Design a 4-bit ALU capable of performing 4 different arithmetic or logical operations using Quartus, implement it using Verilog HDL, and verify it using a timing diagram.

CSE460
VLSI Design Laboratory

Brac University

1st Khairun Nisa
dept. Computer Science Engineering
BRAC University
19101376

2nd Zaima Labiba
dept. Computer Science Engineering
BRAC University
19101284

3rd Hasibul Sakib
dept. Computer Science Engineering
BRAC University
19101283

4th Ishraq Ahmed Esha
dept. Computer Science Engineering
BRAC University
19301261

5th Shamsil Arafin Ullah
dept. Computer Science Engineering
BRAC University
19101164

***Abstract*—The components are referred to as reversible if the input and output values are perfectly mapped one to one. Reversible gates' future is promising in low power VLSI. Their implementation using quantum gates enhances their potential for use in next-generation VLSI applications.**

We have developed a reversible ALU that meets our requirements for hardware complexity, garbage outputs, constant inputs, and the number of reversible gates. The 4-bit ALU design performs three arithmetic and four logical operations. ADD,SUBTRACT, and COMPARE are the four arithmetic operations. AND, OR, XOR, and NOT are the four logical operations with one RESET. It requires a total of five opcodes. Opcode must be consistent.

***Index Terms*—boolean, waveform, quartex, VLSI**

I. Introduction

Arithmetic Logic Unit is a basic unit in many combinational circuits with a number of storage registers connected to it. To perform a micro operation, the contents of specified registers are placed in the inputs of the common ALU. The ALU performs an operation and the result of the operation is then transferred to a destination register. The ALU is a combinational circuit so that the entire registers transfer operation from the source register through the ALU and the destination register can be performed during one clock pulse period. Learning ALU design can help you to design complex circuits. The Carry look ahead adder uses an input logic to perform all arithmetic operations.

II. Operation

The state diagram of the system is shown in the figure above. Here 5 operations are done. SUB,ADD,RESET,XNOR and NAND. For each operation we use 010,100,000,001 and 011 circumstantly.

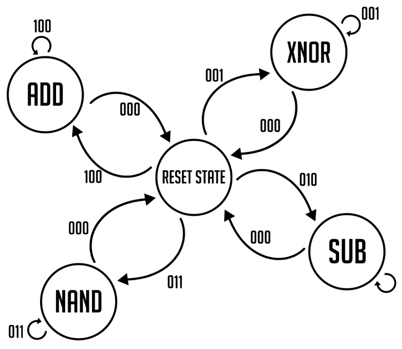

Fig. 1. State Diagram

Identify applicable funding agency here. If none, delete this.

III. Timing Diagram

The timing diagrams are shown from the output of waveform.

A. RESET

Here we set a reset operation with the opcode 000 which is an idle component of our operation . The output C will retain the result of the last operation performed.

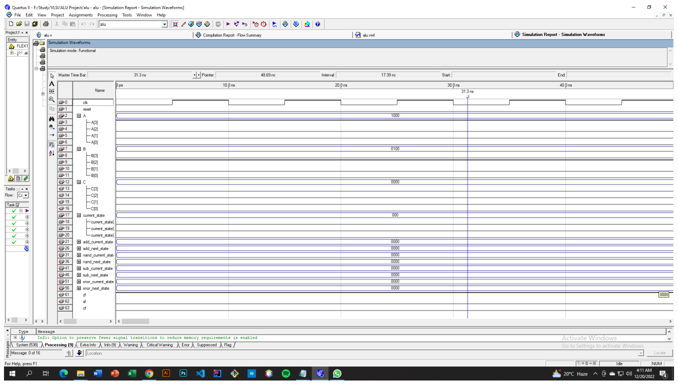

Fig. 2. Timing diagram of RESET

B. SUB

Here we substract our inputs. Firstly we reset the whole state into 000 then we use the opcode 010 for the substraction. It produces bitwise sub. The alu will substract B to A. The operation will be performed serially at the positive edge of input clock during the first clock cycle, the operation will be performed on the LSBs (A0, B0) storing the result in C0 and in the next clock cycle the operation will be performed on A1 , B1 updating C1 and finally, on A3 and B3 saving it to C3 as long as the opcode is active.

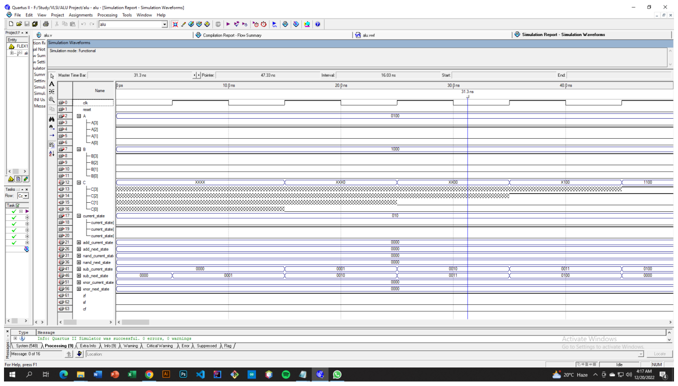

Fig. 3. Timing diagram of SUB

C. XNOR

When the opcode is 001 in XNOR operation. Before starting the XNOR operation we need to set the opcode into reset state .After that we can work in XNOR operation . Here the clock cycle is moving towards to MSB as the figure 4 shown. The ALU will perform bitwise XNOR operation on A, B. The bitwise operation will be performed serially at the positive edge of input clock during the first clock cycle, the operation will be performed on the LSBs (A0,B0) storing the result in C0 and in the next clock cycle the operation will be performed on A1, B1 updating C1 and finally, on A3 and B3 saving it to C3 as long as the opcode is active.

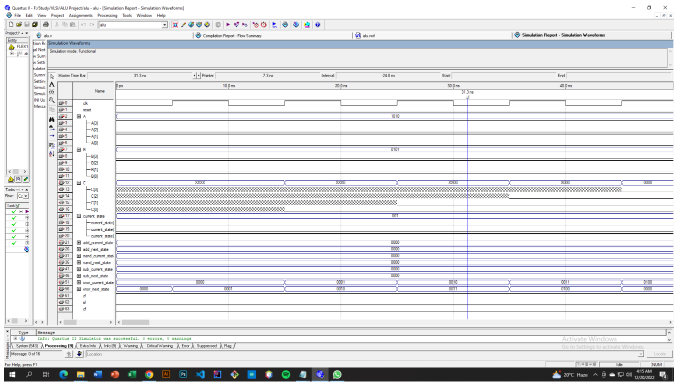

Fig. 4. Timing diagram of XNOR

D. NAND

Here again we set the reset operation with the opcode 000 first. Thenceforth, we start the operation of NAND with the opcode 011.The ALU will perform bitwise NAND operation on A, B. The itwise operation will be performed serially at the positive edge of input clock i.e. during the first clock cycle, the operation will be performed on the LSBs (A0, B0) storing the result in C0 and in the next clock cycle the operation will be performed on A1, B1 updating C1 and finally, on A3 and B3 saving it to C3 as long as the opcode is active.

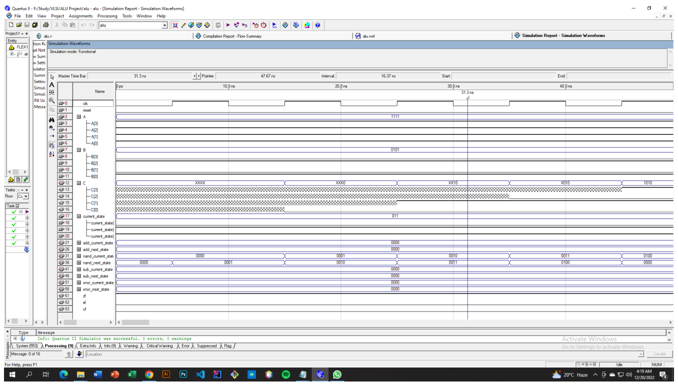

Fig. 5. Timing diagram of NAND

E. ADD

The fifth operation is ADD. The ALU will perform ADD operation on A, B. The operation will be performed serially at the positive edge of input clock during the first clock cycle, the operation will be performed on the LSBs (A0, B0) storing the result in C0 and in the next clock cycle the peration will be performed on A1 , B1 updating C1 and finally, on A3 and B3 saving it to C3 as long as the opcode is active.

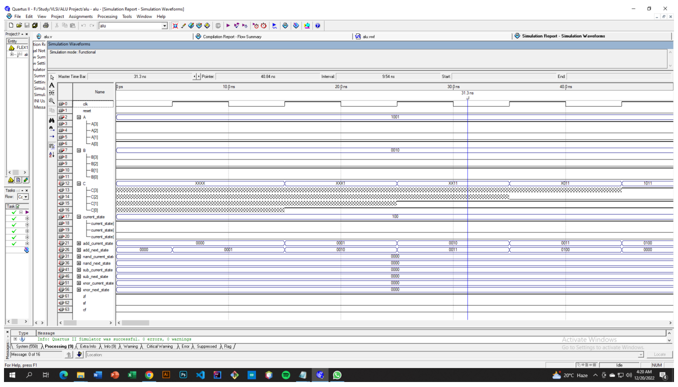

Fig. 6. Timing diagram of ADD

IV. Appendix

module alu (clk,reset,current$_{s}tate, A, B, C, zf, sf, cf, nand_{c}urrent_{s}tate, xnor_{c}urrent_{s}tate, sub_{c}urrent_{s}tate, add_{c}urrent_{s}tate, nand$
$0]current_{s}tate; //assignforopcodeinput[3 : 0]A, B; //assignforinputsoutputreg[3 : 0]C; //assignforoutputoutputregzf, sf, cf;$

output reg [3:0]nand$_{c}urrent_{s}tate, xnor_{c}urrent_{s}tate, sub_{c}urrent_{s}tate, add_{c}urrent_{s}tate, outputreg[3 : 0]nand_{n}ext_{s}tate, xnor_{n}ext_{s}tate, sub_{n}ext_{s}tate, add_{n}ext_{s}tate;$

parameter [3:0]reset$_{s}tate = 3'b000, xnor_{s}tate = 3'b001, sub_{s}tate = 3'b010, nand_{s}tate = 3'b011, add_{s}tate = 3'b100; //mainstatesparameter[3 : 0]sub2_{s}0 = 3'b000, sub2_{s}1 = 3'b001, sub2_{s}2 = 3'b010, sub2_{s}3 = 3'b011, sub2_{s}4 = 3'b100; //sub - substates4bitparameter[3 : 0]xnor2_{s}0 = 3'b000, xnor2_{s}1 = 3'b001, xnor2_{s}2 = 3'b010, xnor2_{s}3 = 3'b011, xnor2_{s}4 = 3'b100; //sub - xnorstates4bitparameter[3 : 0]nand2_{s}0 = 3'b000, nand2_{s}1 = 3'b001, nand2_{s}2 = 3'b010, nand2_{s}3 = 3'b011, nand2_{s}4 = 3'b100; //sub - nandstates4bitparameter[3 : 0]add2_{s}0 = 3'b000, add2_{s}1 = 3'b001, add2_{s}2 = 3'b010, add2_{s}3 = 3'b011, add2_{s}4 = 3'b100; //subaddstates4bit$

// State inside state transition logic (Bitwise AND operartion) always @(posedge clk) begin

if (current$_{s}tate == nand_{s}tate)$

begin nand$_{c}urrent_{s}tate = nand_{n}ext_{s}tate; case(nand_{c}urrent_{s}tate)nand2_{s}0 : nand_{n}ext_{s}tate = nand2_{s}1; nand2_{s}1 : nand_{n}ext_{s}tate = nand2_{s}2; nand2_{s}2 : nand_{n}ext_{s}tate = nand2_{s}3; nand2_{s}3 : nand_{n}ext_{s}tate = nand2_{s}4; nand2_{s}4 : nand_{n}ext_{s}tate = nand2_{s}0; endcaseend$

else if (current$_{s}tate == xnor_{s}tate)$

begin xnor$_{c}urrent_{s}tate = xnor_{n}ext_{s}tate; case(xnor_{c}urrent_{s}tate)xnor2_{s}0 : xnor_{n}ext_{s}tate = xnor2_{s}1; xnor2_{s}1 : xnor_{n}ext_{s}tate = xnor2_{s}2; xnor2_{s}2 : xnor_{n}ext_{s}tate = xnor2_{s}3; xnor2_{s}3 : xnor_{n}ext_{s}tate = xnor2_{s}4; xnor2_{s}4 : xnor_{n}ext_{s}tate = xnor2_{s}0;$

endcase end

else if (current$_{s}tate == sub_{s}tate)$

begin sub$_{c}urrent_{s}tate = sub_{n}ext_{s}tate; case(sub_{c}urrent_{s}tate)sub2_{s}0 : sub_{n}ext_{s}tate = sub2_{s}1; sub2_{s}1 : sub_{n}ext_{s}tate = sub2_{s}2; sub2_{s}2 : sub_{n}ext_{s}tate = sub2_{s}3; sub2_{s}3 : sub_{n}ext_{s}tate = sub2_{s}4; sub2_{s}4 : sub_{n}ext_{s}tate = sub2_{s}0;$

endcase end

else if (current$_{s}tate == add_{s}tate)$

begin add$_{c}urrent_{s}tate = add_{n}ext_{s}tate; case(add_{c}urrent_{s}tate)add2_{s}0 : add_{n}ext_{s}tate = add2_{s}1; add2_{s}1 : add_{n}ext_{s}tate = add2_{s}2; add2_{s}2 : add_{n}ext_{s}tate = add2_{s}3; add2_{s}3 : add_{n}ext_{s}tate = add2_{s}4; add2_{s}4 : add_{n}ext_{s}tate = add2_{s}0;$

endcase end

end

//Output logic based on states (Bitwise AND operartion) always @(current$_{s}tate)begin$

if (current$_{s}tate == reset_{s}tate)beginC = 0; zf = 1; endelseif(current_{s}tate == nand_{s}tate)begincase(nand_{c}urrent_{s}tate)$

nand$2_{s}1 : C[0] = (A[0]B[0]); nand2_{s}2 : C[1] = (A[1]B[1]); nand2_{s}3 : C[2] = (A[2]B[2]); nand2_{s}4 : beginC[3] = (A[3]B[3]); //if(C[0] == 0C[1] == 0C[2] == 0C[3] == 0)//begin//zf = 1; //endendendcaseend//if(C[3] == 1)//begin//sf = 1; //end$

else if(current$_{s}tate == xnor_{s}tate)begincase(xnor_{c}urrent_{s}tate)$

xnor$2_{s}1 : C[0] = (A[0]\ ^{B}[0]); xnor2_{s}2 : C[1] = A[1]\ ^{B}[1]; xnor2_{s}3 : C[2] = A[2]\ ^{B}[2]; xnor2_{s}4 : C[3] = A[3]\ ^{B}[3]; endcaseif(C[0] == 0C[1] == 0C[2] == 0C[3] == 0)beginzf = 1; endend//if(C[3] == 1)//begin//sf = 1; //end$

else if(current$_{s}tate == sub_{s}tate)begincase(sub_{c}urrent_{s}tate)$

sub$2_{s}1 : C[0] = (A[0] - B[0]); sub2_{s}2 : C[1] = A[1] - B[1]; sub2_{s}3 : C[2] = A[2] - B[2]; sub2_{s}4 : C[3] = A[3] - B[3]; endcaseif(C[0] == 0C[1] == 0C[2] == 0C[3] == 0)beginzf = 1; endend//if(C[3] == 1)//begin//sf = 1; //end$

else if(current$_{s}tate == add_{s}tate)begincase(add_{c}urrent_{s}tate)$

add$2_{s}1 : C[0] = (A[0] + B[0]); add2_{s}2 : C[1] = A[1] + B[1]; add2_{s}3 : C[2] = A[2] + B[2]; add2_{s}4 : C[3] = A[3] + B[3]; endcaseif(C[0] == 0C[1] == 0C[2] == 0C[3] == 0)beginzf = 1; endend//if(C[3] == 1)//begin//sf = 1; //endendendmodule$

V. Conclusion

To summarize this project, we can easily implement two 4-bit numbers logically and arithmetically. Five operations are performed by five opcodes: 000, 100, 001, 010, and 011 for RESET based on ADD, XNOR, SUB, NAND. So, based on the timing diagram and state diagram, we can conclude that five operations support Arithmetic Logic Unit

References

[1] Ravi Kant. and Manpreet. Kaur. (2014, May). Design of 4 bit ALU using Reversible Gates.

[2] Robin Mitchell. (2016, August). How to Build Your Own Discrete 4-Bit ALU.

[3] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.