

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, arithmetic, logic, simulation

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*.

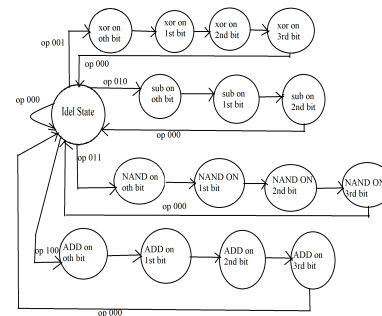


Fig. 1. FSM Diagram

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

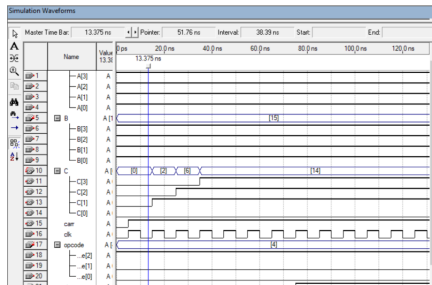


Fig. 2. Timing Diagram for ADD Operation

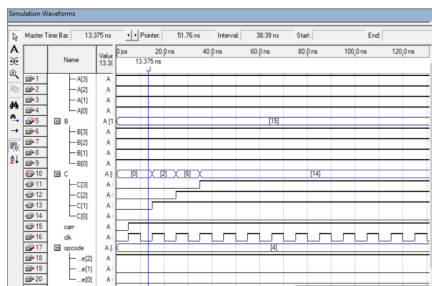


Fig. 3. Broken operation.

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.

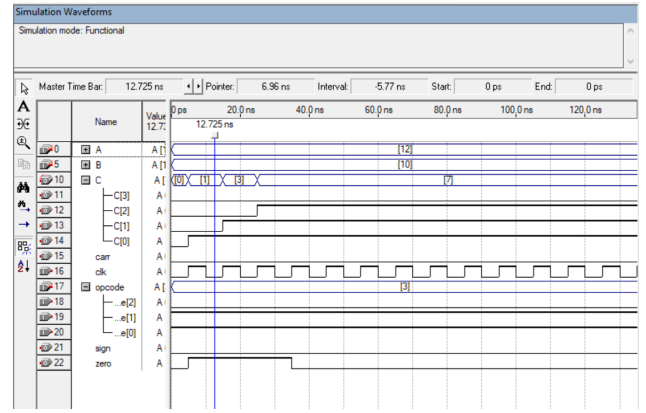


Fig. 4. Timing Daigram for NAND Operation

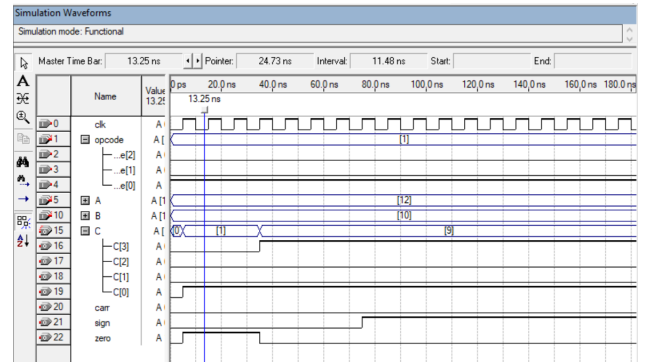


Fig. 5. Timing Diagram for XNOR Operation

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.
```

```

5 reg [1:0] state = 0;
6 always @ (posedge clk) begin
7     case (state)
8         2'b00: begin
9             case (opcode)
10                3'b000: begin
11                    C <= 4'b0000; //RESET
12                    operation
13                    carr <= 1'b0;
14                    sign <= 1'b0;
15                    zero <= 1'b1;
16                end
17                3'b001: begin //XNOR
18                    operation on LSBs
19                    C[0] <= ~(A[0] ^ B[0]);
20                    zero <= C[0] == 1'b0;
21                end
22                3'b010: begin //SUB
23                    operation on LSBs
24                    {carr, C[0]} <= B[0] - A
25                        [0];
26                    zero <= C[0] == 1'b0;
27                end
28                3'b011: begin //NAND
29                    operation on LSBs
30                    C[0] <= ~(A[0] & B[0]);
31                    zero <= C[0] == 1'b0;
32                end
33                3'b100: begin //ADD
34                    operation on LSBs
35                    {carr, C[0]} <= A[0] + B
36                        [0];
37                    zero <= C[0] == 1'b0;
38                end
39            endcase
40            state <= 2'b01;
41        end
42        2'b01: begin
43            case (opcode)
44                3'b001: begin //XNOR
45                    operation on next bit
46                    C[1] <= ~(A[1] ^ B[1]);
47                    zero <= zero & (C[1] ==
48                        1'b0);
49                end
50                3'b010: begin //SUB
51                    operation on next bit
52                    {carr, C[1]} <= B[1] - A
53                        [1] - carr;
54                    zero <= zero & (C[1] ==
55                        1'b0);
56                end
57                3'b011: begin //NAND
58                    operation on next bit
59                    C[1] <= ~(A[1] & B[1]);
60                    zero <= zero & (C[1] ==
61                        1'b0);
62                end
63                3'b100: begin //ADD
64                    operation on next bit
65                    {carr, C[1]} <= A[1] + B
66                        [1] + carr;
67                    zero <= zero & (C[1] ==
68                        1'b0);
69                end
70            endcase
71            state <= 2'b10;
72        end
73    end
74end

```

```

55
56
57     endcase
58     state <= 2'b10;
59 end
60 2'b10: begin
61     case (opcode)
62         3'b001: begin //XNOR
63             operation on next bit
64             C[2] <= ~(A[2] ^ B[2]);
65             zero <= zero & (C[2] ==
66                 1'b0);
67         end
68         3'b010: begin //SUB
69             operation on next bit
70             {carr, C[2]} <= B[2] - A
71                 [2] - carr;
72             zero <= zero & (C[2] ==
73                 1'b0);
74         end
75         3'b011: begin //NAND
76             operation on next bit
77             C[2] <= ~(A[2] & B[2]);
78             zero <= zero & (C[2] ==
79                 1'b0);
80         end
81         3'b100: begin //ADD
82             operation on next bit
83             {carr, C[2]} <= A[2] + B
84                 [2] + carr;
85             zero <= zero & (C[2] ==
86                 1'b0);
87         end
88     endcase
89     state <= 2'b11;
90 end
91 2'b11: begin
92     case (opcode)
93         3'b001: begin //XNOR
94             operation on MSBs
95             C[3] <= ~(A[3] ^ B[3]);
96             sign <= C[3];
97             zero <= C == 4'b0000;
98         end
99         3'b010: begin //SUB
100            operation on MSBs
101            {carr, C[3]} <= B[3] - A[3]
102                - carr;
103            sign <= C[3];
104            zero <= C == 4'b0000;
105            if (B[3] < A[3]) begin //
106                if result is negative,
107                take two's complement
108                of result
109                C <= ~C + 4'b0001;
110                sign <= C[3];
111            end
112        end
113        3'b011: begin //NAND
114            operation on MSBs
115            C[3] <=
116                ~(A
117                    [3] &
118                    B
119                    [3]);
120            sign <= C[3];
121        end
122    endcase
123end

```

```

101         zero <= C == 4'b0000;
102     end
103     3'b100: begin //ADD
104         operation on MSBs
105         {carr, C[3]} <= A[3] + B
106             [3] + carr;
107         sign <= C[3];
108         zero <= C == 4'b0000;
109     end
110 endcase
111 state <= 2'b00;
112 end
113 endcase
114 end
115 endmodule

```

Listing 1. Verilog code for 4-bit ALU