Arithmetic Logic Unit (ALU)
Design and VHDL Implementation
— Documentation —

Student: Alexandra Ilovan

Group: 30432

Academic Year: 2022-2023

Coordinating Teacher: Dr. Ing. Anca Hangan

Table of Contents

I. Introduction

Project Task:

Design an Arithmetic Logic Unit (ALU) that performs the following operations:

- Addition and Subtraction in Two's Complement
- Increment and Decrement Operations
- Logical Bitwise AND, OR & NOT Operations
- Logical Negation Operation
- Left and Right Rotate Operations

The ALU will operate on 32 bit integers represented in Two's Complement.

An accumulator register will be used for storing both one input operand and the result.

The multiplication and division operations will require the usage of an additional circuit.

II. Bibliographical Research

2.1. What is an ALU

An Arithmetic Logic Unit (ALU) is a main component of the central processing unit (CPU) of every computer system. It is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers.

An ALU generally has three inputs: two of the inputs represent its operands (the data to be operated on), and the other input is a code indicating the operation to be performed (opcode). The output of an ALU is represented by the result of the performed operation. In some cases, the ALU may also provide status inputs/outputs, which are used to transmit information about the previous or the current operation between the ALU and external status registers.

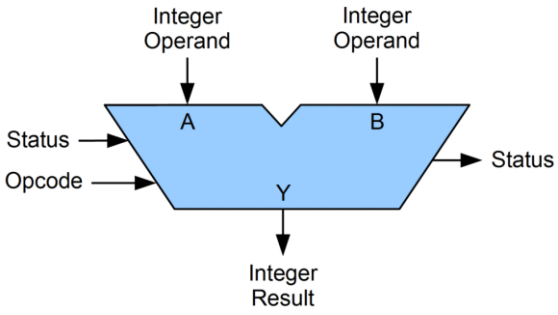

Figure 1. Symbolic representation of an ALU

2.2. How to represent numbers in Two's Complement

Since both the operands and the result of the operations performed by the ALU have to be represented in Two's Complement, it is of utmost importance to understand how to obtain their C2 form. In 2's complement representation, or signed notation, the first bit tells about the sign of the number. The convention is that a number with a leading 1 is negative, while a leading 0 denotes a positive value. For instance, in an 8-bit representation, any number in the range [-128, 127] can be written. The name comes from the fact that a negative number is a two's complement of a positive one. [3]

Positive numbers are represented the same as in the sign magnitude method, but when it comes to negative numbers there are a few additional steps that have to be followed. To find out two's complement of a number, one must first be familiar with one's complement. One's complement of a number can be obtained by simply inverting all the 0's into 1's and vice versa.

Steps to find the 2's complement of any number:

1. Convert the given number into a binary number system or in the base 2 form
2. Find out the 1's complement representation of the given number
3. Add 1 to the least significant bit (LSB) of the number

One important advantage of this system is that -0 and +0 are represented the same as 0 is always considered a positive number, unlike in 1's complement.

Example: finding the representations of 16 and -16 in 2's complement

1. Choose the number of bits in the binaries representation. Let's assume we want values in the 8-bit system.
2. Write down your number, let's say 16. 16 in binary is `1 0000`.
3. Add some leading `0`'s, so that the number has eight digits, `0001 0000`. That's 16 in the two's complement notation.

And what about its counterpart, `-16`?

4. Switch all the digits to their opposite (`0→1` and `1→0`). In our case `0001 0000 → 1110 1111`.
5. Add 1 to this value, `1110 1111 + 1 = 1111 0000`.
6. `1111 0000` in the two's complement representation is `-16` in decimal notation, and is the 2's complement of `0001 0000`.

(Source: https://www.omnicalculator.com/math/twos-complement)

2.3. How to perform arithmetic operations on numbers represented in C2

1. Addition in C2: the addition operation in C2 works the same way as the addition in binary when both of the operands are positive numbers, but when at least one operand is a negative number, some special cases arise.

   Case 1 - Addition of a positive and negative number, when the positive number has greater magnitude:

   - Perform the addition between the positive number and the 2C representation of the negative number
   - Drop the carry bit 1 of the output, so that the result is a positive number

   Special case: if the register size becomes too big, the sign extension method of the MSB has to be applied in order to preserve the sign of the number.

   Case 2 - Addition of a positive and negative number, when the negative number has greater magnitude:

   - Perform the addition between the positive number and the 2C representation of the negative number
   - The result will be the 2C representation of the addition output

   Case 3 – Addition of two negative numbers:

   - Convert both of the operands to 2C and compute the addition
   - Drop the carry bit and then convert the result to 2C [3]

2. Subtraction in C2

To subtract two binary numbers in 2C, the steps below have to be followed:

1. Find the 2C value of the subtraend
2. Add the minuend to the obtained value
3. If the result of the addition has the carry bit 1, drop the carry bit, otherwise the result will be the two's complement of the addition output [3]

3. Multiplication in C2

The methods for multiplying sign-magnitude normally work with 2C numbers if the multiplicand is negative, however some adaptations need to be made if the multiplier is negative. There are two possible methods to handle this situation:

Method 1:

- Check if the multiplier is negative
- If it is, take the 2's complement of both operands before multiplying
- Since both operands have been negated, the result will still have the correct sign

Method 2:

- Subtract the partial product resulting from the MSB instead of adding it like the other partial products
- The multiplicand's sign bit needs to be extended by one position

4. Division in C2 – repeatedly applying 2C Subtraction

1. Take the 2's complement of the divisor and add it to the dividend
2. In the following subtraction cycle, the quotient replaces the dividend
3. Keep subtracting until the quotient becomes too small or 0; if it is not 0, then it is treated as a remainder

2.4. How to perform operations on bitwise integers in VHDL

- By using the provided packages (std_logic_1164, numeric_std, std_logic_arith)
- By using the arithmetic operators (+, -, *, /) and the provided keywords (and, or, not)
- By using comparison operators (for comparing the divisor and dividend during the division operation for example)
- By using shifting operators (<<, >>) for the left and right rotates

III. Project Planning, Proposal and Analysis

3.1. Project Planning

Week	Corresponding Tasks
Weeks 1 & 2 (03.10.2022 – 16.10.2022)	- Choose the theme of the project to be implemented - Write down the requirements of the project implementation process - Research the unknown concepts regarding the theme and general aspects of how to implement it (bibliographical study) - Document yourself on how to represent numbers in Two's complement, and how to perform arithmetic and logical operations on them - Start writing the documentation
Weeks 3 & 4 (17.10.2022 – 30.10.2022)	- Make a detailed weekly plan scheduling the tasks and the progress I want to make every two weeks - Think about the main building blocks of the project and its future design - Research the algorithms that will be used in the making of the project
Weeks 5 & 6 (31.10.2022 – 13.11.2022)	- Make design decisions - Figure out user interaction with the board - Start designing the main components needed for a 1-bit ALU - Design how to interconnect the components in order to obtain a functioning 32-bit ALU
Weeks 7 & 8 (14.11.2022 – 27.11.2022)	- Implement the Arithmetic and Logic Units - Implement the Control Unit and Register Unit
Weeks 9 & 10 (28.11.2022 – 11.12.2022)	- Implement the multiplication and division circuits - Finish implementing the top level of the project - Start testing on the FPGA board and fix potential bugs
Week 11 (12.12.2022 – 25.12.2022)	- Record experimental results and finish the first version of the documentation
Week 12 (09.01.2023 – 16.01.2023)	- Make finishing touches both to the project implementation and documentation, test all cases, fix possible problems - Present the project along with the documentation

3.2. Project Proposal

In order to be able to perform the arithmetic and logical operations on numbers represented on 32 bits, the ALU interconnects 32 components that will perform the corresponding operations on 1-bit numbers, generating intermediate results that will lead to obtaining the final 32 bit result. The building blocks of each such component are represented by logic gates and multiplexers.

The accumulator register has the role of storing both the value of one of the input operands and the value obtained as a result of the ALU operations.

The Control Unit (CU) specifies the operation performed by the ALU from the given operation-set that is available, by generating some control signals that are specific to each operation and that ensures a proper state transition.

The additional circuit required for the multiplication and division operations will be based on the implementation of some hardware multiplication and division algorithms, being therefore designed like a finite state automata, that is also controlled by the CU. The multiplication operation will be performed through the Shift-and-Add method, while the division will make use of the Restore Division algorithm, both of which will be further explained in the Design section of this documentation.

3.2.1. User Interaction with the Basys3 Board of the Artix7 Family

- User can set the value of the operands using the 16 switch inputs provided by the board (each switch represents 1 bit, the most significant bit being the first from the left)
- Once the values of the operators are set, the user must press the load button corresponding to the appropriate operand
- The user can „move” through the operations by using the left and right buttons of the board (the operation will be done instantly)
- The user can use the reset button (the one on the middle) to empty the accumulator and dedicated registers

3.3. Project Analysis

The analysis stage aims to provide a more in-depth description of the list of functionalities provided in the project proposal, as well as explain the algorithms that will be used in the implementation of said functionalities.

3.3.1. Analysis of the basic arithmetic operations

a. Addition operation

The operation of addition is a fundamental operation, since most arithmetic operations are based on the operation of addition: subtraction is an addition with the opposite of the subtractor, multiplication is a repeated addition, and division is a sequence of subtractions and additions. The starting point in the description of the addition of 32- bit numbers is the addition of single-bit numbers. The equations describing this operation are:

$$S = X_1 \oplus X_2 \oplus Cin$$

$$Cout = X_1 X_2 + (X_1 \oplus X_2)\, Cin$$

The algorithm used to describe and implement the addition operation on 32-bit numbers is Ripple-Carry Add. The main idea of the algorithm is the propagation of the output transport generated at a lower rank as the input transport for the immediately following rank. Practically the algorithm follows the logic of decimal addition with carry propagation to the immediately higher rank.

Hence, the equations for adding the corresponding bits on the position *i* become:

$$S_i = X_{1i} \oplus X_{2i} \oplus Cin_i$$

$$Cout_i = X_{1i} X_{2i} + (X_{1i} \oplus X_{2i})\, Cin_i$$

Should the result of the addition be too large to be represented on 32 bits, an overflow occurs. Such an overrun is recorded within the addition operation, only if:

- **the operands have the same sign** (for different signs, the representation range cannot be exceeded, the operation actually representing a subtraction)
- **the sign of the result differs from the signs of the operands** (the available representation range has been exceeded)

Below is a table the displaying all the possible cases when an overflow may or may not occur depending on the signs of the operands:

X_1	X_2	$X_1 + X_2$	Overflow
≥ 0	≥ 0	≥ 0	No
≥ 0	≥ 0	< 0	Yes
≥ 0	< 0	≥ 0	No
≥ 0	< 0	< 0	No
< 0	≥ 0	≥ 0	No
< 0	≥ 0	< 0	No
< 0	< 0	≥ 0	Yes
< 0	< 0	< 0	No

Therefore, overflow detection and signaling is done using the basic combinational logic elements as follows:

$$Overflow+ = X_1 X_2\, not(S) + not(X_1)\, not(X_2) S$$

Basically, overflow occurs when the input and output carry of the most significant rank differ:

$$Overflow+ = Cout_{31} \oplus Cin_{31} = Cout_{31} \oplus Cout_{30}$$

b. Subtraction operation

The subtraction operation exploits the advantages of representing binary numbers in 2's complement, starting from the premise that the opposite of a 2's complement number is obtained by incrementing the 1's complement, obtained by negating all the bits corresponding to the number:

$$-x = not(x) + 1$$

Therefore, the subtraction operation is reduced to a simple addition with the opposite of the subtractor, thus obtained:

$$X1 - X2 = X1 + (not(X_2) + 1)$$

The extension of the subtraction operation from 1 bit to 32 bits is performed similarly to that of the addition operation, the subtraction actually representing an addition.

As for scope overflow, in the subtraction operation, it has the opposite meaning to the overflow encountered in the addition operation, and can only be recorded when:

- **operands have different signs** (in the case of identical signs, the range result cannot be exceeded when subtracting)
- **the sign of the result is identical to the sign of the subtractor** (the absolute value of the subtractor is greater than the absolute value of the subtrahend)

X_1	X_2	X_1 - X_2	Overflow
≥ 0	≥ 0	≥ 0	No
≥ 0	≥ 0	< 0	No
≥ 0	< 0	≥ 0	No
≥ 0	< 0	< 0	Yes
< 0	≥ 0	≥ 0	Yes
< 0	≥ 0	< 0	No
< 0	< 0	≥ 0	No
< 0	< 0	< 0	No

Thus, the logical equation through which an overflow can be obtained from the subtraction operation is:

$$Overflow_{-} = X_1 not(X_2) not(S) + not(X_1) X_2 S$$

c. Increment operation

Incrementation represents the operation by which the operand in question increases its value by 1.

$$x \leftarrow x + 1$$

Since the increment operation reduces to an addition where the second operand has the value equal to 1, the algorithms, rules and overflow cases detailed in the description of the addition operation remain fully valid.

d. Decrement operation

Analogous to the increment operation, decrementation is defined as the operation by which the operand in question decreases its value by 1.

$$x \leftarrow x - 1$$

Since the decrement operation reduces to a subtraction where the second operand has the value equal to 1, the algorithms, rules and overflow cases detailed in the description of the subtraction operation remain fully valid.

e. Negation

The negation operation aims to obtain the opposite of a number.

This uses the properties of the 2's complement representation, which is based on the idea that the opposite of a number is obtained by complementing all the bits in the binary representation of the number and incrementing the resulting value. Basically, the 1's complement of the number whose opposite value is desired is obtained, and then the value of 1 is added to it, according to the following procedure:

$$\overline{x} = \overline{x}_{31}\ \overline{x}_{30} \ldots \overline{x}_2\ \overline{x}_1\ \overline{x}_0$$

$$-x = \overline{x} + 1$$

3.3.2. Analysis of logical bitwise operations

a. Logical AND

The logical AND operation (AND) is a logical operation that produces a result having the value 1-logical only when both bits representing the operands have the value 1-logical.

The bitwise AND operation is defined according to the following truth table:

AND

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

Figure 2. Logical AND Boolean truth table

The result of the AND operation with the operands represented on 32 bits is obtained by applying the AND operation on each pair of bits in the same position in the representation of the operands, as follows:

$$Y_i = X_{1i}\ and\ X_{2i}$$

b. Logical OR

The logical OR operation is defined as the operation that generates a logical 1 when at least one of the operand bits has the value 1-logical.

The truth table for the logical OR function is:

OR

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 3. Logical OR Boolean truth table

c. Logical NOT

The logical NOT operation is a logical operation having only one operand, whose value it negates, according to the following truth table:

NOT

x	x'
0	1
1	0

Figure 4. Logical NOT Boolean truth table

d. Logical Left Rotate

The Left-Rotate Logical operation performs a circular shift to the left of the component bits of the operand by one binary position. On the position of the least significant bit, freed by rotation, the most significant bit which comes out of the operand will arrive:

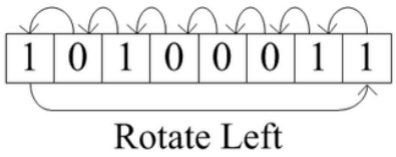

Figure 5. Logical Rotate Left schema

After performing a logical left rotation on a 32-bit number $y = x_{31}x_{30}x_{29} \ldots x_1x_0$, the result will become:

$$y = x_{30}x_{29} \ldots x_1x_0x_{31}$$

e. Logical Right Rotate

The Left-Rotate Logical operation performs a circular shift to the right of the component bits of the operand by one binary position.

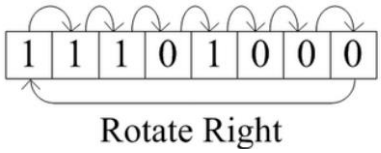

Figure 6. Logical Rotate Right schema

After performing a logical right rotation on a 32-bit number $y = x_{31}x_{30}x_{29} \ldots x_1x_0$, the result will become:

$$y = x_0x_{31} \ldots x_3x_2x_1$$

3.3.3. Analysis of complex arithmetic operations (multiplication and division)

Multiplication and division operations turn out to be the most complex operations performed at the level of the airthmetic logic unit, consisting of a sequence of additions and subtractions, depending on the case.

Thus, for their design and implementation, it is necessary to study and describe specific algorithms by implementing which the desired results can be obtained.

a. Multiplication operation

The multiplication operation is basically repeated addition, the simplest binary multiplication algorithm being **Shift-And-Add Multiplication**. This approach is based on adding the operand X1 to himself X2 times, where X1 represents the multiplicand, and X2 multiplier.

Basically, the strategy uses the same logic as decimal multiplication, and can be described by the following steps:

1. The digits of the multiplier are considered from left to right
2. Each digit of the multiplier is multiplied by each digit of the multiplicand
3. The intermediate result obtained is placed one position further to the left compared to the previous result obtained
4. The intermediate results obtained are added, respecting the indentation obtained by moving the successive intermediate results one position to the left.

In the case of binary multiplication, each step of the algorithm is simplified, since the only digits in the binary numbering system are 0 and 1. Therefore, if the considered digit of the multiplier is 1, the multiplicand will be copied exactly, with a position further to the left of the previously obtained result, and if the considered figure is 0, a number of 0s equal to the number of digits of the product will also be added further to the left.

The sequence of steps followed by the Shift-And-Add Multiplication algorithm can be found in the logic diagram below:

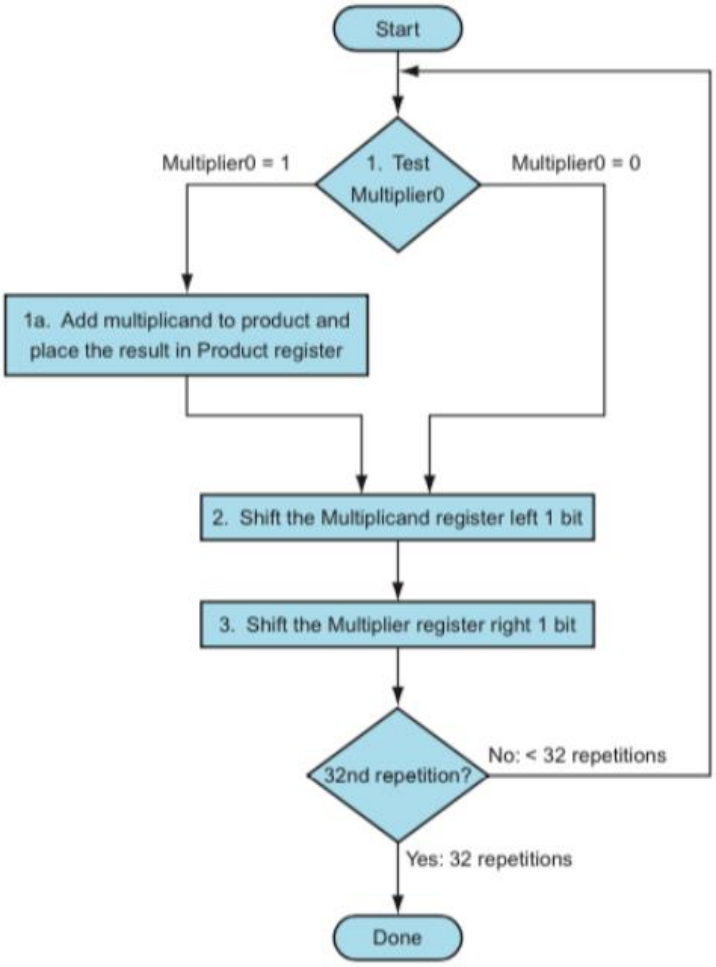

Figure 7. Shift-and-Add Multiplication Algorithm

b. Division operation

The division operation is by far the most cost-intensive and complex one.

In this implementation of the ALU, the algorithm used for binary division will be the **Restoring Division algorithm**. In each step of the algorithm there is a shift of the divisor to the right by one position and a shift of the quotient to the

left by one binary position, the algorithm ending when the divisor becomes less than the remainder of the division.

The basic idea of the algorithm is as follows:

- check if the divisor is less than the remainder
- subtract the divisor from the current value of the remainder
- if the result of the operation is negative, the next step is to restore the previous value by adding the divisor to the value of the remainder and shifting the quotient to the left by one position, so that the bit Q_0 will have the value 0.

The sequence of steps describing the algorithm are described in the logic diagram below:

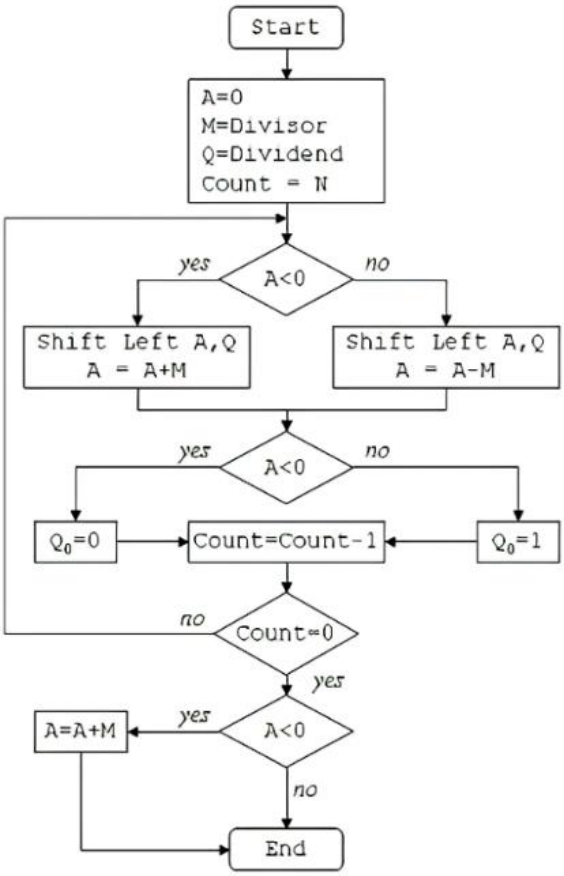

Figure 8. Restoring Division Algorithm

IV. Design

A block diagram containing the main components of the Arithmetic Logic Unit to be designed is presented below:

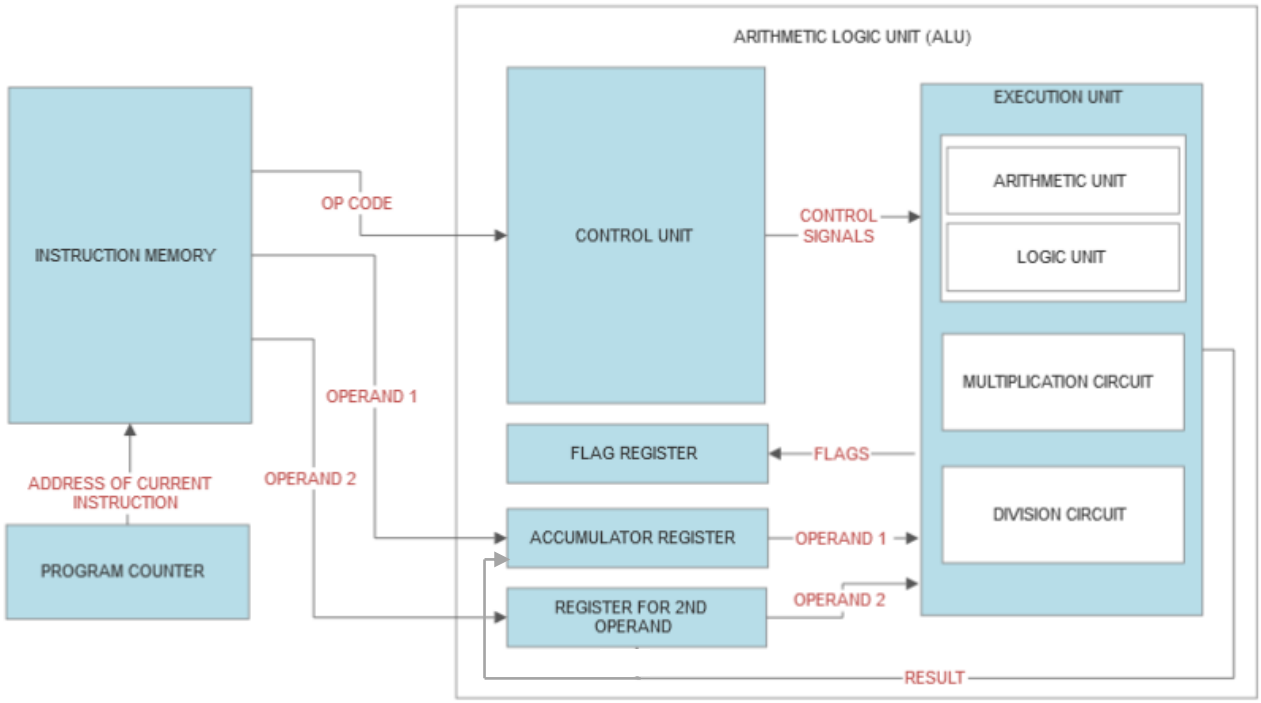

Figure 9. ALU Block Diagram

4.1. Components Presentation

1. **Accumulator Register** – is used for storing both the first 32-bit operand involved and the result of the operation that was chosen to be performed.
2. **Dedicated Register for the second operand** – as the name suggests, this register will store the value of the second 32-bit operand, when needed.
3. **Program Counter** – a counter that ensures an immediate transition from the current instruction in the instruction memory to the next instruction, indicating the location from which the memory read will be made
4. **Instruction Memory** – a ROM memory that stores hardcoded instructions that are to be tested by the ALU. An instruction requires two operand fields (each on 32 bits) and an op-code field, which represents the code of the operation to be performed by the ALU on the given operands.
5. **Control Unit (CU)** – is responsible with selecting the operation to be performed and storing this information in control signals that will then be transmitted to the Execution Unit. It also establishes the sequence of states through which the circuit passes in order to obtain the result of a given operation, thus requiring to be designed as a finite state automaton.

Each operation has a corresponding op-code, which is represented on 4 bits, since there are 12 operations available. Below is a table presenting the operation encodings:

OPERATION	OP CODE
ADD (Addition)	0000
SUB (Subtraction)	0001
INC (Increment)	0010
DEC (Decrement)	0011
AND	0100
OR	0101
NOT	0110
NEG (Negation)	0111
LR (Left Rotate)	1000
RR (Right Rotate)	1001
MUL (Multiplication)	1010
DIV (Division)	1011

6. **Execution Unit (EU)** – manages the logic behind how to perform all the operations in the list of functionalities. The EU, in turn, consists of four components, as follows:

6.1. Arithmetic Unit

- deals with the operations of arithmetic nature:
    a. ADD (Addition)
    b. SUB (Subtraction)
    c. INC (Incrementation)
    d. DEC (Decrementation)
    e. NEG (Negation)
- its design requires the usage of logic gates, multiplexers and adders

6.2. Logic Unit

- deals with the logical and positional operations:
    a. Logical AND
    b. Logical OR
    c. Logical NOT
    d. Left Rotate (LR)
    e. Right Rotate (RR)
- is defined at the most basic level, using interconnected logic gates in order to perform the operations presented above on 32-bit numbers

6.3. Multiplication Circuit – required for performing the multiplication operation (MUL)

6.4. Division Circuit – required for performing the division operation (DIV)

The EU takes the value of the first operand from the accumulator register and the value of the second operand from its dedicated register, performs the operation according to the control signals generated by the Control Unit, writes the result in the accumulator register and, using the flag register, sets flags in order to provide information about status of the obtained result, or in case any exceptions were recorded.

7. **Flag (Status) Register** – stores the values of the flags returned by the Execution Unit post-operation. The following flags can be generated:
   - Sign flag (S) – set to 1 when the result is a negative number
   - Overflow flag (O) – set to 1 when the result is out of range
   - Parity flag (P) – set to 1 when the result is an odd number
   - Zero flag (Z) – set to 1 when the result is 0
   - Divide By Zero flag (D) – (only availabled for the division operation) set to 1 when the divisor has the value 0
   - Auxiliary Carry flag (A) – set to 1 when registering a carry from rank 4 (the least significant 4 bits) to the next immediate rank
   - Carry flag (C) – set to 1 when obtaining a carry for the most significant rank within the result

Figure 10. Structure of the Flag Register

4.2. Functionalities in relation to Design

In this section of the documentation, I will describe how each operation will be implemented from a design standpoint. For each functionality, it will be described which components are needed and how to use them accordingly.

4.2.1. **Design of the Arithmetic Unit** (part of the **Execution Unit)**

1. Addition (ADD):

The addition of two 1-bit numbers will be done using a 1-bit Full Adder, composed of logic gates according to the equations described in section III (3.3.1). In order to perform the addition between two 32-bit numbers, **32 1-bit full adders** will be interconnected, formind a **Ripple Carry Adder**, where the carry-out ouput of the current rank will be connected to the carry-in input of the next rank.

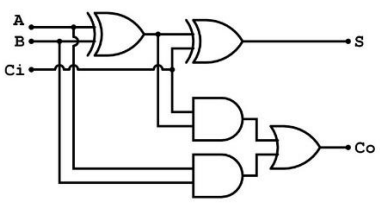

Figure 11. 1-bit Full Adder with logic gates

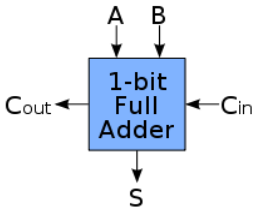

Figure 12. 1-bit Full Adder Block Diagram

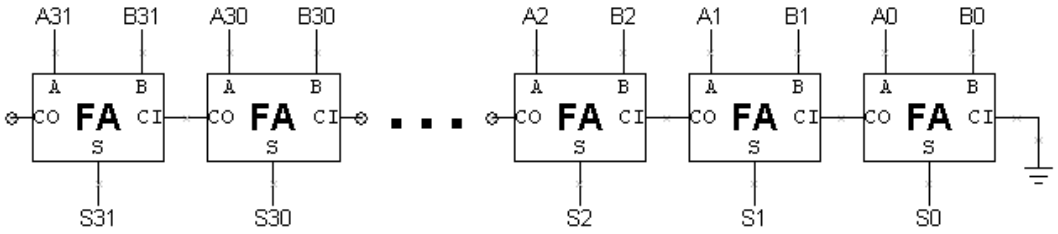

Figure 13. Ripple Carry Adder on 32 bits Block Diagram

2. Subtraction (SUB):

Performing the subtraction operation will require the usage of the same components as the addition, however with a few slight modifications.

The main difference between addition and subtraction consists of the fact that for subtraction, the second operand has to be negated first using two's complement. Only after negating it can we perform the addition between the two operands.

Therefore, from a design point of view, we will need **32 inverters** in order to negate each bit of the second operand (the subtrahend). Additionally, **32 2:1 multiplexers** will be needed in order to select how the second operator will be represented depending on whether addition or subtraction is performed. The selection of one multiplexer, **sel_sub** will be set to 1 when subtraction is performed, and to 0 in the case of addition.

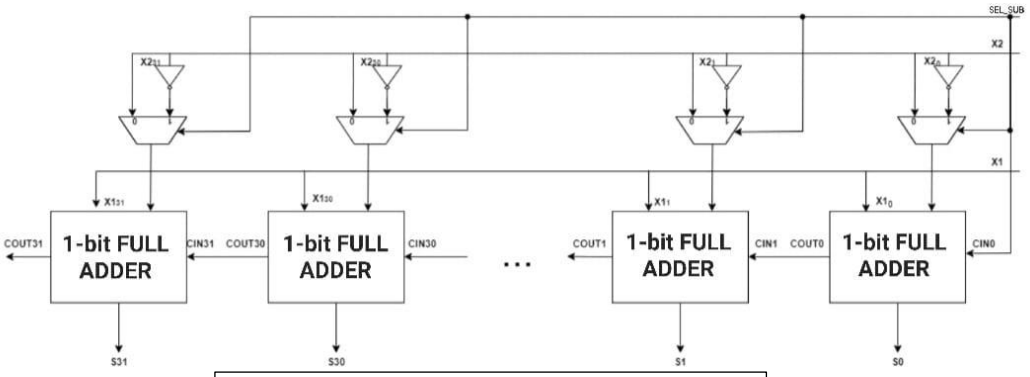

Figure 14. Ripple Carry Adder Subtractor on 32 bits Block

3. Incrementation (INC):

Since the increment operation basically represents an addition where the second operand is equal to 1, the logic will be the same as for the addition. What needs to be done in addition is to always have the value of the register dedicated to the second operand set to the value 1, represented on 32 bits, whenever performing an increment. We do this by using a **2:1 multiplexer**, whose selection will coincide with the control **signal LOAD_OP2_OR_1**. The selection will determine whether the value loaded into the second operand will be its normal value or „00000...01H".

4. Decrementation (DEC):

The decrement operation will be implemented analogous to the increment operation, with the exception that we perform a subtraction with the second operand having the value 1. Setting the value of the second operand will be done through the same

control signal used for the increment, but here the **sel_sub** signal will be also set to logical 1.

5. Negation (NEG):

Negating a number consists of negating all of its bits and adding 1 to the result. Since this operation requires only one operand, we will use the first one, and will have to apply a similar mechanism as the one applied on the second operand for subtraction. We will use once again **32 2:1 multiplexers** in order to select if the input of the adders will take the regular bit values of operand A, or will take their negated values. The dedicated selection for this process will be named **sel_neg** and will be set to 1 to signal that the negation operation is being performed. After negating all the bits of the given number, the LOAD_OP2_OR_1 control signal will also be used for the same purpose as described above, in order to perform the incrementation by 1 that the negation requires.

Finally, the resulting Arithmetic Unit will be:

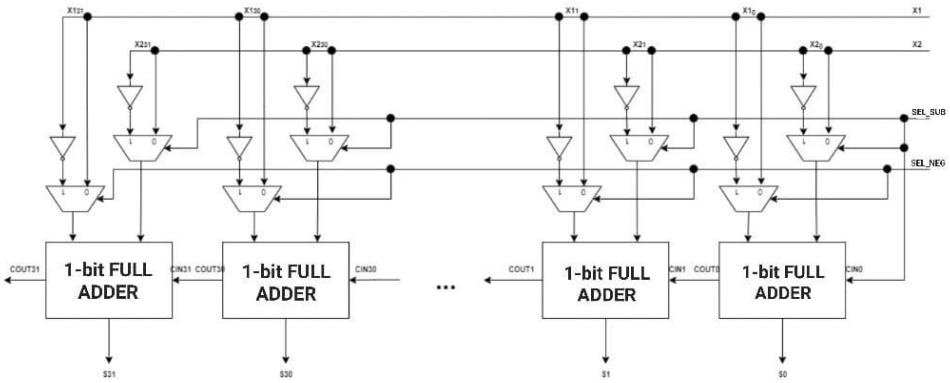

Figure 15. Arithmetic Unit on 32 bits Diagram

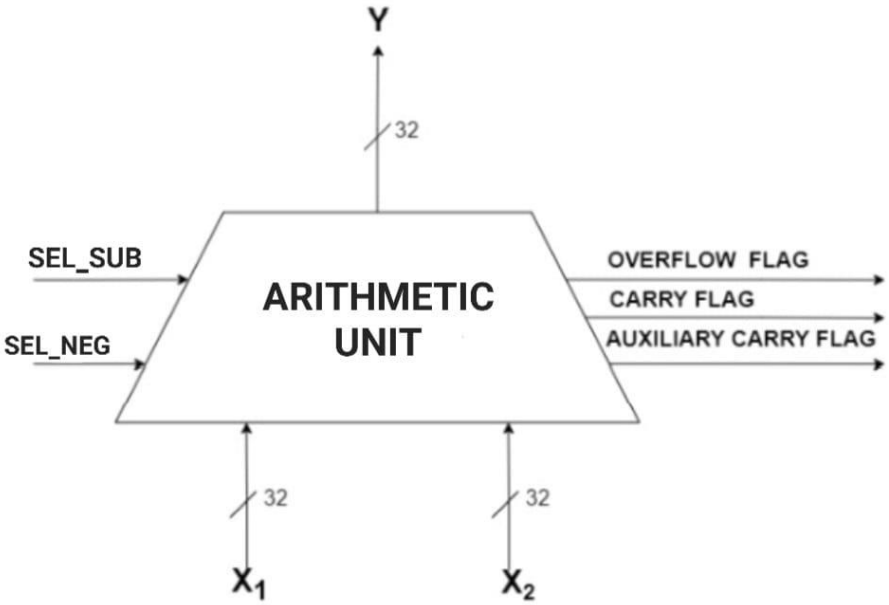

Figure 16. Arithmetic Unit on 32 bits Block Diagram

4.2.2. **Design of the Logic Unit** (part of the **Execution Unit**)

1. Logical AND – will be implemented according to the Analysis Section, by using **32 AND logic gates on 1 bit**, where the inputs will be the bits of the same rank of both operands.

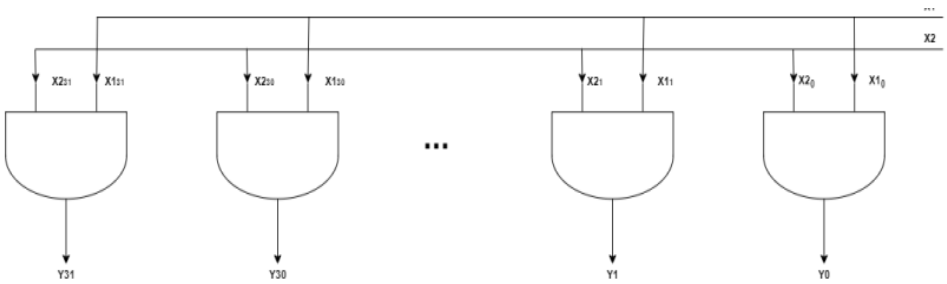

Figure 17. Design for the Logical AND operation on 32 bits

2. Logical OR - analogous to the AND operation, it will require **32 1-bit OR logic gates**.

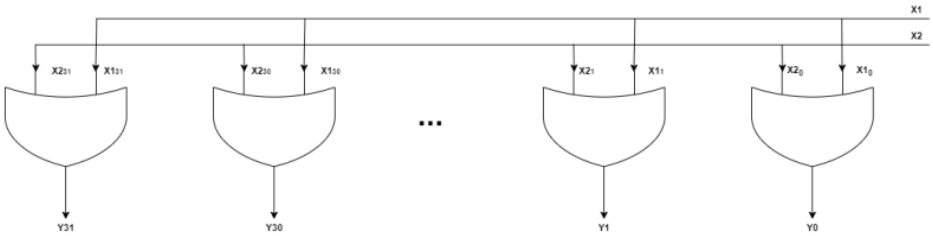

Figure 18. Design for the Logical OR operation on 32 bits

3. Logical NOT – it consists of negating all the bits of the first operand, therefore will require **32 1-bit inverters**.

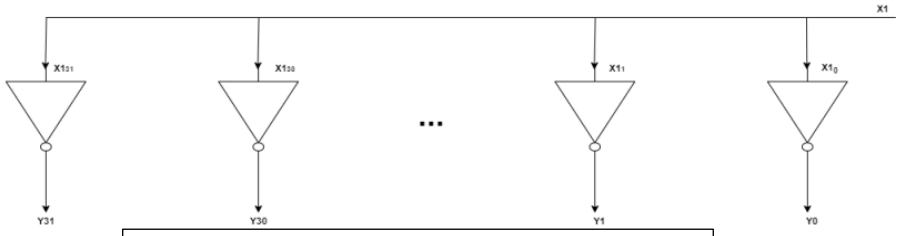

Figure 19. Design for the Logical NOT operation on 32 bits

4. Logical Left Rotate (LR)

The logical rotation to the left will be achieved by interconnecting **32 memory cells** represented by **D flip-flops**, whose serial outputs will be connected to the serial inputs of the next most significant flip-flop, except for the output of the initially most significant latch, which will be connected to the input of the flip-flop on the least significant position. We put the bits of the operand representation on the parallel inputs of the rotation circuit, and the selection between the serial and parallel inputs will be done by means of **32 2:1 multiplexers**, controlled by the LEFT signal. The LEFT signal is active on 0 when loading the bits of the operand in the memory cells. Thus, upon the arrival

of a clock pulse, if the signal LEFT is active on logical 1, the operand will be rotated one position to the left.

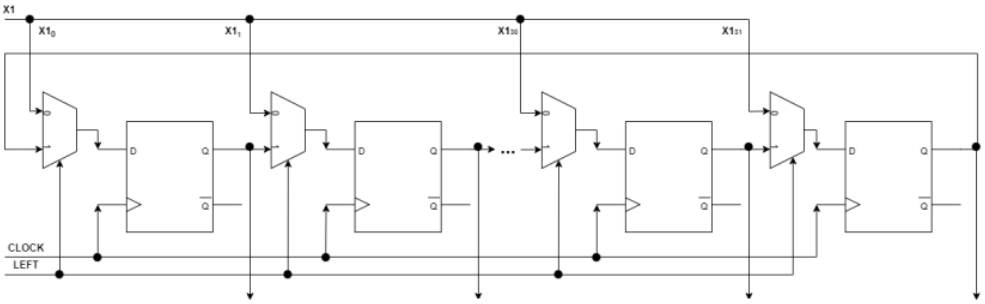

Figure 20. Logical Left Rotation Circuit

5. Logical Right Rotate (RR)

Logical right rotation follows a strategy similar to that of logical right rotation, except that the direction of movement will be reversed. Thus, the serial outputs will connect to the serial inputs of the flip-flops corresponding to the immediately less significant positions, and the output of the flip-flop for the least significant position will connect to the serial input of the flip-flop for the most significant position. Selection between serial and parallel inputs will be done using **32 2:1 multiplexers**, controlled by the signal RIGHT, the reasoning remaining the same as for the logical left rotation.

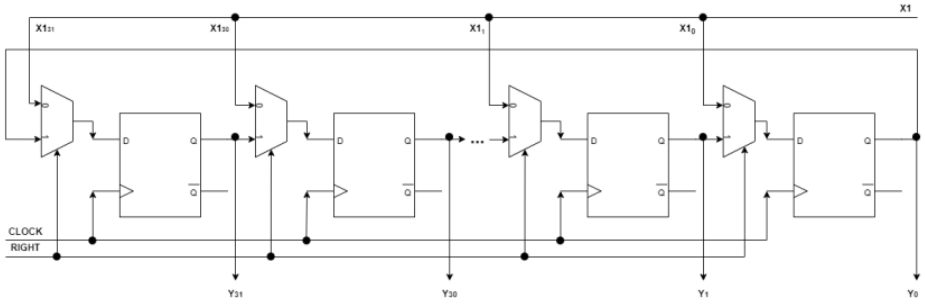

Figure 21. Logical Right Rotation Circuit

In the end, the design of the Logic Unit will look like this:

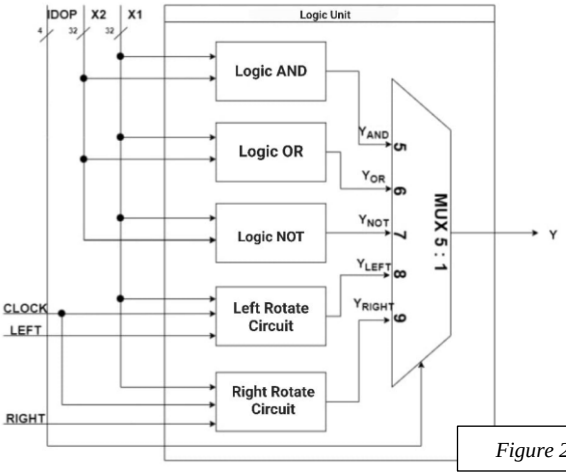

Figure 22. Logic Unit Design

4.2.3. **Design of the Multiplication Circuit** (part of the **Execution Unit**)

In order to figure out how to design and implement the multiplication circuit, we will follow the Shift-and-Add Multiplication method described in the Analysis Section and translate it into hardware language.

The result register RES (represented on 64 bits) is initially set to 0. The RES register is represented on 64 bits to be able to support the basic idea of the algorithm, which, at each step, moves to the left the content of the register D, also defined on 64 bits and intended for storing the divident. Initially X_1, representing the divident, will be stored in the lower half of the register D. The contents of register M, corresponding to the multiplier, will be initialized with the value X_2, and the counter N will be loaded with the value 32. The least significant bit of register M, M_0, will basically determine whether or not the value of the multiplicand will be added to the value stored in the RES register, corresponding to the product. Shifting the contents of the multiplier register to the left will have the effect of shifting the intermediate products to the left by one binary position, and shifting the contents of the D register to the right will allow each digit of the multiplicand to be considered from left to right, similar to the decimal multiplication algorithm.

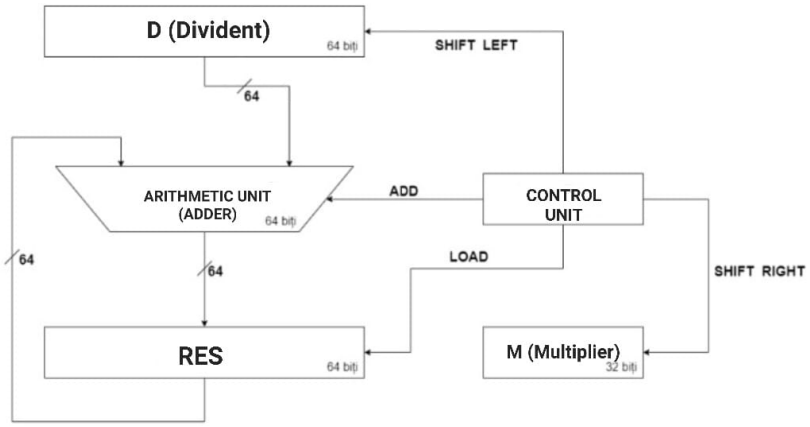

Figure 23. Multiplication Circuit Design

Since the chosen multiplication algorithm operates only on positive integers, we must use some additional negation circuits to ensure that the absolute values of the operands reach the multiplier and that the result of the division is a negative number, should the two terms of the product have different signs, as follows:

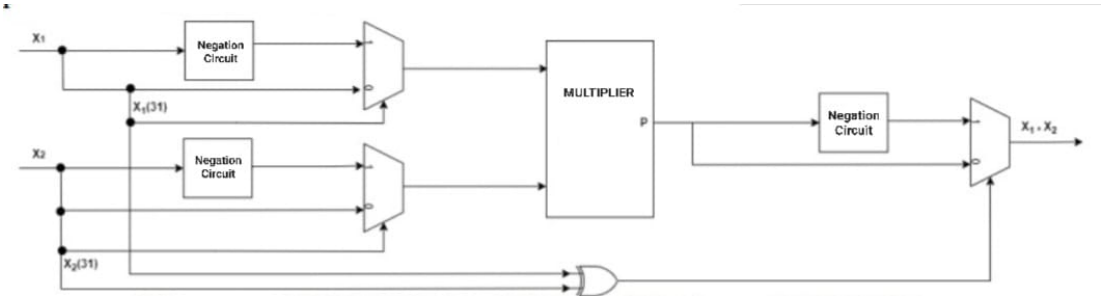

Figure 24. Multiplication Circuit with additional negation circuits

4.2.4. **Design of the Division Circuit** (part of the **Execution Unit**)

The implementation of the division algorithm Restoring Division requires the following design logics: initially, the dividend is loaded into the D register and the divider into the D_2 register, both defined on 64 bits. The 32-bit register Q, corresponding to the division quotient, will be set to 0 and the counter to the value 33. In order to determine whether or not the divider is less than the partial remainder, the contents of the D_2 register will be subtracted from the contents of the D register, where the partial remainders obtained are stored. If the result is negative, the next step is to restore the previous value by adding the value stored in D_2 to the value of the partial remainder and generating a 0 on the least significant position in register Q, Q_0. If the result of the subtraction operation is negative, a value of 1 will be generated on position Q_0. In the next step, the divider is shifted to the right by one binary position, thus aligning it with the contents of the register D for the next iteration of the algorithm.

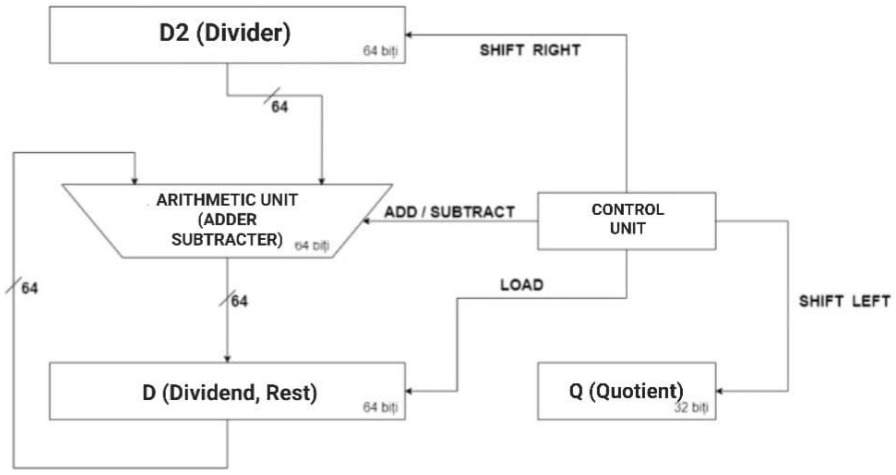

Figure 25. Division Circuit Design

Analogous to the problem encountered in the design of the multiplication circuit, in the case of the division operation it is also necessary to use some negation circuits to ensure that the operands are positive and that the results have the appropriate signs, the quotient being negative if the dividend and the divider have opposite signs , and the rest having the sign of the dividend:

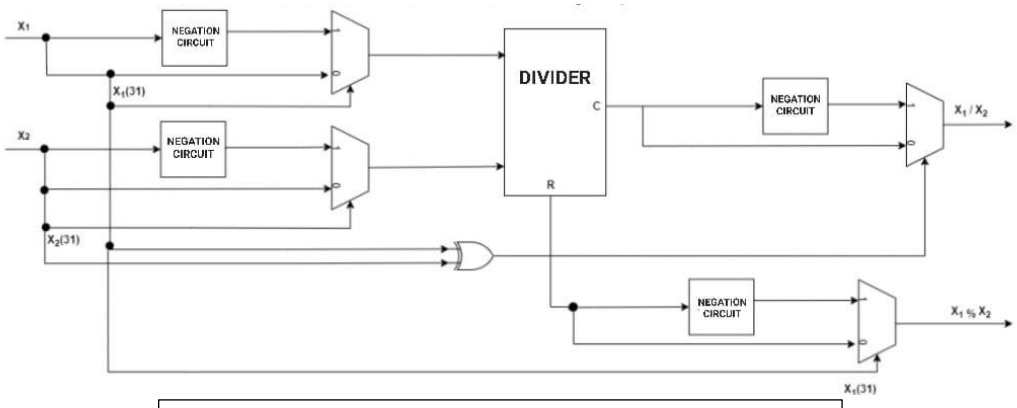

Figure 26. Division Circuit with additional negation circuits

V. Implementation

The implementation of the Arithmetic and Logic Unit respectively follow the description found in chapters III and IV of this documentation. The top level of both units along with their main components are implemented structurally, while the auxiliary circuits such as multiplexers, AND, OR, NOT gates are written following the behavioral method. Regardless, the implementation approach is described for each circuit in its architecture.

5.1. Implementation of the Arithmetic Unit

The Arithmetic Unit is responsible for performing the arithmetic operations of the ALU, that is: addition (ADD), subtraction (SUB), incrementation (INC) and decrementation (DEC), negation (NEG), and its entity is defined the following way:

```
5  -- 32 bit Arithmetic Unit: ADD, SUB, INC, DEC, NEG operations
6  entity arithmetic_unit is
7    Port (x1, x2: in std_logic_vector(31 downto 0);   -- two 32-bit operands
8    sel_sub, sel_neg: in std_logic;                   -- selectors for the Subtraction and Negation operations
9    y: out std_logic_vector(31 downto 0);             -- the 32-bit result
10   flags: out std_logic_vector(6 downto 0));         -- we have 7 possible flags in our Flag Register
11 end arithmetic_unit;
```

Figure 27. Arithmetic Unit Entity definition (Port List)

The Arithmetic Unit has three types of ports: input ports (x1, x2), control ports (sel_sub, sel_neg) and output ports (y, flags) and they serve the following purposes:'

- x1, x2 - represent the two 32-bit operands that the operations of the Arithmetic Unit will be performed on
- sel_sub - when it has the value logic ‚1', it indicates that the operation that is currently performed is the Subtraction (SUB) operation
- sel_neg - when it has the value logic ‚1', it indicates that the operation that is currently performed is the Negation (NEG) operation
- y – the result of the currently performed operation, represented on 32 bits
- flags – represents the flags of the Flag Register (described in the Design chapter) that are set after performing the operations

The architecture of the Arithmetic Unit, defined structurally, begins by defining the components that form the unit:

```
13 architecture Structural of arithmetic_unit is
14 -- components region
15 -- 1 bit Full Adder
16 component full_adder_1bit is
17   Port(a, b, cin: in std_logic;
18   result, cout: out std_logic);
19 end component;
20
21 -- 2:1 Multiplexer
22 component mux_2_1 is
23   Port (x0, x1, sel: in std_logic;
24   y: out std_logic);
25 end component;
26
27 -- 1 bit AND gate with 3 operands
28 component and3_gate is
29   Port (a, b, c: in std_logic;
30   res: out std_logic);
31 end component;
```

```
32
33 -- 1 bit OR gate
34 component or_gate is
35   Port (a, b: in std_logic;
36   res: out std_logic);
37 end component;
38
39 -- 1 bit NOT gate
40 component not_gate is
41   Port (a: in std_logic;
42   res: out std_logic);
43 end component;
```

Figures 28, 29. Arithmetic Unit Components

Following this, it contains a region where the necessary signals are defined:

```
-- signals region
signal mux_x1, mux_x2: std_logic_vector(31 downto 0) := x"00000000";
signal inv_x1, inv_x2: std_logic_vector(31 downto 0) := x"00000000";  -- the inverted values of the two operands
signal result_sig: std_logic_vector(31 downto 0) := x"00000000";
signal carry_sig: std_logic_vector(31 downto 0) := x"00000000";
-- flag signals
signal sign_flag, overflow_flag, parity_flag, zero_flag, div_by_zero_flag, aux_carry_flag, carry_flag, borrow_flag: std_logic := '0';
signal ovf, ovf_add, ovf_sub: std_logic := '0';
```

Figure 30. Arithmetic Unit architecture signals

Finally, the actual implementation of the unit is done by using a for... generate, in which, depending on the index of the unit block, the port maps corresponding to the necessary components are defined.

```
full_arithmetic_unit: for i in 0 to 31 generate

-- implementation of the first block of the Arithmetic Unit
block0: if i = 0 generate
not1: not_gate port map(x1(i), inv_x1(i));
not2: not_gate port map(x2(i), inv_x2(i));

mux_sub: mux_2_1 port map(
     x0 => x2(i),
     x1 => inv_x2(i),
     sel => sel_sub,
     y => mux_x2(i));

mux_neg: mux_2_1 port map(
     x0 => x1(i),
     x1 => inv_x1(i),
     sel => sel_neg,
     y => mux_x1(i));

adder: full_adder_1bit port map(
     a => mux_x1(i),
     b => mux_x2(i),
     cin => sel_sub,
     result => result_sig(i),
```

Figure 31.1. Arithmetic Unit Implementation with for…generate

```
83  -- implementation of the rest of the component blocks of the unit
84  other_blocks: if i > 0 generate
85  not1: not_gate port map(x1(i), inv_x1(i));
86  not2: not_gate port map(x2(i), inv_x2(i));
87
88  mux_sub: mux_2_1 port map(
89       x0 => x2(i),
90       x1 => inv_x2(i),
91       sel => sel_sub,
92       y => mux_x2(i));
93
94  mux_neg: mux_2_1 port map(
95       x0 => x1(i),
96       x1 => inv_x1(i),
97       sel => sel_neg,
98       y => mux_x1(i));
99
100 adder: full_adder_1bit port map(
101      a => mux_x1(i),
102      b => mux_x2(i),
103      cin => carry_sig(i - 1),
104      result => result_sig(i),
105      cout => carry_sig(i));
106 end generate other_blocks;
```

Figure 31.2. Arithmetic Unit Implementation with for…generate

In the end, we compute the flags that are set for this Unit, as well as the outputs:

```
111 -- generating the flags
112 sign_flag <= result_sig(31);
113
114 ovf_add <= (x1(31) and x2(31) and (not result_sig(31))) or ((not x1(31)) and (not x2(31)) and result_sig(31));
115 ovf_sub <= (x1(31) and (not x2(31)) and (not result_sig(31))) or ((not x1(31)) and x2(31) and result_sig(31));
116 ovf <= ovf_add when sel_sub = '0' else ovf_sub;
117 overflow_flag <= '0' when sel_neg = '1' else ovf;
118
119 parity_flag <= result_sig(0);
120
121 zero_flag_proc: process(result_sig)
122 begin
123 case result_sig is
124     when x"00000000" =>
125         zero_flag <= '1';
126     when others =>
127         zero_flag <= '0';
128 end case;
129 end process;
130
131 div_by_zero_flag <= '0';    -- because we don't perform division in this unit
132 aux_carry_flag <= carry_sig(3);
133
```

Figure 32.1. Arithmetic Unit Flag computation

```
133
134 -- if during the SUB op, the 2nd operand > 1st operand, we need to borrow 1,
135 -- thus activating the borrow flag
136 borrow_flag_proc: process(x1, x2)
137 begin
138 if signed(x1) < signed(x2) then
139     borrow_flag <= '1';
140 else
141     borrow_flag <= '0';
142 end if;
143 end process;
144
145 carry_flag <= carry_sig(31) when sel_sub = '0' else borrow_flag;
146
147
148 -- Computing Outputs
149 y <= result_sig;
150 flags <= sign_flag & overflow_flag & parity_flag & zero_flag & div_by_zero_flag & aux_carry_flag & carry_flag;
151
152 end Structural;
```

Figure 32.2. Arithmetic Unit Flag computation

5.2. Implementation of the Logic Unit

The Logic Unit is responsible for performing the logic operations of the ALU, that is: logical AND, logical OR, logical NOT, logical left rotation, logical right rotation, and its entity is defined the following way:

```
4  -- 32 bit Logic Unit: Logic AND, OR NOT, Left & Right rotations
5  entity logic_unit is
6      Port (clk: in std_logic;
7      x1, x2: in std_logic_vector(31 downto 0);
8      id_op: in std_logic_vector(3 downto 0);     -- operation ID
9      left, right: in std_logic;
10     y: out std_logic_vector(31 downto 0);
11     flags: out std_logic_vector(6 downto 0));
12 end logic_unit;
```

Figure 33. Logic Unit Entity definition (Port List)

The Logic Unit has three types of ports: input ports (clk, x1, x2), control ports (id_op, left, right) and output ports (y, flags) and they serve the following purposes:

- clk – input for the clock signals; it is required for the accumulator and dedicated registers
- x1, x2 - represent the two 32-bit operands that the operations of the Logic Unit will be performed on
- id_op - control signal that indicates what operation will be performed
- left – control signal that indicates that a logical left rotation is being performed when set to logic ‚1’
- right - control signal that indicates that a logical right rotation is being performed when set to logic ‚1’
- y – the result of the currently performed operation, represented on 32 bits

- flags – represents the flags of the Flag Register (described in the Design chapter) that are set after performing the operations

The architecture of the Logic Unit, defined structurally, begins by defining the of the unit:

```
13
14  architecture Structural of logic_unit is
15  -- components region
16  component logic_and_op is
17    Port (x0, x1: in std_logic_vector(31 downto 0);
18    y: out std_logic_vector(31 downto 0));
19  end component;
20
21  component logic_or_op is
22    Port (x0, x1: in std_logic_vector(31 downto 0);
23    y: out std_logic_vector(31 downto 0));
24  end component;
25
26  component logic_not_op is
27    Port (x: in std_logic_vector(31 downto 0);
28    y: out std_logic_vector(31 downto 0));
29  end component;
30
31  component left_rotation_circuit is
32    Port (clk: in std_logic;
33    x: in std_logic_vector(31 downto 0);
34    left: in std_logic;                    -- control signal that shows that a shift-left operation is being performed
35    y: out std_logic_vector(31 downto 0));
36  end component;
```

Figure 34.1. Logic Unit Components

```
37
38  component right_rotation_circuit is
39    Port (clk: in std_logic;
40    x: in std_logic_vector(31 downto 0);
41    right: in std_logic;                   -- control signal that shows that a shift-right operation is being performed
42    y: out std_logic_vector(31 downto 0));
43  end component;
44
45  -- 32-bit 5:1 Multiplexer
46  component mux_5_1 is
47    Port (x0, x1, x2, x3, x4: in std_logic_vector(31 downto 0);
48    sel: in std_logic_vector(3 downto 0);
49    y: out std_logic_vector(31 downto 0));
50  end component;
51
52  -- signals region
53  signal y_and, y_or, y_not, y_rot_left, y_rot_right: std_logic_vector(31 downto 0) := x"00000000";
54  signal result_sig: std_logic_vector(31 downto 0) := x"00000000";
55  signal sign_flag, overflow_flag, parity_flag, zero_flag, div_by_zero_flag, aux_carry_flag, carry_flag: std_logic := '0';
```

Figure 34.2. Logic Unit Components & Signals

Following, is the actual implementation of the Logic Unit, containing the port mapping of the components, flag and output computations:

```
begin
logic_and_circuit: logic_and_op port map (x1, x2, y => y_and);
logic_or_circuit: logic_or_op port map (x1, x2, y_or);
logic_not_circuit: logic_not_op port map (x1, y_not);
left_rot_circuit: left_rotation_circuit port map (clk, x1, left, y_rot_left);
right_rot_circuit: right_rotation_circuit port map (clk, x1, right, y_rot_right);
mux5_1: mux_5_1 port map(y_and, y_or, y_not, y_rot_left, y_rot_right, id_op, result_sig);

-- flags computation
sign_flag <= result_sig(31);
overflow_flag <= '0';
parity_flag <= result_sig(0);

zero_flag_proc: process(result_sig)
begin
case result_sig is
    when x"00000000" =>
        zero_flag <= '1';
    when others =>
        zero_flag <= '0';
end case;
end process;
```

Figure 35.1. Logic Unit Implementation

```

div_by_zero_flag <= '0';
aux_carry_flag <= '0';
carry_flag <= '0';

-- output computation
y <= result_sig;
flags <= sign_flag & overflow_flag & parity_flag & zero_flag & div_by_zero_flag & aux_carry_flag & carry_flag;

end Structural;
```

Figure 35.2. Logic Unit Implementation

5.3. Implementation of the Multiplication Circuit

The implementation of the multiplication circuit follows the Shift-and-Add Multiplication algorithm for 32-bit positive binary numbers. The entity of the circuit is defined the following way:

```

entity multiplication_circuit is
    Port (clk: in std_logic;
          clr: in std_logic;  -- clear signal for resetting the operand registers
          x1, x2: in std_logic_vector(31 downto 0);
          clr_res_mul: in std_logic;  -- reset for the mult. result register
          ld_res_mul: in std_logic;   -- parallel load for the result register
          ld_m_mul: in std_logic;     -- parallel load for the multiplier register
          ld_d_mul: in std_logic;     -- parallel load for the multiplicand register
          right_shift_m_mul: in std_logic;    -- right shift signal for shifting the value of the multiplier reg.
          left_shift_d_mul: in std_logic;     -- --||-- of the multiplicand reg.
          y: out std_logic_vector(63 downto 0);
          i0: out std_logic;  -- stores the least significant bit of the current value of the multiplier
          flags: out std_logic_vector(6 downto 0));
end multiplication_circuit;

```

Figure 36. Multiplication Circuit Ports

Following is the architecture of the multiplication circuit, implemented structurally, with the design presented in the previous chapter in mind:

```
19  architecture Structural of multiplication_circuit is
20  -- components region
21  -- Adder on 64 bits
22  component adder_64bits is
23      Port (x1, x2: in std_logic_vector(63 downto 0);
24            sub: in std_logic;
25            cout: out std_logic;
26            y: out std_logic_vector(63 downto 0));
27  end component;
28
29  -- Multiplication register, made generic
30  component register_n_bits is
31      Generic (N : integer := 32);
32      Port (data_in : in std_logic_vector(n - 1 downto 0);        -- parallel input
33            left : in std_logic;                                  -- shift left enable
34            right : in std_logic;                                 -- shift right enable
35            load : in std_logic;                                  -- load enable
36            clr : in std_logic;                                   -- reset
37            serial_in : in std_logic;                             -- serial input
38            clk : in std_logic;
39            data_out : out std_logic_vector(n - 1 downto 0));
40  end component;
41
```

Figure 37.1. Multiplication Circuit Implementation

```
41
42  signal res: std_logic_vector(63 downto 0) := (others => '0');
43  signal ext_x1: std_logic_vector(63 downto 0) := (others => '0');
44  signal multiplicand: std_logic_vector(63 downto 0) := (others => '0');
45  signal multiplier: std_logic_vector(31 downto 0) := (others => '0');
46  signal sum: std_logic_vector(63 downto 0) := (others => '0');
47  signal cout: std_logic := '0';
48  signal carry_flag, aux_carry_flag, overflow_flag, zero_flag, parity_flag, sign_flag, div_by_zero_flag: std_logic := '0';
49
50  begin
51      -- port mapping components
52      res_register: register_n_bits generic map (N => 64) port map (data_in => sum, left => '0', right => '0', load => ld_res_mul,
53      clr => clr_res_mul, serial_in => '0', clk => clk, data_out => res);
54      ext_x1 <= x"00000000" & x1;
55      multiplicand_register: register_n_bits generic map (N => 64) port map (data_in => ext_x1, left => left_shift_d_mul, right => '0',
56      load => ld_d_mul, clr => clr, serial_in => '0', clk => clk, data_out => multiplicand);
57      multiplier_register: register_n_bits generic map (N => 32) port map (data_in => x2, left => '0', right => right_shift_m_mul,
58      load => ld_m_mul, clr => clr, serial_in => '0', clk => clk, data_out => multiplier);
59      adder: adder_64bits port map (x1 => res, x2 => multiplicand, sub => '0', cout => cout, y => sum);
60      i0 <= multiplier(0);
61
```

Figure 37.2. Multiplication Circuit Implementation

```
    -- flags computation
    overflow_flag_proc: process (res)
    begin
        if res(63 downto 32) = x"00000000" then
            overflow_flag <= '0';
        else
            overflow_flag <= '1';
        end if;
    end process overflow_flag_proc;

    carry_flag <= '0';
    aux_carry_flag <= '0';
    parity_flag <= res(0);
    sign_flag <= x1(31) xor x2(31);
    div_by_zero_flag <= '0';

    zero_flag_proc: process (res)
    begin
        if res = x"0000000000000000" then
            zero_flag <= '1';
        else
            zero_flag <= '0';
        end if;
    end process zero_flag_proc;

    y <= res;
    flags <= sign_flag & overflow_flag & parity_flag & zero_flag & div_by_zero_flag & aux_carry_flag & carry_flag;

end Structural;
```

Figure 37.3. Multiplication Circuit Implementation

5.4. Implementation of the Division Circuit

The Division circuit is also implemented structurally, following the Restoring Division algorithm presented in the Analysis chapter. Below I will attach pictures of the port structure, along with the functionality explained correspondingly.

```
entity division_circuit is
    Port (clk: in std_logic;
          clr: in std_logic;
          x1, x2: in std_logic_vector(31 downto 0);
          clr_c_div: in std_logic;     -- reset for the dividend register
          ld_d_div: in std_logic;      -- parallel load of the divident reg.
          sub_div: in std_logic;       -- control signal that indicates the usage of the subtractor
          ld_d2_div: in std_logic;      -- parallel load of the divider reg.
          q0: in std_logic;        -- least significant bit of the quotient
          right_shift_d2_div: in std_logic;     -- right shift circuit for the value of the divider
          left_shift_q_div: in std_logic;      -- --||-- of the quotient
          y: out std_logic_vector(63 downto 0);
          flags: out std_logic_vector(6 downto 0);
          dn: out std_logic); -- sign bit of the multiplicand
end division_circuit;
```

Figure 38. Division Circuit Ports

The division circuit requires the inclusion of the following components: an adder-subtractor on 64 bits, a generic register on n bits (used for storing the divider, divisor, quotient and remainder), and a generic 2:1 multiplexer for n-bit inputs.

```
19
20  architecture Structural of division_circuit is
21  -- components region
22  -- Adder on 64 bits
23  component adder_64bits is
24      Port (x1, x2: in std_logic_vector(63 downto 0);
25            sub: in std_logic;
26            cout: out std_logic;
27            y: out std_logic_vector(63 downto 0));
28  end component;
29
30  component register_n_bits is
31      Generic (N : integer := 32);
32      Port (data_in : in std_logic_vector(n - 1 downto 0);        -- parallel input
33            left : in std_logic;                                   -- shift left enable
34            right : in std_logic;                                  -- shift right enable
35            load : in std_logic;                                   -- load enable
36            clr : in std_logic;                                    -- reset
37            serial_in : in std_logic;                              -- serial input
38            clk : in std_logic;
39            data_out : out std_logic_vector(n - 1 downto 0));
40  end component;
41
42  -- 2 to 1 Multiplexer on n bits
43  component mux_2_1_n_bits is
44      Generic (n: integer := 32);
45      Port (x1: in std_logic_vector(n - 1 downto 0);
46            x2: in std_logic_vector(n - 1 downto 0);
47            sel: in std_logic;
48            y: out std_logic_vector(n - 1 downto 0));
49  end component;
```

Figure 39.1. Division Circuit Implementation

```
51  -- signals region
52  signal dividend : std_logic_vector(63 downto 0) := (others => '0');
53  signal sum_out : std_logic_vector(63 downto 0) := (others => '0');
54  signal divisor : std_logic_vector(63 downto 0) := (others => '0');
55  signal mux_out : std_logic_vector(63 downto 0) := (others => '0');
56  signal ext_x1, ext_x2 : std_logic_vector(63 downto 0) := (others => '0');
57  signal q, r : std_logic_vector(31 downto 0) := (others => '0');
58  signal cout : std_logic := '0';
59  signal carry_flag, aux_carry_flag, overflow_flag, zero_flag, parity_flag, sign_flag, div_by_zero_flag : std_logic := '0';
60
61  begin
62
63      ext_x1 <= x"00000000" & x1;
64      ext_x2 <= x2 & x"00000000";
65
66      mux: mux_2_1_n_bits generic map (n => 64) port map (x1 => sum_out, x2 => ext_x1, sel => ld_d2_div, y => mux_out);
67
68      d2_register: register_n_bits generic map (N => 64) port map (data_in => ext_x2, left => '0', right => right_shift_d2_div,
69      load => ld_d2_div, clr => clr, serial_in => '0', clk => clk, data_out => divisor);
70
71      d_register: register_n_bits generic map (N => 64) port map (data_in => mux_out, left => '0', right => '0', load => ld_d_div,
72      clr => clr, serial_in => '0', clk => clk, data_out => dividend);
73
74      q_register: register_n_bits generic map (N => 32) port map (data_in => x"00000000", left => left_shift_q_div, right => '0',
75      load => '0', clr => clr_c_div, serial_in => q0, clk => clk, data_out => q);
76
77      adder: adder_64bits port map (x1 => dividend, x2 => divisor, sub => sub_div, cout => cout, y => sum_out);
78
79      dn <= dividend(63);
80      r <= dividend(31 downto 0);
81      y <= q & r;
```

Figure 39.2. Division Circuit Implementation

```
    r <= dividend(31 downto 0);
    y <= q & r;

    carry_flag <= '0';
    aux_carry_flag <= '0';
    parity_flag <= q(0);
    sign_flag <= q(31);
    overflow_flag <= '0';

    divide_by_zero_proc: process (x2)
    begin
        if x2 = x"00000000" then
            div_by_zero_flag <= '1';
        else
            div_by_zero_flag <= '0';
        end if;
    end process divide_by_zero_proc;

    zero_flag_proc: process (q)
    begin
        if q = x"00000000" then
            zero_flag <= '1';
        else
            zero_flag <= '0';
        end if;
    end process zero_flag_proc;

    flags <= sign_flag & overflow_flag & parity_flag & zero_flag & div_by_zero_flag & aux_carry_flag & carry_flag;

end Structural;

```

Figure 39.3. Division Circuit Implementation

5.5. Implementation of the Execution Unit

The Execution Units gathers together all the components described in the first four points of this chapter, thus containing the Arithmetic and Logic Units, the Multiplication and Division Circuits. The execution unit receives the two operands represented on 32 bits as input, and provides the result of the currently selected operation (given by the signals that result from the Control Unit and also represent inputs of the EU) in the form of a 64-bit number represented in Two's Complement, along with the flags that were activated accordingly.

```
entity execution_unit is
    Port (x1, x2: in std_logic_vector(31 downto 0);
          clk: in std_logic;
          clr: in std_logic;
          res_sel: in std_logic_vector(1 downto 0);
          idop: in std_logic_vector(3 downto 0);    -- operation id
          sub: in std_logic;
          neg: in std_logic;
          left: in std_logic;
          right: in std_logic;
          -- control signals for the fsm_ctrl multiplier
          clr_res_mul: in std_logic;
          ld_res_mul: in std_logic;
          ld_m_mul: in std_logic;
          ld_d_mul: in std_logic;
          right_shift_m_mul: in std_logic;
          left_shift_d_mul: in std_logic;
          -- control signals for the fsm_ctrl divider
          clr_c_div: in std_logic;
          ld_d_div: in std_logic;
          sub_div: in std_logic;
          ld_d2_div: in std_logic;
          q0: in std_logic;
          right_shift_d2_div: in std_logic;
          left_shift_c_div: in std_logic;
          dn: out std_logic;
          i0: out std_logic;
          flags: out std_logic_vector(6 downto 0);
          y: out std_logic_vector(63 downto 0));  -- result on 64 bits, represented in 2's complement
end execution_unit;

```

Figure 40. Execution Unit Ports

The EU also requires the use of sign extension and zero extension circuits in order to extend the results provided by the Arithmetic and Logic Units represented on 32 bits to 64 bits, since the output signal y will be represented on 64 bits (to fit the results given by the multiplication and division circuits respectively).

Additionally, a complement circuit is needed to transform any negative numbers into positives, since the multiplication and division algorithms are tailored to operate on positive numbers only.

```
102
103  -- additional circuits
104  component complement_circuit is
105      Generic (n: integer := 32);
106      Port (x: in std_logic_vector(n - 1 downto 0);
107            y: out std_logic_vector(n - 1 downto 0));
108  end component;
109
110  -- used for extending the results of the arithmetic/logic units from 32 bits to 64
111  component sign_extend_circuit is
112      Generic (initial_size : integer := 32;
113               extended_size : integer := 64);
114      Port (data_in: in std_logic_vector(initial_size - 1 downto 0);
115            extended_data: out std_logic_vector(extended_size - 1 downto 0));
116  end component;
117
118  component zero_extend_circuit is
119      Port (data_in: in std_logic_vector(31 downto 0);
120            extended_data: out std_logic_vector(63 downto 0));
121  end component;
```

Figure 41. Additional circuits used in the EU

5.6. Implementation of the Control Unit

The Control Unit has an extensive list of ports, including a set of control signals for the arithmetic and logic operations, one set for the multiplication operation, and one for the division operation.

```
4   -- ALU Control Unit
5   entity control_unit is
6       Port (idop: in std_logic_vector(3 downto 0);
7             clk: in std_logic;
8             start: in std_logic;
9             clr: in std_logic;
10            clr_div: in std_logic;
11            clr_mul: in std_logic;
12            i0: in std_logic; -- pentru inmultire (inmultitor(0))
13            dn: in std_logic; -- pentru impartite (deimpartit(31))
14            zero: in std_logic;
15            -- control signals for arithmetic and logical operations
16            sub: out std_logic;
17            neg: out std_logic;
18            left: out std_logic;
19            right: out std_logic;
20            load_acc: out std_logic;
21            load_res_or_op: out std_logic;
22            load_rop2: out std_logic;
23            load_op2_or_1: out std_logic;
24            load_flags: out std_logic;
25            stop: out std_logic;
26            -- control signals for the fsm_ctrl multiplier
27            clr_res_mul: out std_logic;
28            ld_res_mul: out std_logic;
29            ld_m_mul: out std_logic;
30            ld_d_mul: out std_logic;
31            right_shift_m_mul: out std_logic;
32            left_shift_d_mul: out std_logic;
33            -- control signals for the fsm_ctrl divider
34            clr_c_div: out std_logic;
```

Figure 42.1. Control Unit Ports

```
32            left_shift_d_mul: out std_logic;
33            -- control signals for the fsm_ctrl divider
34            clr_c_div: out std_logic;
35            ld_d_div: out std_logic;
36            sub_div: out std_logic;
37            ld_d2_div: out std_logic;
38            q0: out std_logic;
39            right_shift_d2_div: out std_logic;
40            left_shift_c_div: out std_logic;
41            res_sel: out std_logic_vector(1 downto 0));
42    end control_unit;
43
```

Figure 42.2. Control Unit Ports

The Control Unit introduces a new enumerated type *state_type*, which defines all the states that the ALU cycles through when running. A preview of this type would be the following:

```
46  type state_type is (
47      start_state,
48      load_op_add,
49      add_ex,
50      load_op_sub,
51      sub_ex,
52      load_op_inc,
53      inc_ex,
54      load_op_dec,
55      dec_ex,
56      load_op_neg,
57      neg_ex,
58      load_op_and,
59      and_ex,
60      load_op_or,
61      or_ex,
62      load_op_not,
63      not_ex,
64      load_op_rot_left,
65      init_rot_left,
66      rot_left_ex,
67      load_op_rot_right,
68      init_rot_right,
69      rot_right_ex,
70      load_res_alu,
71      load_op_mul,
72      init mul,
```

Figure 43. Control Unit enumerated type

The architecture of the Control Unit is behavioral and it contains two main processes: one that generates the next state of the ALU depending on the current state and on the inputs, at times, and another one that generates the control signals corresponding to each state. The generated control signals are then sent to the Execution Unit as inputs.

5.7. Implementation of the additional components, useful during testing

As can be seen in ***Figure 9*** (*see* ***Design*** *chapter*), there are two components outside of the Arithmetic Logic Unit designated block, those being the Instruction Memory and Program Counter, defined the following way:

```
18  component program_counter is
19    Port (clk, en, clear: in std_logic;
20    addr: out std_logic_vector(31 downto 0));
21  end component;
22
23  component instruction_memory is
24      Port ( address: in std_logic_vector(31 downto 0); -- the current address(the index of the current instruction)
25             data_out: out std_logic_vector(67 downto 0));  -- current instruction, defined on 68 bits
26  end component;
27
```

Figure 44. Additional components ports (PC, ROM)

Those two components are useful during the testing process of the ALU, since the simulation will be conducted in such a manner that the operands and the operations to be performed on them will be stored in the memory, making it easy for the user to switch through results with the press of a button.

The Instruction Memory is a 256x8 ROM memory containing hardcoded data necessary for testing. An instruction contains the hexadecimal values (signed in Two's Complement) of the two operands and a single-digit number representing the ID of the operation to be perfomed. The format of an instruction stored in the memory is presented below:

```
.6  signal rom: rom_array := (
.7      x"0000000A_00000010_0",              -- add (10, 16) = 26 = 0000001Ah
```

Figure 45. ROM instruction example

Here, for instance, the first operand takes the hexadecimal value A (which is 10), the second operand take the hexadecimal value 10 (which is 16). The operation has the ID 0 (all the operation encodings have been presented in the *Design* chapter), which we know represents the addition operation.

The Program Counter gives the addres of the current instruction from the ROM memory described above.

5.8.Implementation of the Top Level Module

The top level module contains all the components presented above, is implemented in a structural manner, and its structure is the following:

```
entity alu_top is
    Port ( next_instr_en: in std_logic;
           clk: in std_logic;
           clr: in std_logic;
           reset: in std_logic;
           start: in std_logic;
           stop: out std_logic;
           result_high: out std_logic_vector(31 downto 0);
           result_low: out std_logic_vector(31 downto 0);
           flags: out std_logic_vector(6 downto 0));
end alu_top;
```

Figure 46. ALU top level ports

Port description:

- *next_instr_en*: an input that enables the transition to the next instruction in the instruction memory
- *clr*: clears the register values for the operands
- *reset*: resets the program counter so that it points to the first instruction in the instruction memory
- *start*: signals that the current instruction has started its execution
- *stop*: signals that the current instruction has been completed
- *result_high*: output representing the most significant 32 bits of the result
- *result_low*: output representing the least significant 32 bits of the result
- *flags*: output of the flag register

5.9. Implementation of additional hardware components

In order to be able to test the ALU on the Basys3 FPGA Board, it was necessary to implement two auxiliary components with the purpose of ensuring the proper functioning of the board so that the user can visualize the generated results.

Therefore, the needed components are a synchronous monopulse generator, which allows the single activation of a signal at the press of the button associated with it and a seven-segment display, which helps view the data flow and the results obtained by the ALU.

```
26
27  component mpg is
28    Port (clk: in std_logic;
29          btn: in std_logic;
30          en: out std_logic);
31  end component;
32
33  component ssd is
34      Port ( digit0 : in STD_LOGIC_VECTOR (3 downto 0);
35             digit1 : in STD_LOGIC_VECTOR (3 downto 0);
36             digit2 : in STD_LOGIC_VECTOR (3 downto 0);
37             digit3 : in STD_LOGIC_VECTOR (3 downto 0);
38             clk : in STD_LOGIC;
39             cat : out STD_LOGIC_VECTOR (6 downto 0);
40             an : out STD_LOGIC_VECTOR (3 downto 0));
41  end component;
```

Figure 47. Additional components ports (MPG, SSD)

VI. Simulation and Tests

As explained in the previous chapter in section 5.7, the tests that will be run on the board/in the testbench are the set of instructions stored in the ROM memory. The tests were selected so that they would showcase a wide variety of output situations, thus aiming to prove the correctitude of the ALU implementation.

The test cases, along with their corresponding results are presented in the following table:

1st operand		2nd operand		Operation	Result		Flags
Hex	Dec	Hex	Dec		Hex	Dec	
A	10	10	16	ADD	1A	26	-
6	6	A	10	ADD	10	16	A
FFFFFFFA	-6	FFFFFFF0	-16	ADD	FFFFFFEA	-22	S,C
7FFFFFFF	2147483647	6FFFFFFF	187904191	ADD	EFFFFFFE	-268435458	S,O,A
A	10	10	16	SUB	FFFFFFFA	-6	S,A,C
A	10	12	18	SUB	FFFFFFF8	-8	S,A,C
80000000	-2147483647	2	2	SUB	7FFFFFFE	2147483646	O,A
A	10	-	-	INC	B	11	P
80000002	-2147483646	-	-	INC	80000003	-2147483645	S,P
FFFFFFF8	-8	-	-	DEC	FFFFFFF7	-9	S,P,A,C
80000000	-2147483648	-	-	DEC	7FFFFFFF	2147483647	O,P,C
A	10	-	-	NEG	FFFFFFF6	-10	S
FFFFFFF6	-10	-	-	NEG	A	10	-
A	10	9	9	AND	8	8	-
ABC500F1	2881814769	A536221	173236769	AND	A410021	172032033	P
A	10	9	9	OR	B	11	P
ABC500F1	2881814769	A536221	173236769	OR	ABD762F1	2883019505	S,P
A	10	-	-	NOT	FFFFFFF5	-11	S,P
FFFFFFF5	-11	-	-	NOT	A	10	-
80000000	2147483648	-	-	LR	1	1	P
B	11	-	-	LR	16	22	-
10	16	-	-	RR	8	8	-
B	11	-	-	RR	80000005	2147483653	S,P
A	10	6	6	MUL	3C	60	-
FFFFFFF6	-10	6	6	MUL	FFFFFFC4	-60	-
A	10	FFFFFFFA	-6	MUL	FFFFFFC4	-60	-
FFFFFFF6	-10	FFFFFFFA	-6	MUL	3C	60	-
A	10	0	0	MUL	0	0	Z

7FFFFFFF	2147483647	3	3	MUL	7FFFFFFD	2147483645	O,P
C	12	6	6	DIV	2 00000000	Q = 2 R = 0	Z
C	12	5	5	DIV	2 00000002	Q = 2 R = 2	P
FFFFFFF4	-12	5	5	DIV	FFFFFFFE FFFFFFFE	Q = -2 R = -2	P
C	12	FFFFFFFB	-5	DIV	FFFFFFFE 00000002	Q = -2 R = 2	P
FFFFFFF4	-12	FFFFFFFB	-5	DIV	2 FFFFFFFE	Q = 2 R = -2	P
C	12	0	0	DIV	0 (division by 0)	0 (division by 0)	P,D

Table 1. Simulation test cases table

6.1. Testbench Simulation Results

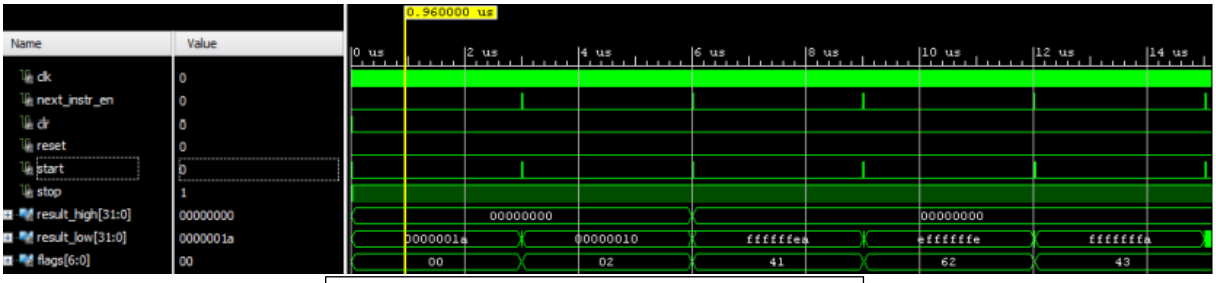

Figure 48.1. Testbench simulation results

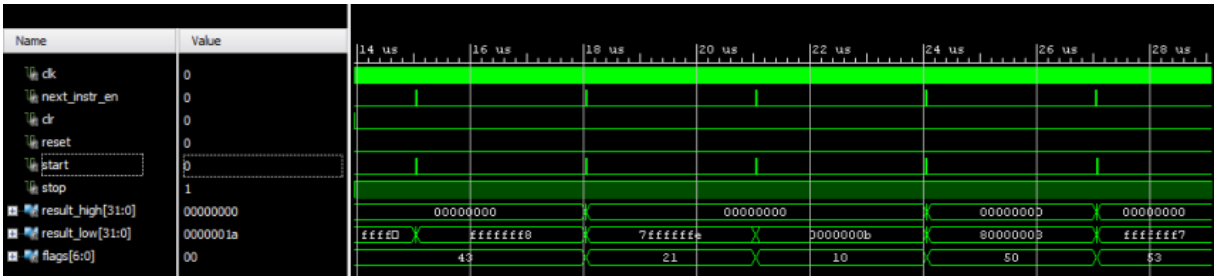

Figure 48.2. Testbench simulation results

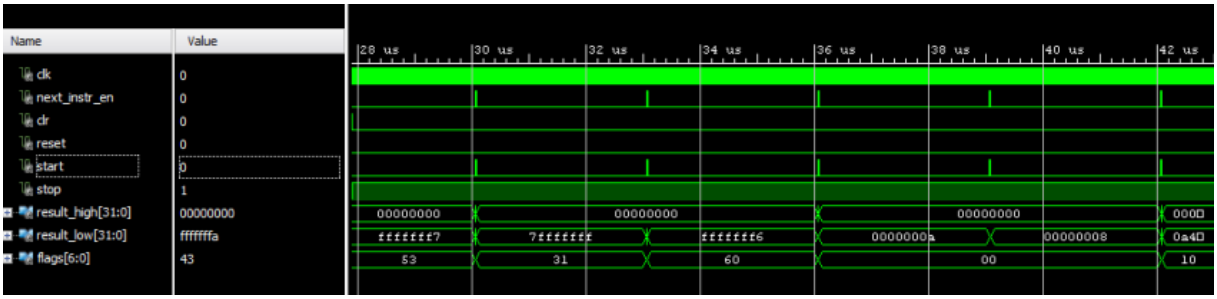

Figure 48.3. Testbench simulation results

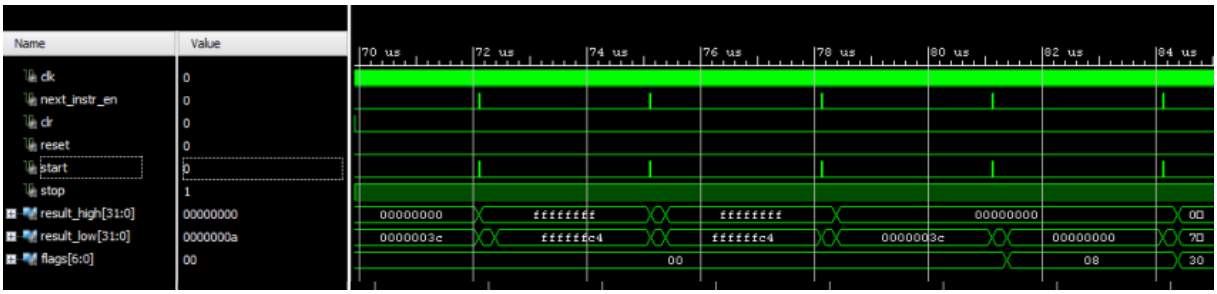

Figure 48.4. Testbench simulation results

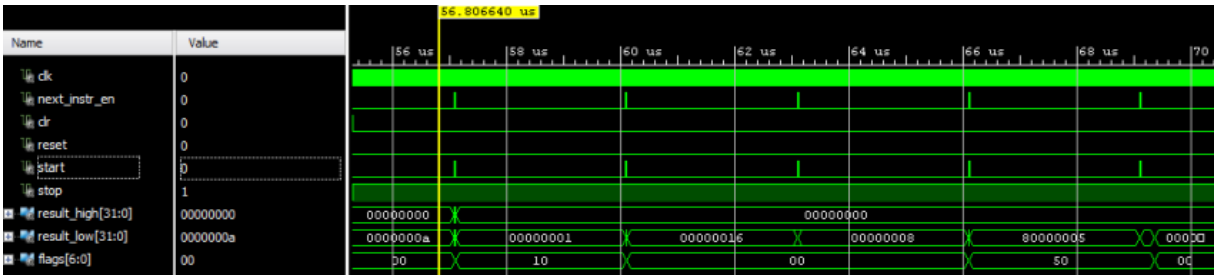

Figure 48.5. Testbench simulation results

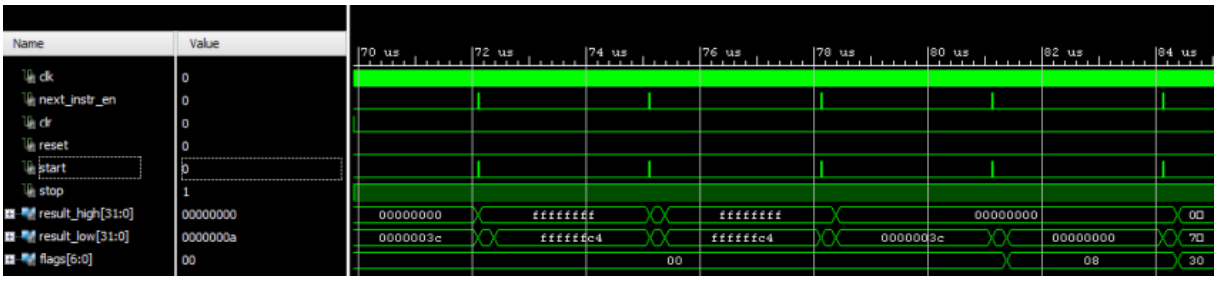

Figure 48.6. Testbench simulation results

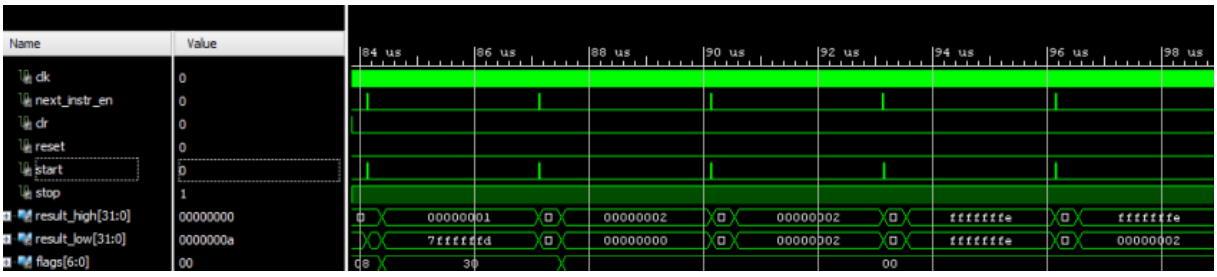

Figure 48.7. Testbench simulation results

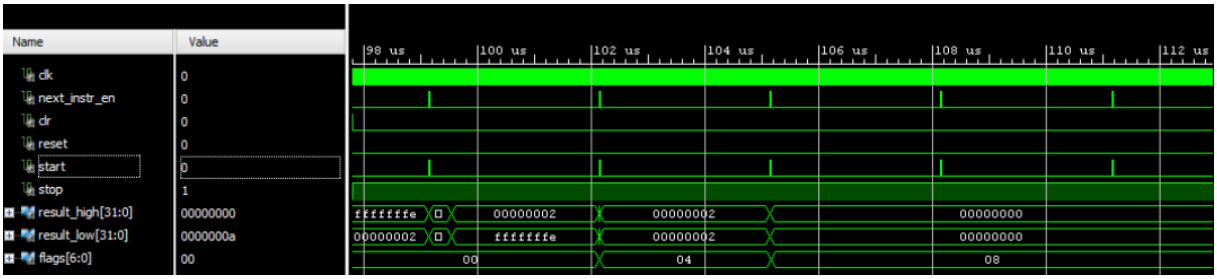

Figure 48.8. Testbench simulation results

6.2. Basys3 Simulation Results

In order to make the testing process easier to grasp, below is a figure showing the main controls of the board:

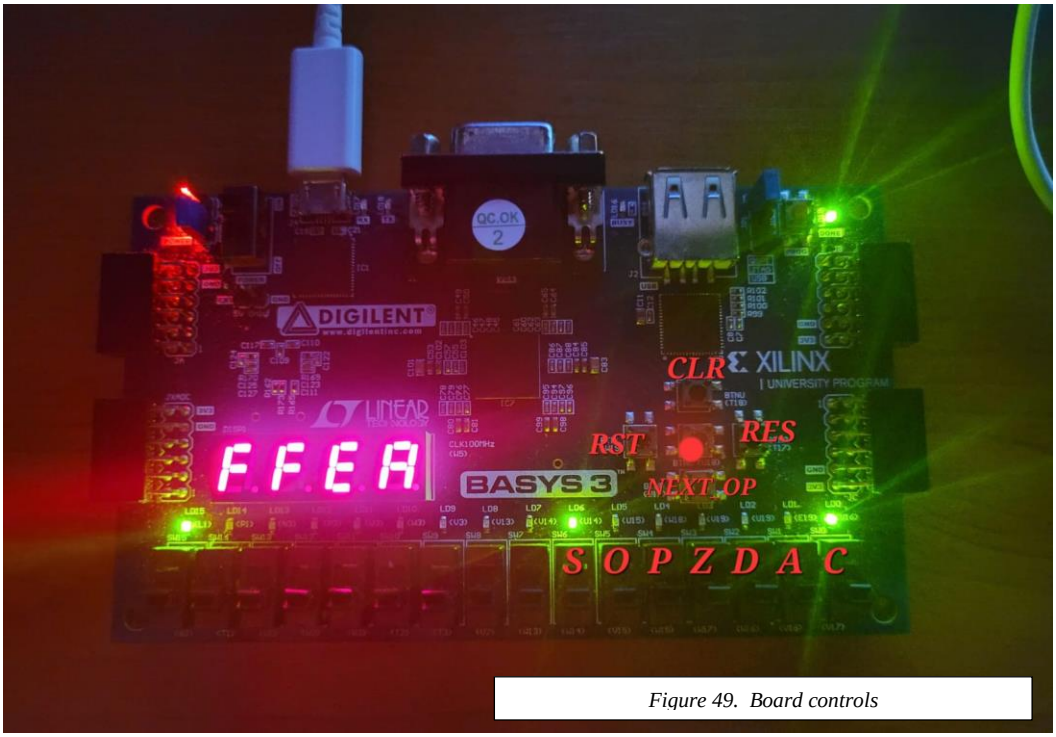

Figure 49. Board controls

Button description:

- CLR – clears the result register: displays „0000” on the seven-segment display
- RST – resets the program counter so that it will point to the first instruction in the ROM
- NEXT_OP – increments the program counter so that it will point to the next instruction in the ROM (moves to the next operation to be performed)
- RES – displays the result of the operation on the SSD

The result requires the use of the switches 0-1 in order to be fully observed:

[SW1, SW0] value	What is being displayed
00	ALU_RES_LOW(15 downto 0)
01	ALU_RES_LOW(31 downto 16)
10	ALU_RES_HIGH(15 downto 0)
11	ALU_RES_HIGH(31 downto 16)

Table 2. Result displayed according to switch values

Another aspect worth mentioning is the fact that the leds [L06, L00] represent the values of the flag outputs:

LED turned on	Corresponding Flag
L06	Sign flag (S)
L05	Overflow flag (O)
L04	Parity flag (P)
L03	Zero flag (Z)
L02	Divide by Zero flag (D)
L01	Auxiliary Carry flag (A)
L00	Carry flag (C)

Table 3. Leds turned on depending on flags

Lastly, here are some of the results displayed on the Basys3 board:

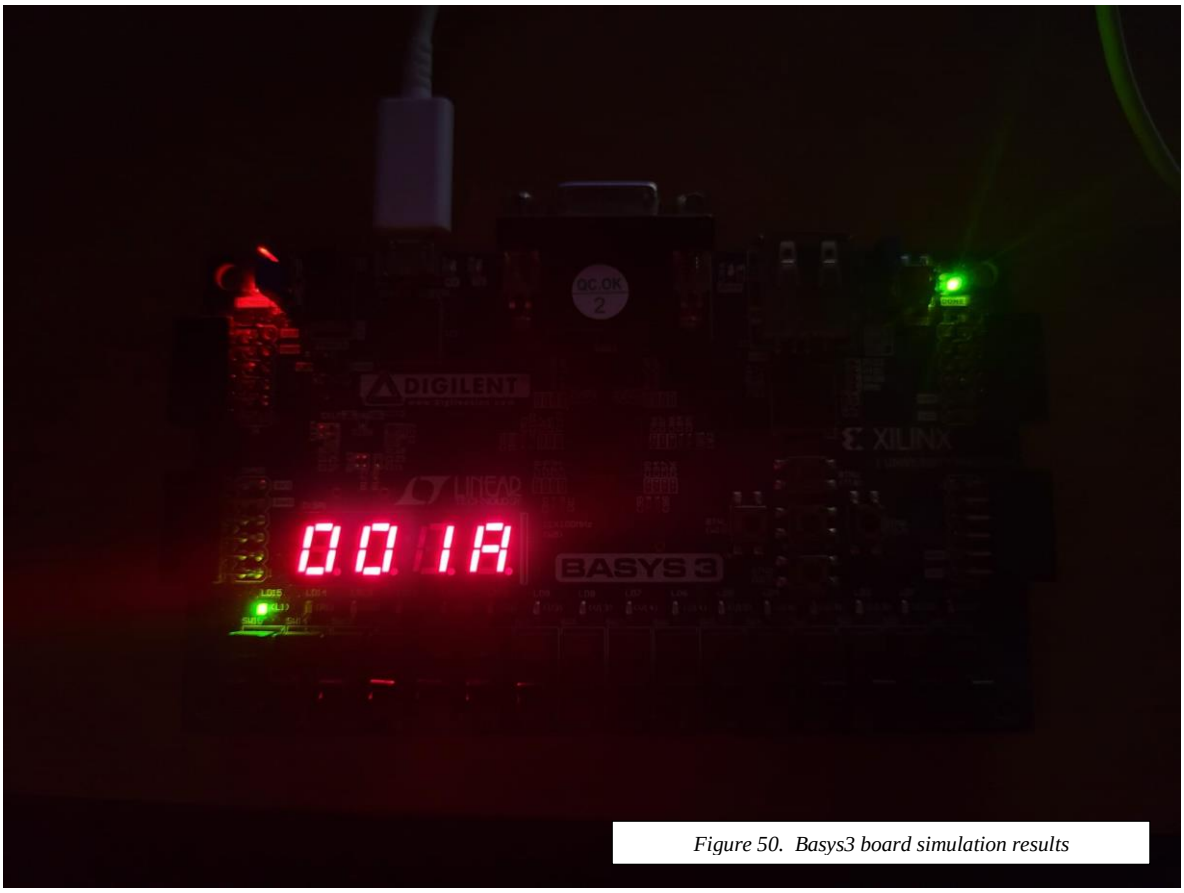

Figure 50. Basys3 board simulation results

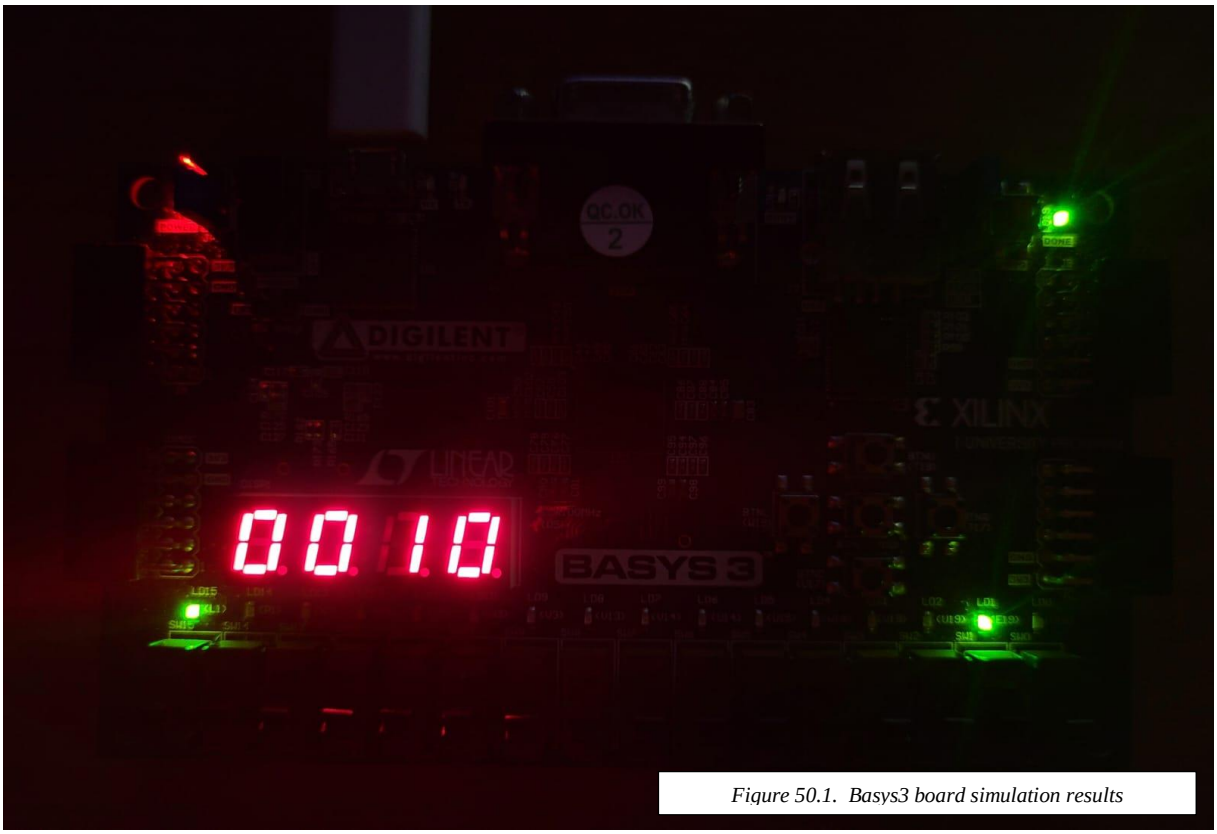

Figure 50.1. Basys3 board simulation results

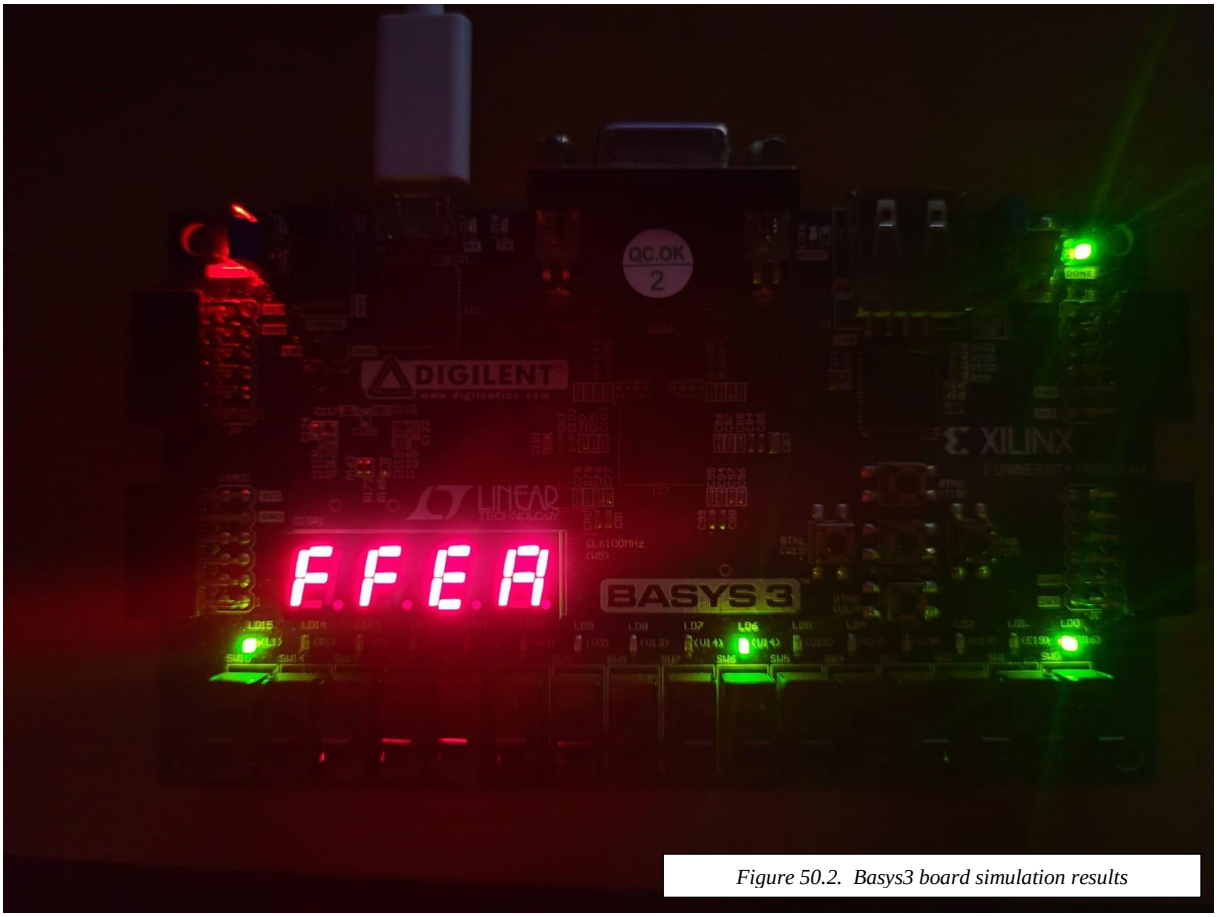

Figure 50.2. Basys3 board simulation results

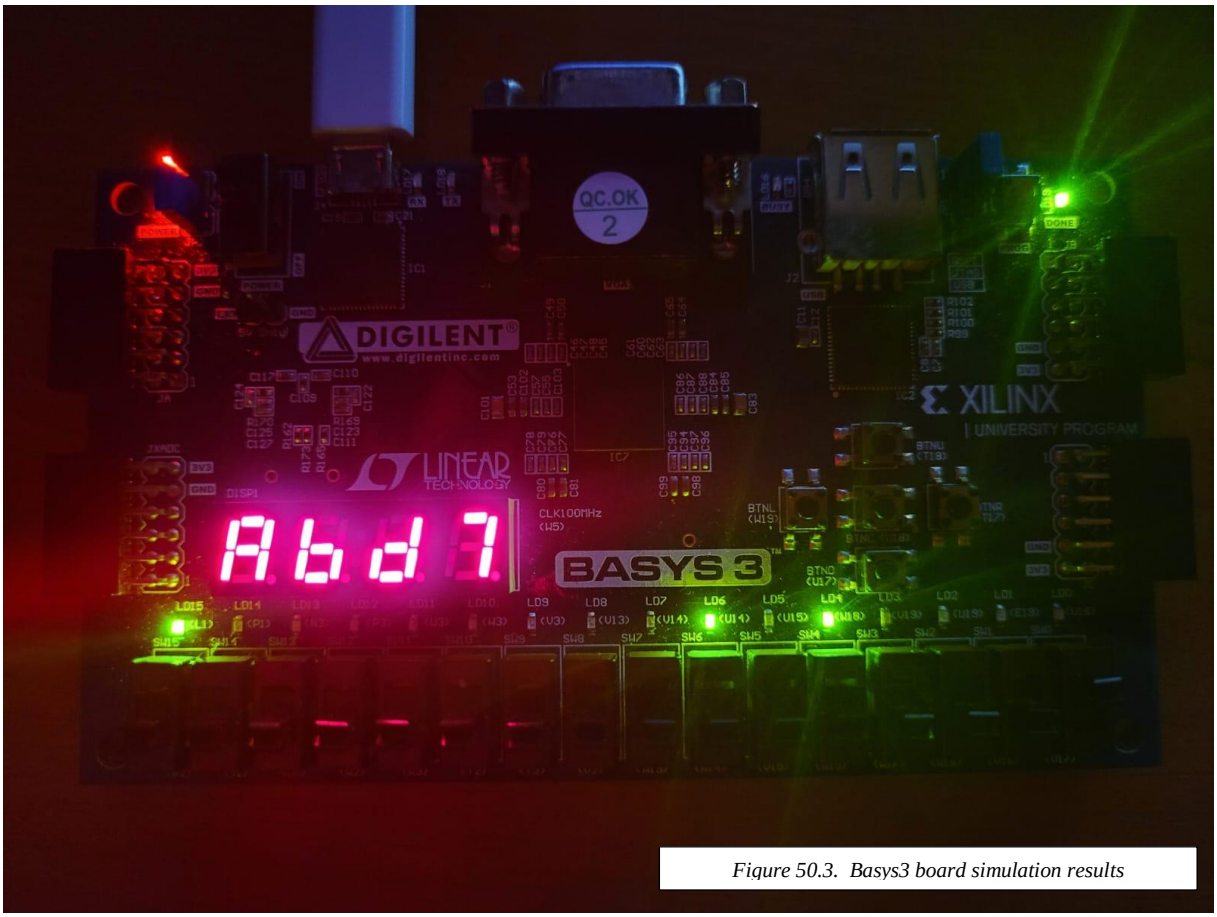

Figure 50.3. Basys3 board simulation results

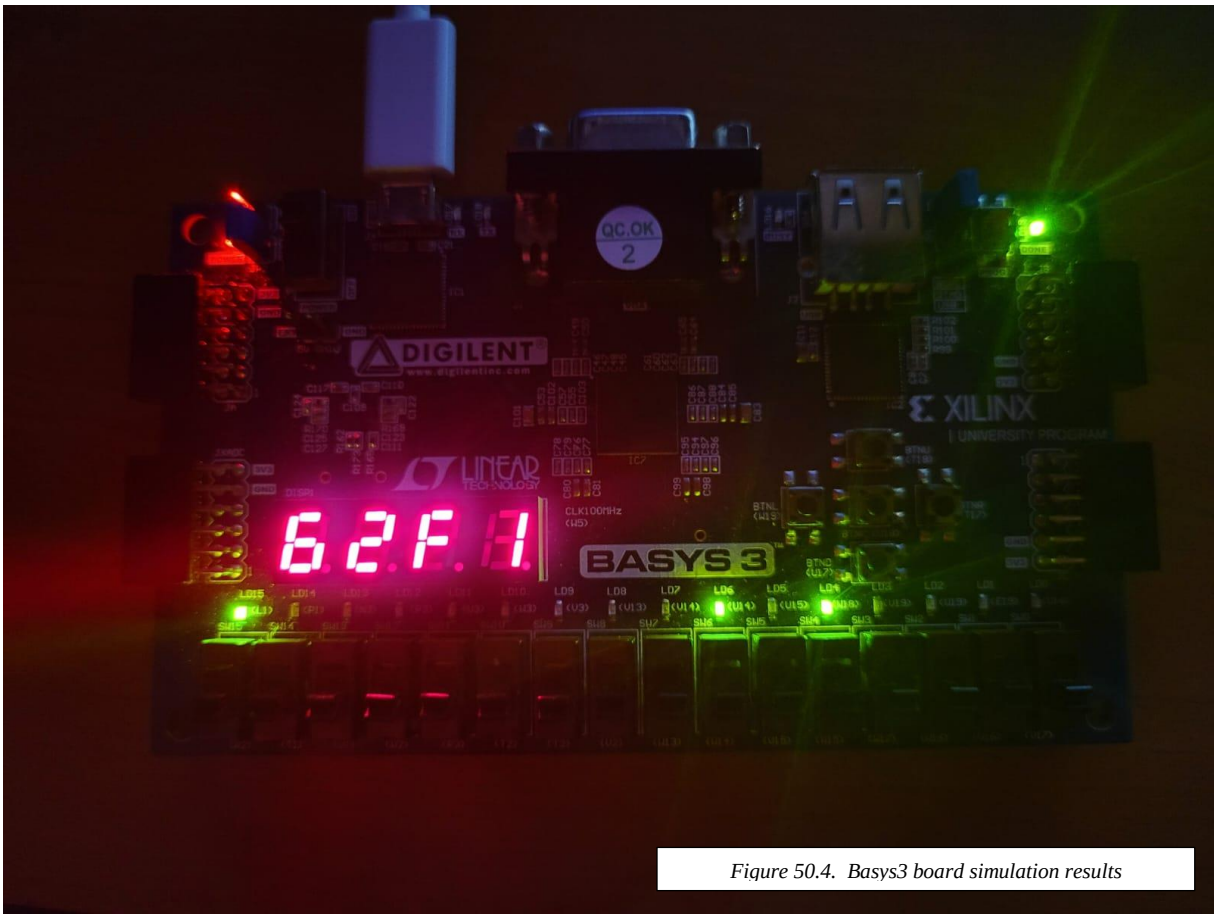

Figure 50.4. Basys3 board simulation results

VII. Conclusion

In conclusion, the Arithmetic Logic Unit solves one of the most important problems encountered by any computing system, that being operations performed on integers, and thus it must ensure correctness in providing results, since the functioning of other components of the system are strongly tied to it.

Working on this project has helped me not only improve my VHDL coding skills, but also get a better understanding of how the manipulation of integers represented in Two's Complement actually works, allthewhile making me appreciate the complexity of the process even more. I have also learned the basic concepts required for the correct and efficient design and implementation of hardware components, as well as how to program an FPGA device, namely the Basys3 board of the Artix 7 FPGA family, by means of interacting with the Vivado software.

Bibliographical Sources

[1] Gojko Babic, Arithmetic Logic Unit – ALU Design, 2005, https://web.cse.ohio-state.edu/~crawfis.3/cse675-02/Slides/CSE675_05_ALUDesign.pdf

[2] Tutorialpoint, Arithmetic Logic Unit (ALU), https://www.tutorialspoint.com/arithmetic-logic-unit-alu

[3] M. Morris Mano, Michael D. Cilentti, Digital Design (Fourth Edition), p. 12-17

[4] Electrical4u, 2's Complement Arithmetic, https://www.electrical4u.com/2s-complement-arithmetic/

[5] Boolean Logic, Princeton University, Boolean Logic (princeton.edu)

[6] Materials provided in course no. 3 and laboratory work no. 4