ARITHMETIC LOGIC UNIT (ALU)

-SEMESTER PROJECT-

Student: Ciocian Roxana-Flavia
Coordinator: Dr. Eng. Hangan Lia-Anca
Technical University of Cluj-Napoca

Table of contents:

1. Introduction

1.1 Project proposal

The goal of this project is to design, implement and test the functionalities of an Arithmetic Logic Unit which will perform the following operations: addition and subtraction, multiplication and division, increments and decrements numbers, and also simple logic operations such as AND, OR, NOT, the rotation to the left or to the right of a number and an operation which negates a number. There are some differences between the usual implementation of an ALU and this one. The addition and subtraction operations will be performed on signed numbers, and the convention used for representing signed binary numbers is called two's complement representation.

1.2 Specifications

This project will be simulated on an IDE provided by Xilinx Vivado. As operands, there will be two numbers on 16 bits each and the result generated after executing an operation will have 32 bits for the multiplication operation, 16 bits for the quotient/reminder of the division operation and only 16 bits for the other operations. Since there are 12 possible operations, they need to be encoded in order to know which operation will take place at a certain moment.

Taking into consideration the brief description of my project, I created a weekly plan needed for designing and implementing it. The plan looks like this:

Week 1 → In this week, we had to choose the project we are going to implement the whole semester.

Weeks 2-3 →These weeks are dedicated to the research part. In the beginning, I need to create a plan for the project's development: establishing the tasks, listing the steps needed to finish the project and making a list of bibliographic references from where I will take all the information needed to complete the tasks.

Weekes 4-5 → In these weeks I will go deeper into the research part. I will take care about 2 part of the project: analysis and design. I will try to find all the components I need for starting to work at this project and I will give a brief explanation about the algorithms I am going to use. Also, I will try to start the coding part if this is possible.

Weekes 6-10 → During these weeks, I will try to finish the implementation of the ALU, and eventually comes up with some improvements if necessary.

Weeks 11-12 → Final weeks are dedicated for testing the project, making some adjustments if the project does not work properly and getting final conclusions.

Week 13 → Project presentation.

2. Bibliographic Study

What is an Arithmetic Logic Unit?

"An arithmetic logic unit (ALU) is a digital circuit used to perform arithmetic and logic operations. It represents the fundamental building block of the central processing unit (CPU) of a computer. Modern CPUs contain very powerful and complex ALUs. In addition to ALUs, modern CPUs contain a control unit (CU).

Most of the operations of a CPU are performed by one or more ALUs, which load data from input registers. A register is a small amount of storage available as part of a CPU. The control unit tells the ALU what operation to perform on that data and the ALU stores the result in an output register. The control unit moves the data between these registers, the ALU, and memory." [1]

What operations can be implemented using the ALU?

An Arithmetic Logic Unit can implement multiple operations, such as arithmetic operations, bitwise logical operations and bit shift operations. Arithmetic operations part includes addition, subtraction, multiplication, division, increment, decrement. Bitwise logical operations part includes AND, OR, NOT, XOR, etc. For the bit shift operations part, it can implement arithmetic shifts, logical shifts and rotations.

What is 2’s complement?

“Two's complement is a mathematical operation on binary numbers, and is an example of a radix complement. It is used in computing as a method of signed number representation. Two's complement is the most common method of representing signed integers on computers. Compared to other systems for representing signed numbers (e.g., ones' complement), two's complement has the advantage that the fundamental arithmetic operations of addition, subtraction, and multiplication are identical to those for unsigned binary numbers.”[2]

How can numbers be represented using 2’s complement?

There is a simple algorithm used to convert a binary number into 2’s complement representation. In order to get the 2’s complement form of a binary number, we need to invert the bits and then add 1 to the least significant bit (LSB) of the given result. The most significant determines the sign of the number, which is called the sign bit. If the sign bit is 0, the number is positive, else the number is negative.

3. Analysis

3.1 Addition and Subtraction

Addition is the operation where digits are added bit by bit, from right to left, with carries passed to the next digit to the left. An overflow occurs when the result from an operation cannot be represented with the available hardware, in this case 8 bits. When adding operands with different sign, as in our case when the operands are in two’s complement, the overflow cannot occur, because the sum of 2 such numbers must be no larger than one of the operands, since the operands are no larger than 8 bits. For example:
5 – 2 = 3. Overflow occurs when adding 2 positive numbers and the sum is negative, or when adding 2 negative numbers and the sum is positive.

In order to implement the addition using 2’s complement, there are 3 possible cases: case 1 - adding a positive number with a negative number when the positive number has a greater magnitude, case 2 - adding a positive number with a negative number when the negative number has a greater magnitude or case 3 - adding 2 negative/positive numbers.

Subtraction is the operation that uses addition, such that the second operand is being negated before being added. In order to do that, 2’s complement transformation will be used on the second operand. In the case where overflow occurs, we have some differences from the addition operation, such that, here the overflow cannot occur when operands have the same sign, because we subtract by negating the second operand and

then add it. For example: 4 – (+2) = 4 – 2 = 2. Overflow occurs when subtracting a negative number from a positive one and get a negative result, or when subtracting a positive number from a negative one and get a positive result.

In order to implement the subtraction algorithm using 2's complement, the next steps must be followed: firstly, we need to find the 2's complement of the second operand, perform the basic addition between them.

3.2 Multiplication and Division

For the multiplication operation, I will use Booth's multiplication algorithm, which is an algorithm that multiplies two signed binary numbers in two's complement notation in efficient way: less number of additions/subtractions required. Booth's algorithm examines adjacent pairs of bits of the N-bit multiplier M in 2's complement representation. Initially, the most significant bits of the product are filled with '0'. To the right of them, append the multiplier. Also, the least significant bit is filled with 0. For each bit M(i), for i running from 0 to N-1, the bits M(i) and M(i-1) are considered. Where those two bits are equal, the product is left unchanged. When M(i) = 1 and M(i-1) = 0, the multiplicand is subtracted from the product, but when M(i) = 0 and M(i-1) = 1, the multiplicand is added to the product.

Example:

3 * (-4), m-the multiplicand, r-the multiplier, p-the product

m = 0011, r = 1100, p = 0000 1100 0

1. p = 0000 110**0 0** → the last two bits are equal => p remains unchanged & perform right shift.
2. p = 0000 011**0 0** → the last two bits are equal => p remains unchanged & perform right shift.
3. p = 0000 001**1 0** → the last two bits are 10 => the multiplicand is subtracted from p => p = 1101 0011 0 & right shift.
4. p = 1110 100**1 1** → the last two bits are equal => p remains unchanged & perform right shift.

- the process was done 4 times (the number of bits) => drop the least significant bit from the product => p = 1111 0100, which is -12.

Division operation is the reciprocal operation of multiplication. There are two operands, named dividend and divisor and a result named quotient accompanied by a second result called the remainder. The relationship between components:
dividend = divisor * quotient + reminder.

For the division multiplication, I will use Long Division algorithm, which *"is the standard algorithm used for "pen-and-paper" division of the multi-digit numbers expressed in decimal notation. It shifts gradually from the left to the right end of the dividend, subtracting the largest possible multiple of the divisor (at the digit level) at each stage, the multiples then become the digits of the quotient, and the final difference is then the reminder. " -[11]*

Example:

n-the dividend, d- the divisor, q-the quotient and r- the reminder.

n = 12 = 1100, d = 4 = 100

-set q and r to 0

-we need a loop from N-1(nr of bits of n - 1) to 0

1. i = 3

r = 00 → left shifted by 1

r = 01 → setting r(0) to n(i)

r < d → skip

2.i = 2

r = 010 → left shift by 1

r = 011 → setting r(0) to n(i)

r < d → skip

3.i = 1

r = 0110 → left shift by 1

r = 0110 → setting r(0) to n(i)

r >= d => r = 10 → (r-d)

q = 10 → setting q(i) to 1

4.i = 0

r = 100 → left shift by 1

r = 100 → setting r(0) to n(i)

r >= d => r = 0 → (r-d)

q = 11 → setting q(i) to 1

end loop => q = 3 = 0011, r = 0 = 0000

3.3 AND, OR, NOT and the negation, rotate left, rotate right, increment and decrement

AND →the logical operation from where the bitwise AND of the two operands appears in the result.
OR → the logical operation from where the bitwise OR of the two operands appears in the result.
NOT → the logical operation, with only one operand, that inverts the bits and replaces every 1 with 0 and every 0 with 1.
Negation → one of the operands is subtracted from 0, which will be the sign change operation.
Rotate left → it is an operation that involves only one operand which is treated as a circular buffer of bits, such as its least and most significant bits are adjacent. For this operation, the most significant bit shifts to the least significant bit, and the rest bits shift left one bit.
Rotate right → it is an operation that involves only one operand which is treated as a circular buffer of bits, such as its least and most significant bits are adjacent. For this operation, the least significant bit shifts to the most significant bit, and the rest bits shift right one bit.
Increment → it is the operation where one operand is incremented by one.
Decrement → it is the operation where one operand is decremented by one.

4. Design

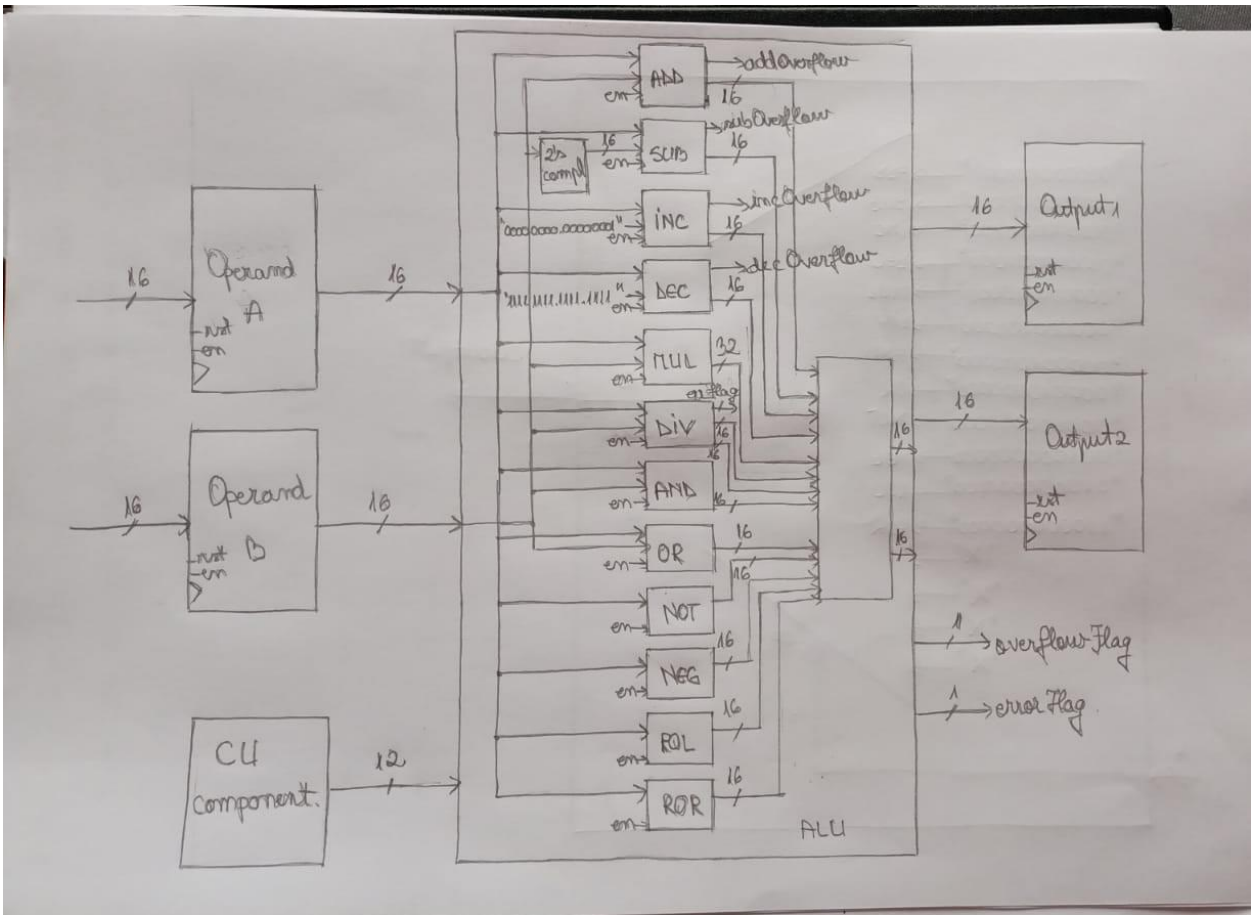
The design of the Arithmetic Logic Unit is based on some components needed in order to complete the implementation of the required operations from the initial task.

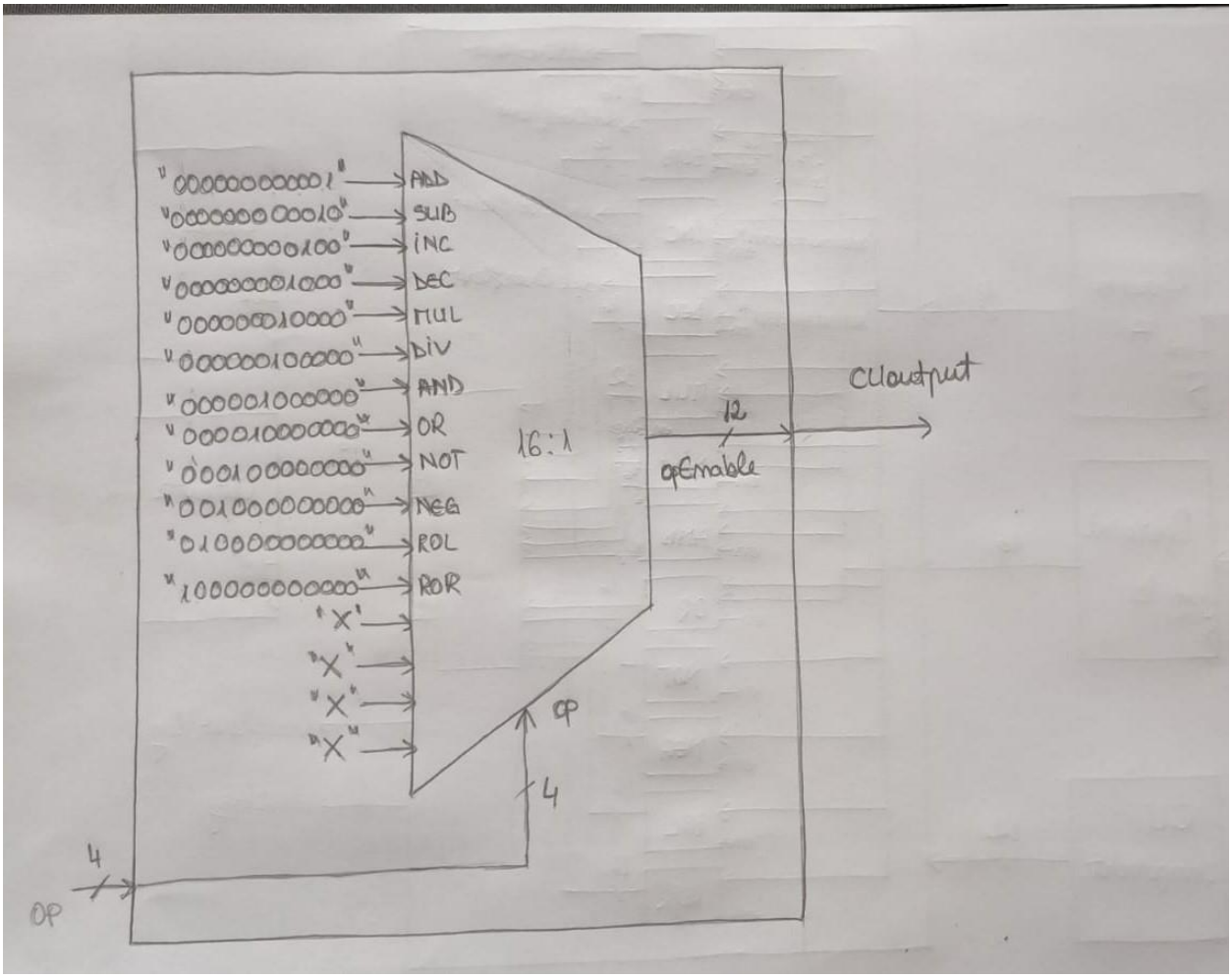
The components needed, in order to design the ALU, are: 16-bit registers, 1-bit full adders, multiplexers.

The Arithmetic Logic Unit will have two inputs, which will be held in two separate registers. Also, there will be others two separate registers for the outputs. I need two registers for the outputs, because of the fact that for the multiplication algorithm, the result will be on 32 bits, therefore, I need two 16-bit registers for storing the result. Moreover. The division operation has two outputs, representing the quotient and the reminder, both on 16 bits, therefore, the two 16-bit registers will be needed as well for storing the result. For the rest of the operations, the result will be only on 16 bits, thus it will be stored on the second 16-bit register, the first one being inactive.

Every operation executed in this project will be represented as components from which the final ALU will be formed.

For the addition, subtraction, increment and decrement, I will need to cascade 16 1-bit full adders. Since the subtraction is an addition, but with the second operand represented in 2's complement, I will need a component that will perform the transformation of a number into the 2's complement representation. The increment and decrement operations, are just simple additions, having the second operand 1, -1 respectively.





In order to determine which operation will be executed, I will need a Control Unit component, which will be basically a 16:1 multiplexer. The operations will be selected based on a code, representing the selection of the multiplexer. Since there are 12 such operations, they need to be encoded on 4 bits, such that there will be a code corresponding to each and every instruction. There will be 12 active inputs for the multiplexer, one for each operation. Based on the selection, the output will be a string of 12 bits (because there are 12 operations). On that string, there is only one '1' that can be on any position, meaning that only one operation is executed. That string will be an enable input for the ALU. Each bit from that string represents an enable input for the instruction components, and only the component where the enable is '1' will perform the operation.

The 4-bit code for each operation is:

- ADD → 0000
- SUB → 0001
- INC → 0010

- DEC → 0011
- MUL → 0100
- DIV → 0101
- AND → 0110
- OR → 0111
- NOT → 1000
- NEG → 1001
- ROL → 1010
- ROR → 1011

5. Implementation

The operations implementation:

- The addition, subtraction, increment and decrement are implemented, as it is already mentioned, by cascading 16 1-bit full adders. There are cases where the overflow occurs. For solving those cases, each component has an overflow flag that is a XOR operation between the 2 most significant bits of the carry out. The overflow occurs when the positive number is larger that 32767, and the negative number is smaller than -32768. On 16 bits, the maximum signed number that can be represented is 32767, -32768 respectively. Also, each component has an enable, and only when it is '1' the result is given.
- The multiplication operation is done using repeated additions/subtractions and right shifts. We need a loop from 0 to 15 (because we have 16 bits) and at each step verify the least two bits from the partial product. The multiplicand is added/subtracted from the partial product depending on those two bits. At the end of the loop, the result will be right shifted by one, giving us the final product. This component has also an enable input, and only when it is '1' the result is given.
- The division operation is done using repeated subtractions and left shifts. We need a loop from 15 to 0, where the reminder it is verified. If it is larger than the dividend, the dividend is subtracted from the reminder, and the quotient receives a 1 on the position of the loop. There is an error flag, which will be activated when the divisor is 0. Moreover, it has an enable flag as well, and the result is given only when the enable flag is active.
- The rest of the operations, are done using logic functions, except for the negation operation, which uses the 2's complement component. Of course, each operation has an enable flag as well.

The Control Unit implementation:

- The Control Unit was implemented as a 16:1 multiplexer. Its output will be an input for the ALU, that will enable the operation wanted to be performed. The output will be on 12 bits, each bit representing the enable input for one operation.

Example: if the selection of the multiplexer is ‘0000’, that means I want to perform the ADD operation. The output is ‘000000000001’, which will enable only the addition operation from ALU.

The ALU implementation:

- The ALU component combines the 12 components of the operations, the CU component and the register component. Initially, the inputs are put into registers. The enable from the CU is given to each operation. Based on the operation performed, the ALU outputs will receive the corresponding results. After that, those result are also put into registers. If the operation is multiplication or division, both output registers will be active, but if the operation is another one, only the second register will be active, the first one taking ‘Z’ (it is not used).

6. Tests and experiments

I created a table for each operation, to exemplify their functionality:

Addition Operation

Add Op	Test 1 (125 + 49)	Test 2(243 + (-71))	Test 3(-117 + 32)	Test 4(-82 + (-27))
Input 1	0000000001111101 (125)	0000000011110011 (243)	1111111110001011 (-117)	1111111110101111 (-81)
Input 2	0000000000110001 (49)	1111111110111001 (-71)	0000000000100000 (32)	1111111111100101 (-27)
Expect. output	0000000010101110 (174)	0000000010101100 (172)	1111111110101011 (-85)	1111111110010100 (-108)
Actual output	0000000010101110 (174)	0000000010101100 (172)	1111111110101011 (-85)	1111111110010100 (-108)

Subtraction Operation

Sub Op	Test 1(67 – (112))	Test 2(211 – (-103))	Test 3(-12 – (165))	Test 4(-53 – (-62))
Input 1	0000000001000011 (67)	0000000011010011 (211)	1111111111110100 (-12)	1111111111001011 (-53)
Input 2	0000000001110000 (112)	1111111110011001 (-103)	0000000010100101 (165)	1111111111000010 (-62)
Expect. output	1111111111010011 (-45)	0000000100111010 (314)	1111111101001111 (-177)	0000000000001001 (9)
Actual output	1111111111010011 (-45)	0000000100111010 (314)	1111111101001111 (-177)	0000000000001001 (9)

And Operation

And Op	Test 1	Test 2	Test 3	Test 4
Input 1	0011011100001011	0000000000000000	1111111111111111	0101010101010101

Input 2	0101111100100000	0111111011010000	1111111111111111	1110110100001001
Expect. output	0001011100000000	0000000000000000	1111111111111111	0100010100000001
Actual output	0001011100000000	0000000000000000	1111111111111111	0100010100000001

Or Operation

Or Op	Test 1	Test 2	Test 3	Test 4
Input 1	0000000000000000	0000000000000000	1111111111111111	0101010101010101
Input 2	0000000000000000	0111111011010000	1111111111111111	1110110100001001
Expect. output	0000000000000000	0111111011010000	1111111111111111	1111110101011101
Actual output	0000000000000000	0111111011010000	1111111111111111	1111110101011101

Not Operation

Not Op	Test 1	Test 2	Test 3	Test 4
Input	0101010101010101	0000000000000000	1111111111111111	0011011100001011
Expect output	1010101010101010	1111111111111111	0000000000000000	1100100011110100
Actual output	1010101010101010	1111111111111111	0000000000000000	1100100011110100

Rotate Left Operation

ROL Op	Test 1	Test 2	Test 3	Test 4
Input	0000101111101111	1111000011110000	0000111100001111	0111100001111000
Expect. output	0001011111011110	1110000111100001	0001111000011110	1111000011110000
Actual output	0001011111011110	1110000111100001	0001111000011110	1111000011110000

Rotate Right Operation

ROR Op	Test 1	Test 2	Test 3	Test 4
Input	0000101111101111	1111000011110000	0000111100001111	0111100001111000
Expect. output	1000010111110111	0111100001111000	1000011110000111	0011110000111100
Actual output	1000010111110111	0111100001111000	1000011110000111	0011110000111100

Increment Operation

Inc Op	Test 1 (143 + 1)	Test 2 (-193 + 1)	Test 3 (83 + 1)	Test 4 (-27 + 1)
Input	0000000010001111 (143)	1111111100111111 (-193)	0000000001010011 (83)	1111111111100101 (-27)
Expect. output	0000000010010000 (144)	1111111101000000 (-192)	0000000001010100 (84)	1111111111100110 (-26)
Actual output	0000000010010000 (144)	1111111101000000 (-192)	0000000001010100 (84)	1111111111100110 (-26)

Decrement Operation

Dec Op	Test 1 (211 – 1)	Test 2 (-15 – 1)	Test 3 (153 – 1)	Test 4 (-137 -1)
Input	0000000011010011 (211)	1111111111110001 (-15)	0000000010011001 (153)	1111111101110111 (-137)
Expect. output	0000000011010010 (210)	1111111111110000 (-16)	0000000010011000 (152)	1111111101110110 (-138)
Actual output	0000000011010010 (210)	1111111111110000 (-16)	0000000010011000 (152)	1111111101110110 (-138)

Negation Operation

Neg Op	Test 1 (132 -> -132)	Test 2(-45 -> 45)	Test 3(73 -> -73)	Test 4(429 -> -420)
Input	0000000010000100 (132)	1111111111010011 (-45)	0000000001001001 (73)	0000000110100100 (420)
Expect. output	1111111101111100 (-132)	0000000000101101 (45)	1111111110110111 (-73)	1111111001011100 (-420)
Actual output	1111111101111100 (-132)	0000000000101101 (45)	1111111110110111 (-73)	1111111001011100 (-420)

Multiplication Operation

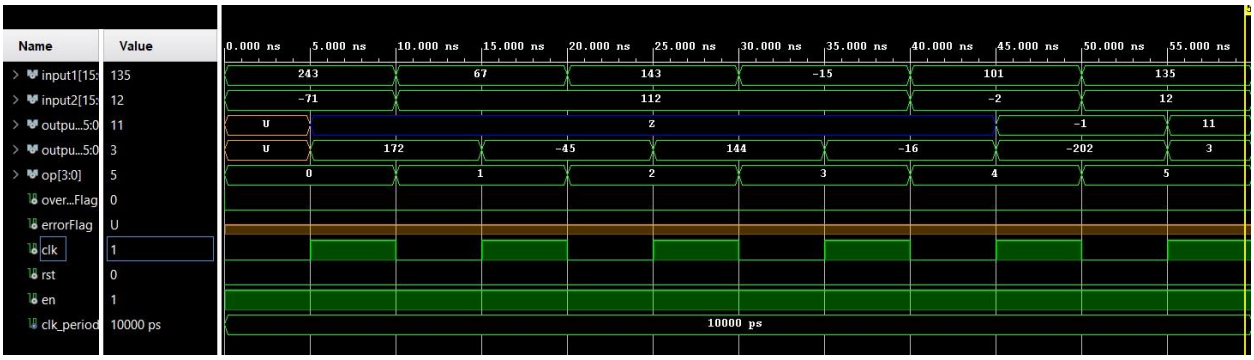
Mul Op	Test 1(33 * 17 = 561)	Test 2(101 * (-2) = -202)	Test 3(-12 *(-6) =72)	Test 4(0 *40 = 0)
Input 1	0000000000100001 (33)	0000000001100101 (101)	1111111111110100 (-12)	0000000000000000 (-8)
Input 2	0000000000010001 (17)	1111111111111110 (-2)	1111111111111010 (-6)	0000000000101000 (40)
Expect output	0000000000000000 0000001000110001 (561)	1111111111111111 1111111100110110 (-202)	0000000000000000 0000000001001000 (72)	0000000000000000 0000000000000000 (0)
Actual output	0000000000000000 0000001000110001 (561)	1111111111111111 1111111100110110 (-202)	0000000000000000 0000000001001000 (72)	0000000000000000 0000000000000000 (0)

Division Operation

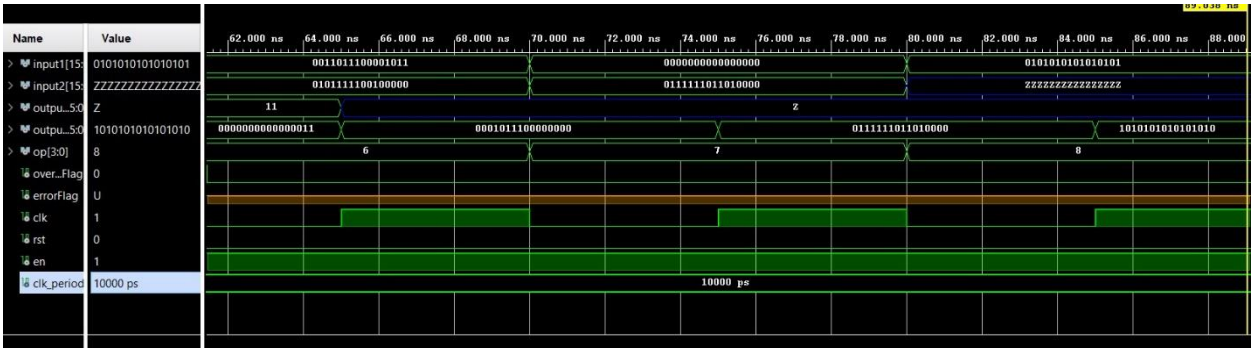
Div Op	Test 1(135:0-> q=’Z’, r=’Z’)	Test 2(83:25 -> q=3, r=8)	Test 3(115:5 -> q=23, r=0)	Test 4(227: 48 -> q=4, r=35)
Input 1	0000000010000111 (135)	0000000001010011 (83)	0000000001110011 (115)	0000000011100011 (227)
Input 2	0000000000000000 (0)	0000000000011001 (25)	0000000000000101 (5)	0000000000110000 (48)
Expect. quotient	ZZZZZZZZZZZZZZZZ	0000000000000011 (3)	0000000000010111 (23)	0000000000000100 (4)
Expect. reminder	ZZZZZZZZZZZZZZZZ	0000000000001000 (8)	0000000000000000 (0)	0000000000100011 (35)
Actual quotient	ZZZZZZZZZZZZZZZZ	0000000000000011 (3)	0000000000010111 (23)	0000000000000100 (4)
Actual reminder	ZZZZZZZZZZZZZZZZ	0000000000001000 (8)	0000000000000000 (0)	0000000000100011 (35)

I will show 4 simulation representing the functionality of the ALU for all operations.

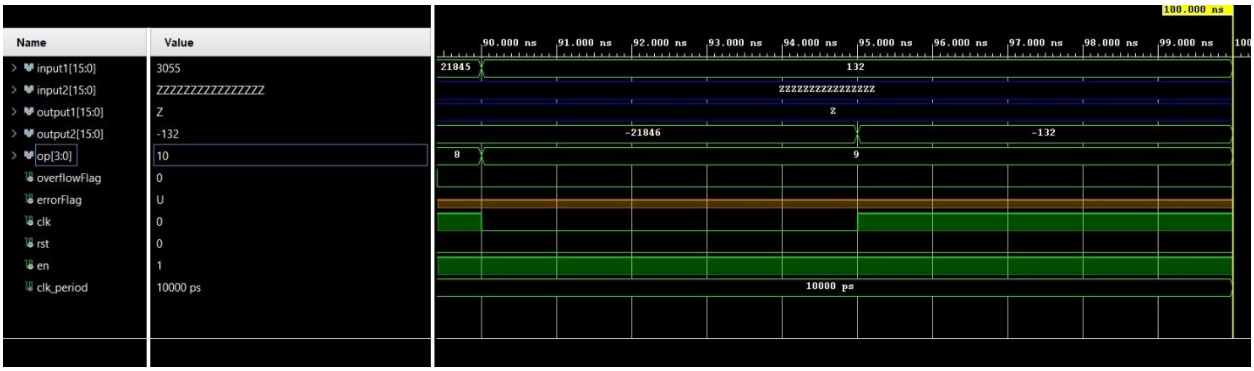
The simulation for the addition, subtraction, increment, decrement, multiplication and division:



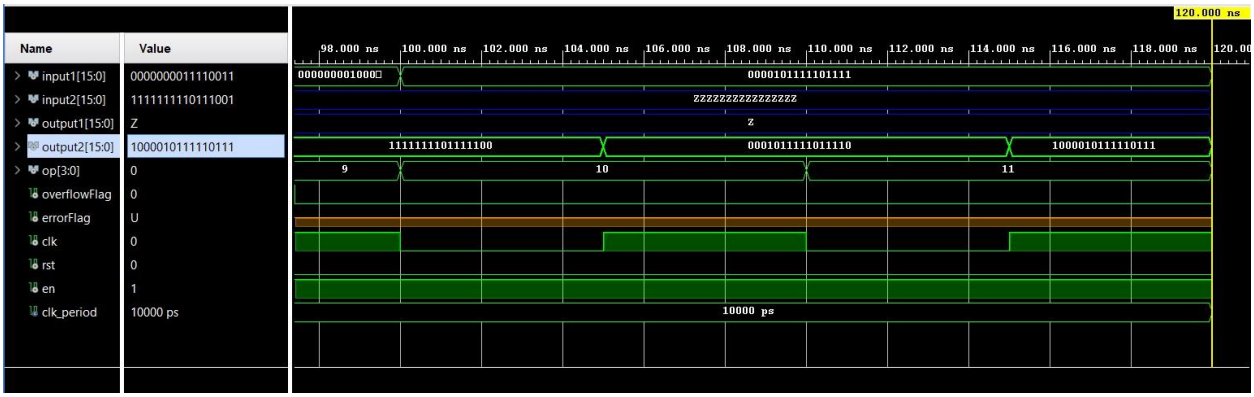
The simulation for and, or, not:



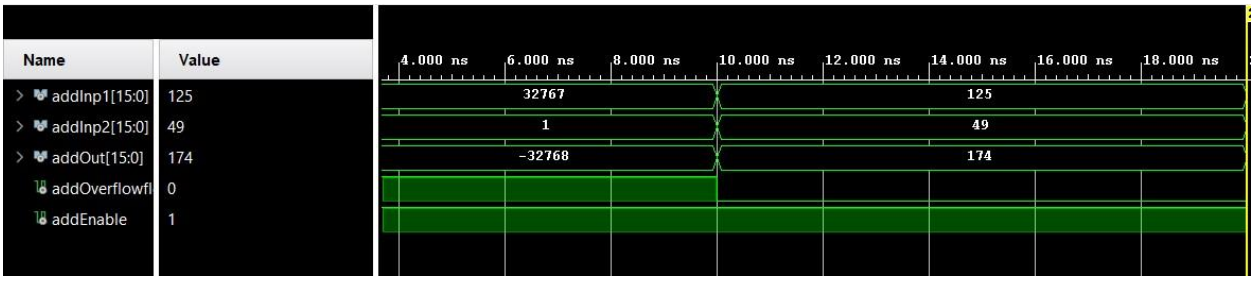
The simulation for negation:



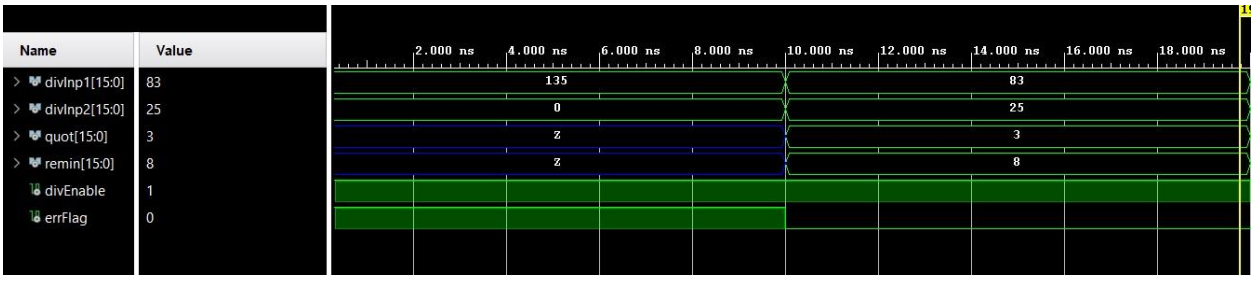
The simulation for rotate left and rotate right:



The simulation for overflow flag:



The simulation for error flag (for division by 0):



7. Conclusions

The goal of this project was to design and implement an Arithmetic Logic Unit, having the inputs on 16 bits. There are many methods for implementing the required operations, but I chose the one that I understood and knew how to implement them.

I liked this project, because it was a challenge for me to create my own version of an Arithmetic Logic Unit. The most challenging task, in my opinion, was to implement the multiplication and division operations. For the multiplication part, I managed to implement it so it can work on signed numbers. Also, to put all the components together in order to obtain the final project, was a challenge too.

This project can be improved by implementing the division operation to work on signed numbers too. Furthermore, it can be practically implemented on a basys3 board, which unfortunately I do not have.

8. Bibliography

[1] – *"Functions of the arithmetic logic unit",* available online:

https://computersciencewiki.org/index.php/Functions_of_the_arithmetic_logic_unit_(ALU)

[2] – *"Two's complement",* available online:

https://en.wikipedia.org/wiki/Two%27s_complement

[3] – *"Arithmetic logic unit",* available online:

https://en.wikipedia.org/wiki/Arithmetic_logic_unit

[4] - David A. Patterson, John L. Hennessy. *Computer Organization and Design: The Hardware/software Interface,3rd Edition.* China Machine Press,2008-4. ISBN: 978-7-111-20214-1

[5] - Gojko Babić, "*Arithmetic / Logic Unit – ALU Design*", available online: https://web.cse.ohio-state.edu/~crawfis.3/cse675-02/Slides/CSE675_05_ALUDesign.pdf

[6] -Baruch, Z. F., *Computer Architecture* (in Romanian), TODESCO, Cluj-Napoca, 2000, ISBN 973-99780-7-x

[7]- Baruch, Z. F., *Structure of Computer Systems* (in English), U. T. PRES, Cluj-Napoca, 2002, ISBN 973-8335-44-2

[8] – Gheorghe Sebestyen-Pal, Structure of Computer Systems (SCS) lectures and laboratories.

[9] – *"Addition and subtraction using 2's complement",* available online:

https://www.javatpoint.com/addition-and-subtraction-using-2s-complement-in-digital-electronics

[10] – "*Booth's multiplication algorithm",* available online:

https://en.wikipedia.org/wiki/Booth%27s_multiplication_algorithm

[11] – "*Division algorithm*", available online:

https://en.wikipedia.org/wiki/Division_algorithm#Pseudocode