Design and Functional Verification of a 32-bit ALU Using Verilog

Objectives: The objective of this project is to design and verify a 32-bit ALU in Verilog that performs arithmetic, logic, and shift operations. A functional testbench was used to check all commands and input combinations, including corner cases like divide-by-zero and underflow, ensuring the correct 64-bit outputs and proper tri-state behaviour.

Key Components of 32-bit ALU Project

1. **ALU RTL Module (alu_32bit)**
    - Implements **32-bit arithmetic operations**: ADD, SUB, INC, DEC, MUL, DIV
    - Implements **logical operations**: AND, OR, XOR, XNOR, INV, NAND, NOR, BUF
    - Implements **shift operations**: SHL (shift left), SHR (shift right)
    - Produces **64-bit output**
    - Supports **tri-state output** with enable (oe)
2. **Testbench (alu_32bit_tb)**
    - Drives **all input combinations** for operands a and b
    - Iterates through **all 16 ALU commands**
    - Controls **output enable** to test tri-state functionality
    - Uses $monitor to display **live outputs for verification**
3. **Tasks for Reusability**
    - initialize → Reset all inputs
    - inputs → Apply operands
    - cmd → Set ALU command

- en_oe → Control output enable
- delay → Introduce simulation timing delays

4. **Functional Verification**
    - Ensures **all operations produce correct outputs**
    - Covers **corner cases**: divide by zero, underflow, inversion
    - Validates **64-bit output results**

RTL

```
module alu_32bit (
   input  [31:0] a_in,
   input  [31:0] b_in,
   input  [3:0]  command_in,
   input         oe,
   output [63:0] d_out
);
   // ALU operation codes
   parameter ADD  = 4'b0000, // Add a and b
             INC  = 4'b0001, // Increment a
             SUB  = 4'b0010, // Subtract b from a
             DEC  = 4'b0011, // Decrement a
             MUL  = 4'b0100, // Multiply a and b
             DIV  = 4'b0101, // Divide a by b
             SHL  = 4'b0110, // Shift left
             SHR  = 4'b0111, // Shift right
             AND  = 4'b1000, // AND
             OR   = 4'b1001, // OR
             INV  = 4'b1010, // Invert a
             NAND = 4'b1011, // NAND
             NOR  = 4'b1100, // NOR
             XOR  = 4'b1101, // XOR
             XNOR = 4'b1110, // XNOR
             BUF  = 4'b1111; // Buffer

   // Internal register
   reg [63:0] out;
   // Combinational logic
   always @(*) begin
      case (command_in)
```

```
            ADD  : out = a_in + b_in;
            INC  : out = a_in + 1;
            SUB  : out = a_in - b_in;
            DEC  : out = a_in - 1;
            MUL  : out = a_in * b_in;
            DIV  : out = (b_in != 0) ? a_in / b_in : 64'd0;
            SHL  : out = a_in << 1;
            SHR  : out = a_in >> 1;
            AND  : out = a_in & b_in;
            OR   : out = a_in | b_in;
            INV  : out = ~a_in;
            NAND : out = ~(a_in & b_in);
            NOR  : out = ~(a_in | b_in);
            XOR  : out = a_in ^ b_in;
            XNOR : out = ~(a_in ^ b_in);
            BUF  : out = a_in;
            default : out = 64'd0;
        endcase
    end
    // Tri-state output
    assign d_out = oe ? out : 64'hZZZZ_ZZZZ_ZZZZ_ZZZZ;
endmodule
```

Testbench

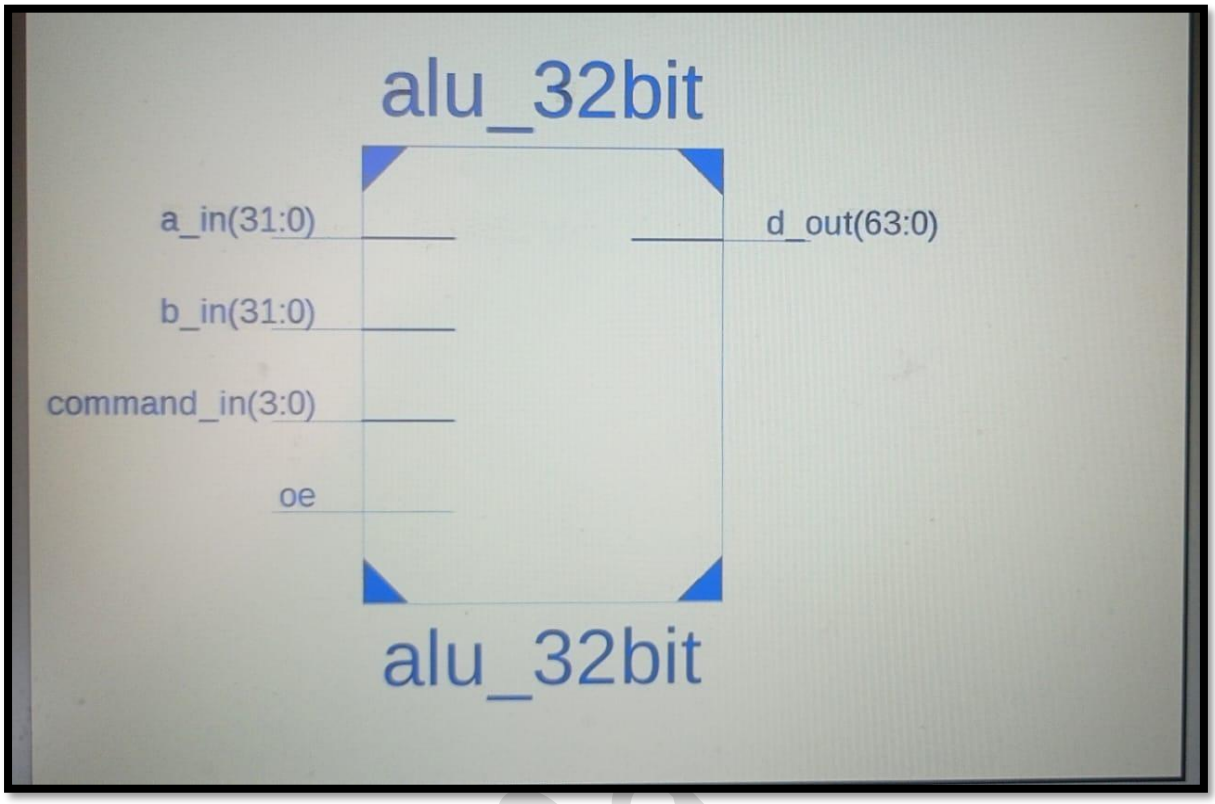
```
`timescale 1ns/1ps
module alu_32bit_tb();
  // Testbench global variables
  reg  [31:0] a, b;
  reg  [3:0]  command;
  reg         enable;
  wire [63:0] out;

  // Variables for iteration of the loops
  integer m, n, o;

  // Parameter constants
  parameter ADD  = 4'b0000,
            INC  = 4'b0001,
            SUB  = 4'b0010,
            DEC  = 4'b0011,
            MUL  = 4'b0100,
            DIV  = 4'b0101,
```

```
        SHL  = 4'b0110,
        SHR  = 4'b0111,
        AND  = 4'b1000,
        OR   = 4'b1001,
        INV  = 4'b1010,
        NAND = 4'b1011,
        NOR  = 4'b1100,
        XOR  = 4'b1101,
        XNOR = 4'b1110,
        BUF  = 4'b1111;

// String for displaying command
reg [8*5:0] string_cmd;

// Step1 : Instantiate the DUT
alu_32bit dut (
  .a_in(a),
  .b_in(b),
  .command_in(command),
  .oe(enable),
  .d_out(out)
);

// Step2 : Initialize task
task initialize;
  begin
    {a, b, enable, command} = 0;
    #10;
  end
endtask

// Enable task
task en_oe(input i);
  begin
    enable = i;
  end
endtask

// Input task
task inputs(input [31:0] j, k);
  begin
    a = j;
    b = k;
```

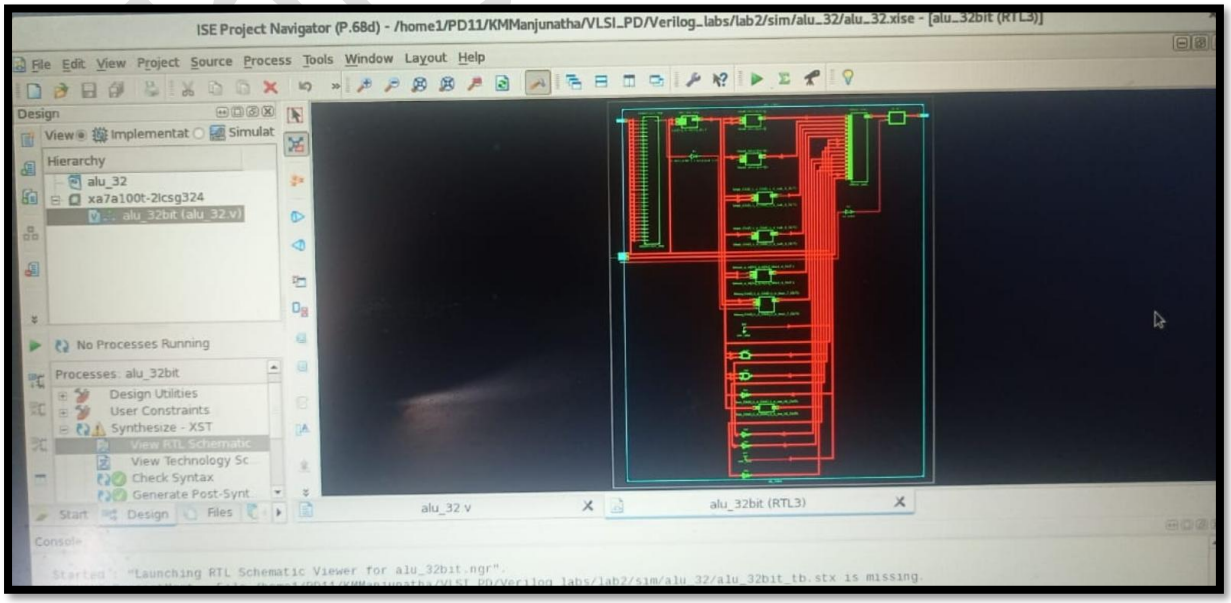
```
  end
endtask

// Command task
task cmd(input [3:0] l);
  begin
    command = l;
  end
endtask

// Delay task
task delay;
  begin
    #10;
  end
endtask

// Command string decoding
always @(command) begin
  case (command)
    ADD  : string_cmd = "ADD";
    INC  : string_cmd = "INC";
    SUB  : string_cmd = "SUB";
    DEC  : string_cmd = "DEC";
    MUL  : string_cmd = "MUL";
    DIV  : string_cmd = "DIV";
    SHL  : string_cmd = "SHL";
    SHR  : string_cmd = "SHR";
    AND  : string_cmd = "AND";
    OR   : string_cmd = "OR";
    INV  : string_cmd = "INV";
    NAND : string_cmd = "NAND";
    NOR  : string_cmd = "NOR";
    XOR  : string_cmd = "XOR";
    XNOR : string_cmd = "XNOR";
    BUF  : string_cmd = "BUF";
    default: string_cmd = "UNK";
  endcase
end

// Stimulus generation
initial begin
  initialize;
```

```
    en_oe(1'b1);

    for (m = 0; m < 16; m = m + 1) begin
      for (n = 0; n < 16; n = n + 1) begin
        inputs(m, n);
        for (o = 0; o < 16; o = o + 1) begin
          command = o;
          delay;
        end
      end
    end

    // Tri-state test
    en_oe(0);
    inputs(32'd20, 32'd10);
    cmd(ADD);
    delay;

    en_oe(1);
    inputs(32'd25, 32'd17);
    cmd(ADD);
    delay;

    $finish;
  end

  // Monitor output
  initial begin
    $monitor("oe=%b a=%d b=%d cmd=%s out=%d",
             enable, a, b, string_cmd, out);
  end

endmodule
```

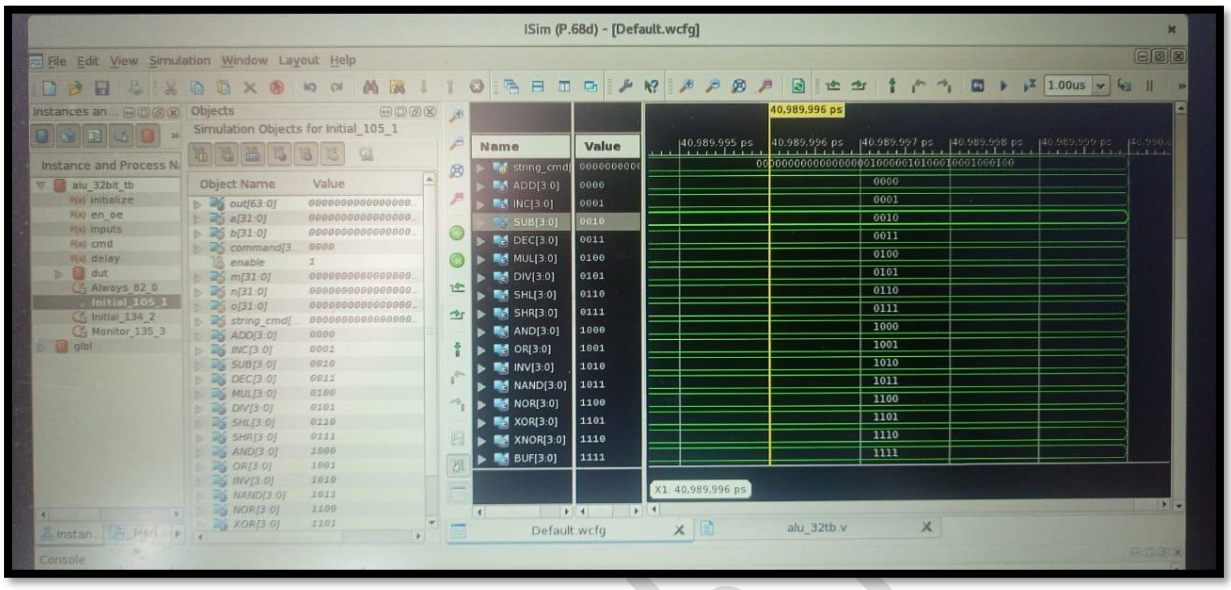
Top – Level Block



Schematic View



Simulation



Console Window Output