Arithmetic Logic Unit Using Verilog RTL and SystemVerilog Verification

(by CH.ANIRUDH)

1.Introduction:

An **Arithmetic Logic Unit** (ALU) is a fundamental component of a **processor**, responsible for performing **arithmetic** and **logical operations**. The ALU is a crucial building block in **microprocessors,** digital signal processors, and embedded systems, enabling efficient computation and data manipulation. The design and verification of an ALU ensure that it operates correctly under all conditions, providing reliable performance in **digital systems**.

This project focuses on the design and verification of an 8-bit ALU using **Verilog** for RTL implementation and **SystemVerilog** for testbench creation. The ALU is designed to execute basic arithmetic operations such as addition, subtraction, multiplication, and division. Verification is performed using a **constrained random testbench**, **functional coverage,** and SystemVerilog **assertions** (SVA) to ensure correctness.

By implementing this project, one gains a deeper understanding of ALU operations, **hardware** design principles, and **verification methodologies.** The project also demonstrates the importance of using SystemVerilog constructs such as constrained random generation, coverage-driven verification, and assertions to validate the correctness and robustness of the ALU.

2.Project Overview:

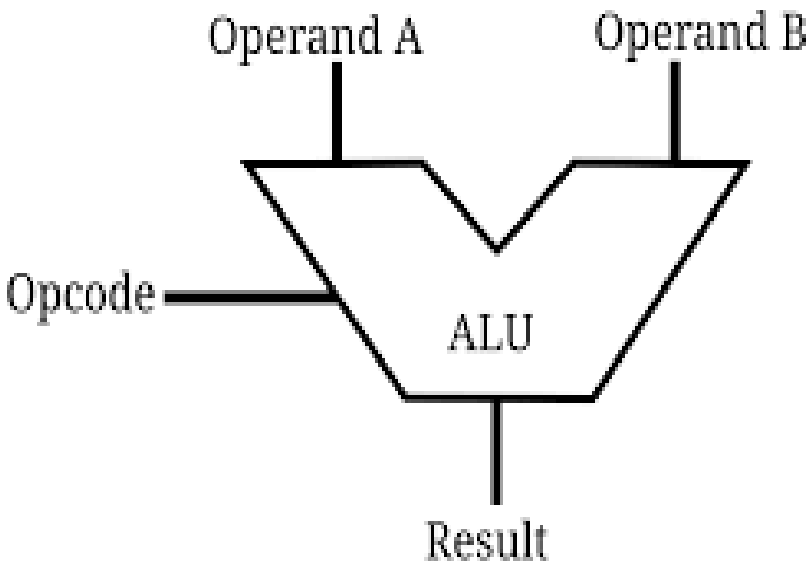
The project consists of the following key components:

- **ALU Design:** Developed using Verilog, supporting operations such as addition, subtraction, multiplication, and division.
- **Testbench Development:** Implemented in SystemVerilog, using constrained random stimulus generation and functional coverage.
- **Verification Methodology:** Employs functional coverage metrics and assertions (SVA) to ensure proper functionality and corner-case testing.
- **Clock and Reset Mechanism:** Provides synchronization for sequential operations and ensures proper initialization.
- **Debugging and Analysis:** Uses testbench messages, functional coverage reports, and assertion failures to debug design issues.

3.Design Implementation:

The ALU is implemented in Verilog and consists of the following key elements:

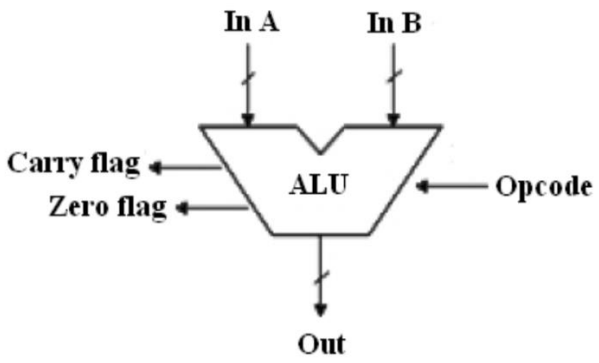
- **Inputs:**
  - opcode (4-bit) – Determines the operation to be performed.
  - a, b (8-bit each) – Operands for arithmetic operations.
- **Outputs:**
  - result (8-bit) – Stores the computed output.
  - valid (1-bit) – Indicates whether the operation was successful.
- **Operations:**
  - Addition (opcode = 0000)
  - Subtraction (opcode = 0001)
  - Multiplication (opcode = 0010)
  - Division (opcode = 0011, with a check for divide-by-zero conditions)



Fig(a): Basic ALU Block Diagram

4.Operation of the ALU:

1. The ALU takes opcode, a, and b as inputs.
2. Based on the **opcode,** the corresponding arithmetic operation is performed.
3. The **valid flag** is set to 1 if the operation is valid; otherwise, it is set to 0 (e.g., divide by zero).
4. The computed result is assigned to the result **output.**



Fig(b): ALU with Status Flags (Carry and Zero Flags)

5.Verification Methodology:

The testbench is written in SystemVerilog and includes the following features:

- **Constrained Random Generation:**
  - Generates random values for opcode, a, and b while ensuring b ≠ 0 for division.
- **Functional Coverage:**
  - Ensures all operations and operand ranges are adequately tested.
- **Assertions (SVA):**
  - Ensures valid flag behaves correctly for supported operations.

- Checks that addition and division operations produce correct results.

- **Clock Generation:**
  - A clock signal is generated to drive the verification environment.
- **Test Execution:**
  - The testbench runs 100 iterations of random input generation, executes operations, collects coverage, and prints the results.

6.Verilog RTL Code:

```
module alu(
   input logic [3:0] opcode,      //4-bit operation code
   input logic [7:0] a, b,        // 8-bit operands
   output logic [7:0] result,     // 8-bit output
   output logic valid           // valid flag (1 if successful, 0 if error)
   );
    always_comb begin
     valid = 1;           //default valid case
      case(opcode)
         4'b0000: result = a + b;   //Add
         4'b0001: result = a - b;   //sub
         4'b0010: result = a * b;   // Mul
         4'b0011: result = (b != 0) ? (a / b) : 8'b0;  //avoid divide by zero
          default: begin
           result = 8'b0;
           valid = 0;    //invalid operation
          end
      endcase
      end
endmodule
```

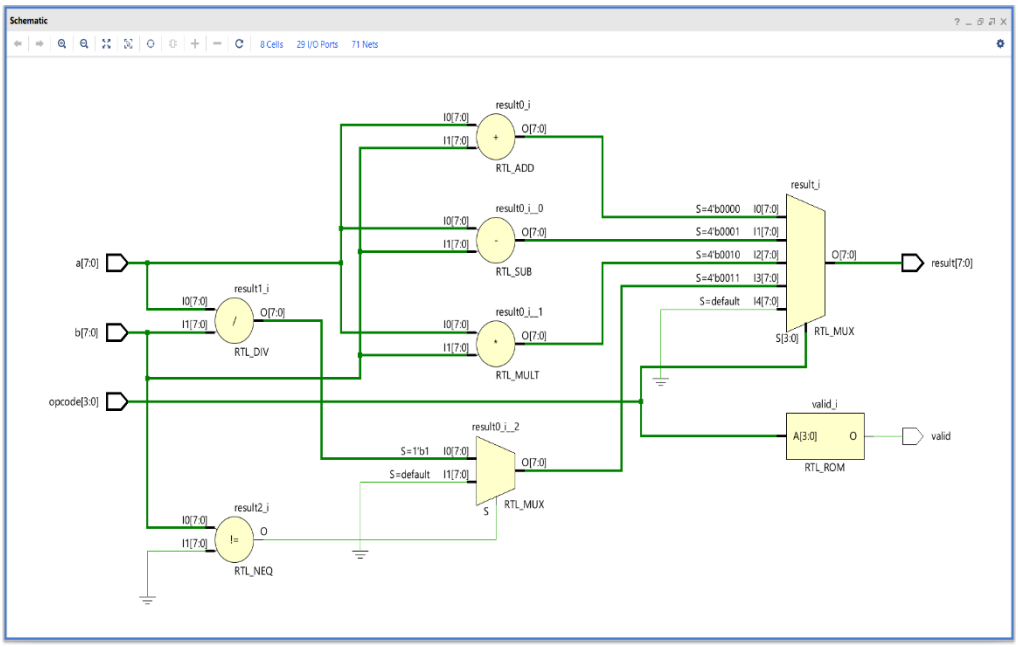
7. SV Testbench:

```
module alu_tb;
    logic clk, rst;
    logic [3:0] opcode;
    logic [7:0] a, b, result;
    logic valid;


    //instantiate ALU (DUT)
```

```
alu dut (
        .opcode(opcode),
        .a(a),
        .b(b),
        .result(result),
        .valid(valid)
);
//constrained random generator
class alu_crt;
  rand logic [3:0] opcode;
  rand logic [7:0] a, b;

  constraint c_opcode { opcode inside {4'b0000, 4'b0001, 4'b0010, 4'b0011};} //only Add, Sub,
Mul, Div
  constraint c_div { if (opcode ==4'b0011) b != 0;} //avoid divide by zero

  function void display ();
  $display ("Random Inputs -> Opcode: %b, A: %d, B: %d", opcode, a,b);
  endfunction
  endclass
  //Functional coverage
  class alu_coverage;
   covergroup cg;
     coverpoint opcode {bins add = {4'b0000}; bins sub = {4'b0001}; bins mul =  {4'b0010}; bins div
= {4'b0011};}
        coverpoint a {bins low ={[0:50]}; bins high = {[51:255]};}
        coverpoint b {bins zero = {0}; bins non_zero = {[1:255]};}
              cross opcode, a, b; // ensure different combinations are tested
        endgroup
        function new();
          cg = new();
        endfunction

        function void sample(input logic [3:0] op,input logic [7:0] aa, input logic [7:0] bb);
         opcode = op;
         a = aa;
         b = bb;
         cg.sample();    //collect data
         endfunction
        endclass
        //assertions (SVA)

        property p_valid_operation;
         @(posedge clk) disable iff (rst) (opcode inside {4'b0000,4'b0001, 4'b0010,4'b0011}) |-> valid;
        endproperty
        assert property (p_valid_operation)
        else $error("Error: Invalid operation detected!");

        property p_add_correct;
         @(posedge clk) disable iff (rst) (opcode == 4'b0000) |-> (result == a + b);
         endproperty
         assert property (p_add_correct)
         else $error("Error: add operation failed!");
```

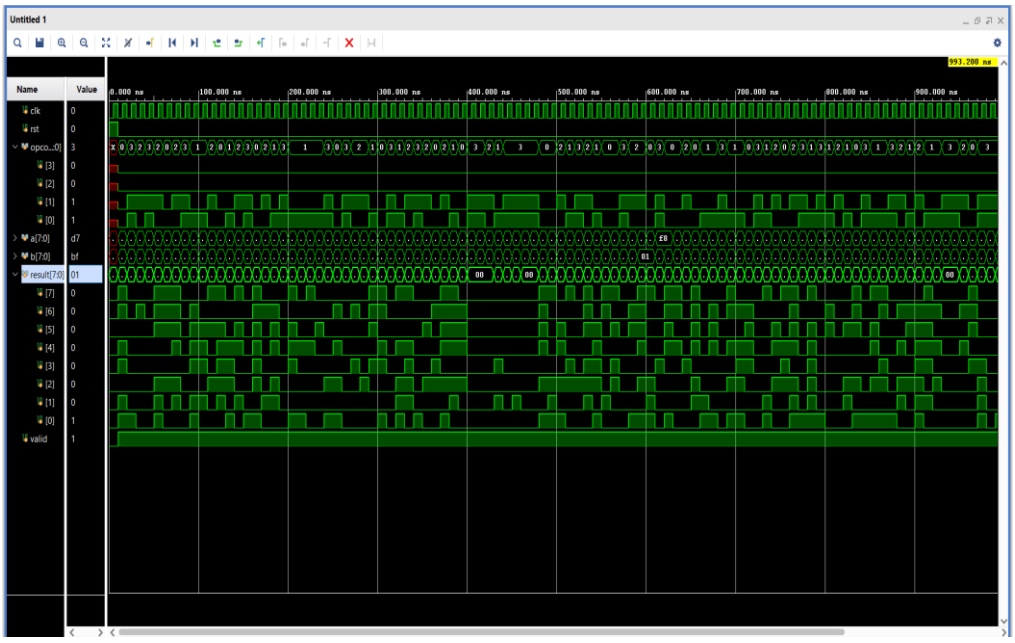
```
        property p_div_valid;
        @(posedge clk) disable iff (rst) (opcode == 4'b0011 && b == 0) |-> (valid == 0);
        endproperty
        assert property (p_div_valid)
        else $error("Error: DIV by zero should be invalid!");

        //Test bench logic
        alu_crt rand_gen;
        alu_coverage cov;

        always #5 clk = ~clk; //clock generation

         initial begin
          clk = 0; rst = 1;
          #10 rst = 0;
          rand_gen = new();
          cov = new();
          repeat (100) begin
           rand_gen. randomize();  //generate the random values
            opcode = rand_gen.opcode;
            a = rand_gen.a;
            b = rand_gen.b;
            #10;   //wait for operation
            cov.sample(opcode, a, b);  //collect coverage
            rand_gen.display();   //display inputs
          end
          $display("Functional coverage: %0.2f%%", cov.cg.get_coverage());
          $finish;
          end
endmodule
```

Elaborated Design:



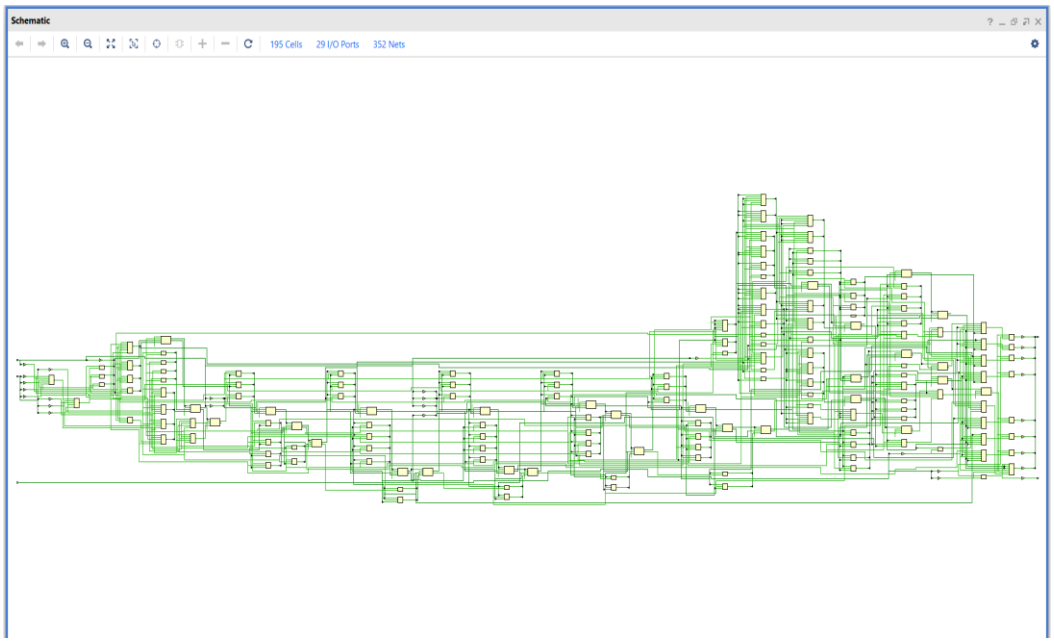
Fig(c): Elaborated dessign og ALU

Simulation:



Fig(d): Simulation of ALU

Synthesis:



Fig(e): Synthesis of ALU

10. Advantages:

- **Automation:** Uses constrained random testing for extensive validation.
- **Comprehensive Coverage:** Ensures all operation scenarios are exercised.
- **Early Error Detection:** Assertions catch design flaws immediately.
- **Scalability:** Can be extended to support additional operations.
-

Disadvantages:

- **Resource Intensive:** Verification requires significant simulation time.
- **Debugging Complexity:** Debugging constrained random test failures can be challenging.
- **Limited Hardware Testing:** Does not include FPGA synthesis and testing.
-

11. Applications:

- **Microprocessor ALUs:** Used in CPUs and DSPs for computation.
- **Embedded Systems:** Applied in embedded controllers requiring arithmetic operations.

- **Digital Signal Processing:** Used for mathematical computations in DSP applications.
- **FPGA Prototyping:** Serves as a learning tool for FPGA-based designs and testing.
-

12. Conclusion:

The **ALU** design and verification project provides a practical approach to understanding **digital arithmetic** operations and verification methodologies. Using **SystemVerilog** for verification ensures thorough testing and correctness validation. This project enhances knowledge in **digital logic design**, verification, and debugging techniques, making it an excellent foundation for further exploration in **VLSI** and **hardware design.**