

# Design a 4-bit ALU

Group: 4 CSE460 Lab Section 9

ATHAR NOOR MOHAMMAD RAFEE

DEPT: CSE

ID: 20101396

Section: 9L

noor.mohammad.rafee@g.bracu.ac.bd

A.S.M MAHABUB SIDDIQUI

DEPT: CSE

ID: 20301040

Section: 9L

asm.mahabub.siddiqui@g.bracu.ac.bd

Ayon Das

DEPT: CSE

ID: 20301099

Section: 9L

ayon.das@g.bracu.ac.bd

MD. SAKIB

DEPT: CSE

ID: 20301180

Section: 9L

md.sakib1@g.bracu.ac.bd

MOHAMMED INZAM UL AZAM

DEPT: CSE

ID: 20101144

Section: 09L

mohammed.inzam.ul.azam@g.bracu.ac.bd

**Abstract**—This project presents the design and implementation of a 4-bit Arithmetic Logic Unit (ALU). The ALU performs arithmetic and logical operations on two 4-bit inputs and produces a 4-bit output. The design is implemented using Verilog hardware description language and simulated using timing function. The ALU supports basic arithmetic operations such as addition and subtraction, as well as logical operations such as ADD, NAND, and XNOR as per requirements of the project. Overall, this project demonstrates the design and implementation of a simple but functional sequential ALU using Verilog HDL.

**Index Terms**—ALU, Verilog HDL, opcode, timing diagram, SUB, XNOR, NAND, ADD, RESET

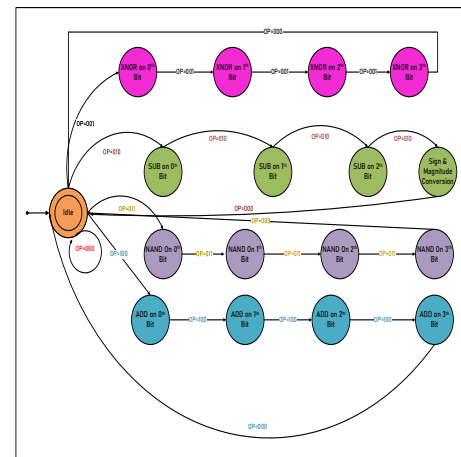
## I. INTRODUCTION

This report presents the design and implementation of a 4-bit ALU using Verilog HDL and Quartus II software. The ALU was designed to perform various arithmetic and logical operations such as **ADDITION**, **SUBTRACTION**, bitwise **AND**, bitwise **NAND**, and bitwise **XNOR**. The design consists of various modules such as the Adder, Subtractor, and logic gates which were generated based on the verilog code. In this report, we provide a detailed description of the design and implementation process, including the Verilog code for each module and the timing diagram for verification. We also discuss the challenges encountered during the design process and how they were overcome. Finally, we present the results of the hardware testing, demonstrating that the ALU is capable of performing the desired operations accurately and efficiently. The design of a 4-bit ALU is an essential component in digital circuit design, and it is a fundamental building block in many larger circuits and VLSI design.

## II. FINITE STATE MACHINE DESIGN AND IMPLEMENTATION

Finite State Machine (FSM) is a model for designing sequential logic circuits, where the circuit's behavior is determined by a finite number of states, inputs and outputs. In this case, the FSM is designed to implement *five* different

operations, namely RESET, XNOR, NAND, SUB, and ADD on two 4-bit inputs A and B. The way we coded the Verilog code represents the implementation of the FSM, which is designed to perform the above-mentioned arithmetic and logical operations on the given input values. From an high level perspective, The FSM has Four states, which are encoded as 2-bit values, as follows:



of the input values and transitions back to the initial state.

Before transition, it also sets the values of *zero* flag, *sign* flag and *carry* flag.

Things are checked and done slightly different based on the *opcode*. It can be observed from the above Fig 1 clearly. The four different operations are implemented using a *case* statement with *opcode* as the selector. Each operation *case* statement contains the logic required to perform the operation on the given input values, and update the output values of the circuit accordingly. For example, for the **ADD** operation, the code first calculates the SUM of the LSBs of the input values, adds the carry value to it (initially 0), and assigns the SUM and the carry value to the output register **C**. Then, it updates the *zero* flag, which is set to 1 if the output is 0, and transitions to the next state which is **IDLE** state that we can see from Figure 1. The outputs of the circuit include **C**, which stores the result of the operation, *carr*, which is the **carry** bit generated during addition or subtraction, *sign*, which is the sign bit of the output value, and *zero*, which is set to 1 if the output is 0000. Overall, the FSM implementation allows the circuit to perform different arithmetic and logical operations on the given input values, and update the output values based on the operation performed.

### III. VERIFICATION

After implementing the verilog code. We used the timing function to verify the working of the code. Below, we are attaching the screenshots from the timing diagram

#### A. ADD Operation:

Below figure 2 shows the **ADD** operation between two binary  $A = 1111$  and  $B = 1111$  numbers where the result is stored in register **C** sequentially in each clock cycles.

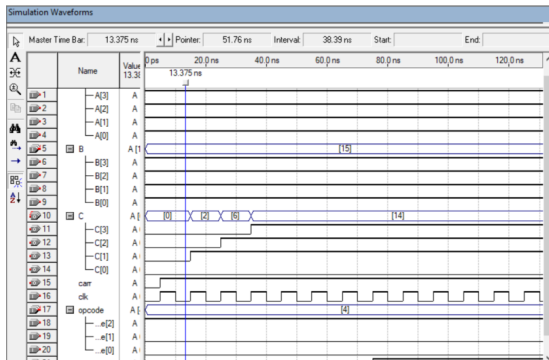


Fig. 2. Timing Diagram for ADD Operation

#### B. SUB Operation

Below figure 3 shows the **SUB** operation between two binary  $A = 0111$  and  $B = 0111$  where the output  $C = 0000$ . As we are working with signed number, the numbers are represented as **2s complement**.

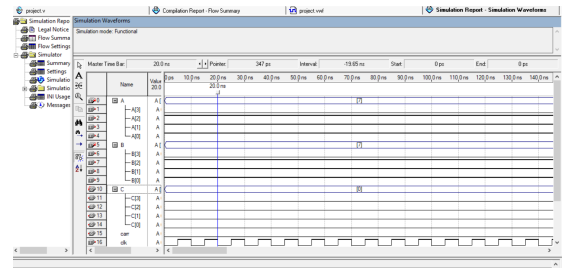


Fig. 3. Timing Diagram for Sub Operation

**NAND** and **XNOR** operation are pretty much straight forward. All we had to do is put the equation in the verilog and then the operation performed as expected. Below timing diagram, those operations are attached.

#### C. NAND Operation

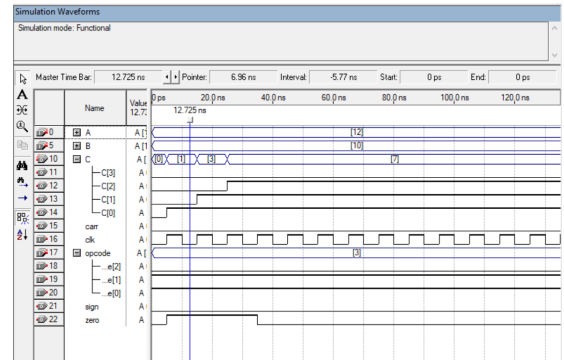


Fig. 4. Timing Diagram for NAND Operation

#### D. XNOR Operation

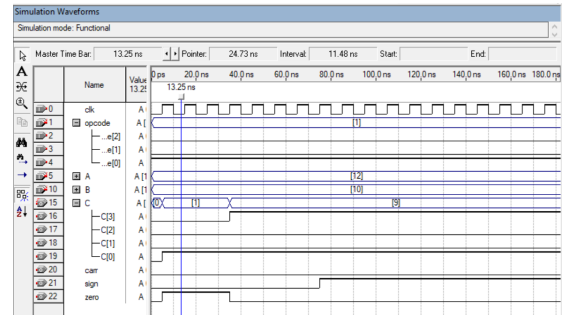


Fig. 5. Timing Diagram for XNOR Operation

#### E. Multiple Operation With Reset

The below attached Figure 6 shows multiple Operations with **SET** and **RESET** functionality. At first the **SUB** operation is performed between binary number  $0111$  and  $0111$ . After that, the **opcode** is changed to reset which is  $000$  during time  $40ns$  to  $50ns$ . Next, the **opcode** was changed to  $011$  and performed **NANAD** operation during the next 4 clock cycles.

After that the mandatory **RESET** and then **ADD** operation for another 4 cycles.

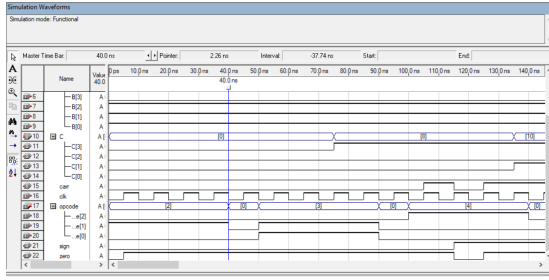


Fig. 6. Timing Daigram with reset

#### IV. CONCLUSION

In conclusion, we have successfully designed and implemented a 4-bit ALU using finite state machines and *Verilog HDL*. We started by defining the project requirements and selecting the appropriate operations to be implemented. Then, we designed the FSM with four states and carefully defined the transitions and outputs for each state. We also implemented the FSM using Verilog HDL and simulated the design using timing diagram to verify its functionality. The simulation results show that our design works correctly for all the selected operations and input combinations. Finally, this project not only provided hands-on experience with Verilog HDL programming and digital circuit design, but also reinforced the importance of a systematic approach to problem-solving and the importance of utilizing efficient design strategies.

#### APPENDIX

##### A. Verilog HDL Code

```

1 module project(input clk, input [3:0] A,
2   input [3:0] B,
3   input [2:0] opcode, output reg [3:0] C,
4   output reg carr, output reg sign, output
5   reg zero);
6 // Will be using state to indicate state of
7 // the machine.
8 reg [1:0] state = 0;
9 // Negative numbers will be represented in 2
10 // s complement.
11 reg signed [3:0] sub_result;
12 reg signed [3:0] sub_a;
13 reg signed [3:0] sub_b;
14 always @ (posedge clk) begin
15   case (state)
16     2'b00: begin
17       case (opcode)
18         3'b000: begin
19           C <= 4'b0000; //RESET
20           operation
21           carr <= 1'b0;
22           sign <= 1'b0;
23           zero <= 1'b1;
24         end
25         3'b001: begin //XNOR
26           operation on LSBs
27           C[0] <= ~(A[0] ^ B[0]);

```

```

28   zero <= C[0] == 1'b0;
29 end
30 3'b010: begin //SUB
31   operation on LSBs
32   sub_a <= A;
33   sub_b <= B;
34   sub_result <= sub_b -
35   sub_a;
36   C[0] <= sub_result[0];
37   zero <= C[0] == 1'b0;
38 end
39 3'b011: begin //NAND
40   operation on LSBs
41   C[0] <= ~(A[0] & B[0]);
42   zero <= C[0] == 1'b0;
43 end
44 3'b100: begin //ADD
45   operation on LSBs
46   {carr, C[0]} <= A[0] + B
47   [0];
48   zero <= C[0] == 1'b0;
49 end
50 endcase
51 state <= 2'b01;
52 end
53 2'b01: begin
54   case (opcode)
55     3'b001: begin //XNOR
56       operation on next bit
57       C[1] <= ~(A[1] ^ B[1]);
58       zero <= zero & (C[1] ==
59       1'b0);
60     end
61     3'b010: begin //SUB
62       operation on next bit
63       sub_a <= A;
64       sub_b <= B;
65       sub_result <= sub_b -
66       sub_a;
67       C[1] <= sub_result[1];
68       zero <= C[1] == 1'b0;
69     end
70     3'b011: begin //NAND
71       operation on next bit
72       C[1] <= ~(A[1] & B[1]);
73       zero <= zero & (C[1] ==
74       1'b0);
75     end
76     3'b100: begin //ADD
77       operation on next bit
78       {carr, C[1]} <= A[1] + B
79       [1] + carr;
80       zero <= zero & (C[1] ==
81       1'b0);
82     end
83   end
84 endcase
85 state <= 2'b10;
86 end
87 2'b10: begin
88   case (opcode)
89     3'b001: begin //XNOR
90       operation on next bit
91       C[2] <= ~(A[2] ^ B[2]);

```

```

74         zero <= zero & (C[2] == 1'b0);
75     end
76     3'b010: begin //SUB
77         operation on next bit
78         sub_a <= A;
79         sub_b <= B;
80         sub_result <= sub_b -
81             sub_a;
82         C[2] <= sub_result[2];
83         zero <= C[2] == 1'b0;
84     end
85     3'b011: begin //NAND
86         operation on next bit
87         C[2] <= ~(A[2] & B[2]);
88         zero <= zero & (C[2] ==
89             1'b0);
90     end
91     3'b100: begin //ADD
92         operation on next bit
93         {carr, C[2]} <= A[2] + B
94             [2] + carr;
95         zero <= zero & (C[2] ==
96             1'b0);
97     end
98 endcase
99 state <= 2'b11;
100 end
101 2'b11: begin
102     case (opcode)
103     3'b001: begin //XNOR
104         operation on MSBs
105         C[3] <= ~(A[3] ^ B[3]);
106         sign <= C[3];
107         zero <= C == 4'b0000;
108     end
109     3'b010: begin //SUB
110         operation on MSBs
111         sub_a <= A;
112         sub_b <= B;
113         {carr, sub_result} <=
114             sub_b - sub_a;
115         C[3] <= sub_result[3];
116         zero <= C == 4'b0000;
117     end
118     3'b011: begin //NAND
119         operation on MSBs
120         C[3] <= ~(A[3] & B[3]);
121         sign <= C[3];
122         zero <= C == 4'b0000;
123     end
124     3'b100: begin //ADD
125         operation on MSBs
126         {carr, C[3]} <= A[3] + B

```

127

128 endmodule

Listing 1. Verilog code for 4-bit ALU