**SV Randomization & Functional Coverage Assignment**

**Abdelrahman Mohamed Ali**

|  |  |
| --- | --- |
| **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) |

**verification Plane flow**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Label | Description | Stimulus Generation | Functional Coverage | Functionality Check |
| ALSU\_1 | Incase of invalid cases do not occur, when opcode is add, then out should perform the addition on ports A and B taking cin if parameter FULL\_ADDER is high | Randomization under constraints on the A and B to have the maximum, minimum and zero values most of the time | Included as coverpoint for A and B. Included with cross coverage when ALU opcode is addition or multiplication | Output Checked against golden model |
| ALSU\_2 | Incase of invalid cases do not occur, when opcode is mult, then out should perform the multiplication on ports A and B | Randomization under constraints on the A and B to have the maximum, minimum and zero values most of the time | Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is addition or multiplication | Output Checked against golden model |
| ALSU\_3 | When invalid cases exist, out output should be low and leds should blink | Randomization under constraints where invalid cases do not occur as frequent as valid cases | Included in a coverpoint for opcode. Included with cross coverage to make sure invalid cases occur | Output Checked against golden model |
| ALSU\_4 | If invalid cases do not occur and the bypass inputs are high, then the output out should by bypass port A or B based on the prioirty if the both bypass ports are high | Randomization for bypass | Included in a coverpoint for bypass | Output Checked against golden model |
| ALSU\_5 | If invalid cases do not occur and the reduction operation are high and opcode nither OR nor XOR, then the output out should by zero and leds should blink as an INVALED case | Randomization for reduction operation | Included in cover group a cross covarage for reduction Invalid (reduction\_invalid) Included with cross coverage when ALU opcode is addition or multiplication | Output Checked against golden model and from invvalid signal |
| ALSU\_6 | Incase of invalid cases do not occur, when opcode is OR and reduction operation is low, then out should perform make A OR B | Randomization under constraints on the A to be invert B (A=~B) | Included in coverpoint for A and B. Included with cross coverage when ALU opcode is arithmatic | Output Checked against golden model |
| ALSU\_7 | Incase of invalid cases do not occur, when opcode is XOR and reduction operation is low, then out should perform A XOR B | Randomization under constraints on the A to be invert B (A=~B) | Included in coverpoint for A and B. Included with cross coverage when ALU opcode is arithmatic | Output Checked against golden model |
| ALSU\_8 | Incase of invalid cases do not occur, when opcode either OR or XOR and reduction operation is high for port A, then out should perform reduction OR or XOR on ports A onley | Randomization under constraint the input B most of the time to have one bit high in its 3 bits while constraining the A to be low | Included in coverpoint for A. when A is {001, 010, 100} and only the red\_op\_A is high | Output Checked against golden model |
| ALSU\_9 | Incase of invalid cases do not occur, when opcode either OR or XOR and reduction operation is high for port B, then out should perform reduction OR or XOR on ports B onley | Randomization under constraint the input A most of the time to have one bit high in its 3 bits while constraining the B to be low | Included in coverpoint for A. when B is {001, 010, 100} and onley red\_op\_A is low and red\_op\_B is high | Output Checked against golden model |
| ALSU\_10 | Incase of invalid cases do not occur, when opcode is SHIFT and direction is low, then out should perform shift left logical for last out with serial in | Randomization for A and B with no constraint | Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is SHIFT or ROTATE | Output Checked against golden model |
| ALSU\_11 | Incase of invalid cases do not occur, when opcode is SHIFT and direction is high, then out should perform shift right logical for last out with serial in | Randomization for A and B with no constraint | Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is SHIFT or ROTATE | Output Checked against golden model |
| ALSU\_12 | Incase of invalid cases do not occur, when opcode is ROTATE and direction is high, then out should perform the MSB to be LSB | Randomization for A and B with no constraint | Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is SHIFT or ROTATE | Output Checked against golden model |
| ALSU\_13 | Incase of invalid cases do not occur, when opcode is ROTATE and direction is low, then out should perform the LSB to be MSB | Randomization for A and B with no constraint | Included in coverpoint2 for A and B. Included with cross coverage when ALU opcode is SHIFT or ROTATE | Output Checked against golden model |
| ALSU\_14 | rst is high | Randomization under constraints on rst to be low most of time and make the first randomized value is high in post\_randomize | include in coverpoint with weight zero | Output Checked from wave |
| ALSU\_15 | making opcode always valid | randomize an array and make it unique with no INVALID value | included in coverpoint ALU\_cp | Output Checked against golden model |

**Do file**

vlib work

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

vlog -coveropt 3 +cover +acc {Codes/pkgs/transaction\_pkg.sv}

vlog -coveropt 3 +cover +acc {Codes/pkgs/coverage\_pkg.sv}

##

vlog -coveropt 3 +cover +acc {Codes/Design/ALSU.v}

vlog -coveropt 3 +cover +acc {Codes/Design/ALSU\_sva.sv}

##

vlog -coveropt 3 +cover +acc {Codes/golden\_model/ALUS\_ref.sv}

##

vlog -coveropt 3 +cover +acc {Codes/testbench/testbench.sv}

vsim -voptargs=+acc work.testbench

add wave \*

add wave /testbench/assert\_invaled

add wave /testbench/assert\_invaled\_opcode

add wave /testbench/assert\_invaled\_ref\_op

add wave /testbench/assert\_out

add wave /testbench/sva/redA\_OR\_assert

add wave /testbench/sva/redB\_OR\_assert

add wave /testbench/sva/redA\_XR\_assert

add wave /testbench/sva/redB\_XR\_assert

add wave /testbench/sva/Invalid\_assert

add wave /testbench/sva/OR\_assert

add wave /testbench/sva/XOR\_assert

add wave /testbench/sva/add\_assert

add wave /testbench/sva/mult\_assert

add wave /testbench/sva/shiftL\_assert

add wave /testbench/sva/shiftR\_assert

add wave /testbench/sva/rotateL\_assert

add wave /testbench/sva/rotateR\_assert

add wave /testbench/sva/reset\_assertion/rst\_out\_assert

add wave /testbench/sva/reset\_assertion/rst\_leds\_assert

# free thw next command and comment the quit -sim to see the wave form

#run -all

vsim -coverage -vopt work.testbench -c -do "coverage save -onexit -du ALSU -directive -codeAll cover.ucdb; run -all"

#

coverage report -detail -cvg -directive -comments -output {Reports\FUNCTION\_COVER\_ALSU.txt} {}

#

quit -sim

#

vcover report cover.ucdb -details -all -annotate -output  {Reports\cover\_alsu.txt}

#

vcover report -html cover.ucdb -output  {Reports\html\_report\.}

**Shared\_pkg:**

package shared\_pkg;

    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 RST\_ACTIVATE = 5;

    parameter INVALID\_ACTIVATE = 5;

    parameter BYPASS\_ACTIVATE = 10;

    parameter REDACTION\_ACTIVATE = 20;

    bit first\_rst = 0;

    opcode\_e valid\_arr[] = {OR, XOR, ADD, MULT, SHIFT, ROTATE};

endpackage

**coverage\_pkg:**

package coverage\_pkg;

import shared\_pkg::\*;

import transaction\_pkg::\*;

class ALSU\_coverage;

    ALSU\_transaction tr = new();

    covergroup cvr\_gp;

        // input A bins

            A\_cp: coverpoint tr.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 tr.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 tr.red\_op\_A {

                bins one = {1};

                bins zero = {0};

            }

            op\_B\_cp: coverpoint tr.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 tr.opcode (ALU)

            ALU\_cp: coverpoint tr.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 = (OR => XOR => ADD => MULT => SHIFT => ROTATE);

            }

        // cover point for c\_in

            cin\_cp: coverpoint tr.cin;

        // cover point for tr.serial\_in

            serial\_cp: coverpoint tr.serial\_in;

        // cover point for tr.direction

            direction\_cp: coverpoint tr.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}, tr.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}, tr.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 tr.rst

            rst\_cp: coverpoint tr.rst;

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

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

    endgroup

    function new();

        cvr\_gp = new();

    endfunction //new()

    function void COV\_sample(input ALSU\_transaction take\_tr);

        tr = take\_tr;

        cvr\_gp.sample();

    endfunction

endclass //coverage

endpackage

**transaction\_pkg:**

package transaction\_pkg;

import shared\_pkg::\*;

class ALSU\_transaction;

    rand opcode\_e opcode;

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

    rand bit signed [2:0] A, B;

    bit [1:0]red\_op\_parameterTest;

    constraint rules1\_7 {

        //rst constraint

            rst dist {0:=100-RST\_ACTIVATE, 1:=RST\_ACTIVATE};

        // Invalid cases constraint

            opcode dist {INVALID\_6:=INVALID\_ACTIVATE, INVALID\_7:=INVALID\_ACTIVATE, [0:5]:/100-2\*INVALID\_ACTIVATE};

        // 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:=100-BYPASS\_ACTIVATE, 1:=BYPASS\_ACTIVATE};

            bypass\_B dist {0:=100-BYPASS\_ACTIVATE, 1:=BYPASS\_ACTIVATE};

        // red\_op constraint

            red\_op\_A dist {0:=100-REDACTION\_ACTIVATE, 1:=REDACTION\_ACTIVATE};

            red\_op\_B dist {0:=100-REDACTION\_ACTIVATE, 1:=REDACTION\_ACTIVATE};

        // 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 {valid\_arr};

        unique {arr};

    }

    function new();

    endfunction //new()

    function void post\_randomize();

        if(!first\_rst) rst = 1;

        first\_rst = 1;

        if (red\_op\_parameterTest!=2'b11 && !rst && !bypass\_A && !bypass\_B) begin

            if (opcode==OR) begin

                red\_op\_A = 1;

                red\_op\_B = 1;

                red\_op\_parameterTest[0] = 1;

            end

            if (opcode==XOR) begin

                red\_op\_A = 1;

                red\_op\_B = 1;

                red\_op\_parameterTest[1] = 1;

            end

        end

    endfunction

endclass //ALSUtransaction

endpackage

**Testbench code:**

import shared\_pkg::\*;

import coverage\_pkg::\*;

import transaction\_pkg::\*;

`define reset disable iff(rst)

`define mk\_assertion(sva\_assert) assert property (@(posedge clk) disable iff(rst) (sva\_assert))

module testbench ();

logic clk, 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, leds\_ref;

logic [5:0] out, out\_ref;

ALSU\_transaction tr = new();

ALSU\_coverage cov = new();

ALSU DUT(.\*);

ALSU\_sva sva(.\*);

ALUS\_ref REF(

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

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

);

initial begin

    clk = 0;

    forever

        #1 clk = ~clk;

end

initial begin

    // First loop turn off constraint 8

    tr.rules\_8.constraint\_mode(0);

    repeat(500\_000) begin

        randomization;

        @(negedge clk);

        sample;

    end

    // Forced rst

    bypass\_A = 0; bypass\_B = 0;

    red\_op\_A = 0; red\_op\_B = 0;

    rst = 1; @(negedge clk); rst = 0;

    // Seconed loop turn ON constraint 8 and turn OFF other constraints

    tr.rules1\_7.constraint\_mode(0);

    tr.rules\_8.constraint\_mode(1);

    repeat(10\_000) begin

        randomization(1);

        foreach(tr.arr[i]) begin

            opcode = tr.arr[i];

            tr.opcode = tr.arr[i];

            @(negedge clk);

            sample;

        end

        $display("arr = %p",tr.arr);

    end

    $stop;

end

task randomization(bit con\_rule8 = 0);

    assert(tr.randomize());

    if (!con\_rule8) begin

        rst = tr.rst;

        bypass\_A = tr.bypass\_A;

        bypass\_B = tr.bypass\_B;

        red\_op\_A = tr.red\_op\_A;

        red\_op\_B = tr.red\_op\_B;

    end

    cin = tr.cin;

    direction = tr.direction;

    serial\_in = tr.serial\_in;

    opcode = tr.opcode;

    A = tr.A;

    B = tr.B;

endtask

task sample;

    if (~tr.rst||~bypass\_A||~bypass\_B) cov.COV\_sample(tr);

endtask

assert\_invaled:         assert property (@(posedge clk) disable iff(rst) REF.invalid |-> DUT.invalid);

assert\_invaled\_opcode:  assert property (@(posedge clk) disable iff(rst) REF.invalid\_opcode |-> DUT.invalid\_opcode);

assert\_invaled\_ref\_op:  assert property (@(posedge clk) disable iff(rst) REF.invalid\_red |-> DUT.invalid\_red\_op);

assert\_out:             assert property (@(posedge clk) (out == out\_ref));

endmodule

**Design code:**

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

// Author: Kareem Waseem

// Course: Digital Verification using SV & UVM

//

// Description: ALSU Design

//

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

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

parameter INPUT\_PRIORITY = "A";

parameter FULL\_ADDER = "ON";

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[2] & opcode\_reg[3]; // Wrong

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;

    A\_reg <= 0;

    B\_reg <= 0;

  end

  else begin

    // if (invalid) // Wrong

    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 (invalid) // Wrong

    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 begin

        //case (opcode) // wrong

        case (opcode\_reg) // FIX

          3'h0: begin // Fix

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

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

           // out = (INPUT\_PRIORITY == "A")? &A\_reg: &B\_reg; // Wrong

            else if (red\_op\_A\_reg)

              out <= |A\_reg; // FIX

           // out <= &A\_reg; // Wrong

            else if (red\_op\_B\_reg)

              out <= |B\_reg; // FIX

           // out <= &B\_reg; // Wrong

            else

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

           // out <= A\_reg & B\_reg; // Wrong

          End

          3'h1: begin // Fix

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

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

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

            else if (red\_op\_A\_reg)

              out <= ^A\_reg;

           // out <= |A\_reg;

            else if (red\_op\_B\_reg)

              out <= ^B\_reg;

           // out <= |B\_reg;

            else

              out <= A\_reg ^ B\_reg;

           // out <= A\_reg | B\_reg;

          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 mk\_assert(condition) assert property (@(posedge clk) disable iff(rst) condition)

`define mk\_cover(condition) cover property (@(posedge clk) disable iff(rst) condition)

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 opcode\_e 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: `mk\_assert(bypass\_A |-> ##2 out==$past(A,2));

bypass\_B\_assert: `mk\_assert((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: `mk\_assert(redValid(red\_op\_A,OR,0)        |-> ##2 out==|($past(A,2)));

redB\_OR\_assert: `mk\_assert(redValid(red\_op\_B,OR,red\_op\_A) |-> ##2 out==|($past(B,2)));

redA\_XR\_assert: `mk\_assert(redValid(red\_op\_A,XOR,0)        |-> ##2 out==^($past(A,2)));

redB\_XR\_assert: `mk\_assert(redValid(red\_op\_B,XOR,red\_op\_A) |-> ##2 out==^($past(B,2)));

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

// assertion of invalid cases //

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

sequence Invalid\_seq;

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

endsequence

Invalid\_assert: `mk\_assert(Invalid\_seq |-> ##[0:2](out==0 && leds=='hffff));

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

// 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:   `mk\_assert(op\_assert(0) |-> ##2  out== $past(A,2) | $past(B,2) );

XOR\_assert:  `mk\_assert(op\_assert(1) |-> ##2  out==($past(A,2)^$past(B,2)) );

add\_assert:  `mk\_assert(op\_assert(2) |-> ##2  out==($past(A,2)+$past(B,2)+$past(cin,2)) );

mult\_assert: `mk\_assert(op\_assert(3) |-> ##2  out==($past(A,2)\*$past(B,2)) );

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

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

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

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

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

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

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

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

// cover of bypass operation //

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

bypass\_A\_cover: `mk\_cover( bypass\_A |-> ##2 out==$past(A,2));

bypass\_B\_cover: `mk\_cover( (bypass\_B && !bypass\_A) |-> ##2 out==$past(B,2));

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

// cover of reduction operation //

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

redA\_OR\_cover: `mk\_cover( redValid(red\_op\_A,0,0)        |-> ##2 out==|($past(A,2)));

redB\_OR\_cover: `mk\_cover( redValid(red\_op\_B,0,red\_op\_A) |-> ##2 out==|($past(B,2)));

redA\_XR\_cover: `mk\_cover( redValid(red\_op\_A,1,0)        |-> ##2 out==^($past(A,2)));

redB\_XR\_cover: `mk\_cover( redValid(red\_op\_B,1,red\_op\_A) |-> ##2 out==^($past(B,2)));

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

// cover of invalid cases //

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

Invalid\_cover: `mk\_cover( Invalid\_seq |-> ##[0:2](out==0 && leds=='hffff));

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

// 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:   `mk\_cover( op\_assert(0) |-> ##2  out== $past(A,2) | $past(B,2) );

XOR\_cover:  `mk\_cover( op\_assert(1) |-> ##2  out==($past(A,2) ^ $past(B,2)) );

add\_cover:  `mk\_cover( op\_assert(2) |-> ##2  out==($past(A,2) + $past(B,2) + $past(cin,2)) );

mult\_cover: `mk\_cover( op\_assert(3) |-> ##2  out==($past(A,2) \* $past(B,2)) );

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

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

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

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

endmodule

**Golden model:**

import shared\_pkg::\*;

module ALUS\_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 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;

opcode\_e opcode\_reg;

wire invalid, invalid\_opcode, invalid\_red;

assign invalid\_opcode = (opcode\_reg == INVALID\_6 || opcode\_reg == INVALID\_7);

assign invalid\_red = (red\_op\_A\_reg || red\_op\_B\_reg) && (opcode\_reg != OR && opcode\_reg != XOR);

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

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;

        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 == OR) 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 == XOR) 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 == ADD)

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

        else if (opcode\_reg == MULT)

            out <= A\_reg \* B\_reg;

        else if (opcode\_reg == SHIFT) begin

            if (direction\_reg)

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

            else

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

        end

        else if (opcode\_reg == ROTATE) begin

            if (direction\_reg)

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

            else

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

        end

    end

end

endmodule

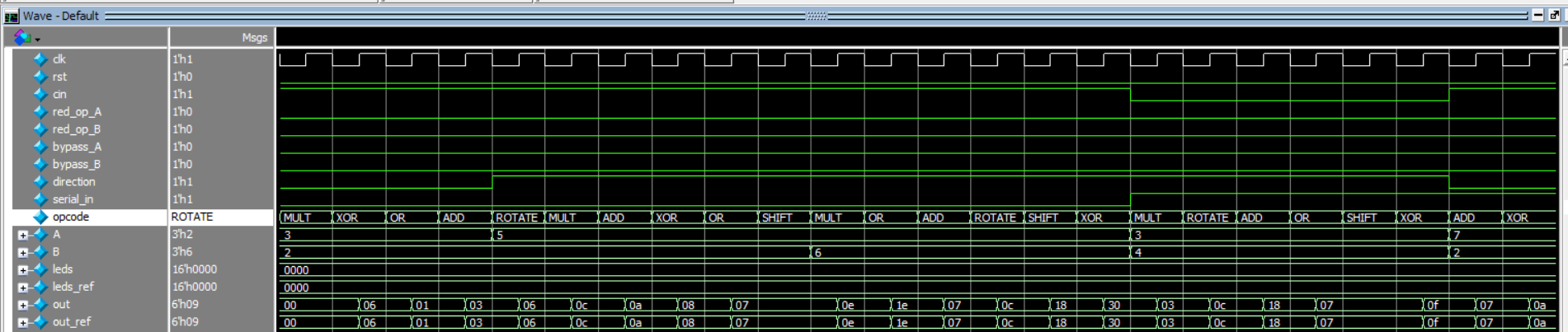
**Bugs Report:**

|  |  |  |  |
| --- | --- | --- | --- |
| 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 |

**QuestaSim Snippet:**

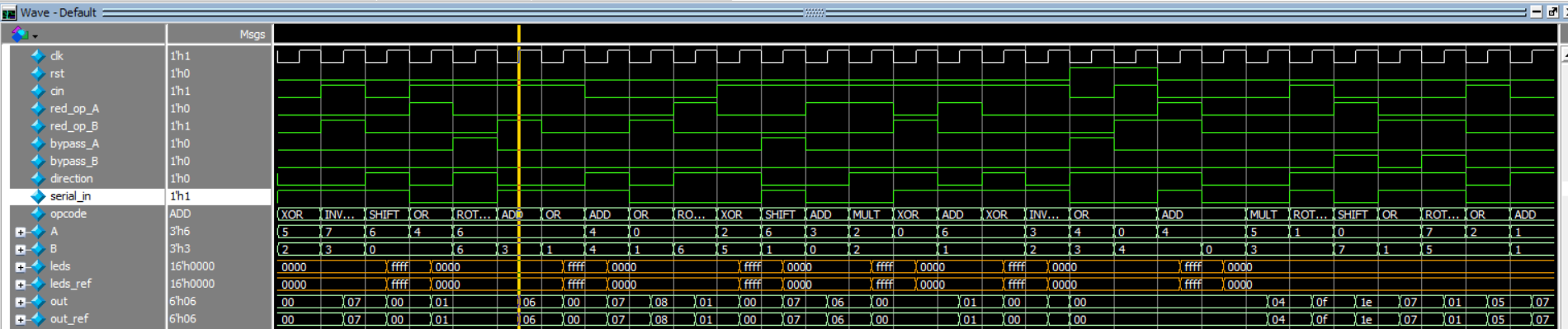
Constraint rule\_8 No invalid cases



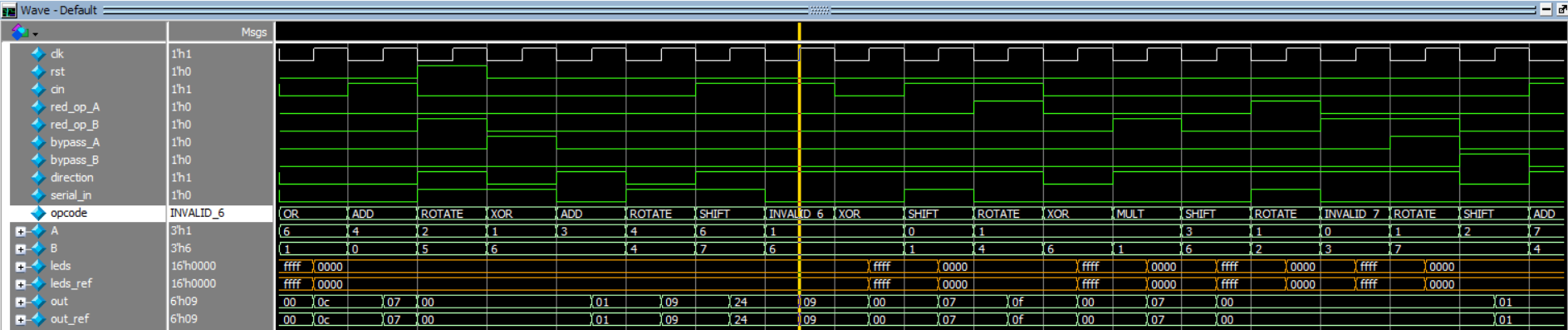
rst constraint most of the time Inactive



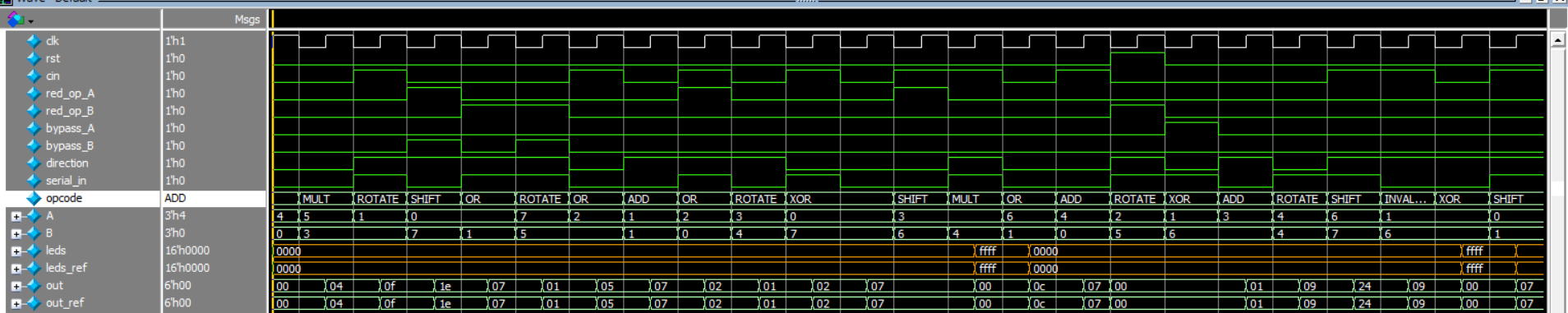
red\_op\_A is active while opcode is ADD 🡺 Invalid case 🡺 leds blink and out = zero



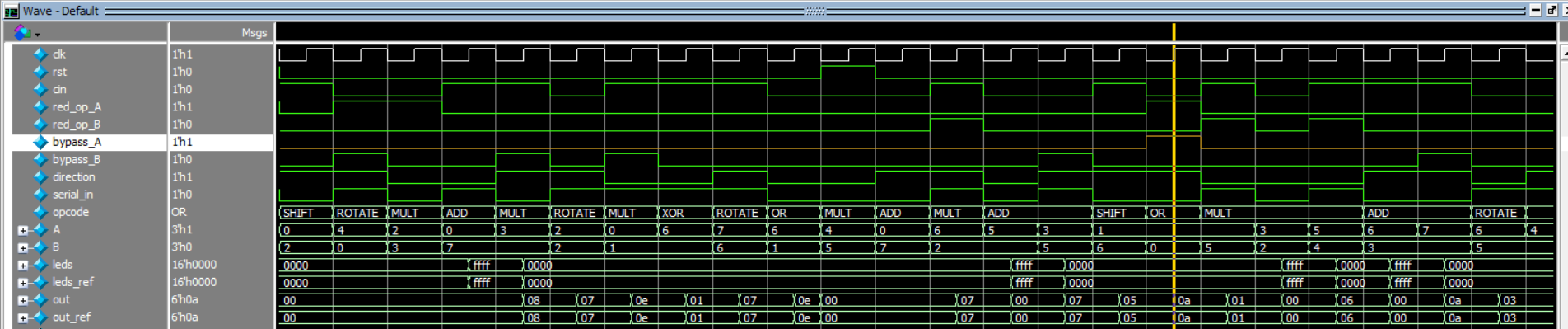
Opcode = Invalid case



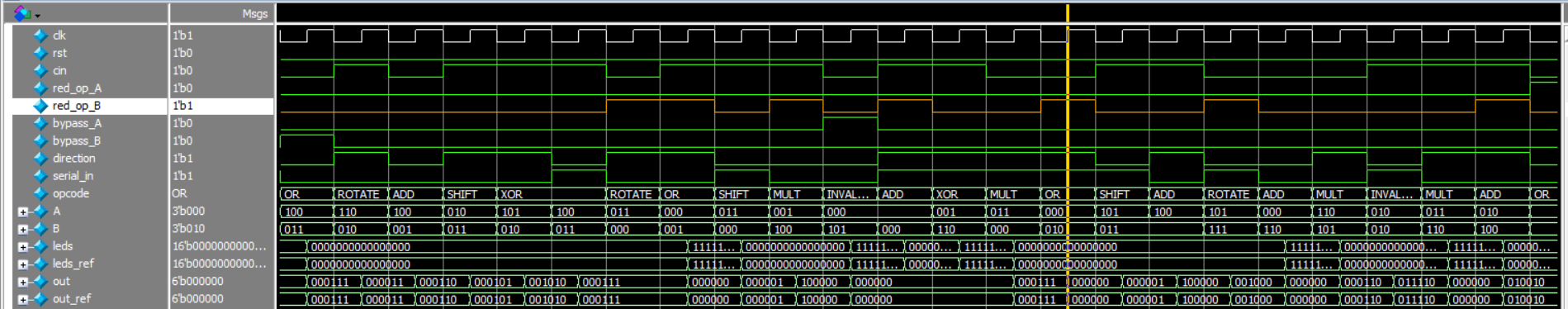
Valid Opcode Cases



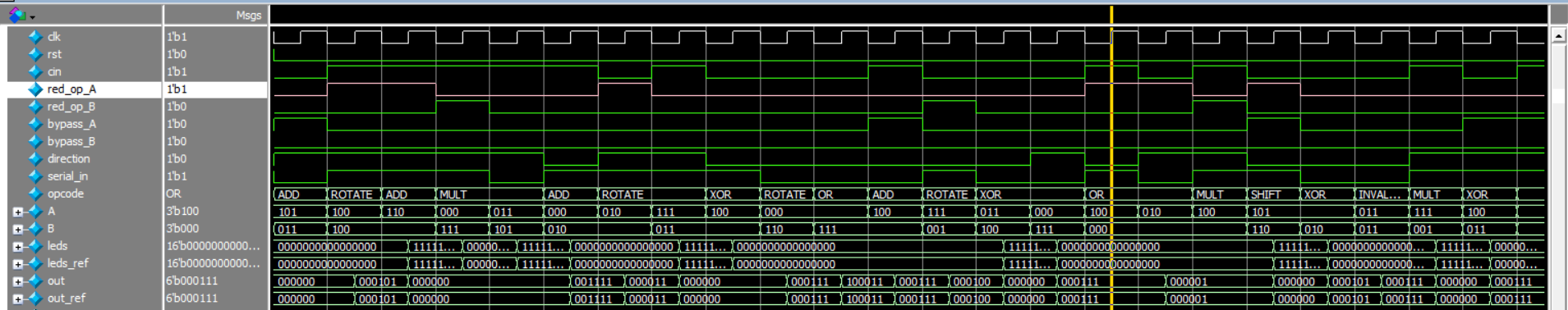
Bypass\_A



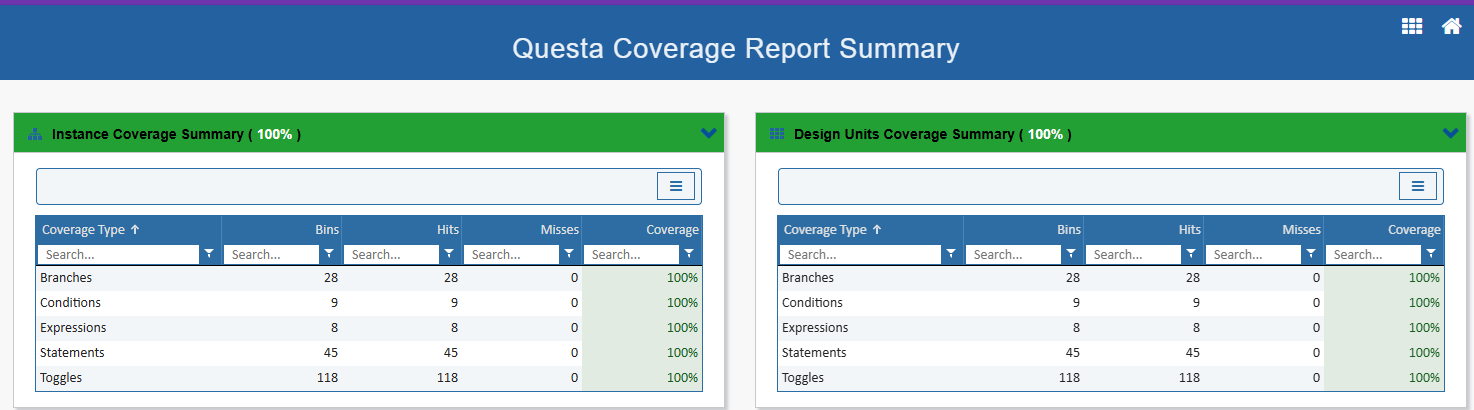
Opcode = OR, red\_op\_B active



Opcode = OR, red\_op\_A active

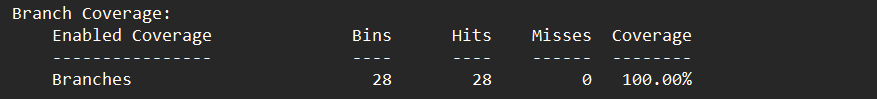


**Code Coverage HTML report**



**Code Coverage txt report**

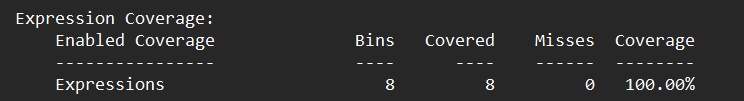
* Branches



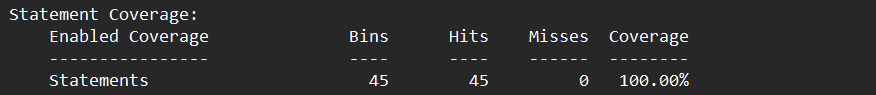
* Conditions



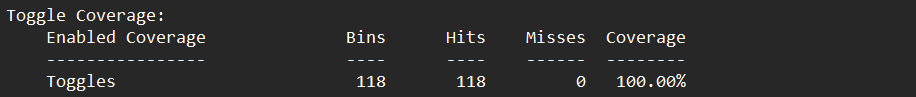
* Expression



* Statement

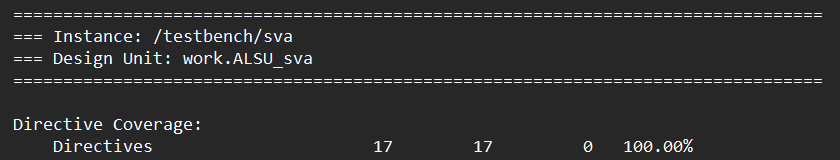


* Toggles



**FUNCTION\_COVER report**

* Directive Coverage



* Covergroup Coverage

