4-bit ALU Arithmetic Operations

Arjita Saha
dept. of CSE
Brac University
Dhaka, Bangladesh
arjita.saha@g.bracu.ac.bd

Dipak Debnath Arka
dept. of CSE
Brac University
Dhaka, Bangladesh
dipak.debnath.arka@g.bracu.ac.bd

Rawnak Muntaha Anan
dept. of CSE
Brac University
Dhaka, Bangladesh
rawnak.muntaha.anan@g.bracu.ac.bd

MD. Shafiul Alam
dept. of CSE
Brac University
Dhaka, Bangladesh
md.shafiul.alam@g.bracu.ac.bd

Foyzunnesa Swarna
dept. of CSE
Brac University
Dhaka, Bangladesh
foyzunnesa.swarna@g.bracu.ac.bd

***Abstract*—The group6 project of CSE460 implementation of a ALU that takes two, four-bit inputs: A, B, and a three-bit operation code (opcode) also produces a four-bit output, C. Based on the opcode,it performs five different operations RESET,XOR,ADD,AND and SUB. The operations are performed based on the inserted 3 bit opcode-001 for XOR, 010 for ADD, 011 for AND and 100 for SUB. In addition, opcode 000 is set as a RESET operation which refers to a no operation state, retaining the last operation's output.Depending on the result of a particular operation, the ALU also produces three flags: carry, zero and sign flag.Zero Flag is set to 1 when the output C is 0.Sign Flag is set to 1 when the MSB of the output C is 1.Carry Flag is set to 1 when output carry/borrow is 1.The ALU has been implemented in the Quartus software using Verilog HDL and is verified by analyzing the timing diagram for the current project.**

***Index Terms*—ALU, MSB, LSB, bitwise operation, opcode.**

I. Introduction

The Arithmetic Logical Unit, or ALU, is a digital circuit that performs arithmetic and logic operations. It is often compared as a computer's mathematical brain. It serves as a basic building-block of a CPU. ALUs are typically found on computer processors, which are extremely powerful, complicated in design, and perform a wide range of operations. A comparatively basic 4-bit ALU has been implemented using Verilog in Quartus that performs 4 different operations- XOR, ADD, AND and SUB. Let's explore its details. To implement our project, firstly, the inputs (two 4-bit inputs A,B and a 3-bit opcode) and outputs (a 4-bit output C, carry flag, zero flag and sign flag) have been defined in the verilog code. Then, the specific 3-bit opcode values are set to perform specific tasks (000 for reset, 001 for XOR,010 for ADD, 011 for AND and 100 for SUB). Further, a 5-bit ALU Result variable has been created which includes the output of the operation. The carry flag is then set to access the 5th bit (MSB) of the ALU Result and show result 1 in case the 5th bit is 1 there (which basically means that output exceeds the 4 bit operation and has a carry). The sign flag is set to access the 4th bit (from backwards) of the ALU Result and show result 1 in case the 4th bit is 1. In addition, the zero flag outputs 1 when the operation output is 0. Lastly, the codes for bitwise logical operations (XOR, ADD, AND and SUB) have been added. This is how we have implemented our desired ALU.

II. METHOLOGY

The arithmetic logic unit (ALU), which is a computer's central processing unit, is extremely important. On the instruction words, it performs the appropriate arithmetic and logical operations. The arithmetic A unit (AU) and the logic unit (LU) are two ways that the arithmetic logic unit (ALU) might be divided in some microprocessor architectures. Engineers can create an ALU to calculate any operation. The ALU gets more expensive, occupies more CPU space, and generates more heat as the processes get more complicated. Engineers ensure that the ALU is strong enough for the CPU to be equally strong and rapid while not being overly complex to make it prohibitively expensive or have other issues.

III.

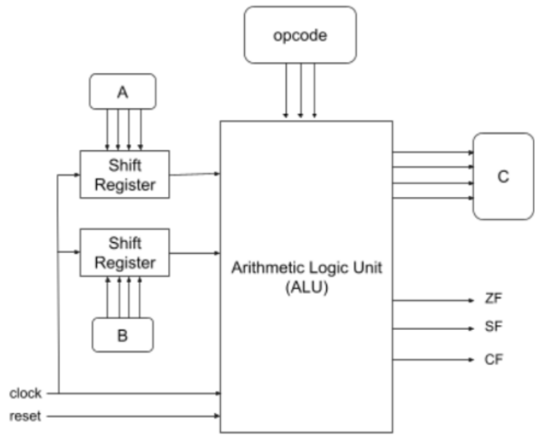
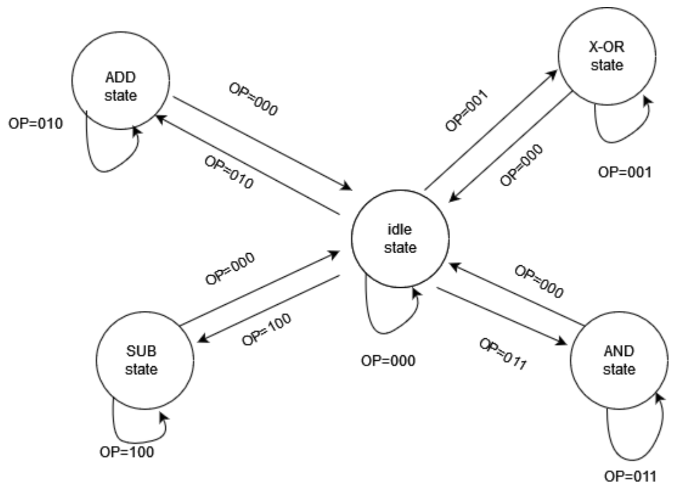


Fig. 1: Block diagram of the ALU.

The ALU takes the following inputs: two four-bit inputs (opcode), A, B, a three-bit operation code. It generates the four-bit output C by performing five different operations on A and B in accordance with the opcode. Depending on how an operation turns out, the ALU also produces three more flags: the carry, zero, and sign flags.

IV. OPERATION

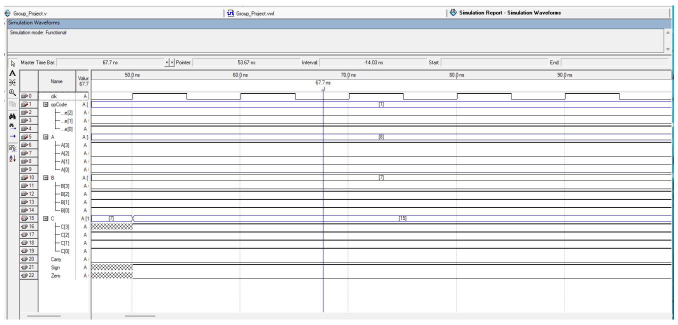
A. *State Diagram*



As the state diagram suggests, the ALU stays at the idle state by default. The next states are determined based on the different 3-bit opcode inputs given. While in idle state, if the given opcode is 001, then the next state is the XOR State where XOR operation is executed. for opcode 010, Add State is the next state where ADD operation is executed.For opcode 011, the next state is the AND State where the AND operation is performed. For opcode 100, the next state is the SUB State where the SUB operation is performed.Inside every state, the Carry flag, sign flag and zero flag are calculated after bitwise operation. If the opcode is similar to the current state's opcode, then it'll go into a self-loop. For opcode 000, a RESET operation is performed and the ALU loops backs to the idle state. Giving the opcode- 000 from any of the existing states in the ALU will result in transition to idle state. And giving opcodes assigned to XOR, ADD, AND and SUB states from any of the stages will result in transition towards these stages depending on the opcode.

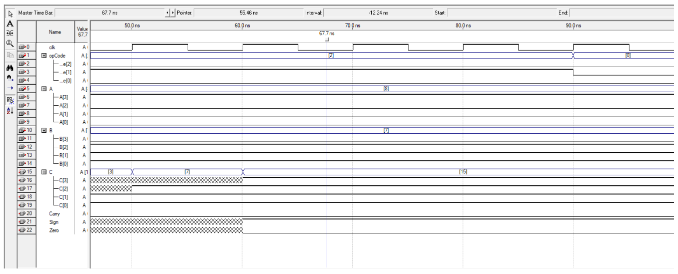
V. TIMING DIAGRAM

A. *XOR*



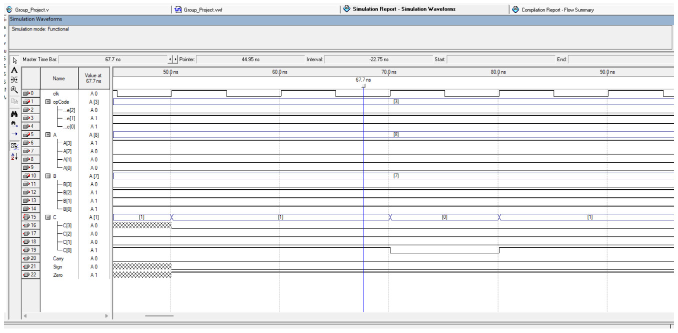
From 50ns to 90ns our opcode is 001.So the alu will perform xor during this cycle. The value of A is 1000 and B is 0111.It will take 5 clock cycles to get our result as we are doing bit wise xor operation. In our timing diagram, we can see that the first clock cycle (50-60ns), LSB bit (A[0]$\wedge B[0]) = 1, So, outputLSBbitC[0] = 1.Insecondclockcycle(60 - 70ns), (A[1] \wedge B[1]) = 1, So, C[1] = 1.Inthirdclockcycle(70 - 80ns), (A[2] \wedge B[2]) = 1, So, C[2] = 1.Inthirdclockcycle(80 - 90ns), (A[3] \wedge B[3]) = 1, So, C[3] = 1.wegetourfullresults, C = 1111, whichmatchesourtheoreticalvalue.$

B. *ADD*



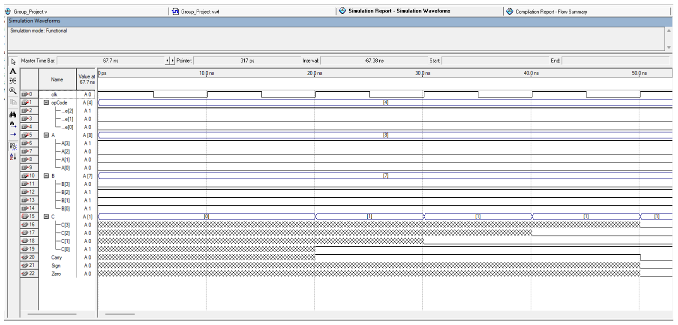
From 50 ns to 90 ns our opcode is 010. So the ALU will perform an add operation during this cycle. The value of A is 1000 and B is 0111. It will take 5 clock cycles to get our result as we are doing bitwise add operation. In our timing diagram, we can see that the first clock cycle (50-60ns), LSB bit A[0] + B[0] = 1 , So, output LSB bit C[0] = 1 and Carry=0 for this sub state. In second clock cycle (60-70ns) , A[1] + B[1] = 1 , So,C[1] =1 and Carry=0 In third clock cycle (70-80ns) , A[2] + B[2] = 1 ,So,C[2] =1 and Carry=0 In third clock cycle (80-90ns) , A[3] + B[3] =1,So,C[3] =1 and Carry=0 From 50ns we get our full results, C = 1111 which matches our theoretical value. Since, MSB is 1 after doing all bit operation, we can say the Sign flag will be 1 in 5th cycle and the Zero flag will be 0 as the result is nonzero value which matches with our timing diagram.

C. AND


Simulation Report - Simulation Waveforms
Simulation mode: Functional
clk
opCode
A
B
C
Carry
Sign
Zero

From 50ns to 90ns our opcode is 011. So the alu will perform during this cycle. The value of A is 1000 and B is 0111.It will take 5 clock cycles to get our result as we are doing bitwise and operation. In our timing diagram, we can see that the first clock cycle (50-60ns), LSB bit (A[0]&B[0]) = 0 , So, output LSB bit C[0] = 0 In second clock cycle (60-70ns) , (A[1]&B[1]) = 0 , So,C[1] =0 In third clock cycle (70-80ns) , (A[2]&B[2]) = 0 ,So,C[2] =0 In the third clock cycle (80-90ns) , (A[3]&B[3]) = 0,So,C[3] =0 .We get our full results, C = 0000 which matches our theoretical value.

D. SUB


Simulation Report - Simulation Waveforms
Simulation mode: Functional
clk
opCode
A
B
C
Carry
Sign
Zero

From 50ns to 90ns our opcode is 100.So the alu will perform subtraction during this cycle. The value of A is 1000 and B is 0111.It will take 5 clock cycles to get our result as we are doing bitwise sub operation. In our timing diagram, we can see that the first clock cycle (50-60ns), LSB bit A[0] - B[0] = 1 , So, output LSB bit C[0] = 1 and Carry=1 for this sub state. In second clock cycle (60-70ns) , A[1] - B[1]=0 , So,C[1] =0 and Carry=1 In third clock cycle (70-80ns) ,A[2] - B[2] = 0 ,So,C[2] =0 and Carry=1 In third clock cycle (80-90ns) , A[3] - B[3] = 0,So,C[3] =0 and Carry=0 we get our full results, C = 0001 which matches our theoretical value. Since, MSB is 0 after doing all bit operation, we can say the Sign flag will be 0 in 5th cycle and the Zero flag will be 0 as the result is nonzero value which matches with our timing diagram.

VI. CONCLUSION

Logical and arithmetic operations can be used for numerous tasks such as circuit build-up, hardware level programming,real life problem implementation, control-system manufacturing etc. As our project ALU performs XOR,ADD, AND and SUB operations efficiently (in reference to the timing diagrams and truth tables), it can be used for practical cases that have usage of these operations. Furthermore, the ALU can also be upgraded to perform more operations in the future. In conclusion, we have successfully implemented a 4-bit ALUthat performs XOR, ADD, AND and SUB operations which may serve many practical engineering needs.

VII. APPENDIX (VERILOG CODE)

```
    module Group_Project( A,B,opCode,clk,C,Carry,
    Sign,Zero);
    input clk;
    input [3:0] A,B;
   input [2:0] opCode;
   reg [1:0] tmp,tmp2,tmp3,tmp4;
    reg [1:0] temp,temp2,temp3,temp4;
   output reg [3:0] C;
   output reg Carry,Sign,Zero;

    parameter do_reset = 3'b000, do_x_or = 3'b001,
    do_sub=3'b100, do_and=3'b011, do_add=3'b010;

    reg[2:0] current_state,sub_state,
    sub_next_state,sub_current_state;

    parameter sub2_S0 = 3'b000, sub2_S1 = 3'b001,
    sub2_S2=3'b010, sub2_S3=3'b011,
    sub2_S4=3'b100;

    reg[2:0] and_next_state,and_current_state;
parameter and2__S0 = 3'b000,
 and2__S1 = 3'b001, and2__S2=3'b010,
 and2__S3=3'b011, and2__S4=3'b100;

reg[2:0] add_next_state,
 add_current_state;
parameter add2_S0 = 3'b000,
 add2_S1 = 3'b001, add2_S2=3'b010,
 add2_S3=3'b011, add2_S4=3'b100;

reg[2:0] x_or_next_state,
 x_or_current_state;
parameter x_or2_S0 = 3'b000,
 x_or2_S1 = 3'b001, x_or2_S2=3'b010,
 x_or2_S3=3'b011, x_or2_S4=3'b100;

always @(posedge clk)
begin
//Sub_Start
if (opCode == do_sub)
begin
current_state = do_sub;
sub_current_state = sub_next_state;
case(sub_current_state)
sub2_S0: sub_next_state = sub2_S1;
sub2_S1: sub_next_state = sub2_S2;
```

```
sub2_S2: sub_next_state = sub2_S3;
sub2_S3: sub_next_state = sub2_S4;
sub2_S4: sub_next_state = sub2_S0;
endcase
end

if (opCode == do_and)
begin
current_state = do_and;
and_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;
endcase
end
if (opCode == do_add)
begin
current_state = do_add;
add_current_state = add_next_state;
case(add_current_state)
add2_S0: add_next_state = add2_S1;
add2_S1: add_next_state = add2_S2;
add2_S2: add_next_state = add2_S3;
add2_S3: add_next_state = add2_S4;
add2_S4: add_next_state = add2_S0;
endcase
end
if (opCode == do_x_or)
begin
current_state = do_x_or;
x_or_current_state = x_or_next_state;
case(x_or_current_state)
x_or2_S0: x_or_next_state = x_or2_S1;
x_or2_S1: x_or_next_state = x_or2_S2;
x_or2_S2: x_or_next_state = x_or2_S3;
x_or2_S3: x_or_next_state = x_or2_S4;
x_or2_S4: x_or_next_state = x_or2_S0;
endcase
end



end


//Output logic based on states (Bitwise AND
operartion)
always @(current_state)
    begin
if(current_state == do_sub)
case(sub_current_state)
//and2_S0: C = 0;
sub2_S1:
begin
temp = A[0] - B[0];
C[0]=temp[0];
Carry = temp[1];
end
sub2_S2:
begin
temp2=A[1] - B[1] - temp[1];
C[1]=temp2[0];
Carry = temp2[1];
end
sub2_S3:
begin
temp3=A[2] - B[2] - temp2[1];
C[2]=temp3[0];
Carry = temp3[1];
end
sub2_S4:
begin
temp4=A[3] - B[3] - temp3[1];
C[3]=temp4[0];
Carry = temp4[1];
Zero = C == 0?1:0;
Sign = C[3] == 0?0:1;
end
endcase
if(current_state == do_and)
case(and_current_state)
//and2_S0: C = 0;
and2__S1:
begin
C[0]=(A[0] & B[0]);
Carry = 0;
end
                and2__S2:
begin
C[1]=(A[1] & B[1]);
Carry = 0;
end
                and2__S3:
begin
C[2]=(A[2] & B[2]);
Carry = 0;
end
                and2__S4:
                begin
C[3]=(A[3] & B[3]);
Carry = 0;
Zero = C == 0?1:0;
Sign = C[3] == 0?0:1;
end
endcase

if(current_state == do_add)
case(add_current_state)
//and2_S0: C = 0;
```

```
add2_S1:
begin
tmp = (A[0] + B[0]);                              endmodule
C[0]=tmp[0];
Carry = tmp[1];
end
    add2_S2:
    begin
tmp2 = (A[1] + B[1] + tmp[1]);
C[1]=tmp2[0];
Carry = tmp2[1];
end
    add2_S3:
    begin
tmp3 = (A[2] + B[2] + tmp2[1]);
C[2]=tmp3[0];
Carry = tmp3[1];
end
    add2_S4:
begin
tmp4 = (A[3] + B[3] + tmp3[1]);
C[3]=tmp4[0];
Carry = tmp4[1];
Zero = C == 0?1:0;
Sign = C[3] == 0?0:1;
end
endcase

if(current_state == do_x_or)
case(x_or_current_state)
//and2_S0: C = 0;
x_or2_S1:
begin
C[0]=(A[0] ^ B[0]);
Carry = 0;
end
                x_or2_S2:
                begin
C[1]=(A[1] ^ B[1]);
Carry = 0;
end
                x_or2_S3:
                begin
C[2]=(A[2] ^ B[2]);
Carry = 0;
end
                x_or2_S4:
begin
C[3]=(A[3] ^ B[3]);
Carry = 0;
Zero = C == 0?1:0;
Sign = C[3] == 0?0:1;
end
endcase
end
```