Designing a 4-bit ALU

Group-5 CSE 460 Lab Section(9)

Nusrat Zaman Raya
dept. of CSE
BRAC University
ID: 20301110
Email: nusrat.zaman.raya@g.bracu.ac.bd

Saif Mokarrom
dept. of CSE
BRAC University
ID: 20301121
Email: md.saif.mokarrom@g.bracu.ac.bd

Al Shahriar Him
dept. of CSE
BRAC University
ID: 20301131
Email: al.shahriar.him@g.bracu.ac.bd

Farhan Faruk
dept. of CSE
BRAC University
ID: 20301137
Email: farhan.faruk@g.bracu.ac.bd

Ismail Hossain Saihan
dept. of CSE
BRAC University
ID: 20301159
Email: ismail.hossain.saihan@g.bracu.ac.bd

***Abstract*—This paper centers on the arithmetic operations of the Arithmetic Logic Unit (ALU). The ALU is a critical component responsible for mathematical and logical computations in the central processor unit (CPU) of a computer. In modern CPU's, ALUs are exceptionally robust and sophisticated. The objective of this project is to design a 4-bit ALU that can perform four arithmetic operations, utilizing adder logic. To achieve this, the logic of adders and multiplexers was modified to create the ALU. The four operations that this ALU can execute are ADD, NOR, SUB, and XNOR. The significance of this study is that it utilizes adders and multiplexers instead of logic gates for each operation, providing a unique approach to designing an ALU.**

***Index Terms*——component, arithmetic logic, multiplexers**

I. Introduction

Computers are intricate machines that operate on binary code, consisting of a series of 0s and 1s. To perform complex calculations, a computer relies on a key component known as the Arithmetic Logic Unit (ALU), which is responsible for executing mathematical and logical operations on binary numbers. While the ALU is designed to work with binary numbers, the Floating Point Unit (FPU) specializes in decimal calculations, enabling it to perform operations on non-integer numbers. The CPU, FPU, and GPU work in tandem to make up the ALU, which can consist of multiple units within a single CPU or FPU. The data on which the ALU operates is known as operands, which are input into the unit to execute specific actions. The output of the ALU is the result of the action taken on the input operands. Additionally, the ALU includes registers that store the results of past or recent operations, allowing for faster and more efficient processing. Registers are used by the CPU to store, retrieve, and process data. Processor registers are specialized registers used by the CPU for processing operations. The modern ALU is a highly advanced component of computers and may also include a Control Unit (CU) that manages and coordinates the flow of data between the ALU, memory, and registers. Overall, the ALU is a critical component of modern computers that enables them to perform complex mathematical and logical operations quickly and efficiently.

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

II. Operation

A. State Diagram

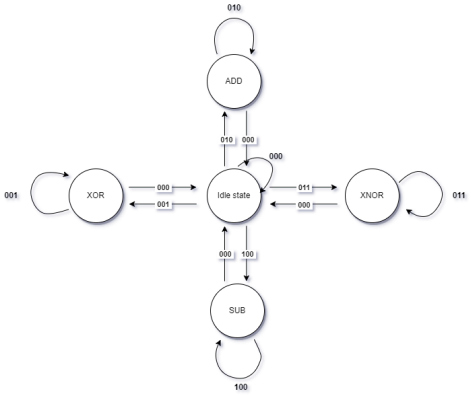

Fig. 1. State Diagram

Here is a simplified explanation of the state diagram. Initially, the system is in an idle state. It will work based on the opcode. There are four opcodes to choose from: ADD, SUB, NOR, and XNOR, and each one does a different task. The opcode should stay the same during the task. Once the task is done, the opcode needs to be reset to 000 by entering it

into the timing diagram after a certain number of clock cycles. Then the system goes back to doing nothing.

B. Timing Diagram

1. XOR:

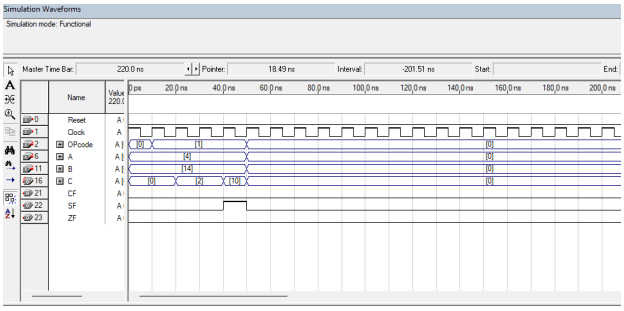

Fig. 2. Timing Diagram Of XOR

Here we give the 2 inputs (A, B) and a given opcode for XOR operation which is 001. If we look into the timing diagram, we can see that for given input and opcode we get our desired output. As it is a bitwise possible clock ALU. So, the output will be based on it. As it is a XOR operation the sign flag, zero flag, and carry flag will not work for it. This operation will only need 2 types of input one is the 4-bit A, B, and 3-bit opcode, and will work on a positive clock bitwise operation.

2. ADD:

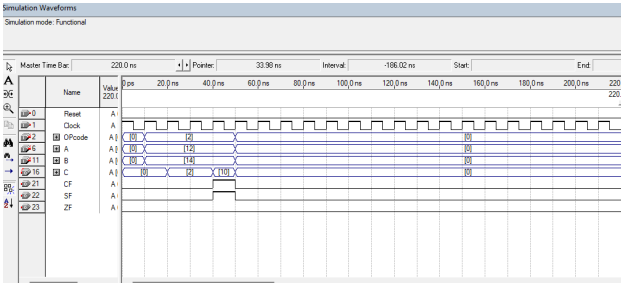

Fig. 3. Timing Diagram Of ADD

Here we give the 2 inputs (A, B), where the given opcode for ADD operation is 010. If we look into the timing diagram, we can see that for given input and opcode we get our desired output. Here it is a bitwise possible clock ALU. So, the output will be based on it. There will be a sign flag, a zero flag, and a carry flag for the operation of the ALU.

3. XNOR:

Here we give the 2 inputs (A, B) and a given opcode for the XNOR operation which is 011. If we look into the timing diagram, we can see that for given input and opcode we get our desired output. As it is a bitwise possible clock ALU. So, the output will be based on it.As it is a XNOR operation the sign flag, zero flag, and carry flag will not work for it. This operation will only need 2 types of input one is the 4-bit A, B, and 3-bit opcode, and will work on a positive clock bitwise operation.

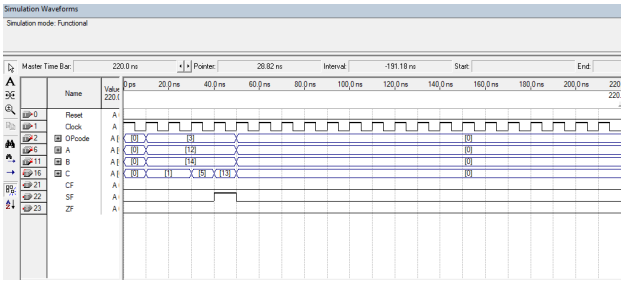

Fig. 4. Timing Diagram Of XNOR

4. SUB:

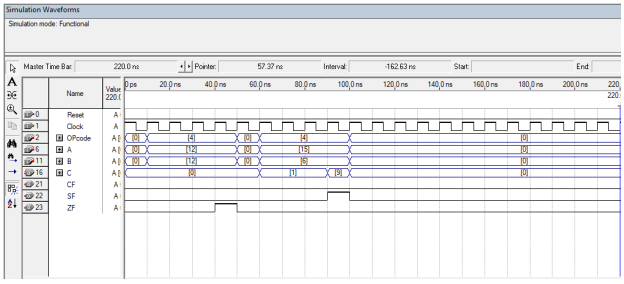

Fig. 5. Timing Diagram Of SUB

Here we give the 2 inputs (A, B) and a given opcode for SUB operation which is 100. If we look into the timing diagram, we can clearly see that for given input and opcode we get our desired output. Here it is a bitwise possible clock ALU. So, the output will be based on it. There will be a sign flag, a zero flag, and a carry flag for the operation of the ALU.

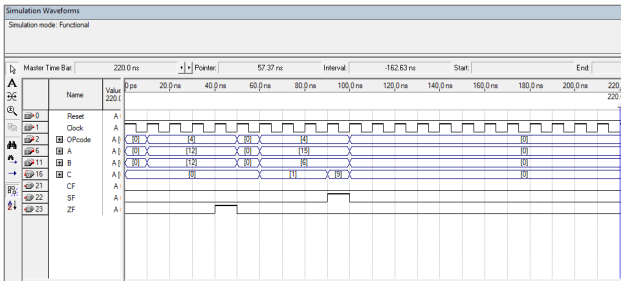

Fig. 6. Timing Diagram Of All 4 Operation

III. Conclusion:

Logical and mathematical operations are versatile tools that can facilitate various activities, including circuit construction, hardware programming, problem-solving, and control system design. Our project's ALU is adept at performing critical operations, such as XNOR, SUB, NOR, and ADD operations, which are essential in real-world applications that require precision and accuracy, as it complies with timing diagrams and truth tables. Additionally, the ALU has the potential for future enhancements, allowing it to execute more tasks effectively. Our team has succeeded in developing a 4-bit ALU

that can proficiently execute XNOR, SUB, NOR, and ADD operations, providing a valuable resource for fulfilling an array of engineering needs.

IV. Appendix (Verilog Code)

```
module fourBitALU (input Clock,
    input Reset, input[3:0]A,
    input[3:0]B, input[2:0]OPcode,
    output reg[3:0]C, output reg ZF,
    output reg CF, output reg SF);

reg[1:0]temp;
reg car;

initial car = 0;
reg [2:0] count;

initial temp = 0;
initial count = 0;

parameter [2:0]
RESET = 3'b000,
XOR = 3'b001,
ADD = 3'b010,
XNOR = 3'b011,
SUB = 3'b100;

always @(posedge Clock)
begin
if (Reset == 0)
begin
temp = 0;
count = 1;
end
else
begin
if (OPcode == RESET)
begin
temp = 0;
count = 1;
C = 0;
ZF = 0;
SF = 0;
CF = 0;
car = 0;
end
else if (OPcode == XOR)
begin
if (count == 1)
begin
C[0] = A[0] ^ B[0];
count = 2;
end
else if (count == 2)
begin
C[1] = A[1] ^ B[1];
count = 3;
end
else if (count == 3)
begin
C[2] = A[2] ^ B[2];
count = 4;
end
else if (count == 4)
begin
C[3] = A[3]^B[3];
if (C == 0)
ZF = 1;
else
ZF = 0;
SF = C[3];
end
end
else if (OPcode == SUB)
begin
if (count == 1)
begin
temp = A[0] - B[0];
C[0] = temp[0];
car = temp[1];
count = 2;
end
else if (count == 2)
begin
temp = A[1] - B[1] - car;
C[1] = temp[0];
car = temp[1];
count = 3;
end
else if (count == 3)
begin
temp = A[2] - B[2] - car;
C[2] = temp[0];
car = temp[1];
count = 4;
end
else if (count==4)
begin
temp = A[3] - B[3] - car;
C[3] = temp[0];
car = temp[1];
if (C == 0)
ZF = 1;
else
ZF = 0;
SF = C[3];
CF = car;
end
end
else if (OPcode == XNOR)
begin
if (count == 1)
```

```
begin
C[0] = A[0] ~^ B[0];
count = 2;
end
else if (count == 2)
begin
C[1] = A[1] ~^ B[1];
count = 3;
end
else if (count == 3)
begin
C[2] = A[2] ~^ B[2];
count = 4;
end
else if (count == 4)
begin
C[3] = A[3]~^B[3];
if (C == 0)
ZF = 1;
else
ZF = 0;
SF = C[3];
end
end
else if (OPcode == ADD)
begin
if (count == 1)
begin
temp = A[0] + B[0];
C[0] = temp[0];
car = temp[1];
count = 2;
end
else if (count == 2)
begin
temp = A[1] + B[1] + car;
C[1] = temp[0];
car = temp[1];
count = 3;
end
else if (count == 3)
begin
temp = A[2] + B[2] + car;
C[2] = temp[0];
car = temp[1];
count = 4;
end
else if (count == 4)
begin
temp = A[3] + B[3] + car;
C[3] = temp[0];
car = temp[1];
if (C == 0)
ZF = 1;
else
ZF = 0;
SF = C[3];
CF = car;
end
end
end
end
endmodule
```