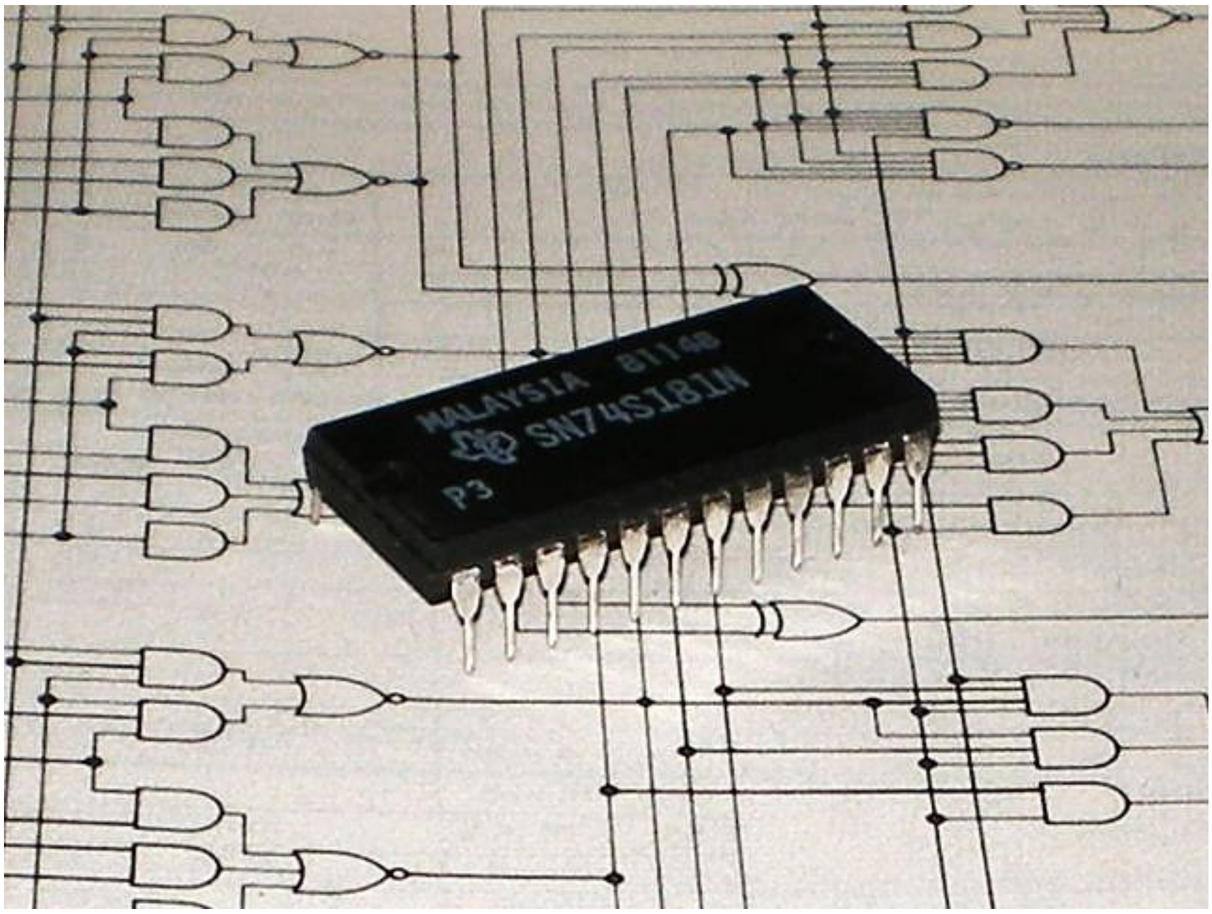
ALU: Arithmetic Logic Unit



Loga Darius
Technical University of Cluj - Napoca

Contents

1. Introduction

1.1. Context

The goal of this project is to design, implement and test several operations that can be included in an Arithmetic Logic Unit (ALU). Those operations have, at maxium, two inputs on 4-bits and only one output on 4 or 5-bits.

This unit can be used by people who need a simple way of using a different version of a calculator that can add, substract, multiply, divide, opposite of a number or use logic operations on numbers. It can also be used as an addition to other devices, for example, it could be integrated in any sort of processor or microprocessor that could use or benefit from having the capability of doing these operations or can be implemented in a test environment to discorver future bugs that can be later fixed.

1.2. Specifications

This unit will be simulated in the IDE provided by VIVADO and then programmed into a Xilinx Basys 3 board. It will be able to perform the operations mentioned above inside a multiplexer architecture based unit where if one input for the selection is displayed, the operation assign to that number of the selection will be performed and then the result will be displayed in the simulation file and also on the board, not on the same time.

1.3. Objectives

Design and implement a arithmetic logic unit that can compute the addition and subtraction of two numbers in two’s complement, increment or decrement one number, opposite of one number, logic AND, OR of two numbers, logic NOT for only one number at a time, rotatin on left or right of one number, use one accumulator as input from the output and multiplication and division of two numbers implemented in different files

2. Bibliographic study

For this part, I searched and documented myself about the requirements needed to implement all the operations detailed in the **Objectives** subsection. Addition and substraction in two's complement differ a lot from the normal addition or substraction, so I tried to find a proper way to convert the inputs in two's complement, then apply the operations and the result can remain in the two's complement form without making another conversion. For the increment or decrement, I just add one or substract one until I reach the limit of the result design which will be 8-bits. The opposite of a number will be represented in two's complement and displayed in this form.

Logic AND and OR of two numbers will be implemented bitwise and display the result after all comparisons are done and for NOT it is just the oppposite of all bits. The principle of the rotations is that after one bit is left or right shifted, it should appear to the most significant bit for left and to the least significant bit for the right and continue to be shifted until it reaches the exact number that was introduced as input. For the accumulator, I will use the counter principle, but instead, I will add the accumulator's value and the one from the other input.

The last operations are multiplication and division using different files which means that in those files I create entities for those and make them components in the main file and call them using one function to map them into the specific inputs and outputs to have the best result. For the multiplication, I will use the Wallace Tree method with half and full adders and for division, I searched that for the binary division is based on repeated substractions of the divisor from the partial remainder and are executed only if divisor is lower than the partial remainder.

3. Analisys

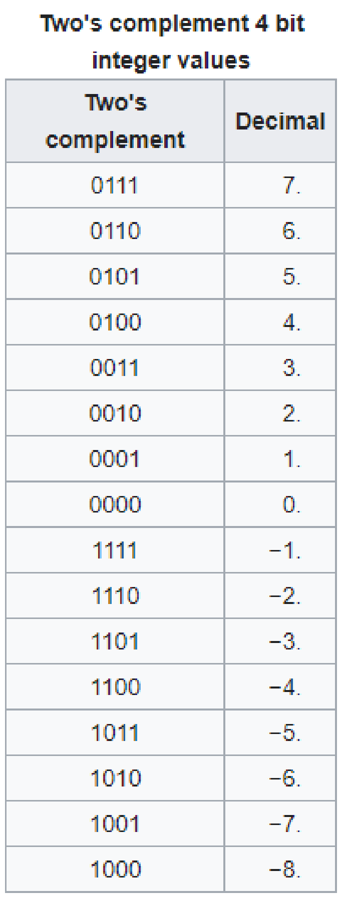
In my further analisys, I assumed that all the inputs related to the numbers are on 4-bits and the output is on 8-bits. All the operations will be explained in this chapter and for some of then, I will put, in the next chapter, a picture with the block diagram or the internal structure to clarify the explanations.

Operation 1 - 2.) Addition/Substraction in Two's Complement

To explain what Two's Complement is, I can simply say that I can perform operations using unsigned or signed numbers. Below is an image that represents better what I am saying. The result of the addition or substraction, if it is on the values from those 4-bits, there is no need to represent the carry-out bit, therefore, there will not be any overflow. The overflow occurs if we add two positive numebers, substract from a positive number a negative number then the result is negative or if we add two negative numbers, substract from a positive number from a negative number then the result will be positive.

Might also occurs if we compute those operations, where the result cannot be written on that bits, so we need to increase the number of bits accepted by the result in order to obtain the correct result. In those situations, we need to display the carry-out bit to know exactly the length of the result. Below is an inequality that helps us to understand what is the limit of bits for each number that we want to know, input or result, where n is the number of bits for a number:

$$-2^{n-1} \leq \text{Two's Complement} \leq 2^{n-1} - 1.$$



Two's complement 4 bit integer values

Two's complement	Decimal
0111	7.
0110	6.
0101	5.
0100	4.
0011	3.
0010	2.
0001	1.
0000	0.
1111	−1.
1110	−2.
1101	−3.
1100	−4.
1011	−5.
1010	−6.
1001	−7.
1000	−8.

Fig. 1 - representation of the two's complement for 4-bit values.

Operation 3 - 4.) Increment/Decrement

For this operation we simply take one of the input numbers and increment or decrement it by 1. The result will be on 4 used bits out of 8.

Operation 5.) Logic AND

This operation will be performed just like a normal AND Gate, but bitwise between the inputs and the result will be just one of 4 used bits out of 8.

Operation 6.) Logic OR

This operation will be performed just like a normal OR Gate, but bitwise between the inputs and the result will be just one of 4 used bits out of 8.

Operation 7.) Logic NOT

This operation will be performed just like a normal NOT Gate for all bits of one of the inputs and the result will be one of 4 used bits out of 8.

Operation 8.) Opposite of a number

We choose between one of the unsigned inputs and negate all the bits and then add 1 to that negation. This should be the opposite, with respect to the inequation shown before at the addition/substraction for the result. The numbers of bits for the output will be 5.

Operation 9 – 10.) Rotation left or right

One of the inputs will have the possibility to be rotated in one of those directions with one bit. So, for one of this, the LSB should become the MSB or vice-versa and all other bits are shifted according to the direction specified by us.

Operation 11.) Unsigned Up Accumulator

This operation is similar with a normal counter, but instead of adding just one, we can add a variable to this accumulator and obtain the sum between one input and the variable, rewrite the value of the accumulator with this new one and display the result in a limit of numbers of 8-bits.

Operation 12 – 13.) Multiplication and Substraction for Unsigned numbers

For the multiplication of the unsigned numerbs we can use the Wallace Tree method because is efficient and has three simple steps: 1.) multiply each bit of one number to the ones of the other number; 2.) The partial products can be done using sequences of half and full adders 3.) Group the final results into two numbers and add them with an adder. We should have 3 stages in which we make those additions with those adders. One approach should be similar to this one, just like in the image below:

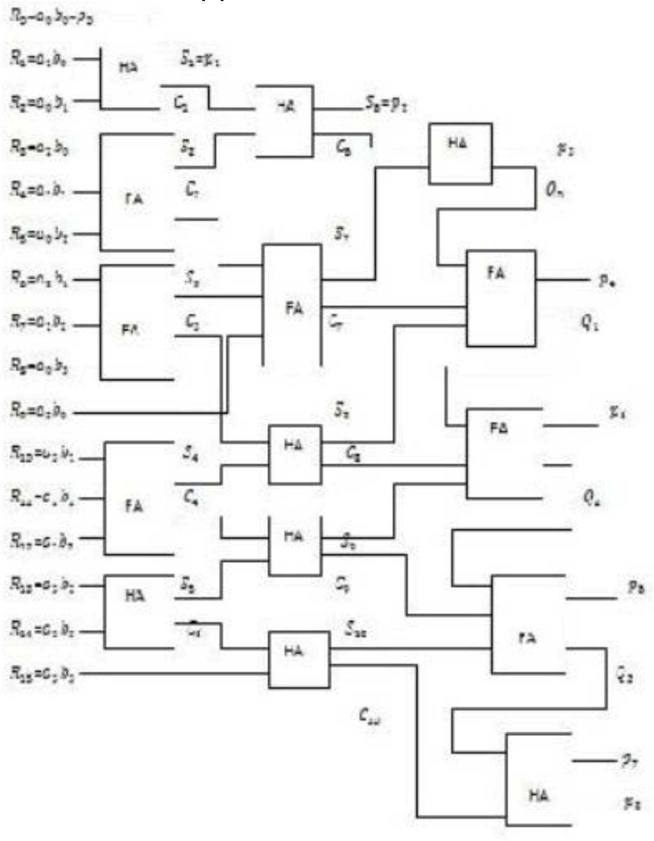


Fig. 2 - Black box for the multiplication.

I think I wiill try to simplify this approach and came with a better one for the implementation part. For the division, as I explain earlier in the documentation, I have to make repeated substractions to the divisor. The starting formula is: A = B * Q + R, where A is the number, B is the other number, Q is the quotient and R is the rest Substraction and shift operations are the basic operations for the division algorithm. For this algorithm, we can use counter and one value in which we store the substractions and increase the counter and for the result I will obtain the Q and R, after different comparisons. All of these operations can survey, at some point, the overflow part, but as I said, we make these operations for unsinged numbers.

4. Design

Below is the top level of the main application which includes the selection of each operation, inputs, outputs and also, if it can work for my Basys 3 board, the BCD convertor for the display.

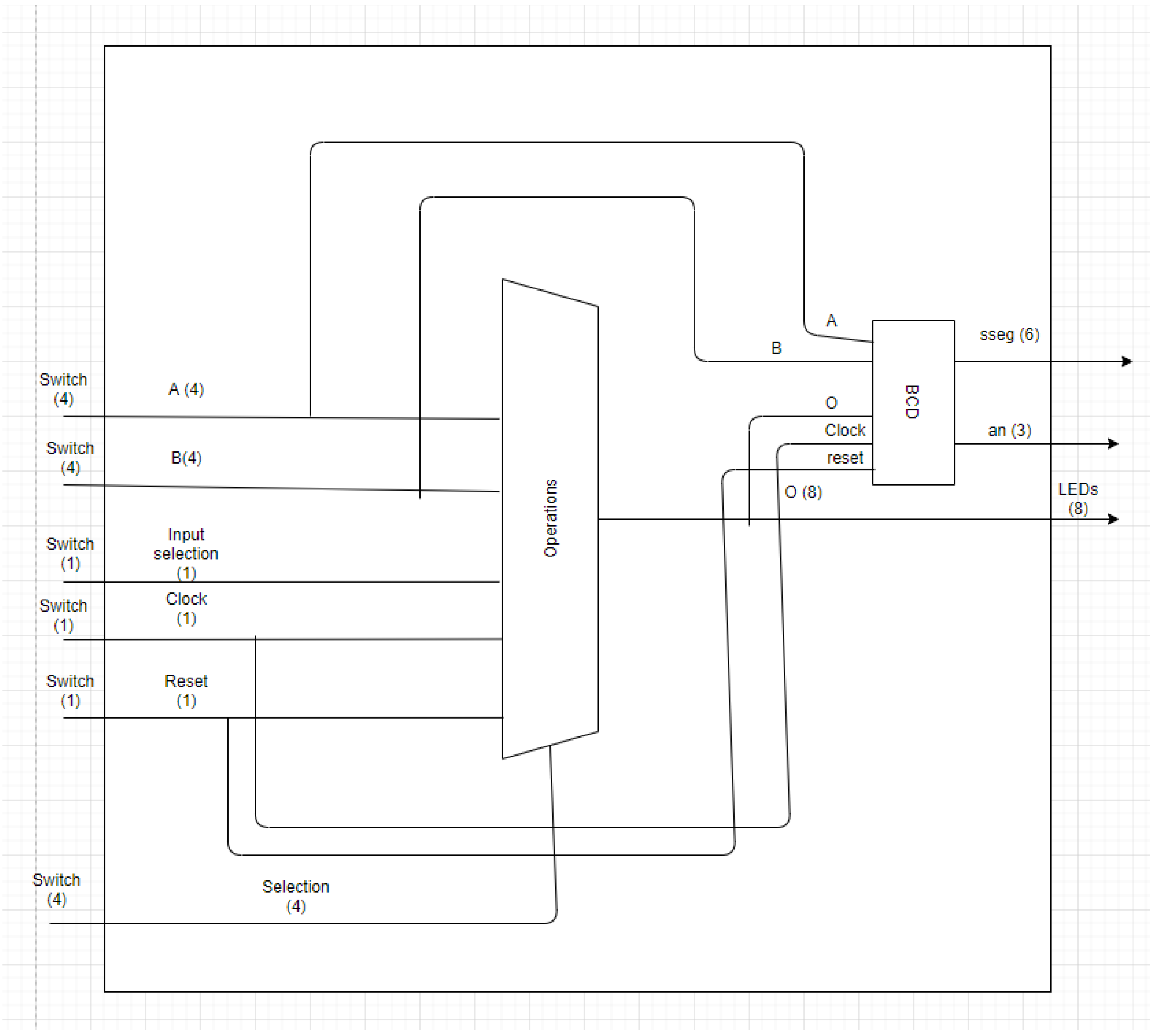
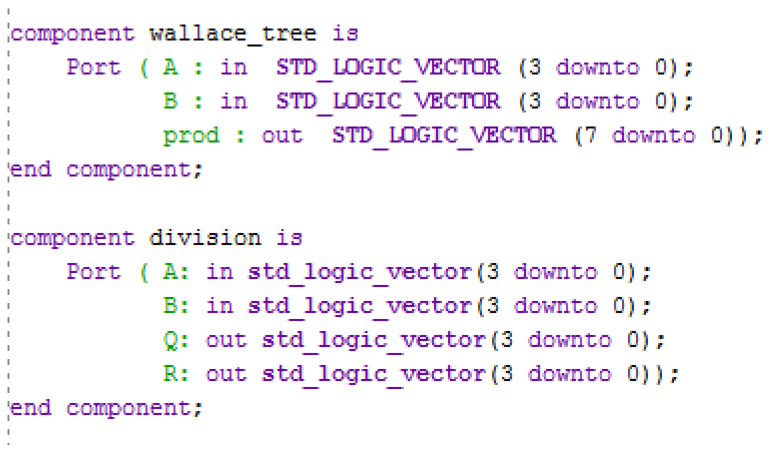


Fig. 3 - The black box for the whole project

The Multiplexer based approach that I have chosen for the operations is this because it is really easy for me to make a selection, using 4-bits, and also implemented by switches on the board, do the operation and display the output on the LEDs or on the board's digital diplay. Clock and Reset are synchronous for both components and the Input Selection bit is used for the selection of either A or B input for the operations that require just one of them. For all the inputs I chose to have switches on the board.

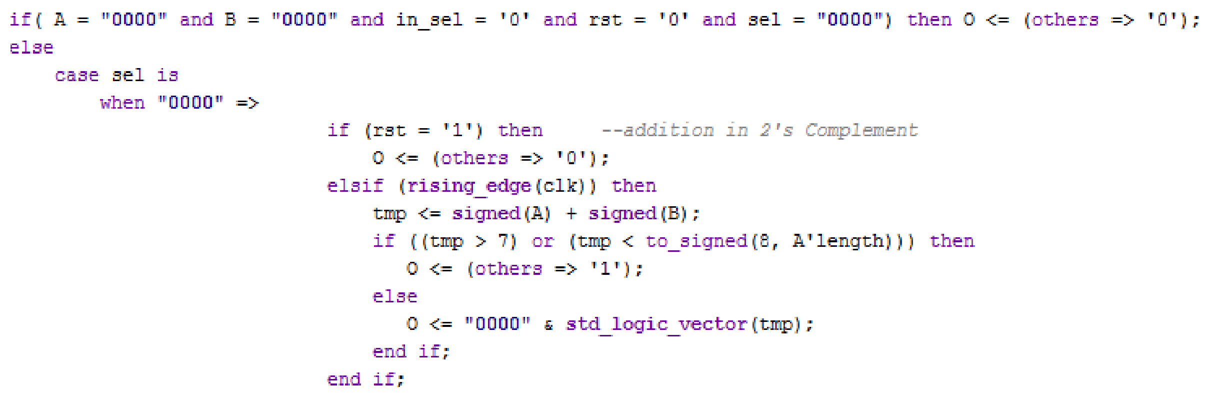
5. Implementation

This chapter is closely retalted to the Design one, but I have a final version in mind on how to approach this project. So, I started by thinking of all the inputs and outputs with the help of the already created black box and how to shape that "MUX" component. One big case statement was used in my project to fit every functionality presented in the chapters above. Also, I used **numeric_std** library which helped me so much with the computation of addition and substraction in Two's Complement. One input, at a time, was used for several functions, for example: increasing, decreasing, rotation left/right, Up accumulator etc.
For the multiplication I sticked to the representation done in one laboratory work using Wallace tree, but for the division, after a lot of tries, I managed to find a nicer and simpler version, using the newly listed library, just by dividing to obtain the quotient (**Q**) and for the rest (**R**) I used a function called "**rem**" which basically functions just like pen-and-paper division when we substract (**A – B * Q**) if **B * Q < A** to obtain it, converted into unsigned numbers and then back to std_logic_vector. The **Q** is saved on first 4 bits, starting from MSB, and **R** is on the last 4 bites of the final displayable result. Below, I will insert some images resembling code snippets from the actual top level domain and also from the testbench.
On the Basys3 I used switches, LEDs and board input for clock.



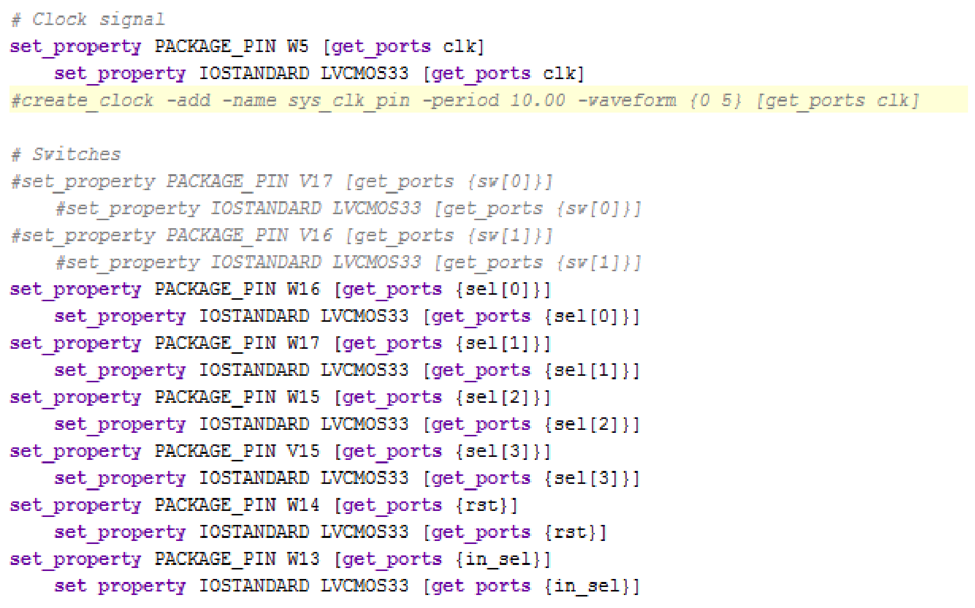
```
component wallace_tree is
    Port ( A : in  STD_LOGIC_VECTOR (3 downto 0);
           B : in  STD_LOGIC_VECTOR (3 downto 0);
           prod : out  STD_LOGIC_VECTOR (7 downto 0));
end component;

component division is
    Port ( A: in std_logic_vector(3 downto 0);
           B: in std_logic_vector(3 downto 0);
           Q: out std_logic_vector(3 downto 0);
           R: out std_logic_vector(3 downto 0));
end component;
```

Fig. 4 - Multiplication and division as components for the main domain.



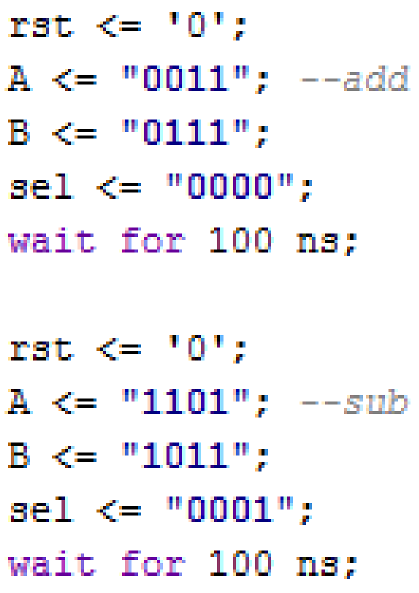
```
if( A = "0000" and B = "0000" and in_sel = '0' and rst = '0' and sel = "0000") then O <= (others => '0');
else
    case sel is
        when "0000" =>
                        if (rst = '1') then     --addition in 2's Complement
                            O <= (others => '0');
                        elsif (rising_edge(clk)) then
                            tmp <= signed(A) + signed(B);
                            if ((tmp > 7) or (tmp < to_signed(8, A'length))) then
                               O <= (others => '1');
                            else
                               O <= "0000" & std_logic_vector(tmp);
                            end if;
                        end if;
```

Fig. 5 – case statement with a verification at the begining for the Basys3 board and also usage of the signed values and different conversions



```
# Clock signal
set_property PACKAGE_PIN W5 [get_ports clk]
    set_property IOSTANDARD LVCMOS33 [get_ports clk]
#create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports clk]

# Switches
#set_property PACKAGE_PIN V17 [get_ports {sw[0]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {sw[0]}]
#set_property PACKAGE_PIN V16 [get_ports {sw[1]}]
    #set_property IOSTANDARD LVCMOS33 [get_ports {sw[1]}]
set_property PACKAGE_PIN W16 [get_ports {sel[0]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sel[0]}]
set_property PACKAGE_PIN W17 [get_ports {sel[1]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sel[1]}]
set_property PACKAGE_PIN W15 [get_ports {sel[2]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sel[2]}]
set_property PACKAGE_PIN V15 [get_ports {sel[3]}]
    set_property IOSTANDARD LVCMOS33 [get_ports {sel[3]}]
set_property PACKAGE_PIN W14 [get_ports {rst}]
    set_property IOSTANDARD LVCMOS33 [get_ports {rst}]
set_property PACKAGE_PIN W13 [get_ports {in_sel}]
    set_property IOSTANDARD LVCMOS33 [get_ports {in_sel}]
```

Fig. 6 – Constraints used for the connections with the board'

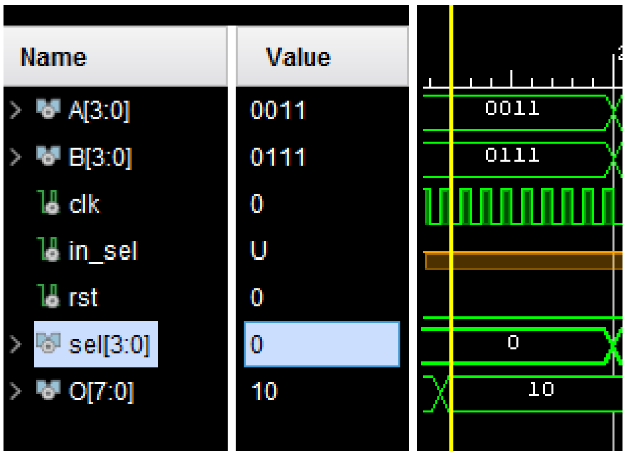


```
rst <= '0';
A <= "0011"; --add
B <= "0111";
sel <= "0000";
wait for 100 ns;

rst <= '0';
A <= "1101"; --sub
B <= "1011";
sel <= "0001";
wait for 100 ns;
```

Fig. 7 – Inputs used in the testbench for additon and substraction in Two's Complement

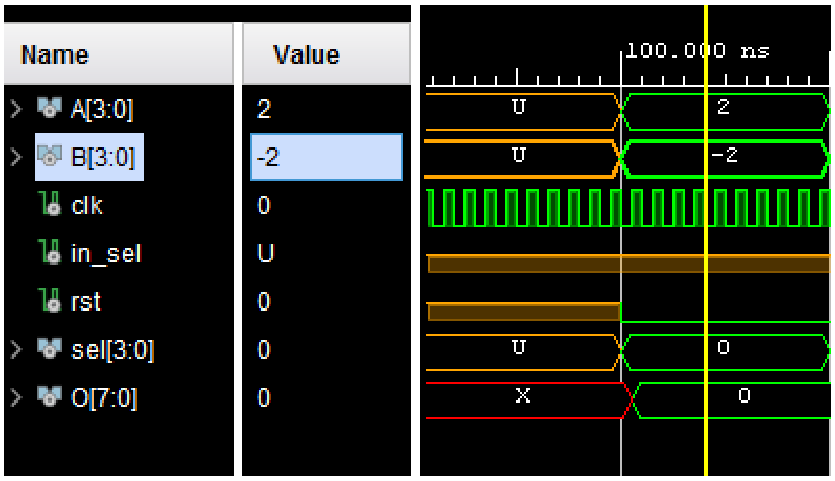
6. Testing and Validation

A series of images for each implemented functionality, from the testbench, will be inserted below even if the validation is not correct.



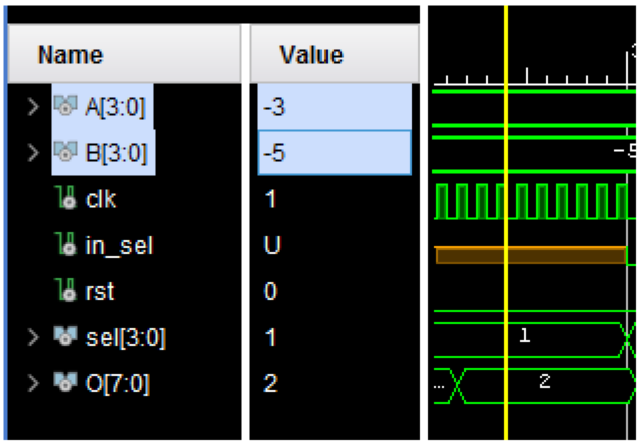
Name	Value
A[3:0]	0011
B[3:0]	0111
clk	0
in_sel	U
rst	0
sel[3:0]	0
O[7:0]	10

Fig. 8 – even if I put a validation not to surpass certain integer values, the simulator compiles it as valid for the Two's Complement addition

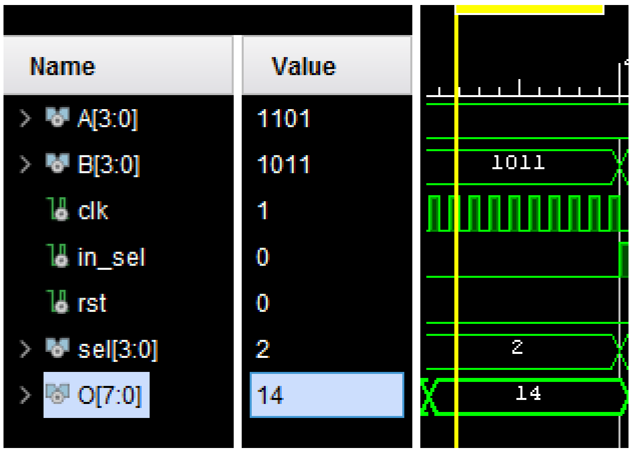


Name	Value
A[3:0]	2
B[3:0]	-2
clk	0
in_sel	U
rst	0
sel[3:0]	0
O[7:0]	0

Fig. 9 – this is the one good output for the same addition

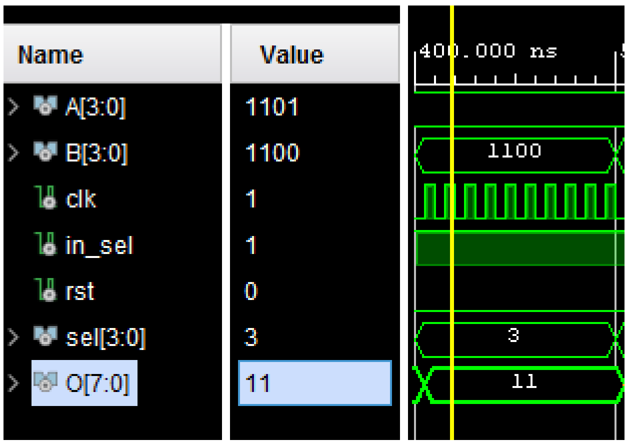


Name	Value
A[3:0]	-3
B[3:0]	-5
clk	1
in_sel	U
rst	0
sel[3:0]	1
O[7:0]	2



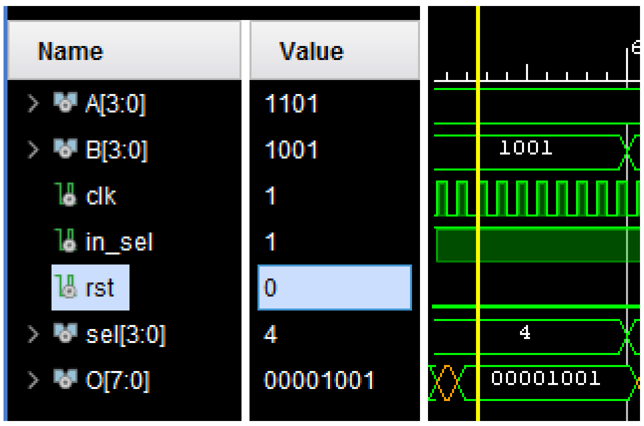
Name	Value
A[3:0]	1101
B[3:0]	1011
clk	1
in_sel	0
rst	0
sel[3:0]	2
O[7:0]	14

Fig. 10 – valid Two's Comp substraction **Fig. 11 –** Inncrement by 1



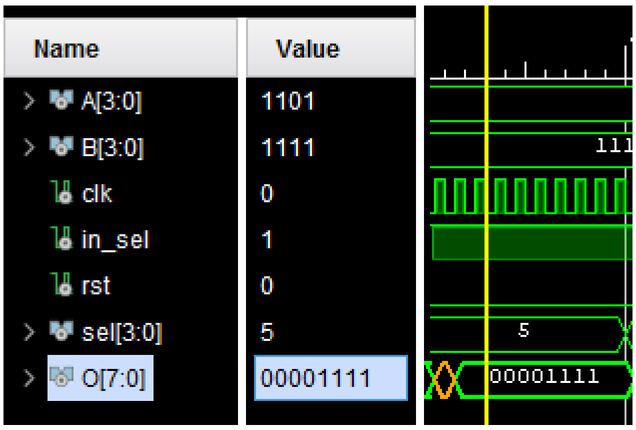
Name	Value
A[3:0]	1101
B[3:0]	1100
clk	1
in_sel	1
rst	0
sel[3:0]	3
O[7:0]	11

Fig. 12 – Decrement by 1



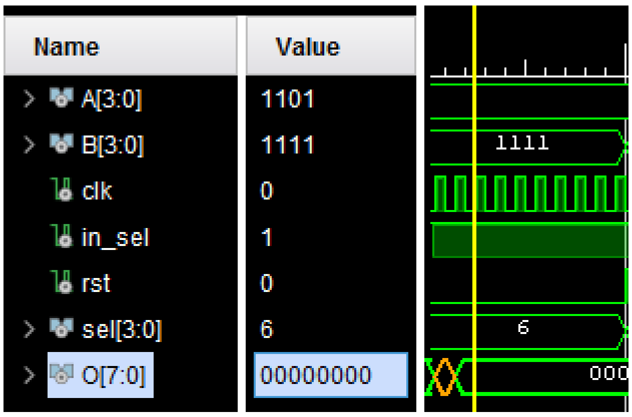
Name	Value
A[3:0]	1101
B[3:0]	1001
clk	1
in_sel	1
rst	0
sel[3:0]	4
O[7:0]	00001001

Fig. 13 – Logic AND



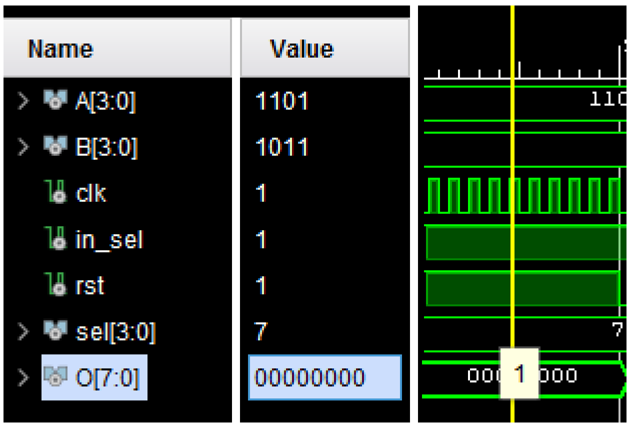
Name	Value
A[3:0]	1101
B[3:0]	1111
clk	0
in_sel	1
rst	0
sel[3:0]	5
O[7:0]	00001111

Fig. 14 – Logic OR

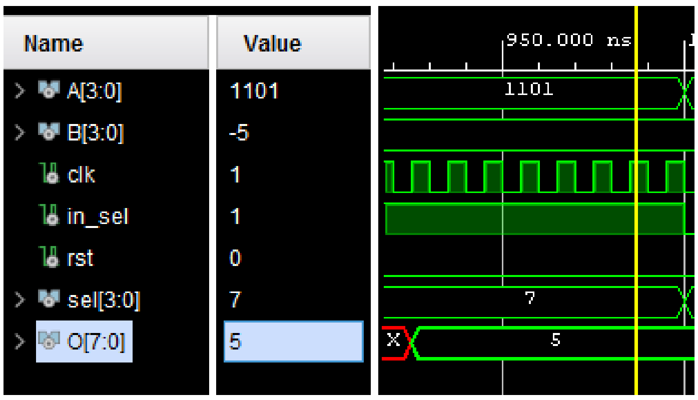


Name	Value
A[3:0]	1101
B[3:0]	1111
clk	0
in_sel	1
rst	0
sel[3:0]	6
O[7:0]	00000000

Fig. 15 – Logic NOT for the B input

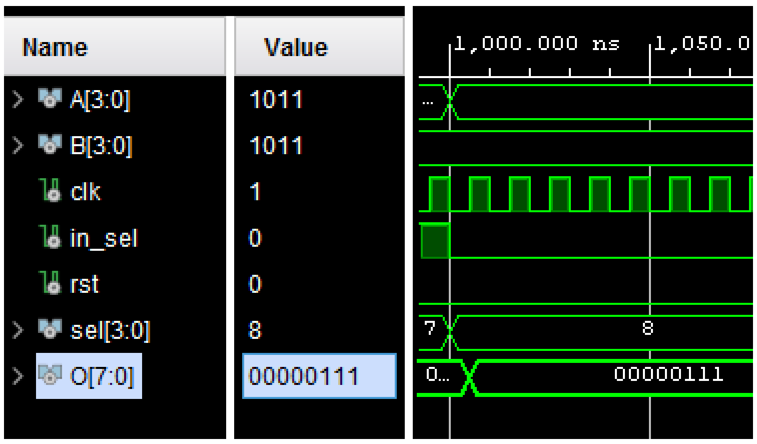


Name	Value
A[3:0]	1101
B[3:0]	1011
clk	1
in_sel	1
rst	1
sel[3:0]	7
O[7:0]	00000000



Name	Value
A[3:0]	1101
B[3:0]	-5
clk	1
in_sel	1
rst	0
sel[3:0]	7
O[7:0]	5

Fig. 16 – Oppsoite of a number with reset on high **Fig. 17 –** Oppsoite of a number with reset on low



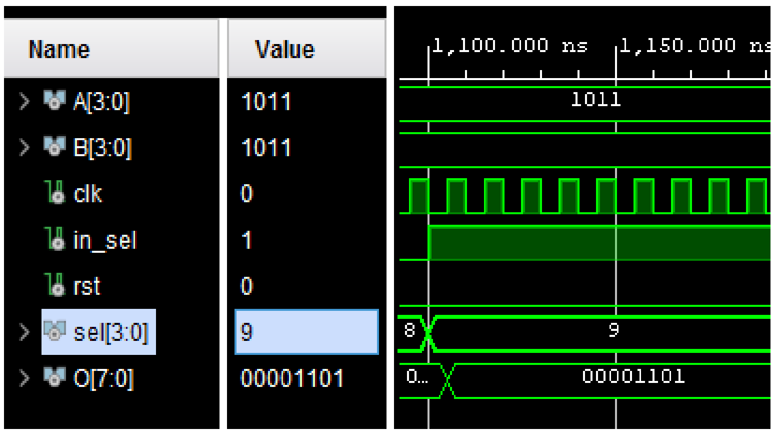


Fig. 18 – Rotation left

Fig. 19 – Rotation right

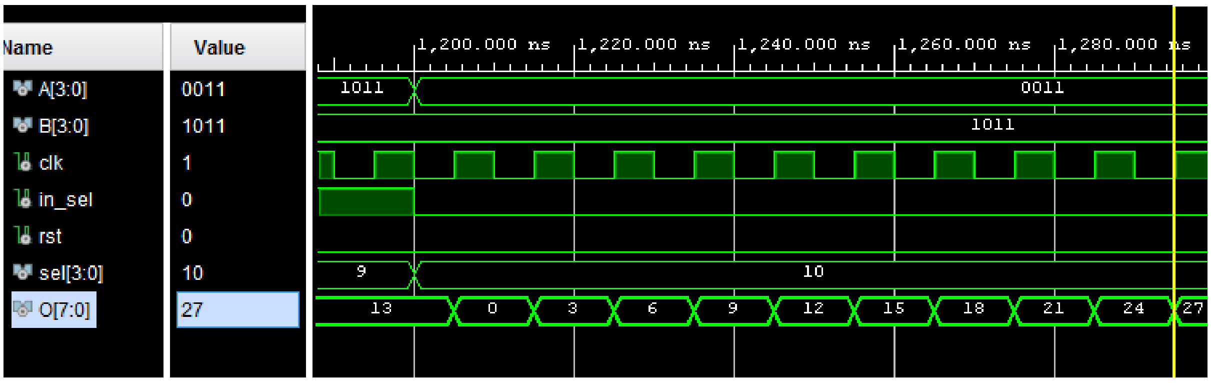


Fig. 20 – Up accumulator only for 100 ns, it can go to the maximum nb of bits (8 in my case)

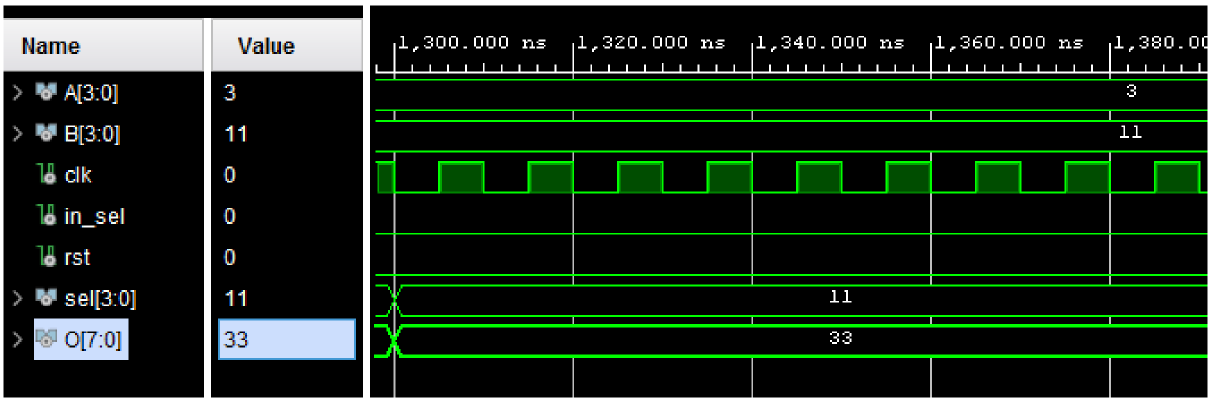


Fig. 20 – Multiplication using Wallace tree for unsigned inputs

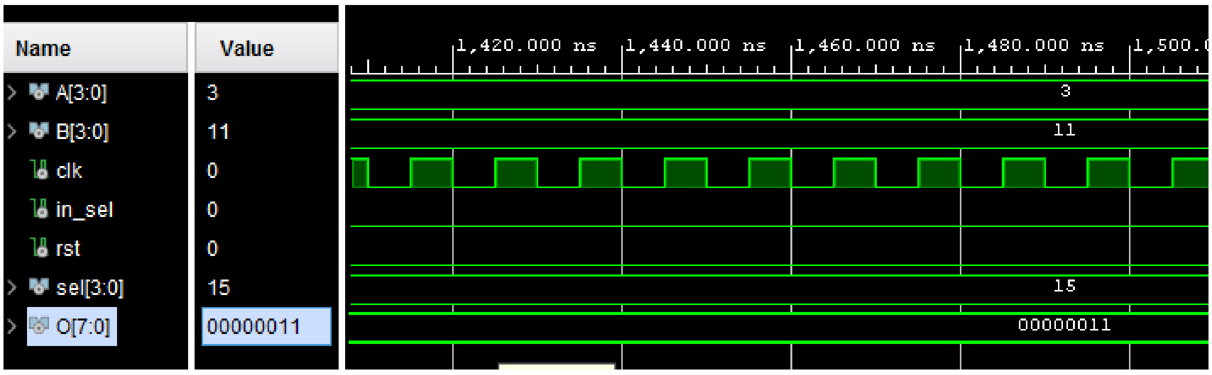


Fig 21. – Division for unsigned inputs

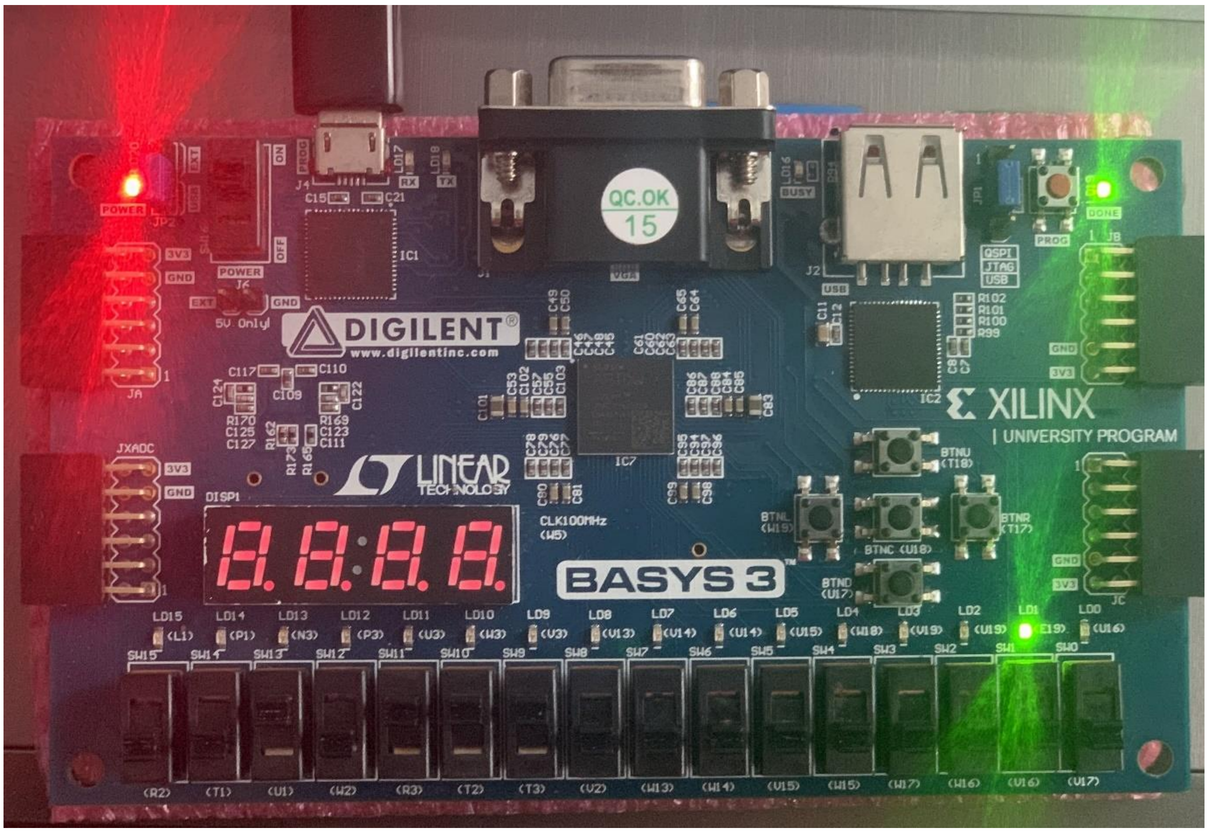


Fig. 22 – Image from my Basys3 doing something, not showing the right input

It might be because of the internal clock or some misleading code in the constraint, because I had no issues with the simulator, except one if.

7. Conclusions

To sum up everything, it was a challenging project, full of enjoyment and waiting because of the slow computation of the simulation, but now I know what is required to design and build an VHDL project with the usage of a board and also to be prepared in implementing such a handful of functionalities.
In the future, I would like to expend the project, to fix all the bugs, errors, and more validations, functionalities that can cover a lot more inputs from the board. This project helped me understand the value of time and also that a larger project require much more attention to details and preparation for the "final combat".

8. Bibilography

https://en.wikipedia.org/wiki/Two%27s_complement
https://www.doc.ic.ac.uk/~eedwards/compsys/arithmetic/index.html
https://www.ijser.org/paper/Structural-VHDL-Implementation-of-Wallace-Multiplier.html
https://app.diagrams.net/
https://www.nandland.com/vhdl/examples/example-signed-unsigned.html
https://www.nandland.com/vhdl/tips/tip-convert-numeric-std-logic-vector-to-integer.html#Numeric-Std_Logic_Vector-To-Signed
Own laboratory works
https://arith-matic.com/notebook/img/alu/74181-ALU.jpg