

# 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 OR, and bitwise XOR. 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 four different

operations, namely RESET, XNOR, 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. The FSM has four states, which are encoded as 2-bit values, as follows:

- State 0 (2'b00): In this state, the circuit performs the selected operation on the first bit of the input values and transitions to the next state.
- State 1 (2'b01): In this state, the circuit performs the selected operation on the second bit of the input values and transitions to the next state.
- State 2 (2'b10): In this state, the circuit performs the selected operation on the third bit of the input values and transitions to the next state.
- State 3 (2'b11): In this state, the circuit performs the selected operation on the fourth and most significant bit 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*.

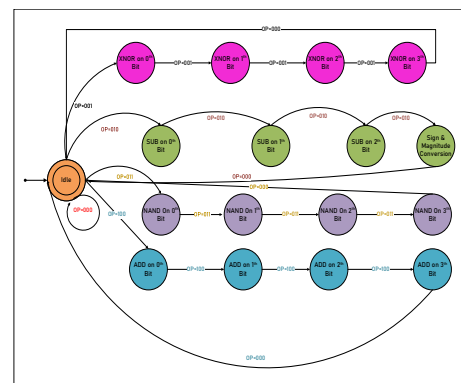


Fig. 1. FSM Diagram

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 we can see from Fig 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 zero. 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  $B = 1111$  where the result is stored C sequentially in each clock cycles.

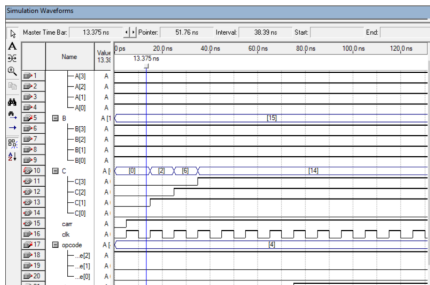


Fig. 2. Timing Daigram for ADD Operation

#### B. SUB Operation

Below figure 3 shows the **SUB** operation between two binary  $A = 0111$  and  $B = 0111$

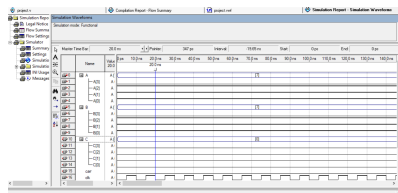


Fig. 3. Timing Diagram for Sub Operation

**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

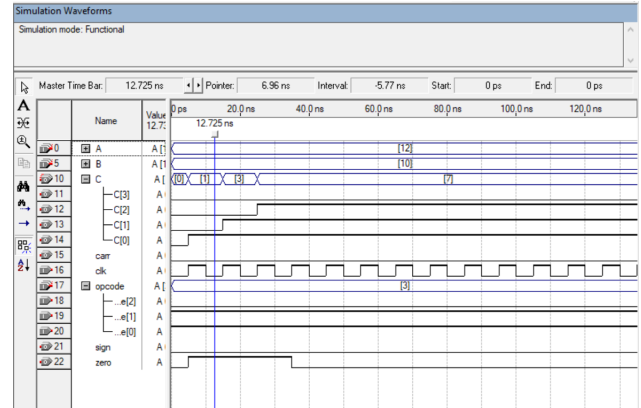


Fig. 4. Timing Daigram for NAND Operation

#### D. XNOR Operation

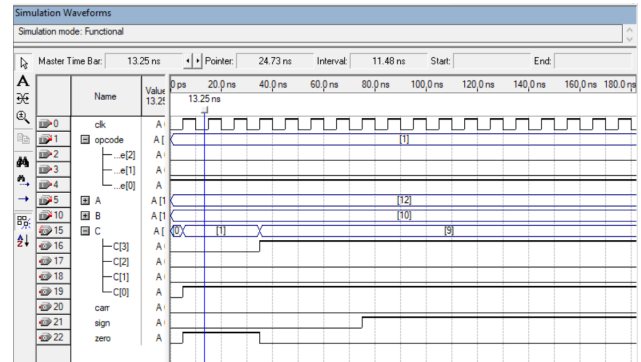


Fig. 5. Timing Daigram for XNOR Operation

#### 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 an **RESET** and then **ADD** operation for another 4 cycles.

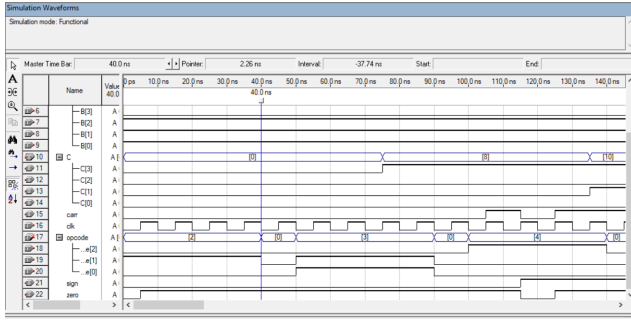


Fig. 6. Timing Daigram with reset

#### 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 always @ (posedge clk) begin
10     case (state)
11         2'b00: begin
12             case (opcode)
13                 3'b000: begin
14                     C <= 4'b0000; //RESET
15                     operation
16                     carr <= 1'b0;
17                     sign <= 1'b0;
18                     zero <= 1'b1;
19                 end
20                 3'b001: begin //XNOR
21                     operation on LSBs
22                     C[0] <= ~(A[0] ^ B[0]);
23                     zero <= C[0] == 1'b0;
24                 end
25                 3'b010: begin //SUB
26                     operation on LSBs
27                     {carr, C[0]} <= B[0] - A
28                     [0];
29                     zero <= C[0] == 1'b0;
30                 end
31                 3'b011: begin //NAND
32                     operation on LSBs
33                     C[0] <= ~(A[0] & B[0]);
34                     zero <= C[0] == 1'b0;
35                 end
36                 3'b100: begin //ADD
37                     operation on LSBs
38                     {carr, C[0]} <= A[0] + B
39                     [0];
40                     zero <= C[0] == 1'b0;
41                 end
42                 3'b101: begin //SUB
43                     operation on next bit
44                     {carr, C[1]} <= B[1] - A
45                     [1] - carr;
46                     zero <= zero & (C[1] ==
47                     1'b0);
48                 end
49                 3'b110: begin //NAND
50                     operation on next bit
51                     C[1] <= ~(A[1] & B[1]);
52                     zero <= zero & (C[1] ==
53                     1'b0);
54                 end
55                 3'b111: begin //NAND
56                     operation on next bit
57                     C[2] <= ~(A[2] & B[2]);
58                     zero <= zero & (C[2] ==
59                     1'b0);
60                 end
61             endcase
62             state <= 2'b01;
63         endcase
64     end
65 end
66
67 2'b01: begin
68     case (opcode)
69         3'b001: begin //XNOR
70             operation on next bit
71             C[1] <= ~(A[1] ^ B[1]);
72             zero <= zero & (C[1] ==
73             1'b0);
74         end
75         3'b010: begin //SUB
76             operation on next bit
77             {carr, C[1]} <= B[1] - A
78             [1] - carr;
79             zero <= zero & (C[1] ==
80             1'b0);
81         end
76         3'b011: begin //NAND
77             operation on next bit
78             C[1] <= ~(A[1] & B[1]);
79             zero <= zero & (C[1] ==
80             1'b0);
81         end
82         3'b100: begin //ADD
83             operation on next bit
84             {carr, C[1]} <= A[1] + B
85             [1] + carr;
86             zero <= zero & (C[1] ==
87             1'b0);
88         end
89         3'b101: begin //SUB
90             operation on next bit
91             {carr, C[2]} <= B[2] - A
92             [2] - (carr & ~zero);
93             zero <= zero & (C[2] ==
94             1'b0);
95         end
96         3'b110: begin //NAND
97             operation on next bit
98             C[2] <= ~(A[2] & B[2]);
99             zero <= zero & (C[2] ==
100            1'b0);
101         end
102     endcase
103     state <= 2'b10;
104 end
105
106 2'b10: begin
107     case (opcode)
108         3'b001: begin //XNOR
109             operation on next bit
110             C[2] <= ~(A[2] ^ B[2]);
111             zero <= zero & (C[2] ==
112             1'b0);
113         end
114         3'b010: begin //SUB
115             operation on next bit
116             {carr, C[2]} <= B[2] - A
117             [2] - (carr & ~zero);
118             zero <= zero & (C[2] ==
119             1'b0);
120         end
121         3'b011: begin //NAND
122             operation on next bit
123             C[2] <= ~(A[2] & B[2]);
124             zero <= zero & (C[2] ==
125             1'b0);
126         end
127         3'b100: begin //ADD
128             operation on next bit
129             {carr, C[2]} <= A[2] + B
130             [2] + carr;
131             zero <= zero & (C[2] ==
132             1'b0);
133         end
134         3'b101: begin //SUB
135             operation on next bit
136             {carr, C[3]} <= B[3] - A
137             [3] - carr;
138             zero <= zero & (C[3] ==
139             1'b0);
140         end
141         3'b110: begin //NAND
142             operation on next bit
143             C[3] <= ~(A[3] & B[3]);
144             zero <= zero & (C[3] ==
145             1'b0);
146         end
147         3'b111: begin //NAND
148             operation on next bit
149             C[3] <= ~(A[3] & B[3]);
150             zero <= zero & (C[3] ==
151             1'b0);
152         end
153     endcase
154     state <= 2'b11;
155 end
156
157 2'b11: begin
158     case (opcode)
159         3'b001: begin //XNOR
160             operation on next bit
161             C[3] <= ~(A[3] ^ B[3]);
162             zero <= zero & (C[3] ==
163             1'b0);
164         end
165         3'b010: begin //SUB
166             operation on next bit
167             {carr, C[3]} <= B[3] - A
168             [3] - carr;
169             zero <= zero & (C[3] ==
170             1'b0);
171         end
172         3'b011: begin //NAND
173             operation on next bit
174             C[3] <= ~(A[3] & B[3]);
175             zero <= zero & (C[3] ==
176             1'b0);
177         end
178         3'b100: begin //ADD
179             operation on next bit
180             {carr, C[3]} <= A[3] + B
181             [3] + carr;
182             zero <= zero & (C[3] ==
183             1'b0);
184         end
185         3'b101: begin //SUB
186             operation on next bit
187             {carr, C[4]} <= B[4] - A
188             [4] - carr;
189             zero <= zero & (C[4] ==
190             1'b0);
191         end
192         3'b110: begin //NAND
193             operation on next bit
194             C[4] <= ~(A[4] & B[4]);
195             zero <= zero & (C[4] ==
196             1'b0);
197         end
198         3'b111: begin //NAND
199             operation on next bit
200             C[4] <= ~(A[4] & B[4]);
201             zero <= zero & (C[4] ==
202             1'b0);
203         end
204     endcase
205     state <= 2'b00;
206 end
207 end

```

```

23 end
24 3'b011: begin //NAND
25     operation on LSBs
26     C[0] <= ~(A[0] & B[0]);
27     zero <= C[0] == 1'b0;
28 end
29 3'b100: begin //ADD
30     operation on LSBs
31     {carr, C[0]} <= A[0] + B
32     [0];
33     zero <= C[0] == 1'b0;
34 end
35
36 endcase
37 state <= 2'b01;
38 end
39 2'b01: begin
40     case (opcode)
41         3'b001: begin //XNOR
42             operation on next bit
43             C[1] <= ~(A[1] ^ B[1]);
44             zero <= zero & (C[1] ==
45             1'b0);
46         end
47         3'b010: begin //SUB
48             operation on next bit
49             {carr, C[1]} <= B[1] - A
50             [1] - carr;
51             zero <= zero & (C[1] ==
52             1'b0);
53         end
54         3'b011: begin //NAND
55             operation on next bit
56             C[1] <= ~(A[1] & B[1]);
57             zero <= zero & (C[1] ==
58             1'b0);
59         end
60         3'b100: begin //ADD
61             operation on next bit
62             {carr, C[1]} <= A[1] + B
63             [1] + carr;
64             zero <= zero & (C[1] ==
65             1'b0);
66         end
67         3'b101: begin //SUB
68             operation on next bit
69             {carr, C[2]} <= B[2] - A
70             [2] - carr;
71             zero <= zero & (C[2] ==
72             1'b0);
73         end
74         3'b110: begin //NAND
75             operation on next bit
76             C[2] <= ~(A[2] & B[2]);
77             zero <= zero & (C[2] ==
78             1'b0);
79         end
80         3'b111: begin //NAND
81             operation on next bit
82             C[2] <= ~(A[2] & B[2]);
83             zero <= zero & (C[2] ==
84             1'b0);
85         end
86     endcase
87     state <= 2'b10;
88 end
89
90 2'b10: begin
91     case (opcode)
92         3'b001: begin //XNOR
93             operation on next bit
94             C[2] <= ~(A[2] ^ B[2]);
95             zero <= zero & (C[2] ==
96             1'b0);
97         end
98         3'b010: begin //SUB
99             operation on next bit
100            {carr, C[2]} <= B[2] - A
101            [2] - carr;
102            zero <= zero & (C[2] ==
103            1'b0);
104         end
105         3'b011: begin //NAND
106             operation on next bit
107             C[2] <= ~(A[2] & B[2]);
108             zero <= zero & (C[2] ==
109             1'b0);
110         end
111         3'b100: begin //ADD
112             operation on next bit
113             {carr, C[2]} <= A[2] + B
114             [2] + carr;
115             zero <= zero & (C[2] ==
116             1'b0);
117         end
118         3'b101: begin //SUB
119             operation on next bit
120             {carr, C[3]} <= B[3] - A
121             [3] - carr;
122             zero <= zero & (C[3] ==
123             1'b0);
124         end
125         3'b110: begin //NAND
126             operation on next bit
127             C[3] <= ~(A[3] & B[3]);
128             zero <= zero & (C[3] ==
129             1'b0);
130         end
131         3'b111: begin //NAND
132             operation on next bit
133             C[3] <= ~(A[3] & B[3]);
134             zero <= zero & (C[3] ==
135             1'b0);
136         end
137     endcase
138     state <= 2'b11;
139 end
140
141 2'b11: begin
142     case (opcode)
143         3'b001: begin //XNOR
144             operation on next bit
145             C[3] <= ~(A[3] ^ B[3]);
146             zero <= zero & (C[3] ==
147             1'b0);
148         end
149         3'b010: begin //SUB
150             operation on next bit
151             {carr, C[3]} <= B[3] - A
152             [3] - carr;
153             zero <= zero & (C[3] ==
154             1'b0);
155         end
156         3'b011: begin //NAND
157             operation on next bit
158             C[3] <= ~(A[3] & B[3]);
159             zero <= zero & (C[3] ==
160             1'b0);
161         end
162         3'b100: begin //ADD
163             operation on next bit
164             {carr, C[3]} <= A[3] + B
165             [3] + carr;
166             zero <= zero & (C[3] ==
167             1'b0);
168         end
169         3'b101: begin //SUB
170             operation on next bit
171             {carr, C[4]} <= B[4] - A
172             [4] - carr;
173             zero <= zero & (C[4] ==
174             1'b0);
175         end
176         3'b110: begin //NAND
177             operation on next bit
178             C[4] <= ~(A[4] & B[4]);
179             zero <= zero & (C[4] ==
180             1'b0);
181         end
182         3'b111: begin //NAND
183             operation on next bit
184             C[4] <= ~(A[4] & B[4]);
185             zero <= zero & (C[4] ==
186             1'b0);
187         end
188     endcase
189     state <= 2'b00;
190 end
191 end

```

```

71         C[2] <= ~(A[2] & B[2]);
72         zero <= zero & (C[2] ==
73             1'b0);
74     end
75     3'b100: begin //ADD
76         operation on next bit
77         {carr, C[2]} <= A[2] + B
78             [2] + carr;
79         zero <= zero & (C[2] ==
80             1'b0);
81     end
82     endcase
83     state <= 2'b11;
84 end
85 2'b11: begin
86     case (opcode)
87     3'b001: begin //XNOR
88         operation on MSBs
89         C[3] <= ~(A[3] ^ B[3]);
90         sign <= C[3];
91         zero <= C == 4'b0000;
92     end
93     3'b010: begin //SUB
94         operation on MSBs
95         {carr, C[3]} <= B[3] - A[3]
96             - carr;
97         sign <= C[3];
98         zero <= C == 4'b0000;
99         if (B[3] < A[3]) begin //
100             if result is negative,
101             take two's complement
102             of result
103             C <= ~C + 4'b0001;
104             sign <= C[3];
105         end
106     end
107     3'b011: begin //NAND
108         operation on MSBs
109         C[3] <=
110             ~(A
111                 [3] &
112                 B
113                 [3]);
114         sign <= C[3];
115         zero <= C == 4'b0000;
116     end
117     3'b100: begin //ADD
118         operation on MSBs
119         {carr, C[3]} <= A[3] + B
120             [3] + carr;
121         sign <= C[3];
122         zero <= C == 4'b0000;
123     end
124     endcase
125     state <= 2'b00;
126 end
127 endcase
128 end
129 endmodule

```

Listing 1. Verilog code for 4-bit ALU