Four Bit Arithmetic And Logical Unit

*A 4-bit ALU capable of performing 4 different arithmetic or logical operations using Quartus, implemented using Verilog HDL and verified by timing diagram.

1st Shah Imran
ID:18301149
Group-5
Section-8
shah.imran@g.bracu.ac.bd

2nd Istiak Ahmed Alin
ID-19201087
Group-5
Section-8
istiak.ahmed.alin@g.bracu.ac.bd

3rdSherajus Salehin
ID-19101311
Group-5
Section-8
sherajus.salehin@g.bracu.ac.bd

4th Moonami Sharmita Azad
ID-16201039
Group-5
section-8
moonami.sharmita.azad@g.bracu.ac.bd

5th Lamia Rahman
ID-14101225
Group-5
Section-8
lamia.rahman@g.bracu.ac.bd

6th Md.Muhtasim Hossain
ID-19101263
Group-5
Section-8
md.muhtasim.hossain@g.bracu.ac.bd

***Abstract*—The following 4-bit ALU (Arithmetic and Logical Unit) takes two 4 bit inputs and results in a 4-bit output with 3-bit flag output. Primarily, the first four pins take the first number in 4-bit binary form. Similarly, the second input pin takes the second number. Based on the opcode (provided by the next 3-bit inputs), output pins show the result of the selected operation. There are a total of 5 operable opcodes that perform reset, X-OR, ADD, XNOR & SUB. There are also 3 flags ZF, SF and CF that indicates whether the output is zero(ZF=1) or not, the sign is positive(SF=1) and whether the output has carry (CF=1) or not. The whole operation works at the positive edge of a given clock at pin 9.**

The machine has multiple states for operating a single while the opcode is active. Apart from the idle state, Most operations require two sub-states (current and next) to fully perform the operation. Each sub-state is then divided into four parts to perform a bitwise operation.

***Index Terms*—4-bit, ALU, XOR, XNOR, SUM, SUB**

I. Introduction

Using the Verilog compiler we have built an ALU which can calculate 5 different operations and we could get time diagrams for different operations and the circuit through the RTL viewer.

A. ALU Part

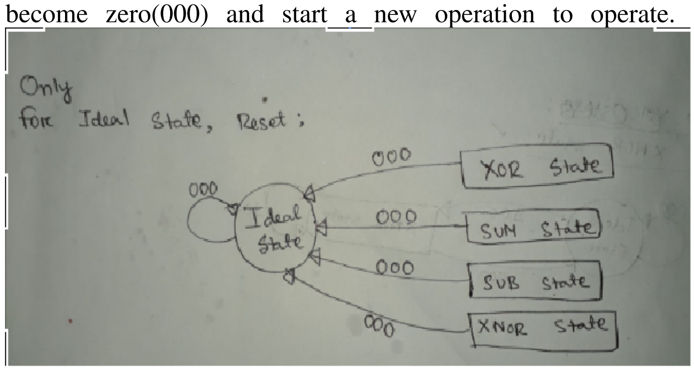
An ALU (Arithmetic-logic unit) is a part of a CPU(Central Processing Unit) which carries out arithmetic and logic operations on the operands in computer instruction. In our project we are building an ALU which solves XOR,ADD,XNOR,SUB and Reset the problem when the operation is done.

Input: 2 inputs were taken which are A and B. They are 4-bit each.
Output : 1 output was taken which is C and it is 4-bit.

II. Operation details:

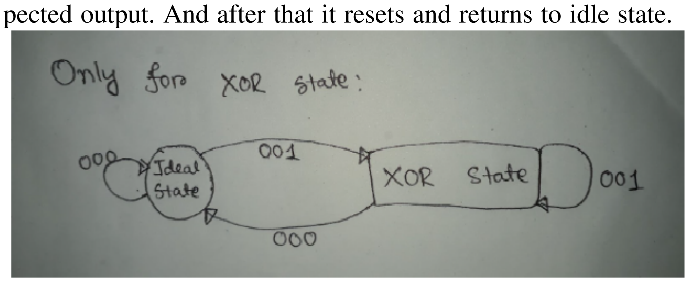
A. Reset (000)

According to our project, If an operation is successfully done and the output was given, the Reset resistor will become zero(000) and start a new operation to operate.



B. XOR(001)

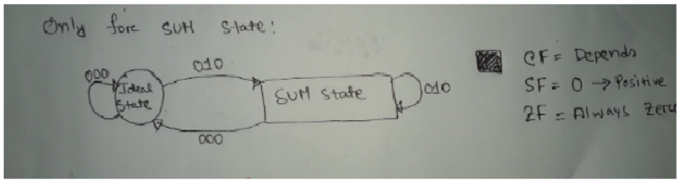
By performing this operation in ALU, through input sending 1-bit each time it performs XOR operation and provides expected output. And after that it resets and returns to idle state.



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

C. ADD(010)

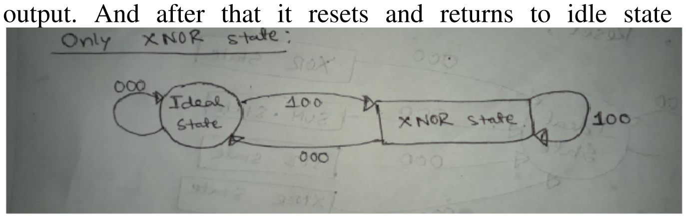
According to this operation in ALU, through input sending 1-bit each time it if MSB becomes overflow 1 that means Carry Bit turns into 1 and the output will be provided accordingly.



D. XNOR(011)

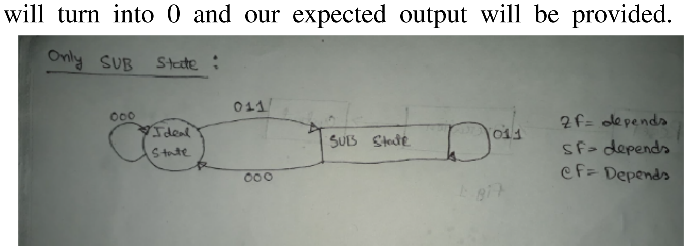
By performing this operation in ALU,through input sending 1-bit each time it

performs XNOR operation and provides expected output. And after that it resets and returns to idle state



E. SUB(100)

According to this operation in ALU, sending 1-bit each time Alu will perform SUB operation and if output,C=000 then zero flag will turn into 1. If the sign flag turns into 1 then it is a negative number and if it is positive the sign flag will turn into 0 and our expected output will be provided.



F. FLAG details

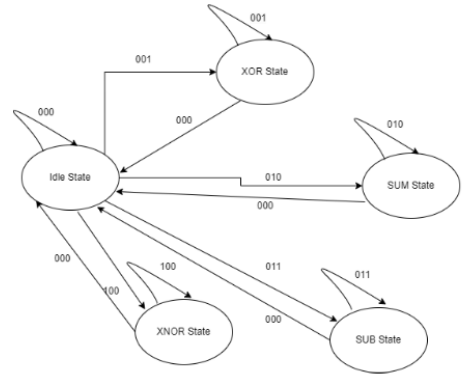
These flags are 2-bit each.

1) Zero Flag (ZF): The zero flag is 1 when the output is 0.

2) Sign Flag (SF): The sign flag is 1 when the output is 1.

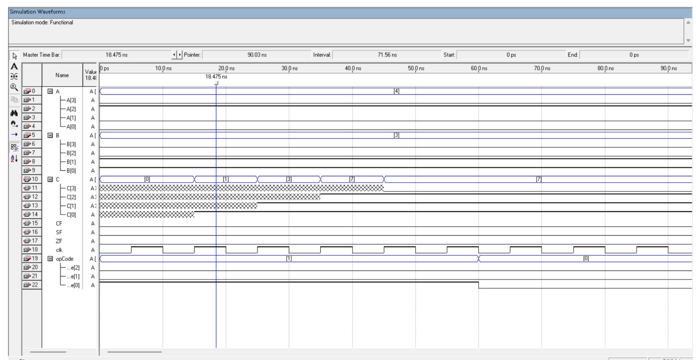
3) Carry Flag(CF): The carry flag is 1 then the output carry is 1

III. STATE DIAGRAM

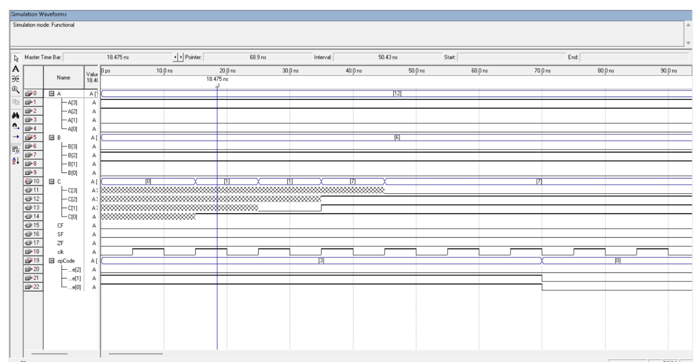


IV. TIMING DIAGRAM

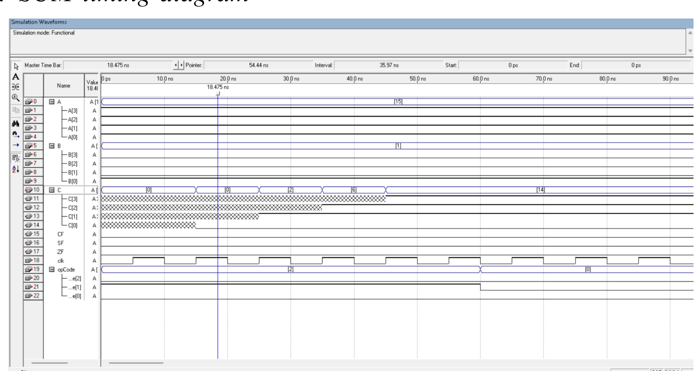
A. XOR timing diagram



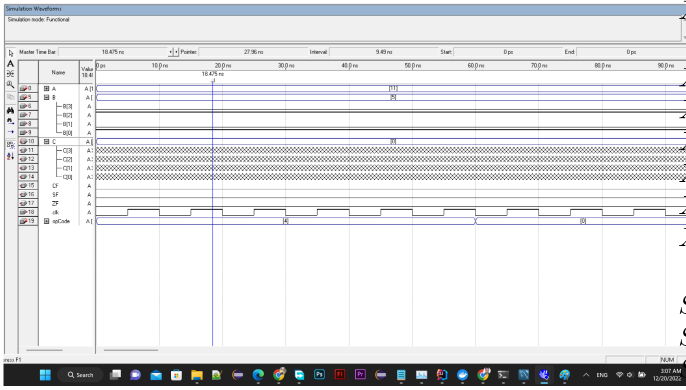
B. XNOR timing diagram



C. SUM timing diagram



D. SUB timing diagram



V. Conclusion

By solving this problem using the Verilog compiler we have built an ALU and simulated a code which could compile 5 different by taking 2 4-bit input and it also can 3 different flags which can show sign, carry and the output is zero or not. The project is like a calculator which could operate according to the user and shows a timing diagram to explain the process.To explain furthermore , RTL inverter shows how this circuit was built and how it instructs and its process.

VI. Appendix

module Final$_ALU(clk, opCode, A, B, C, CF, ZF, SF)$;

input clk; input [2:0] opCode; input [3:0] A; input [3:0] B;

//output wire [3:0] R; output reg [3:0] C; output reg CF, SF, ZF;

reg cin, cout;

reg [2:0] current$_state, /* and_current_state, and_next_state, */XOR_current_state, XOR_next_state, XNOR_current_state, XNOR_next_state, SUM_current_state, S$

parameter idle$_state = 3'b000, XOR_state = 3'b001, SUM_state = 3'b010, XNOR_state = 3'b011, SUB_state = 3'b100, and_state = 3'b101; //parameterand2_S0 = 3'b000, and2_S1 = 3'b001, and2_S2 = 3'b010, and2_S3 = 3'b011, and2_S4 = 3'b100; parameterXOR_S0 = 3'b000, XOR_S1 = 3'b001, XOR_S2 = 3'b010, XOR_S3 = 3'b011, XOR_S4 = 3'b100; parameterXNOR_S0 = 3'b000, XNOR_S1 = 3'b001, XNOR_S2 = 3'b010, XNOR_S3 = 3'b011, XNOR_S4 = 3'b100; parameterSUM_S0 = 3'b000, SUM_S1 = 3'b001, SUM_S2 = 3'b010, SUM_S3 = 3'b011, SUM_S4 = 3'b100; parameterSUB_S0 = 3'b000, SUB_S1 = 3'b001, SUB_S2 = 3'b010, SUB_S3 = 3'b011, SUB_S4 = 3'b100;$

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

if (opCode == idle$_{state})$

begin /* To do idle state*/ end

if (opCode == XOR$_{state})$

begin XOR$_current_state = XOR_next_state; case(XOR_current_state)XOR_S0 : XOR_next_state = XOR_S1; XOR_S1 : XOR_next_state = XOR_S2; XOR_S2 : XOR_next_state = XOR_S3; XOR_S3 : XOR_next_state = XOR_S4; XOR_S4 : XOR_next_state = XOR_S0; endcaseend$

if (opCode == XNOR$_state)beginXNOR_current_state = XNOR_next_state; case(XNOR_current_state)XNOR_S0 : XNOR_next_state = XNOR_S1; XNOR_S1 : XNOR_next_state = XNOR_S2; XNOR_S2 : XNOR_next_state = XNOR_S3; XNOR_S3 : XNOR_next_state = XNOR_S4; XNOR_S4 : XNOR_next_state = XNOR_S0; endcaseend$

if (opCode == SUM$_state)beginSUM_current_state = SUM_next_state; case(SUM_current_state)SUM_S0 : SUM_next_state = SUM_S1; SUM_S1 : SUM_next_state = SUM_S2; SUM_S2 : SUM_next_state = SUM_S3; SUM_S3 : SUM_next_state = SUM_S4; SUM_S4 : SUM_next_state = SUM_S0; endcaseend$

if (opCode == SUB$_state)beginSUB_current_state = SUB_next_state; case(SUB_current_state)SUB_S0 : SUB_next_state = SUB_S1; SUB_S1 : SUB_next_state = SUB_S2; SUB_S2 : SUB_next_state = SUB_S3; SUB_S3 : SUB_next_state = SUB_S4; SUB_S4 : SUB_next_state = SUB_S0; endcaseend$

/*if (opCode == and$_state)beginand_current_state = and_next_state; case(and_current_state)and2_S0 : and_next_state = and2_S1; and2_S1 : and_next_state = and2_S2; and2_S2 : and_next_state = and2_S3; and2_S3 : and_next_state = and2_S4; and2_S4 : and_next_state = and2_S0; endcaseend */$

end

//Output logic based on states (Bitwise AND operartion) always @(XOR$_current_state|XNOR_current_state|SUM_current_state)beginif($

XOR$_state)case(XOR_current_state)//XOR_S0 : C = 0; XOR_S1 : C[0] = A[0]^B[0]; XOR_S2 : C[1] = A[1]^B[1]; XOR_S3 : C[2] = A[2]^B[2]; XOR_S4 : C[3] = A[3]^B[3]; endcase$

if(opCode == XNOR$_state)case(XNOR_current_state)//XOR_S0 : C = 0; XNOR_S1 : C[0] = (A[0]^B[0]); XNOR_S2 : C[1] = (A[1]^B[1]); XNOR_S3 : C[2] = (A[2]^B[2]); XNOR_S4 : C[3] = (A[3]^B[3]); endcase$

if(opCode == SUM$_state)cin = 0; case(SUM_current_state)SUM_S1 : if(cin == 0)beginC[0] = A[0]^B[0]^cin; cout = ((A[0]B[0])|(cin * (A[0]B[0]))); cin = cout; end$

SUM$_S2 : if(cin == 0)beginC[1] = A[1]^B[1]^cin; cout = ((A[1]B[1])|(cin * (A[1]B[1]))); cin = cout; endelsebeginC[1] = A[1]^B[1]^cin; cout = ((A[1]B[1])|(cin * (A[1]B[1]))); cin = cout; endSUM_S3 : if(cin == 0)beginC[2] = A[2]^B[2]^cin; cout = ((A[2]B[2])|(cin * (A[2]B[2]))); cin = cout; endelsebeginC[2] = A[2]^B[2]^cin; cout = ((A[2]B[2])|(cin * (A[2]B[2]))); cin = cout; endSUM_S4 : if(cin == 0)beginC[3] = A[3]^B[3]^cin; cout = ((A[3]B[3])|(cin * (A[3]B[3]))); cin = cout; endelsebeginC[3] = A[3]^B[3]^cin; cout = ((A[3]B[3])|(cin * (A[3]B[3]))); cin = cout; endendcase$

//if(opCode == SUB$_state)//case(SUB_current_state)/ *$
$TodoSubstate * ///endcase$

/*if(opCode == and$_state)case(and_current_state)//and2_S0 :$
$C = 0; and2_S1 : C[0] = A[0]B[0]; and2_S2 : C[1] =$
$A[1]B[1]; and2_S3 : C[2] = A[2]B[2]; and2_S4 : C[3] =$
$A[3]B[3]; endcase * /$

end

endmodule