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 *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:

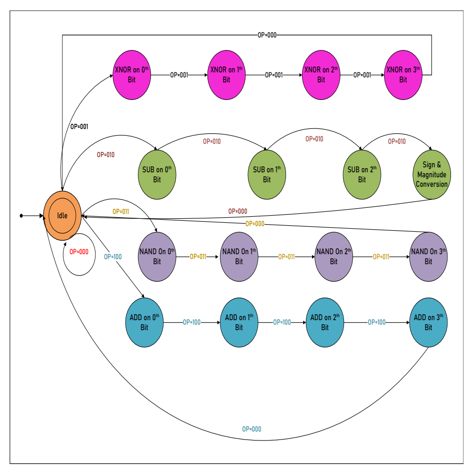

Fig. 1. FSM Diagram

- 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.

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* where the result is stored C sequentially in each clock cycles.

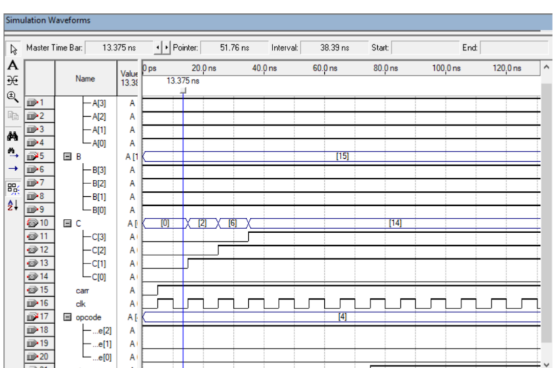

Fig. 2. Timing Daigram for ADD Operation

B. SUB Operation

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

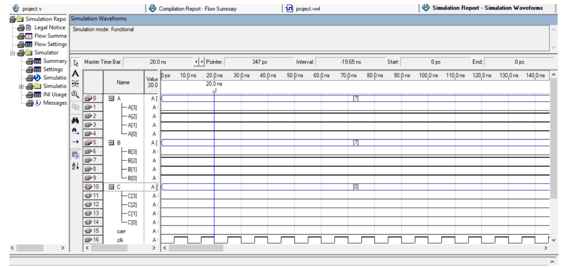

Fig. 3. Timing Diagram for Sub Operation

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

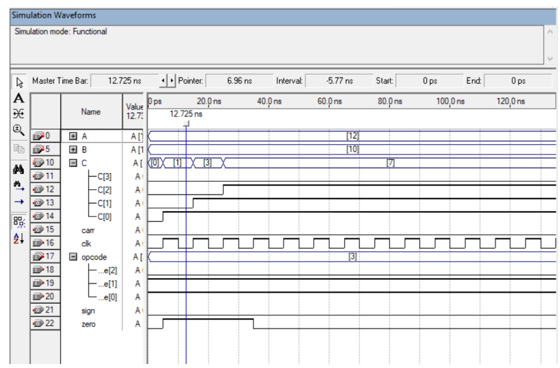

Fig. 4. Timing Daigram for NAND Operation

D. XNOR Operation

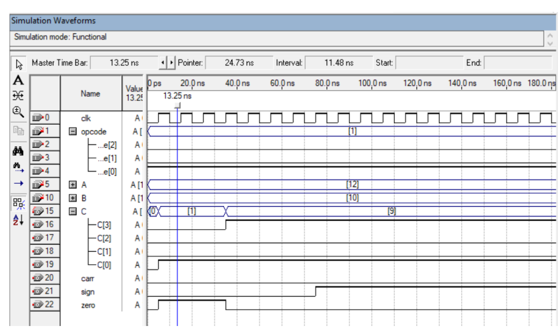

Fig. 5. Timing Daigram for XNOR Operation

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

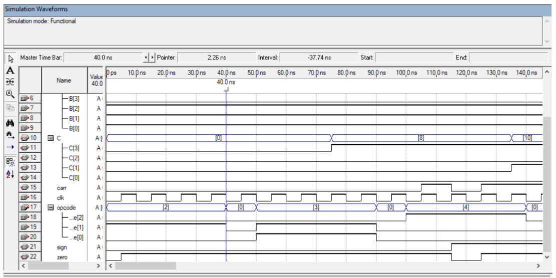

Fig. 6. Timing Daigram with reset

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

```
module project(input clk, input [3:0] A, input [3:0] B,
  input [2:0] opcode, output reg [3:0] C,
   output reg carr, output reg sign, output reg zero);
// Will be using state to indicate state of the machine.
reg [1:0] state = 0;
always @ (posedge clk) begin
    case (state)
        2'b00: begin
            case (opcode)
                3'b000: begin
                    C <= 4'b0000; //RESET operation
                    carr <= 1'b0;
                    sign <= 1'b0;
                    zero <= 1'b1;
                end
                3'b001: begin //XNOR operation on LSBs
                    C[0] <= ~(A[0] ^ B[0]);
                    zero <= C[0] == 1'b0;
                end
                3'b010: begin //SUB operation on LSBs
                    {carr, C[0]} <= B[0] - A[0];
                    zero <= C[0] == 1'b0;
               end
                3'b011: begin //NAND operation on LSBs
                    C[0] <= ~(A[0] & B[0]);
                    zero <= C[0] == 1'b0;
                end
                3'b100: begin //ADD operation on LSBs
                    {carr, C[0]} <= A[0] + B[0];
                    zero <= C[0] == 1'b0;
                end


            endcase
            state <= 2'b01;
        end
        2'b01: begin
            case (opcode)
                3'b001: begin //XNOR operation on next bit
                    C[1] <= ~(A[1] ^ B[1]);
                    zero <= zero & (C[1] == 1'b0);
                end
                3'b010: begin //SUB operation on next bit
                    {carr, C[1]} <= B[1] - A[1] - carr;
                    zero <= zero & (C[1] == 1'b0);
                end
                3'b011: begin //NAND operation on next bit
                    C[1] <= ~(A[1] & B[1]);
                    zero <= zero & (C[1] == 1'b0);
                end
                3'b100: begin //ADD operation on next bit
                    {carr, C[1]} <= A[1] + B[1] + carr;
                    zero <= zero & (C[1] == 1'b0);
                end


            endcase
            state <= 2'b10;
        end
        2'b10: begin
            case (opcode)
                3'b001: begin //XNOR operation on next bit
                    C[2] <= ~(A[2] ^ B[2]);
                    zero <= zero & (C[2] == 1'b0);
                end
                3'b010: begin //SUB operation on next bit
                    {carr, C[2]} <= B[2] - A[2] - (carr & ~zero);
                    zero <= zero & (C[2] == 1'b0);
```

```
                end
                3'b011: begin //NAND
                    operation on next bit
                    C[2] <= ~(A[2] & B[2]);
                    zero <= zero & (C[2] ==
                        1'b0);
                end
                3'b100: begin //ADD
                    operation on next bit
                    {carr, C[2]} <= A[2] + B
                        [2] + carr;
                    zero <= zero & (C[2] ==
                        1'b0);
                end

            endcase
            state <= 2'b11;
        end
        2'b11: begin
            case (opcode)
                3'b001: begin //XNOR
                    operation on MSBs
                    C[3] <= ~(A[3] ^ B[3]);
                    sign <= C[3];
                    zero <= C == 4'b0000;
                end
                3'b010: begin //SUB
                    operation on MSBs
                 {carr, C[3]} <= B[3] - A[3]
                      - carr;
                 sign <= C[3];
                 zero <= C == 4'b0000;
                   if (B[3] < A[3]) begin //
                       if result is negative,
                        take two's complement
                        of result
                       C <= ~C + 4'b0001;
                       sign <= C[3];
                   end
                end
                3'b011: begin //NAND
                    operation on MSBs
                                    C[3] <=
                                        ~(A
                                        [3] &
                                         B
                                        [3]);
                    sign <= C[3];
                    zero <= C == 4'b0000;
                end
                3'b100: begin //ADD
                    operation on MSBs
                    {carr, C[3]} <= A[3] + B
                        [3] + carr;
                    sign <= C[3];
                    zero <= C == 4'b0000;
                end

            endcase
            state <= 2'b00;
        end
    endcase
end
endmodule
```

Listing 1. Verilog code for 4-bit ALU