Notre Dame University Bangladesh

Digital System Design Lab
CSE4106

Sequential ALU Design and Implementation

Submitted To :

Prof. Dr. Shaheena Sultana
Professor
Department of CSE

Submitted By :

Abrity Paul Chowdhury
ID : 0692130005101005
Batch : CSE 17
Submission Date: November 23, 2024

Contents

0.1 Introduction

The Sequential Arithmetic Logic Unit (ALU) is a digital circuit designed to perform arithmetic and logical operations. This system integrates a range of functionalities such as addition, subtraction, multiplication, division, logical comparisons, and bitwise operations. The schematic and provided Verilog files detail the design and implementation of a highly modular ALU, featuring JK Flip-Flops for control and sequential behavior.

0.2 Objective

The objective of the Sequential Arithmetic Logic Unit (ALU) project is to design, implement, and validate a modular digital circuit capable of performing a wide range of arithmetic and logical operations efficiently. By integrating sequential control through JK Flip-Flops, the design aims to achieve state retention and synchronization for step-by-step execution. This project seeks to provide a versatile and scalable computational unit suitable for microprocessors, embedded systems, and educational purposes, emphasizing modularity, accuracy, and functionality across diverse operations.

0.3 System Components

0.3.1 Arithmetic Logic Unit (ALU)

The ALU here is the core computational module with the following features:

- **Arithmetic Operations**
    1. Addition (RTL_ADD)
    2. Subtraction (RTL_SUB)
    3. Multiplication (RTL_MULT)
    4. Division (RTL_DIV)
- **Logical Operations**
    1. Bitwise AND (RTL_AND)
    2. OR (RTL_OR)
    3. XOR (RTL_XOR)
    4. NOT (RTL_INV)
- **Shifting Operations**
    1. Logical Left Shift (RTL_LSHIFT)
    2. Logical Right Shift (RTL_RSHIFT)

- **Comparison Operations**
  1. Greater Than (RTL_GT)
  2. Less Than (RTL_LT)
  3. Equal To (RTL_EQ)

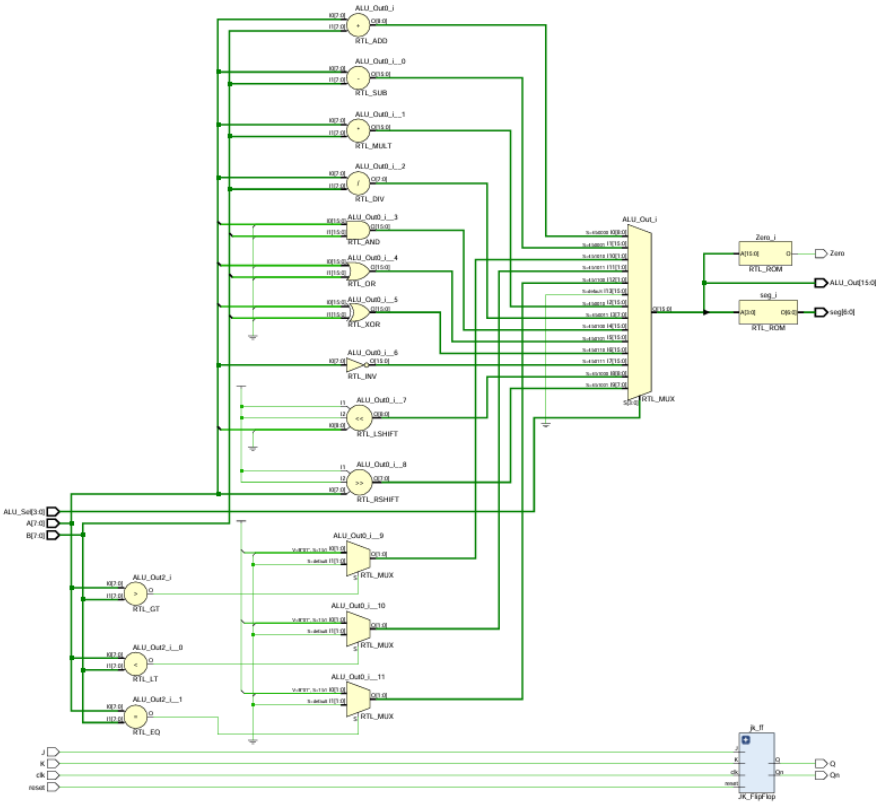

Figure1:Circuit Design of ALU

0.3.2 Control Unit

The JK Flip-Flops act as a sequential control mechanism, providing synchronization and state retention for the ALU.

0.3.3 Input and Output

- Inputs
  1. A (8 bits): First operand.

2. B (8 bits): Second operand.
3. ALU_Sel (4 bits): Selector for the operation.
4. Clock (clk) and Reset: Synchronization signals.

- Outputs

1. ALU_Out (16 bits): Result of the selected operation.
2. Zero: Indicates if the result is zero.
3. Q, Qn: State outputs from the JK Flip-Flops.
4. seg (7 bits): For display integration.

0.4 Design Highlights

0.4.1 Modularity

The design is divided into multiple modules, each corresponding to a specific operation. This modularity simplifies debugging and testing while allowing for easy extension of functionalities.

0.4.2 Sequential Design

Using JK Flip-Flops ensures that the design is capable of sequential operation. The states of the flip-flops are synchronized with the clock signal, making the ALU suitable for applications requiring step-by-step execution.

0.4.3 ROM Integration

The inclusion of ROM blocks for certain outputs indicates additional flexibility, likely for predefined data storage or look-up operations.

0.5 Schematic Review

The schematic illustrates the connectivity between ALU modules, control signals, and input/output ports.
Key observations include:

- Clear separation of arithmetic and logical operations.
- Comprehensive wiring of selector signals (ALU_Sel) to corresponding modules.
- Effective use of buses to handle multi-bit data.

```
module Sequential_ALU (
    input [7:0] A, B,
    input [3:0] ALU_Sel,
    input J, K, clk, reset, // Inputs for JK flip-flop
    output reg [15:0] ALU_Out,
    output reg Zero,
    output reg [6:0] seg, // 7-segment display output
    output Q, Qn // Outputs from JK flip-flop
);

// Instantiate the JK flip-flop
JK_FlipFlop jk_ff (
    .J(J),
    .K(K),
    .clk(clk),
    .reset(reset),
    .Q(Q),
    .Qn(Qn)
);

always @(*) begin
    case (ALU_Sel)
        4'b0000: ALU_Out = A + B; // Addition
        4'b0001: ALU_Out = A - B; // Subtraction
        4'b0010: ALU_Out = A * B; // Multiplication
        4'b0011: ALU_Out = A / B; // Division
        4'b0100: ALU_Out = A & B; // AND
        4'b0101: ALU_Out = A | B; // OR
        4'b0110: ALU_Out = A ^ B; // XOR
        4'b0111: ALU_Out = ~A;    // NOT
        4'b1000: ALU_Out = A << 1; // Left Shift
        4'b1001: ALU_Out = A >> 1; // Right Shift
        4'b1010: ALU_Out = (A > B) ? 16'b1 : 16'b0; // Greater Than
        4'b1011: ALU_Out = (A < B) ? 16'b1 : 16'b0; // Less Than
        4'b1100: ALU_Out = (A == B) ? 16'b1 : 16'b0; // Equal To
        default: ALU_Out = 16'b0; // Default case
    endcase
```

Figure2: Initial Design of ALU part 1

```
    // Zero flag
    if (ALU_Out == 16'b0)
        Zero = 1;
    else
        Zero = 0;

    // 7-segment display logic (example for displaying ALU_Out[3:0])
    case (ALU_Out[3:0])
        4'b0000: seg = 7'b1000000; // 0
        4'b0001: seg = 7'b1111001; // 1
        4'b0010: seg = 7'b0100100; // 2
        4'b0011: seg = 7'b0110000; // 3
        4'b0100: seg = 7'b0011001; // 4
        4'b0101: seg = 7'b0010010; // 5
        4'b0110: seg = 7'b0000010; // 6
        4'b0111: seg = 7'b1111000; // 7
        4'b1000: seg = 7'b0000000; // 8
        4'b1001: seg = 7'b0010000; // 9
        4'b1010: seg = 7'b0001000; // A
        4'b1011: seg = 7'b0000011; // b
        4'b1100: seg = 7'b1000110; // C
        4'b1101: seg = 7'b0100001; // d
        4'b1110: seg = 7'b0000110; // E
        4'b1111: seg = 7'b0001110; // F
        default: seg = 7'b1111111; // Blank
    endcase
end

endmodule
```

Figure3: Initial Design of ALU part 2

JK_FlipFlop.v

C: > Users > Abrity > Downloads > JK_FlipFlop.v

```
1   module JK_FlipFlop (
2       input J, K, clk, reset,
3       output reg Q, Qn
4   );
5       always @(posedge clk or posedge reset) begin
6           if (reset) begin
7               Q <= 0;
8               Qn <= 1;
9           end else begin
10              case ({J, K})
11                  2'b00: ; // No change
12                  2'b01: Q <= 0;
13                  2'b10: Q <= 1;
14                  2'b11: Q <= ~Q;
15              endcase
16              Qn <= ~Q;
17          end
18      end
19  endmodule
20
```

Figure4: Initial Design of JK Flip Flop

0.6 Testing Framework

The provided testbench (ALU_tb.v) ensures thorough verification of the ALU's functionality by:

- Applying varied input combinations to test all operations.
- Verifying the correctness of outputs, including edge cases like division by zero and overflows.
- Simulating the sequential behavior of the ALU under varying clock and reset conditions

```
`timescale 1ns / 1ps

module ALU_tb;
    // Testbench signals
    reg [7:0] A, B;
    reg [3:0] ALU_Sel;
    reg J, K, clk, reset;
    wire [15:0] ALU_Out;
    wire Zero;
    wire [6:0] seg;
    wire Q, Qn;

    // Instantiate the ALU module
    Sequential_ALU uut (
        .A(A),
        .B(B),
        .ALU_Sel(ALU_Sel),
        .J(J),
        .K(K),
        .clk(clk),
        .reset(reset),
        .ALU_Out(ALU_Out),
        .Zero(Zero),
        .seg(seg),
        .Q(Q),
        .Qn(Qn)
    );

    // Generate a clock signal
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // Clock with a period of 10 units
    end

    // Test scenarios
    initial begin
```

Figure5: Testbench of ALU part1

```
module ALU_tb;
    $dumpfile("dump.vcd");  // Specifies the name of the VCD file
    $dumpvars(0, ALU_tb);    // Dumps variables from the testbench module
end
    initial begin
        // Initialize inputs
        A = 8'b00000010; // Example value for A
        B = 8'b00000001; // Example value for B
        ALU_Sel = 4'b0000; // Initialize ALU_Sel
        J = 0;
        K = 1;
        reset = 1; // Start with reset active

        #10 reset = 0; // Release reset after 10 time units

        // Test addition
        ALU_Sel = 4'b0000; // Addition operation
        #10; // Wait 10 time units

        // Test subtraction
        ALU_Sel = 4'b0001; // Subtraction operation
        #10;

        // Test multiplication
        ALU_Sel = 4'b0010; // Multiplication operation
        #10;

        // Test division
        ALU_Sel = 4'b0011; // Division operation
        #10;

        // Test AND operation
        ALU_Sel = 4'b0100;
        #10;

        // Test OR operation
        ALU_Sel = 4'b0101;
```

Figure6: Testbench of ALU part2

```
initial begin
    // Test OR operation
    ALU_Sel = 4'b0101;
    #10;

    // Test XOR operation
    ALU_Sel = 4'b0110;
    #10;

    // Test NOT operation
    ALU_Sel = 4'b0111;
    #10;

    // Test Greater Than
    ALU_Sel = 4'b1010;
    #10;

    // Test Equal To
    ALU_Sel = 4'b1100;
    #10;

    // Toggle JK flip-flop inputs to test flip-flop behavior
    J = 1;
    K = 0;
    #10;
    J = 1;
    K = 1;
    #10;

    // End the simulation
    $finish;
end

// Monitor the outputs
initial begin
    $monitor("Time=%0t | A=%0d B=%0d ALU_Sel=%b | ALU_Out=%0d Zero=%b seg=%b | Q=%b Qn=%b",
```

Figure7: Testbench of ALU part3

```
        // End the simulation
        $finish;
    end

    // Monitor the outputs
    initial begin
        $monitor("Time=%0t | A=%0d B=%0d ALU_Sel=%b | ALU_Out=%0d Zero=%b seg=%b | Q=%b Qn=%b",
                 $time, A, B, ALU_Sel, ALU_Out, Zero, seg, Q, Qn);
    end

endmodule
```

Figure8: Testbench of ALU part4

0.7 Conclusion

The Sequential ALU design effectively combines arithmetic and logical operations with sequential control, providing a versatile and modular solution. The provided implementation, supported by a detailed schematic and testing framework, is well-suited for both practical and educational purposes.