Arithmetic Logical Unit Design (ALU)

Anshuman Banik
Department of Instrumentation and Electronics
Jadavpur University

October 2024

1 Problem Statement

Realize a miniature 4-bit unsigned ALU that performs the following operations depending upon a 4-bit op-code "OP":

OP	Action
0000	NOP
0001	Add two operands (operand1 + operand2)
0010	Subtract two operands (operand1 - operand2)
0011	Bit-wise AND the two operands
0100	Bit-wise XOR the two operands
0101	Bit-wise OR the two operands
0110	Generates the 2s-complement of operand1
0111	Bit-wise inverts operand1
1000	Shift operand1 left by operand2 bits. (operand2 ¡ 4)
1001	Shift operand1 right by operand2 bits. (operand2 ¡ 4)
1010	Rotate operand1 left by operand2 bits. (operand2 ¡ 4)
1011	Rotate operand1 right by operand2 bits. (operand2 ¡ 4)

1.1 Solution

The following Verilog code implements the 4-bit unsigned ALU as specified:

```
module alu (
    input [3:0] operand1,
    input [3:0] operand2,
    input [3:0] OP,
    output reg [3:0] result
);
    always @(*) begin
        case (OP)
            4'b0000: result = operand1;
            4'b0001: result = operand1 + operand2;
            4'b0010: result = operand1 - operand2;
            4'b0011: result = operand1 & operand2;
            4'b0100: result = operand1 ^ operand2;
            4'b0101: result = operand1 | operand2;
            4'b0110: result = ~operand1 + 4'b0001;
            4'b0111: result = ~operand1;
            4'b1000: result = operand1 << operand2[1:0];
            4'b1001: result = operand1 >> operand2[1:0];
            4'b1010: result = (operand1 << operand2[1:0]) | (
                operand1 >> (4 - operand2[1:0]));
            4'b1011: result = (operand1 >> operand2[1:0]) | (
                operand1 << (4 - operand2[1:0]));
            default: result = 4'b0000;
        endcase
    end
endmodule
```

1.2 Code Explanation

The Verilog code provided defines a 4-bit unsigned Arithmetic Logic Unit (ALU). Below is an explanation of the main components and functionality of the code:

- **Module Definition:** The ALU is defined as a module named `alu`. It has three inputs: `operand1`, `operand2`, and `OP`, all of which are 4-bit wide. The output `result` is also a 4-bit register.
- **Always Block:** The `always @(*)` block triggers whenever any of the inputs change, ensuring that the output `result` is updated accordingly.
- **Case Statement:** The `case` statement checks the value of the 4-bit opcode `OP`:
  - `4'b0000:` Outputs `operand1` (No operation).
  - `4'b0001:` Performs addition of `operand1` and `operand2`.
  - `4'b0010:` Performs subtraction of `operand2` from `operand1`.
  - `4'b0011:` Performs a bitwise AND operation between the two operands.
  - `4'b0100:` Performs a bitwise XOR operation.
  - `4'b0101:` Performs a bitwise OR operation.
  - `4'b0110:` Computes the 2's complement of `operand1`.
  - `4'b0111:` Computes the bitwise inversion of `operand1`.
  - `4'b1000:` Shifts `operand1` left by the number of bits specified in the lower 2 bits of `operand2`.
  - `4'b1001:` Shifts `operand1` right by the number of bits specified in the lower 2 bits of `operand2`.
  - `4'b1010:` Rotates `operand1` left by the number of bits specified in `operand2`.
  - `4'b1011:` Rotates `operand1` right by the number of bits specified in `operand2`.
  - `default:` In case of an unrecognized opcode, it sets the result to `4'b0000`.

1.3 Testbench Code

The following testbench code verifies the functionality of the 4-bit unsigned ALU:

```
module tb_alu_4bit;
    // Inputs
    reg [3:0] operand1;
    reg [3:0] operand2;
    reg [3:0] OP;

    // Outputs
    wire [3:0] result;

    // Instantiate the ALU
    alu dut (
        .operand1(operand1),
        .operand2(operand2),
        .OP(OP),
        .result(result)
    );

    // Test sequence
    initial begin
        $monitor("OP=%b operand1=%b operand2=%b result=%b", OP,
            operand1, operand2, result);

        // Test NOP (0000)
        operand1 = 4'b1010; operand2 = 4'b0001; OP = 4'b0000; #10;
        // Test Add (0001)
        operand1 = 4'b0011; operand2 = 4'b0101; OP = 4'b0001; #10;
        // Test Subtract (0010)
        operand1 = 4'b1001; operand2 = 4'b0011; OP = 4'b0010; #10;
        // Test AND (0011)
        operand1 = 4'b1100; operand2 = 4'b1010; OP = 4'b0011; #10;
        // Test XOR (0100)
        operand1 = 4'b0110; operand2 = 4'b1101; OP = 4'b0100; #10;
        // Test OR (0101)
        operand1 = 4'b0011; operand2 = 4'b0111; OP = 4'b0101; #10;
        // Test 2's complement (0110)
        operand1 = 4'b0001; operand2 = 4'b0000; OP = 4'b0110; #10;
        // Test Invert (0111)
        operand1 = 4'b0101; operand2 = 4'b0000; OP = 4'b0111; #10;
        // Test Shift Left (1000)
        operand1 = 4'b0011; operand2 = 4'b0010; OP = 4'b1000; #10;
        // Test Shift Right (1001)
        operand1 = 4'b1100; operand2 = 4'b0010; OP = 4'b1001; #10;
        // Test Rotate Left (1010)
        operand1 = 4'b1001; operand2 = 4'b0001; OP = 4'b1010; #10;
        // Test Rotate Right (1011)
        operand1 = 4'b1001; operand2 = 4'b0010; OP = 4'b1011; #10;

        $finish;
    end
endmodule
```

1.4 ALU Schematic

Figure 2 shows the design of the 4-bit ALU.

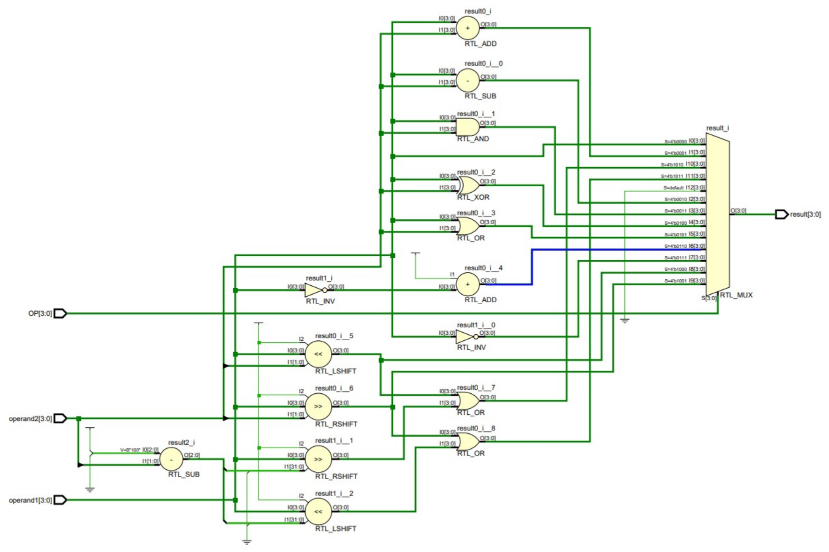

Figure 1: 4-bit ALU Design

1.5 Behavioral Simulation

Figure 2 Timing Simulation

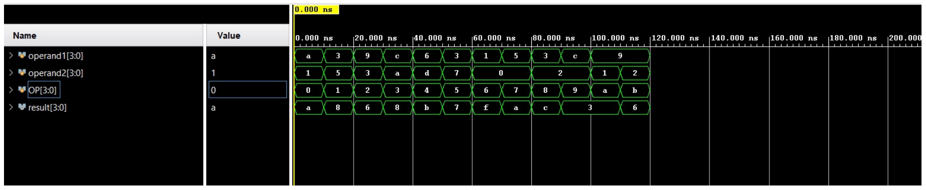

Figure 2: Simulation