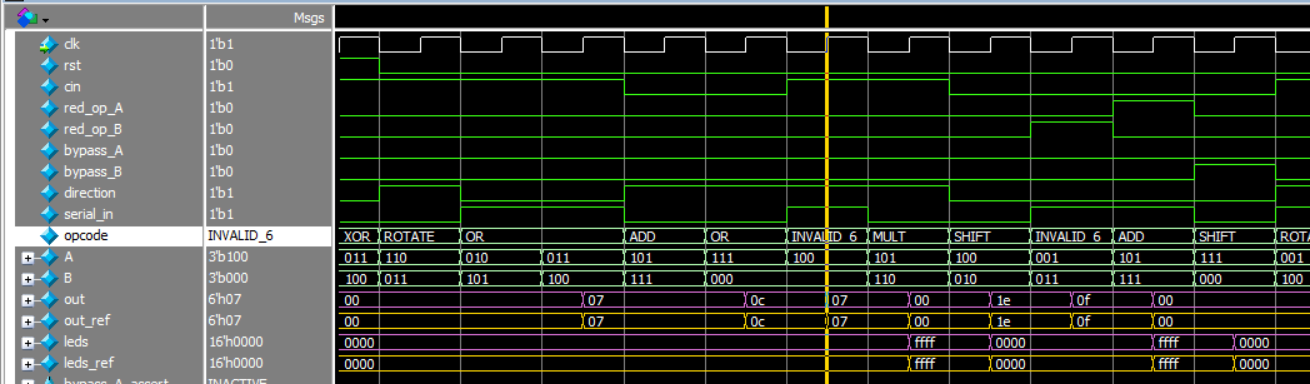
* Constraint reset most of the time is inactive



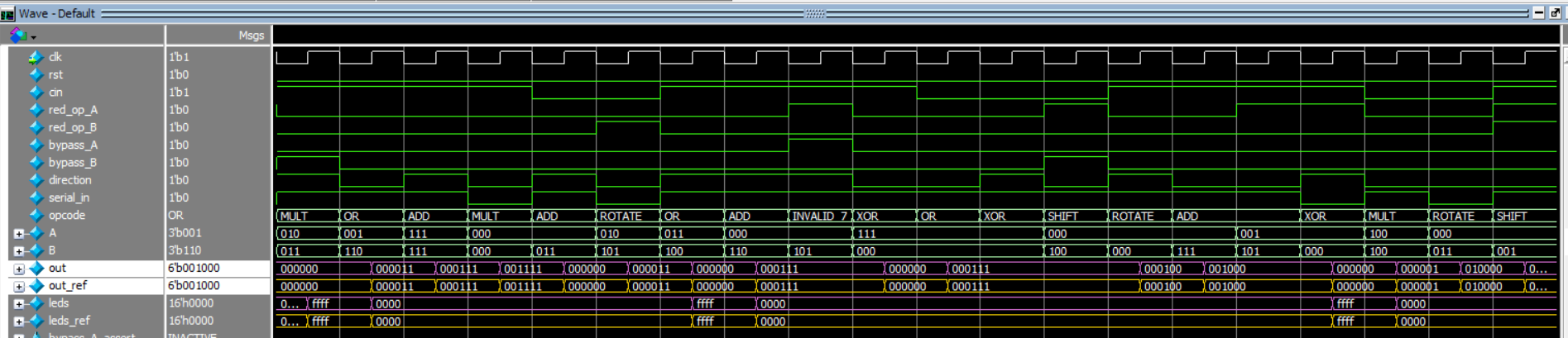
* Invalid reduction



* Invalid opcode



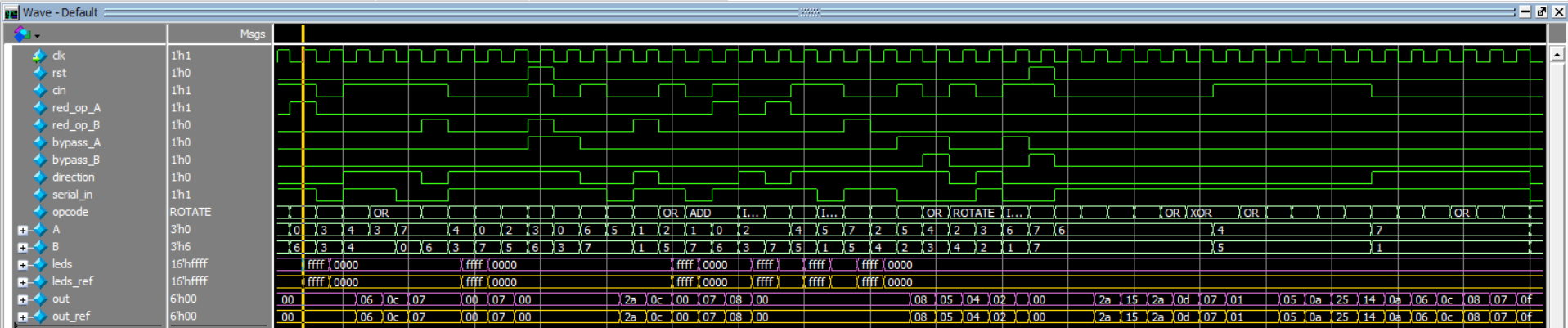
|  |  |  |
| --- | --- | --- |
| **- OR** | **- XOR** | **- ADD** |
| **- MULT** | **- SHIFT** | **- ROTATE** |



3. Add the signals of the interface to wave and make sure that the inputs are driven. Take a screenshot and add it to your PDF.

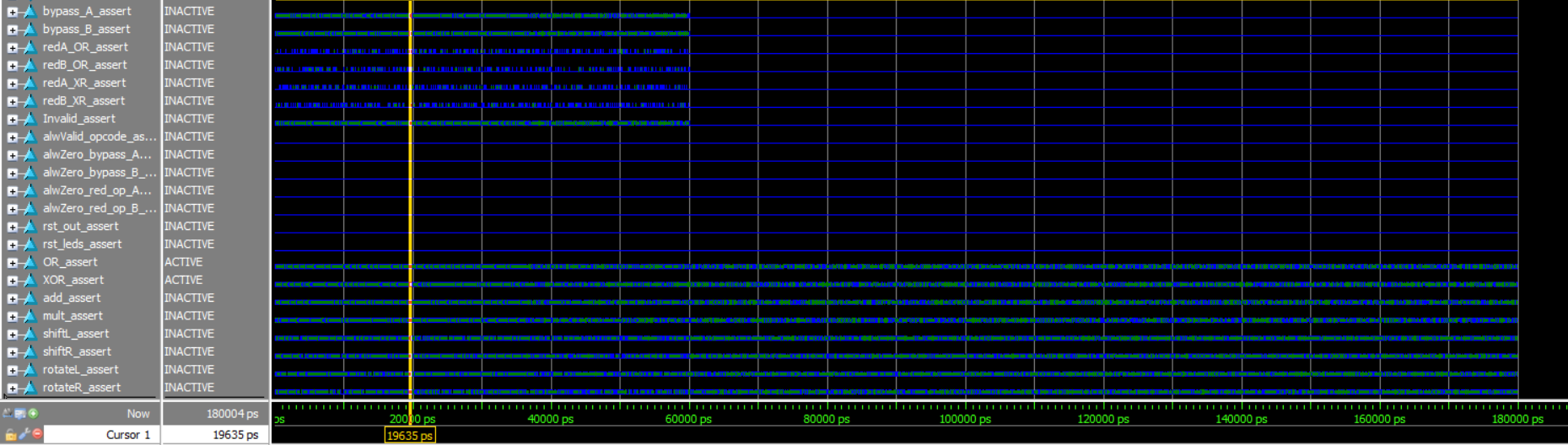
Do file:

add wave -position insertpoint sim:/top/intf/\*



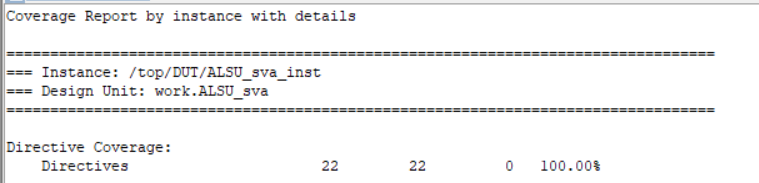
4. Take screenshots of the code coverage, functional coverage, sequential domain (assertions) coverage for ALSU

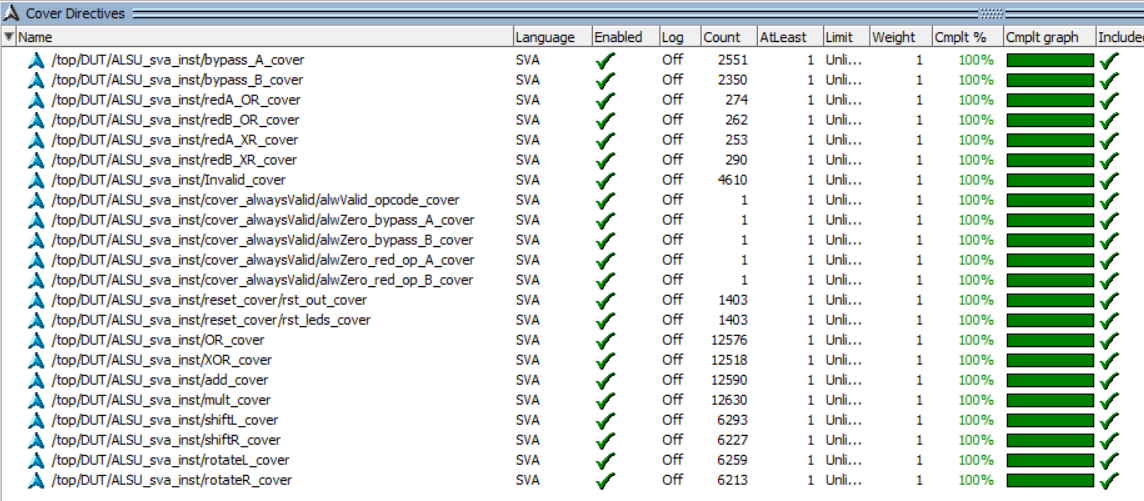
* Assertion





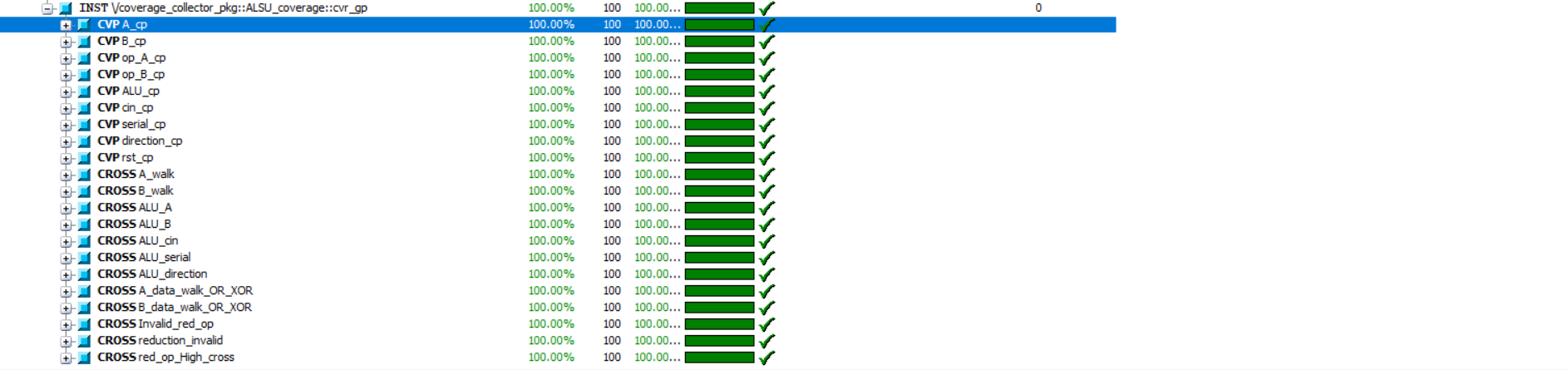
* Coverage directives



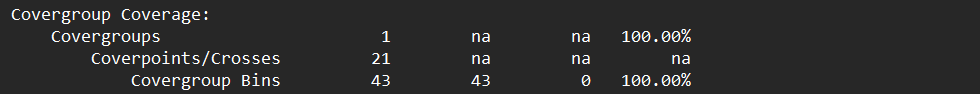


* Coverage

1. Questa Group Coverage

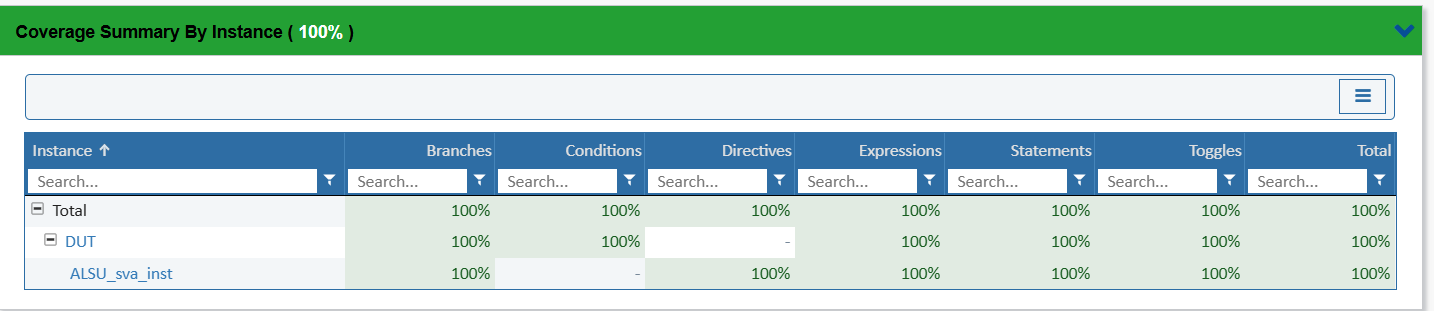


2. Functional coverage

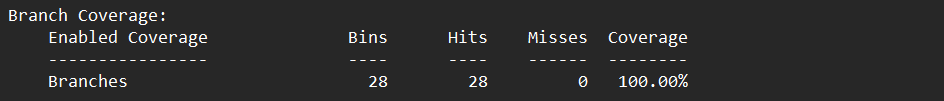


3. Code coverage

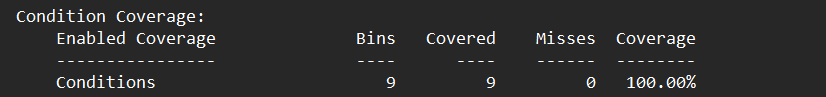
* HTML report



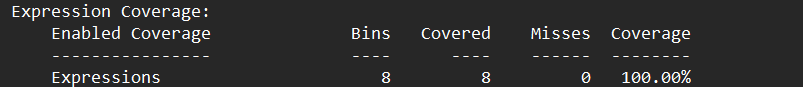
* Txt report
  + Branch Coverage



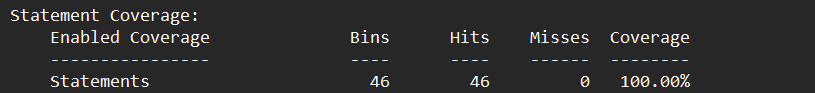
* + Condition Coverage



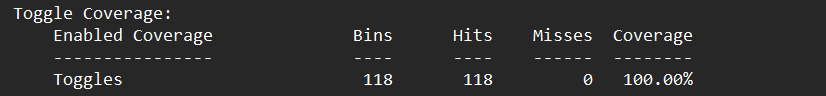
* + Expression Coverage



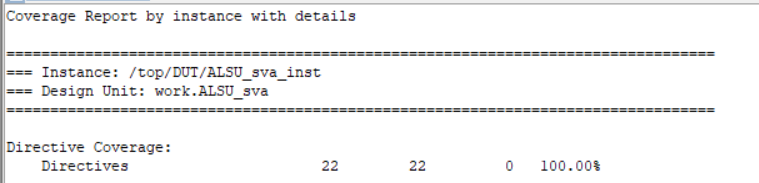
* + Statement Coverage



* + Toggle Coverage

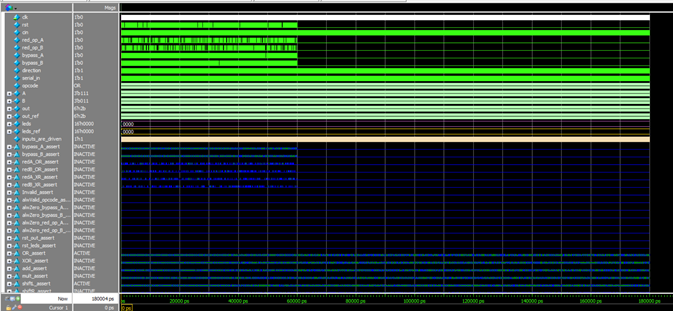


4. Sequential domain coverage





**///// using factory property to override only valid sequencs, Leds never get blink/////**



**/// All cover directives were covered but the Invalid one.**

C:\Users\abdel\AppData\Local\Microsoft\Windows\Clipboard\HistoryData\{0B039DF7-3245-46AF-9973-CDA9F973228B}\{0AB48B66-BD76-4647-A459-B70EDB3BF296}\ResourceMap\{C74EAE1B-8D87-4A35-9E74-8134BEB17BBD}

**////   Code**

        function void build\_phase(uvm\_phase phase);

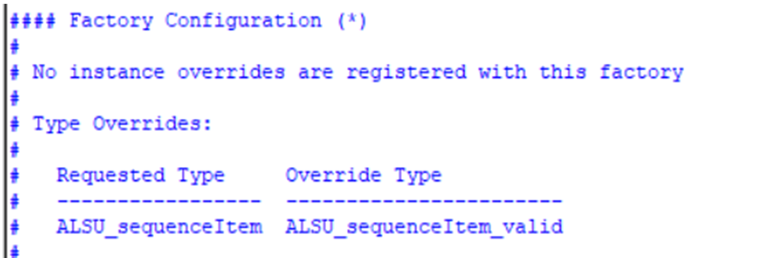
            uvm\_factory factory = uvm\_factory::get();

            super.build\_phase(phase);

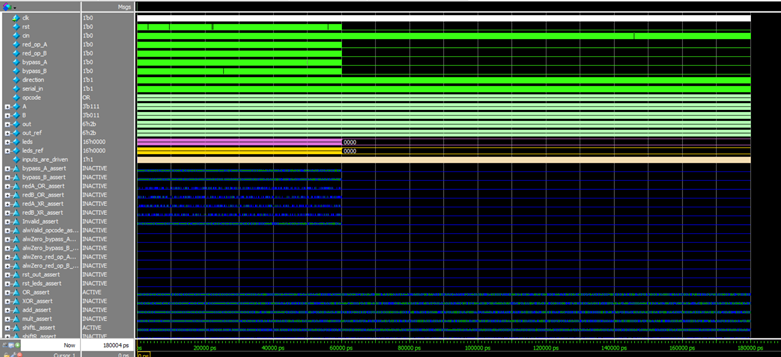
            set\_type\_override\_by\_type(ALSU\_sequenceItem::get\_type(), ALSU\_sequenceItem\_valid::get\_type());

            factory.print();

**/// Factory report**



**///  without the override method call, Leds blink then No blink (Valid and Invalid cases) ///**

****

**/// Invalid cases was covered**

**C:\Users\abdel\AppData\Local\Microsoft\Windows\Clipboard\HistoryData\{0B039DF7-3245-46AF-9973-CDA9F973228B}\{0AB48B66-BD76-4647-A459-B70EDB3BF296}\ResourceMap\{42FB69F3-A950-4996-A7B5-AF0F9B158D43}**

**/// Code**

        function void build\_phase(uvm\_phase phase);

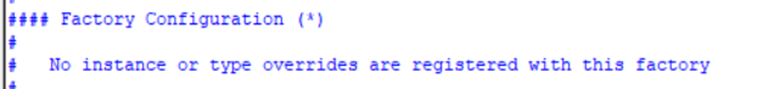
            uvm\_factory factory = uvm\_factory::get();

            super.build\_phase(phase);

            //set\_type\_override\_by\_type(ALSU\_sequenceItem::get\_type(), ALSU\_sequenceItem\_valid::get\_type());

            factory.print();

**/// Factory report**

****