MINISTRY OF EDUCATION AND RESEARCH

TECHNICAL UNIVERSITY
OF CLUJ-NAPOCA, ROMANIA

STRUCTURE OF COMPUTER SYSTEMS

ARITHMETIC LOGIC UNIT (ALU) SOLUTION DESCRIPTION DOCUMENT

Student: Mihai Cristina-Mădălina
Teaching assistant: Mădălina Neagu

Faculty of Automation and Computer Science
Computer Science Department (En)
3rd year of study, gr. 30434

Content:

1. Introduction

1.1 Context

- **The main objective** of this laboratory work is to **design, implement and test 12 operations** on the two's complement format on **8 bits.** The system will implement the following operations:
  - addition;
  - subtraction;
  - incrementation;
  - decrementation;
  - logical AND, OR, NOT;
  - negation;
  - left shift;
  - right shift;
  - multiplication;
  - division.
- The project will implemented as a **Project** using **Vivado application** which includes a simulation tool to test the implementation.

1.2 Specifications

The system will be simulated in the IDE provided by **Vivado** and then tested on a **Basys3 board**. It will be able to represent internally the numbers in the two's complement representation, perform the required operations and display the input and output in decimal format on the board.

1.3 Objectives

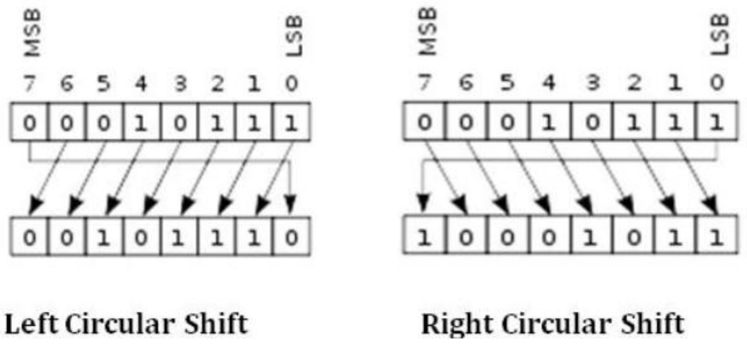
The aim of the project is to implement a functional system using minimum complexity and resources. Based on user choice, the system will be able to compute the required operation on the specified numbers. The input and the output will be visible into the simulation tool and on the board.

2. Bibliographic study

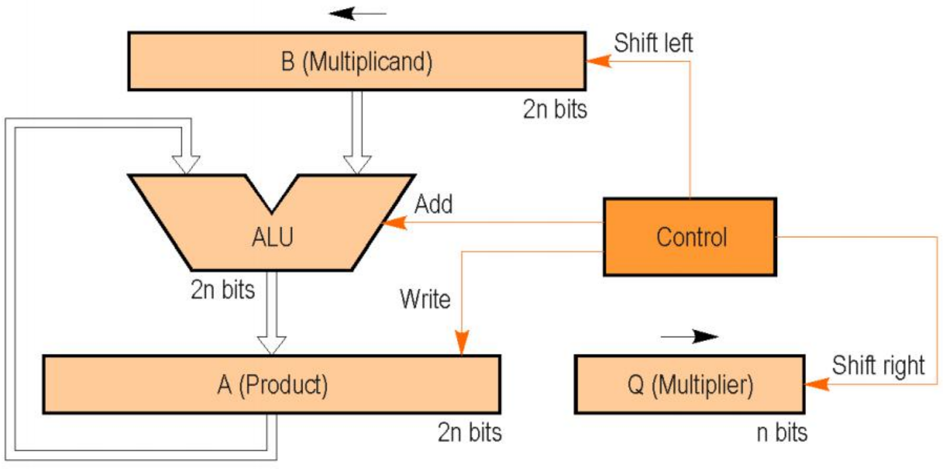
The ALU (Arithmetic and Logic Unit) is a digital circuit that performs **arithmetic and logical operations**. It must be able to calculate most operations.
For simple operations (addition, subtraction, logical AND, logical OR and logical NOT) there will be used simple notations available in Vivado to perform the operations ("+", " -", "and", "or", "not") in order to reduce complexity and improve readability.

For **negation**, the following method will be used:
- negate all the bits of the number (using logical not);
- add 1 to the negated number.

For **rotation**, logical shift and concatenation will be performed according to the following scheme:

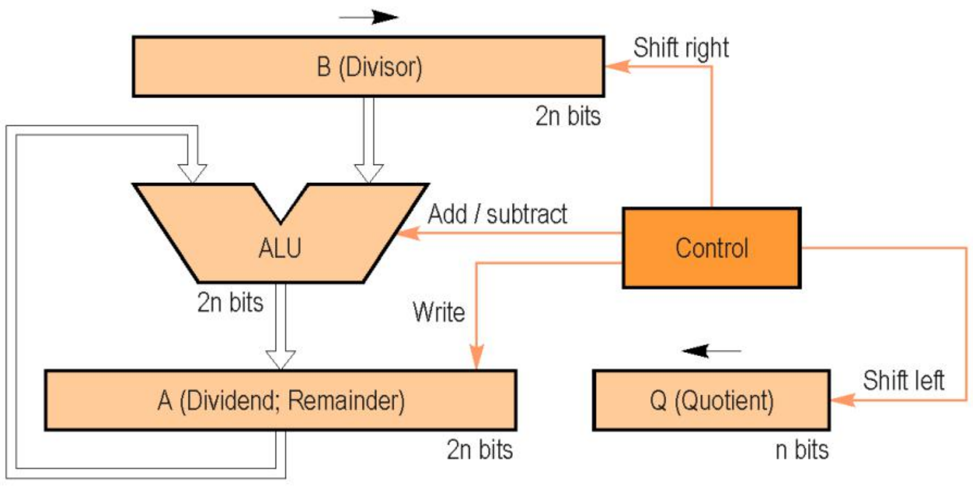


For **multiplication**, the algorithm presented in the laboratory work will be used:
Shift-and-Add Multiplication is one of the basic and simplest method for adding 2 numbers.Basically, the whole idea is to add the multiplicand (let's say X) to itself for Y (multiplier) times. The algorithm is based on taking each digit of the multiplicand in turn and multiplying it by a single digit of the multiplier. Each intermediate product is placed in the appropriate positions, to the left of the earlier results. Finally, all the intermediate products are added together, to get the final result. The block design of the shift-and-add multiplication technique is illustrated in the figure below.



For **division**, there will be used the same algorithm as in the laboratory work mentioned above:
In any division operation, we have the first operand called *dividend* (X), the second operand called *divisor* (Y) and the results are the *quotient* (Q) and *remainder* (R). The mathematical expression is: X = Q * Y + R, R < Y.
The algorithm for decimal division is explained below:
1. Choose a digit and subtract the product between this digit and the divisor from the partial remainder.
2. If the result is smaller than the divisor, the digit was chosen correctly.
3. Otherwise, choose another digit and repeat the subtraction.

The binary division algorithm is based on repeated subtractions of the divisor Y from the partial remainder R, but are executed only if Y ≤ R, which results in a quotient digit of 1 (otherwise is 0).
The basic block design of the division operation is illustrated below in figure below:



3. Analysis

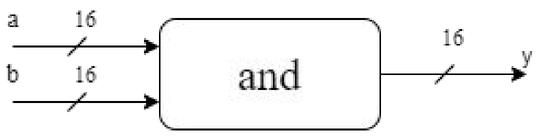
3.1 Logical operations

For the implementation of the logical operations, there has been included in the project the library "**IEEE.std_logic_1164.all**" which allowed the programmer to work on bits. Hence, for each of the following operation, there will be presented the function used for computing the required operation, the block diagram and additional explanation:

➢ **AND**

Function: "and" (l, r : std_logic_vector) RETURN std_logic_vector;

Block diagram:



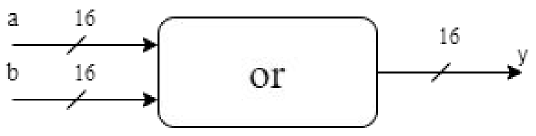
Explanation: The system applies the **"and"** function on numbers **a** and **b** which are on 16 bits in our case and returns a new number, **y**. The function is applied to each pair of individual bits according to the following truth table:

Input		**Output**
a	**b**	**y**
0	0	0
0	1	0
1	0	0
1	1	1

➢ **OR**

Function: "or" (l, r : std_logic_vector) RETURN std_logic_vector;

Block diagram:



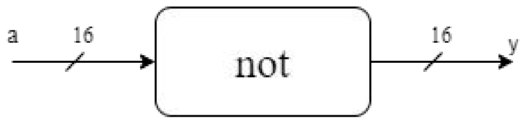
Explanation: The system applies the **"or"** function on numbers **a** and **b** which are on 16 bits in our case and returns a new number, **y**. The function is applied to each pair of individual bits according to the following truth table:

Input		**Output**
a	**b**	**y**
0	0	0
0	1	1
1	0	1
1	1	1

➢ **NOT**

Function: "not" (l : std_logic_vector) RETURN std_logic_vector;

Block diagram:

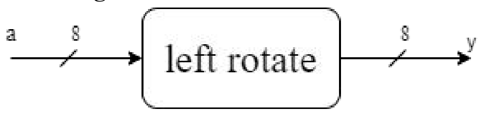


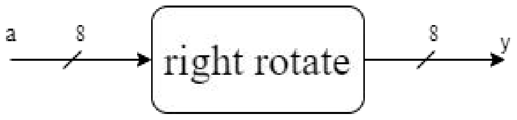
Explanation: The system applies the **"or"** function on number **a** which is on 16 bits in our case and returns a new number, **y**. The function is applied to individual bit according to the following truth table:

Input	Output
a	**y**
0	1
1	0

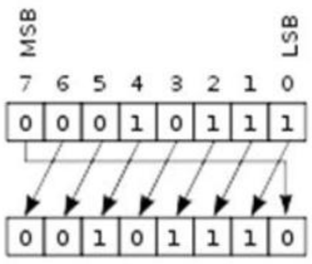
➢ **LEFT/RIGHT ROTATION**

Block diagrams:

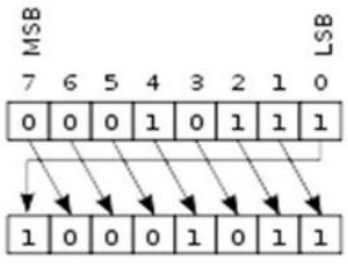




Explanation: The system rotates to the left/right the bits of the input number **a** and returns a new number, **y**, according to the following example:



Left Rotation



Right Rotation

Where:
- MSB is the Most Significant Bit;
- LSB is the Least Significant Bit;

In the *left rotation*, bits from index 0 to 6 are shifted one position to the left and the MSB becomes the LBS.
In the *right rotation*, bits from index 1 to 7 are shifted one position to the right and the LSB becomes the MSB.

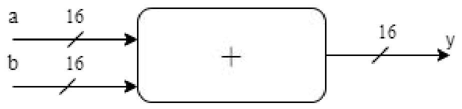
3.2 Arithmetic operations

For the implementation of arithmetic operation, there has been included in the project the library “**IEEE.std_logic_signed.all**” which provides signed numerical computation on type *std_logic_vector*. Hence, for each of the following operation, there will be presented the function/algorithm used for computing the required operation, the block diagram/flowchart and additional explanation:

➢ ADDITION

Function: "+" (L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
return STD_LOGIC_VECTOR;

Block diagram:

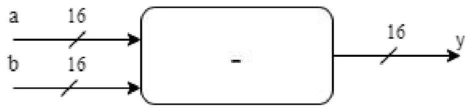


Explanation: The system applies the **“+”** function on numbers **a** and **b** which are on 16 bits in our case and returns a new number, **y**. The input numbers and the result are represented in two’s complement.

➢ SUBTRACTION

Function: "-" (L: STD_LOGIC_VECTOR; R: STD_LOGIC_VECTOR)
return STD_LOGIC_VECTOR;

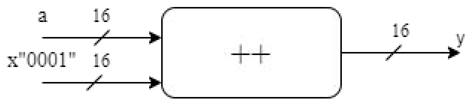
Block diagram:



Explanation: The system applies the **“-”** function on numbers **a** and **b** which are on 16 bits in our case and returns a new number, **y**. The input numbers and the result are represented in two’s complement.

➢ INCREMENTATION

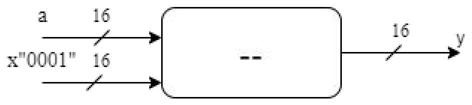
Block diagram:



Explanation: The system applies the **“++”** function on numbers **a** and x”0001” which is the hexadecimal representation of the decimal number 1. In simple terms, there is performed a simple addition by 1.

➢ DECREMENTATION

Block diagram:



Explanation: The system applies the **“--”** function on numbers **a** and x”0001” which is the hexadecimal representation of the decimal number 1. In simple terms, there is performed a simple subtraction by 1.

➢ MULTIPLICATION

There has been used the *Shift-and-Add Multiplication* algorithm because it can be easily modified to perform division as well. It is similar to the multiplication performed by paper and pencil. This method adds the multiplicand X to itself Y times, where Y denotes the multiplier. To multiply two numbers by paper and pencil, the algorithm is to take the digits of the multiplier one at a time from right to left, multiplying the multiplicand by a single digit of the multiplier and placing the intermediate product in the appropriate positions to the left of the earlier results.

As an example, consider the multiplication of two unsigned 4-bit numbers, 8 (1000) and 9 (1001).

```
Multiplicand                1000 ×
Multiplier                  1001
                            1000
                         0000
                        0000
                       1000
Product                1001000
```

In the case of binary multiplication, since the digits are 0 and 1, each step of the multiplication is simple. If the multiplier digit is 1, a copy of the multiplicand (1 × multiplicand) is placed in the proper positions; if the multiplier digit is 0, a number of 0 digits (0 × multiplicand) are placed in the proper positions.

- Consider the multiplication of positive numbers. The first version of the multiplier circuit, which implements the shift-and-add multiplication method for two n-bit numbers, is shown in *Figure 1*:

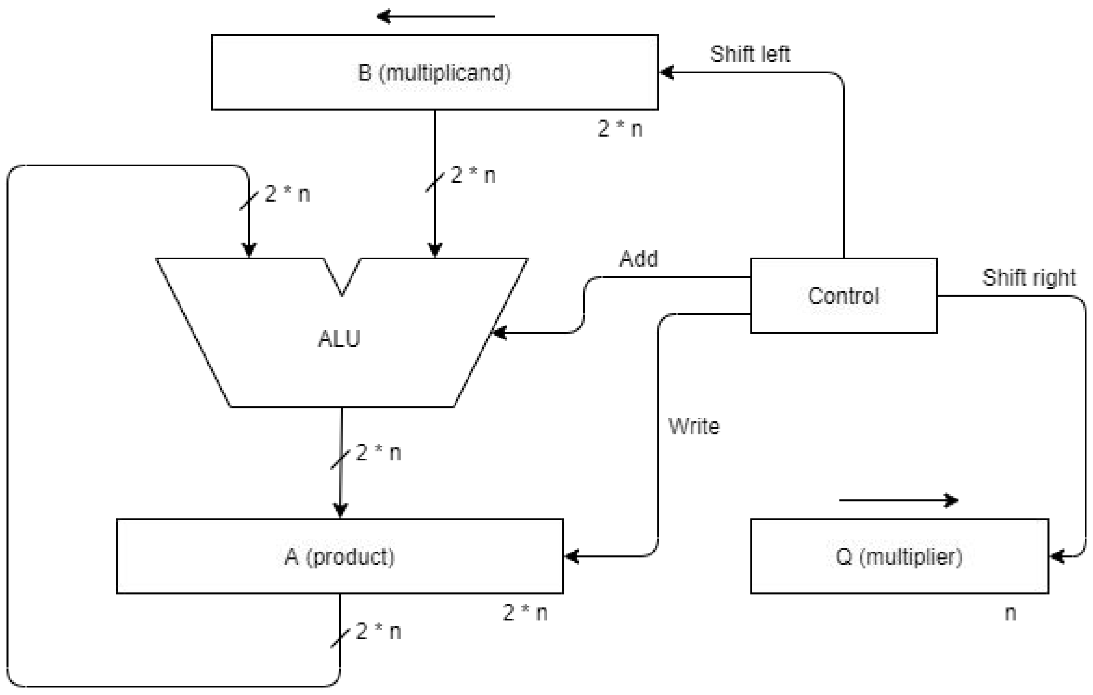


Figure 1. Block design of the shift-and-add multiplication technique

- The 2n-bit product register (A) is initialized to 0. Since the basic algorithm shifts the multiplicand register (B) left one position each step to align the multiplicand with the sum being accumulated in the product register, we use a 2n-bit multiplicand register with the multiplicand placed in the right half of the register and with 0 in the left half.

\

The flowchart from *Figure 2* presents the basic steps needed for multiplication:

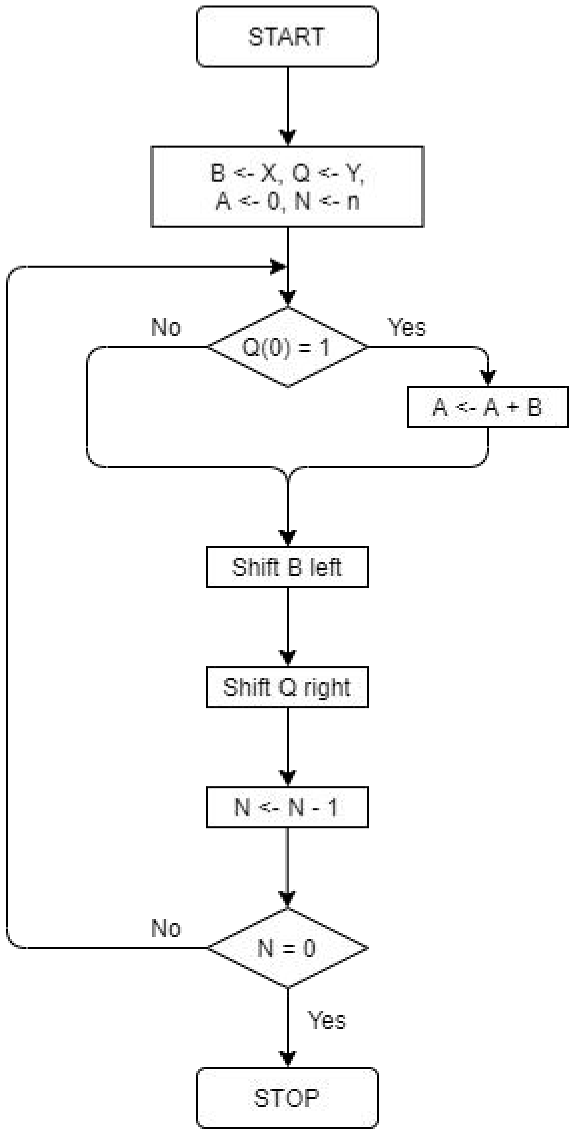


Figure 2. Flowchart of the shift-and-add multiplication algorithm

The algorithm starts by loading the multiplicand into the B register, loading the multiplier into the Q register, and initializing the A register to 0. The counter N is initialized to n. The least significant bit of the multiplier register (Q0) determines whether the multiplicand is added to the product register. The left shift of the multiplicand has the effect of shifting the intermediate products to the left, just as when multiplying by paper and pencil. The right shift of the multiplier prepares the next bit of the multiplier to examine in the following iteration.

➢ DIVISION

In any division operation, we have the first operand called *dividend* (X), the second operand called *divisor* (Y) and the results are the *quotient* (Q) and *remainder* (R). The mathematical expression is:
X = Q * Y + R, R < Y.
The algorithm for decimal division is explained below:

1. Choose a digit and subtract the product between this digit and the divisor from the partial remainder.
2. If the result is smaller than the divisor, the digit was chosen correctly.
3. Otherwise, choose another digit and repeat the subtraction.

The binary division algorithm is based on repeated subtractions of the divisor Y from the partial remainder R, but are executed only if Y ≤ R, which results in a quotient digit of 1 (otherwise is 0). The basic block design of the division operation is illustrated in the *Figure 3*.

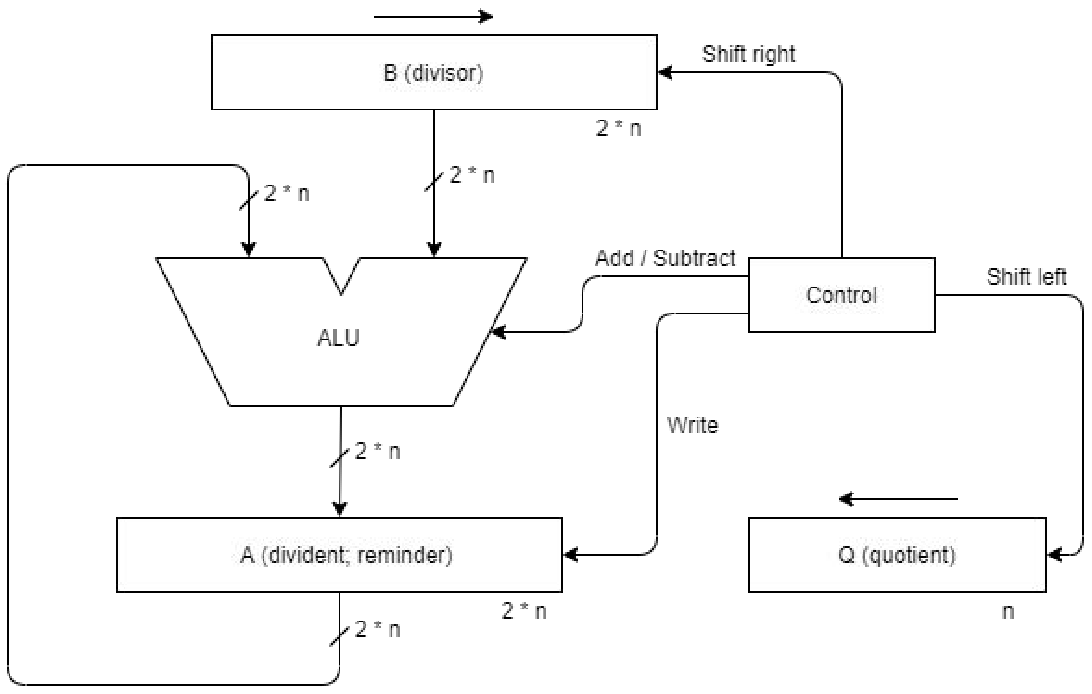


Figure 3. Block design of the division technique

The flowchart in the *Figure 4* presents the basic steps needed for division:

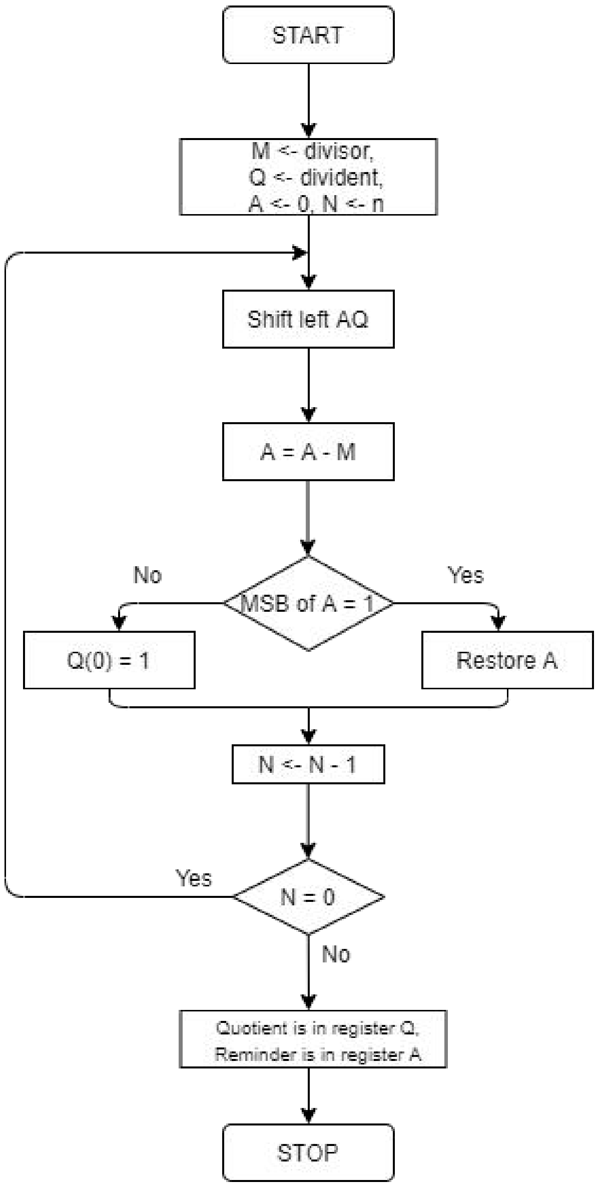


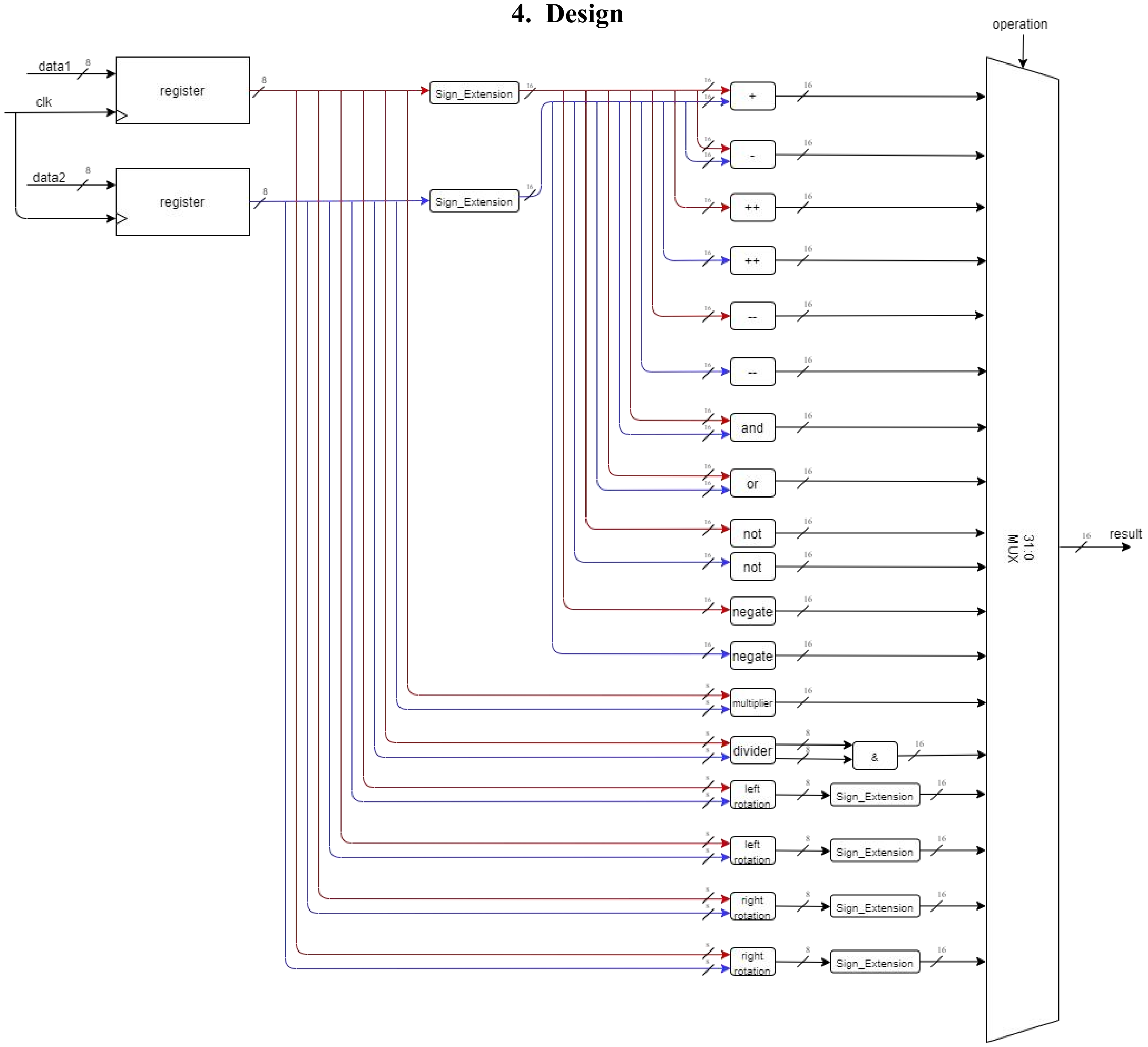
Figure 4. Flowchart of the division algorithm

Step-1: First, the registers are initialized with corresponding values (*Q* = Dividend, *M* = Divisor, *A* = 0, *n* = number of bits in dividend).
Step-2: Then the content of register *A* and *Q* is shifted left as if they are a single unit.
Step-3: Then content of register *M* is subtracted from *A* and result is stored in *A*.
Step-4: Then the most significant bit of the *A* is checked if it is 0 the least significant bit of *Q* is set to 1 otherwise if it is 1 the least significant bit of *Q* is set to 0 and value of register *A* is restored i.e. the value of *A* before the subtraction with *M*.
Step-5: The value of counter *n* is decremented.
Step-6: If the value of *n* becomes zero we get of the loop otherwise we repeat from *Step-2*.
Step-7: Finally, the register *Q* contain the quotient and *A* contain remainder.

➢ NEGATION

The following method will be used:
- negate all the bits of the number (using the logical operation **“not”**);
- add 1 to the negated number (using the arithmetic operation **“and”**).

4. Design



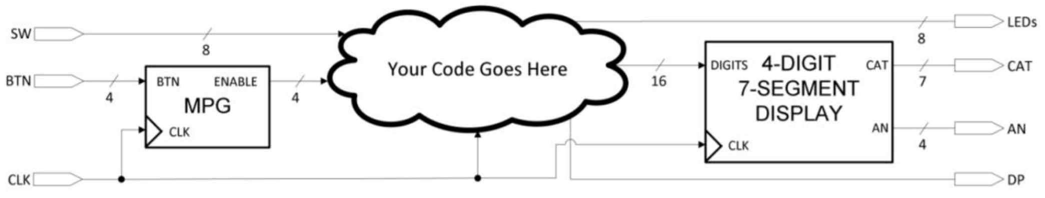
The data flow and through each component will be explained from left to right:

- *data1* and *data2* represent the first number and the second number given by the user. They are on 8 bits and internally, the numbers are interpreted in the two's complement format;
- *operation* represents the number of the operation (see table below) which will be performed on the number(s). It is represented on 5 bits;
- *result* is the final result after performing the operation and it is represented on 16 bits.

- For the operations *addition, subtraction, incrementation, decrementation, and, or, not, negate* there has been used *data1* and *data2* extended to 16 bits, which means that the MSB has been concatenated to the left of the initial number 8 times.
- For the *multiplication*, the component already takes as input two numbers on 8 bits and returns a result on 16 bits. Hence, there is no need to change the input data or the result.
- For *division*, the component takes as input two numbers on 8 bits and returns a quotient and a reminder (both on 8 bits). Since all the other operations return a single result, the quotient and the reminder have been concatenated into a single result.
- For *left/right rotation*, there has been given as input the original data input. Since the final result is set to 16 bits for the simplicity of the project, the rotated number will be extended to 16 bits.

Operation	**Meaning**
00000	addition
00001	subtraction
00010	increment first number
00011	increment second number
00100	decrement first number
00101	decrement second number
00110	AND
00111	OR
01000	NOT first number
01001	NOT second number
01010	negate first number
01011	negate second number
01100	right rotate of first number
01101	right rotate of first number
01110	left rotate of first number
01111	left rotate of second number
10000	multiplication
10001	division
others	0

For the **implementation on the board**, there has been designed the following scheme:



The *MPG* represents a frequency divider (*'debouncer'*). It receives as input the intern clock of the board of 100MHz frequency and returns a clock signal of 2000Hz due to the internal functionality which will be explained later.
The *4-digit 7-segment display* takes care of the translation of the internal output of the system to the physical representation on the board in decimal.

5. Implementation

Due to the requirement to implement different components for the multiplication and addition operation, the final system was designed both structural and behavioral.

The structural part presented below is described in the *case* structure. The libraries presented previously are the main reason most of the operations could be implemented in one line of code.

```vhdl
process(data_1, data_2, operation, mul_result, div_quotient, div_reminder)
    variable data_1_extended, data_2_extended : std_logic_vector(15 downto 0);
begin
    data_1_mul <= data_1;
    data_2_mul <= data_2;
    data_1_div <= data_1;
    data_2_div <= data_2;
    data_1_extended := std_logic_vector(resize(signed(data_1), data_1_extended'length));
    data_2_extended := std_logic_vector(resize(signed(data_2), data_2_extended'length));
    case operation is
        when "00000" => result <= data_1_extended + data_2_extended; -- addition
        when "00001" => result <= data_1_extended - data_2_extended; -- subtraction
        when "00010" => result <= data_1_extended + x"01"; -- increment first number
        when "00011" => result <= data_2_extended + x"01"; -- increment second number
        when "00100" => result <= data_1_extended - x"01"; -- decrement first number
        when "00101" => result <= data_2_extended - x"01"; -- decrement second number
        when "00110" => result <= data_1_extended and data_2_extended; -- AND
        when "00111" => result <= data_1_extended or data_2_extended; -- OR
        when "01000" => result <= not data_1_extended; -- NOT first number
        when "01001" => result <= not data_2_extended; -- NOT second number
        when "01010" => result <= (not data_1_extended) + x"01"; -- negate first number
        when "01011" => result <= (not data_2_extended) + x"01"; -- negate second number
        when "01100" => result <= std_logic_vector(resize(signed((data_1(0) & data_1(7 downto 1))), 16)); -- right rotate of first number
        when "01101" => result <= std_logic_vector(resize(signed((data_2(0) & data_2(7 downto 1))), 16)); -- right rotate of second number
        when "01110" => result <= std_logic_vector(resize(signed((data_1(6 downto 0) & data_1(7))), 16)); -- left rotate of first number
        when "01111" => result <= std_logic_vector(resize(signed((data_2(6 downto 0) & data_2(7))), 16)); -- left rotate of second number
        when "10000" => result <= mul_result; -- first number * second number
        when "10001" => result <= div_quotient & div_reminder; -- first number / second number = quotient & reminder
        when others => result <= (others => '0');
    end case;
end process;
```

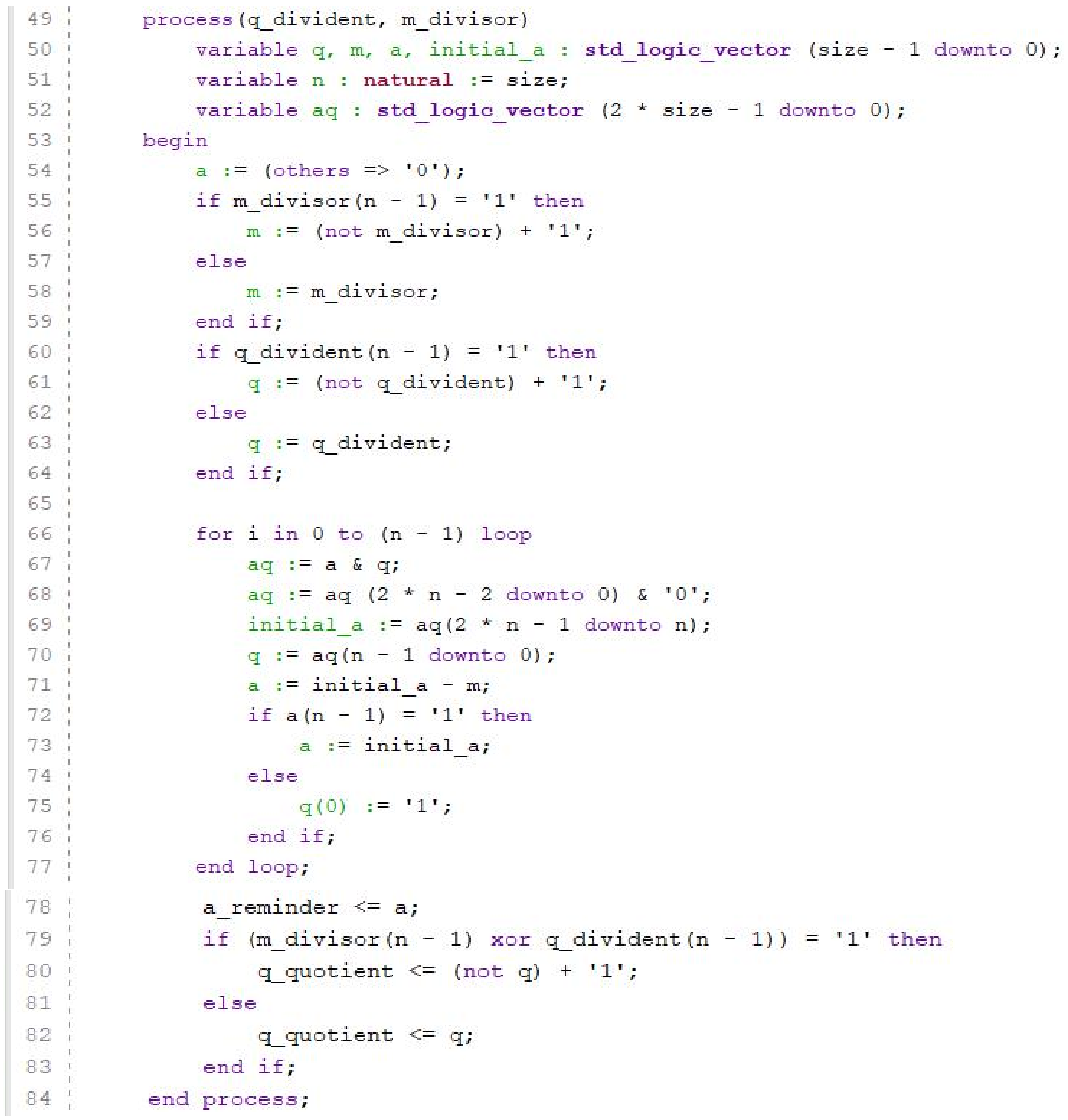
For **the behavioral part**, there has been designed individual components for multiplication and division.

The *multiplication* was implemented according to the corresponding flowchart presented previously. There is just a slight modification at the initialization of *q* and *b* variables.

Since the two's complement representation is used, the algorithm has to be modified so that the sign rules for the multiplication remains unchanged. Thus, for the case when both numbers are negative, they are changed into positive (lines 53-54) ones and the algorithm proceeds as planned. For the rest of the cases, the algorithms follows the natural flow of the steps.

```vhdl
47      process (b_port, q_multiplier)
48          variable a, b : std_logic_vector (2 * size - 1 downto 0);
49          variable q : std_logic_vector (size - 1 downto 0);
50          variable n : natural := size;
51      begin
52          if (b_port(size - 1) = '1' and q_multiplier(size - 1) = '1') then
53              b := std_logic_vector(resize(signed((not b_port) + x"01"), b'length));
54              q := (not q_multiplier) + x"01";
55          else
56              b := std_logic_vector(resize(signed(b_port), b'length));
57              q := q_multiplier;
58          end if;
59          a := (others => '0');
60          for i in 0 to (n - 1) loop
61              if q(0) = '1' then
62                  a := a + b;
63              end if;
64              b := b(2 * n - 2 downto 0) & '0';
65              q := '0' & q(n - 1 downto 1);
66          end loop;
67          a_port <= a;
68      end process;
```

The *division* was implemented according to the corresponding flowchart presented previously as well. There is just a slight modification at the initialization of *q* and *m* variables.
Since the two's complement representation is used, the algorithm has to be modified so that the sign rules for the division remains unchanged. Thus, for the case when the input numbers are negative, they are changed into positive ones (lines 56 and 61) and the algorithm proceeds as planned until returning the final result because, for the case when one of the number was negative, the quotient has to be negative, so it has to be negated (lines 79-80). For the rest of the cases, the algorithms follows the natural flow of the steps.



```
49      process(q_divident, m_divisor)
50          variable q, m, a, initial_a : std_logic_vector (size - 1 downto 0);
51          variable n : natural := size;
52          variable aq : std_logic_vector (2 * size - 1 downto 0);
53      begin
54          a := (others => '0');
55          if m_divisor(n - 1) = '1' then
56              m := (not m_divisor) + '1';
57          else
58              m := m_divisor;
59          end if;
60          if q_divident(n - 1) = '1' then
61              q := (not q_divident) + '1';
62          else
63              q := q_divident;
64          end if;
65
66          for i in 0 to (n - 1) loop
67              aq := a & q;
68              aq := aq (2 * n - 2 downto 0) & '0';
69              initial_a := aq(2 * n - 1 downto n);
70              q := aq(n - 1 downto 0);
71              a := initial_a - m;
72              if a(n - 1) = '1' then
73                  a := initial_a;
74              else
75                  q(0) := '1';
76              end if;
77          end loop;
78          a_reminder <= a;
79          if (m_divisor(n - 1) xor q_divident(n - 1)) = '1' then
80              q_quotient <= (not q) + '1';
81          else
82              q_quotient <= q;
83          end if;
84      end process;
```

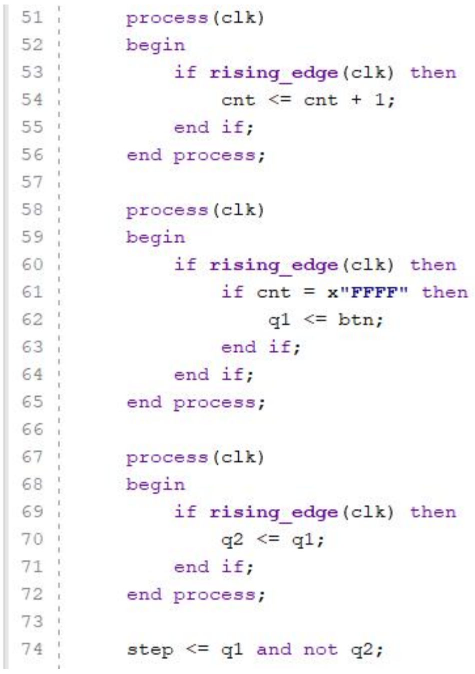
Implementation on the board requires the *MPG* and *SSD* components:

The **MPG** works in the following way:

- The *cnt* signal is incremented at each clock pulse of the internal clock (*clk* in our case).

- When *cnt* reaches the maximum value possible (namely “ffff” in hexadecimal representation), the logical value ‘1’ will be returned.

=> Thus, the frequency of the internal clock of the board is divided into a much smaller one and the user can easily interact with the board and understand the functionality of the project.

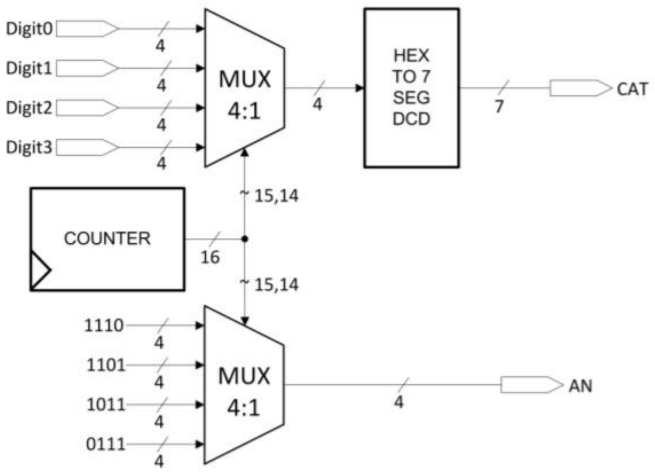


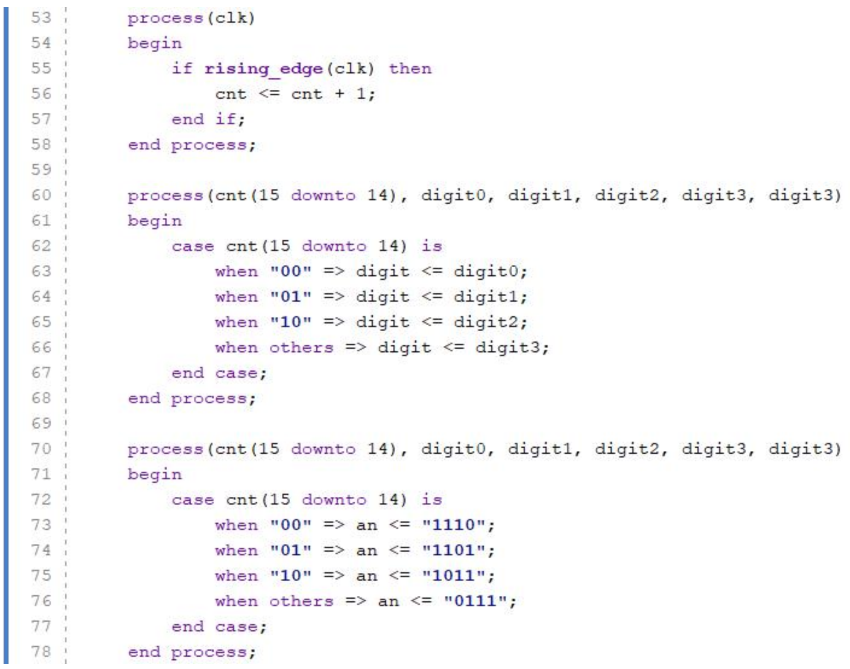
```vhdl
    process(clk)
    begin
        if rising_edge(clk) then
            cnt <= cnt + 1;
        end if;
    end process;

    process(clk)
    begin
        if rising_edge(clk) then
            if cnt = x"FFFF" then
                q1 <= btn;
            end if;
        end if;
    end process;

    process(clk)
    begin
        if rising_edge(clk) then
            q2 <= q1;
        end if;
    end process;

    step <= q1 and not q2;
```

The **SSD** works according to the following scheme:

- The inputs are 4 4-bit signals (*Digit0, Digit1, Digit2, Digit3*) and the clock signal.

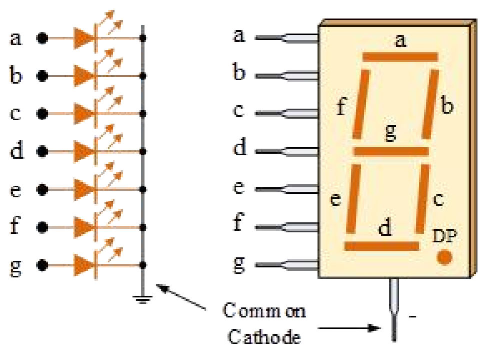
- The outputs are represented by the anode (*an*) and cathode (*cat*) signals (active low).

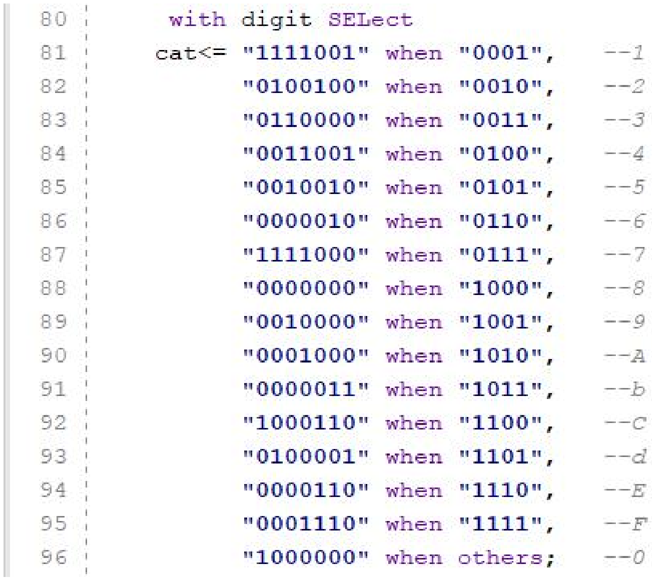




```
    process(clk)
    begin
        if rising_edge(clk) then
            cnt <= cnt + 1;
        end if;
    end process;

    process(cnt(15 downto 14), digit0, digit1, digit2, digit3, digit3)
    begin
        case cnt(15 downto 14) is
            when "00" => digit <= digit0;
            when "01" => digit <= digit1;
            when "10" => digit <= digit2;
            when others => digit <= digit3;
        end case;
    end process;

    process(cnt(15 downto 14), digit0, digit1, digit2, digit3, digit3)
    begin
        case cnt(15 downto 14) is
            when "00" => an <= "1110";
            when "01" => an <= "1101";
            when "10" => an <= "1011";
            when others => an <= "0111";
        end case;
    end process;
```

The final step was to make the transition from the internal representation of the digits into a human understandable representation on the board. This step was implemented using a *with - select* structure:

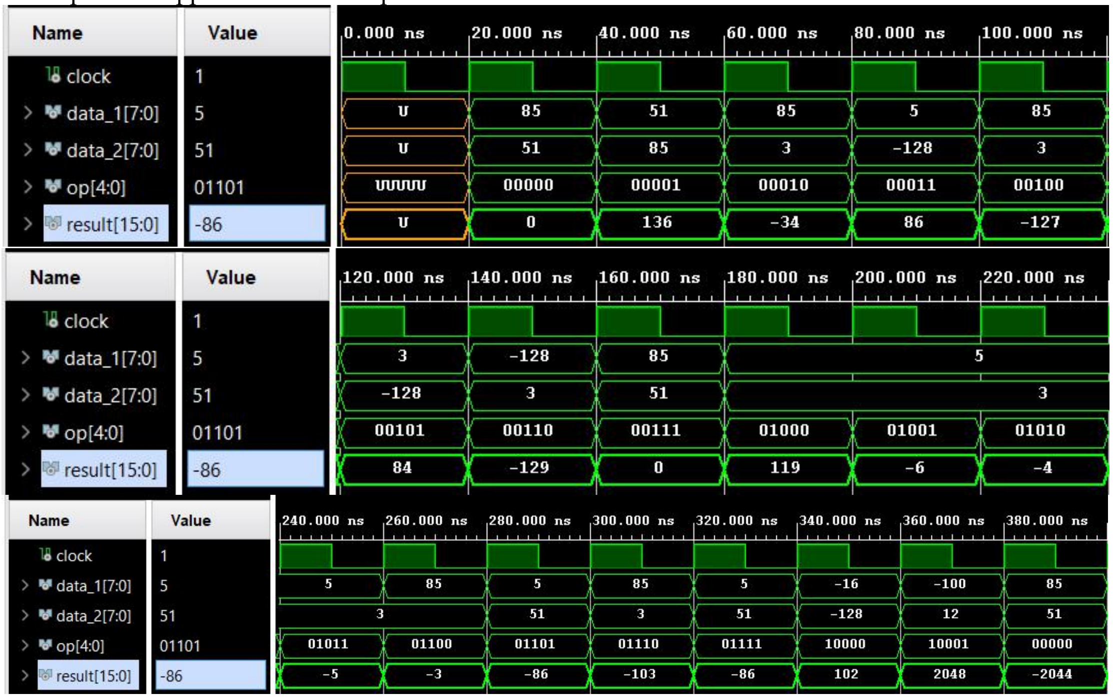




```
    with digit SELect
   cat<= "1111001" when "0001",   --1
         "0100100" when "0010",   --2
         "0110000" when "0011",   --3
         "0011001" when "0100",   --4
         "0010010" when "0101",   --5
         "0000010" when "0110",   --6
         "1111000" when "0111",   --7
         "0000000" when "1000",   --8
         "0010000" when "1001",   --9
         "0001000" when "1010",   --A
         "0000011" when "1011",   --b
         "1000110" when "1100",   --C
         "0100001" when "1101",   --d
         "0000110" when "1110",   --E
         "0001110" when "1111",   --F
         "1000000" when others;   --0
```

6. Testing and validation

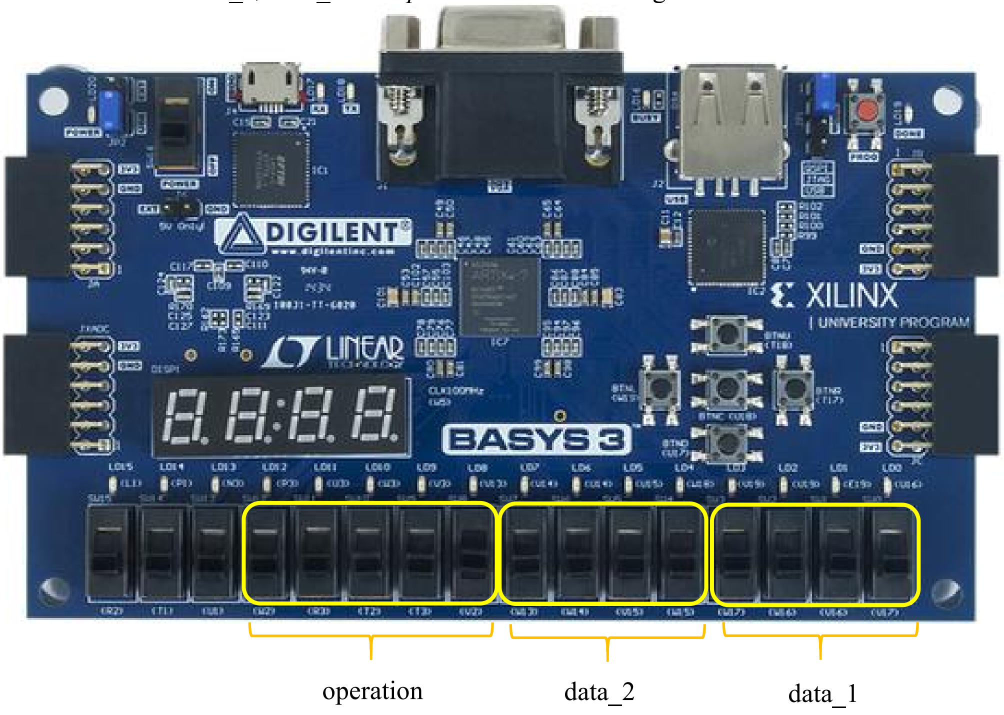
There has been run the following test cases using the simulator available in Vivado:
The *data_1, data_2* and *result* are represented in signed decimal and the *op* in binary. The result for each operation appear after a clock pulse.



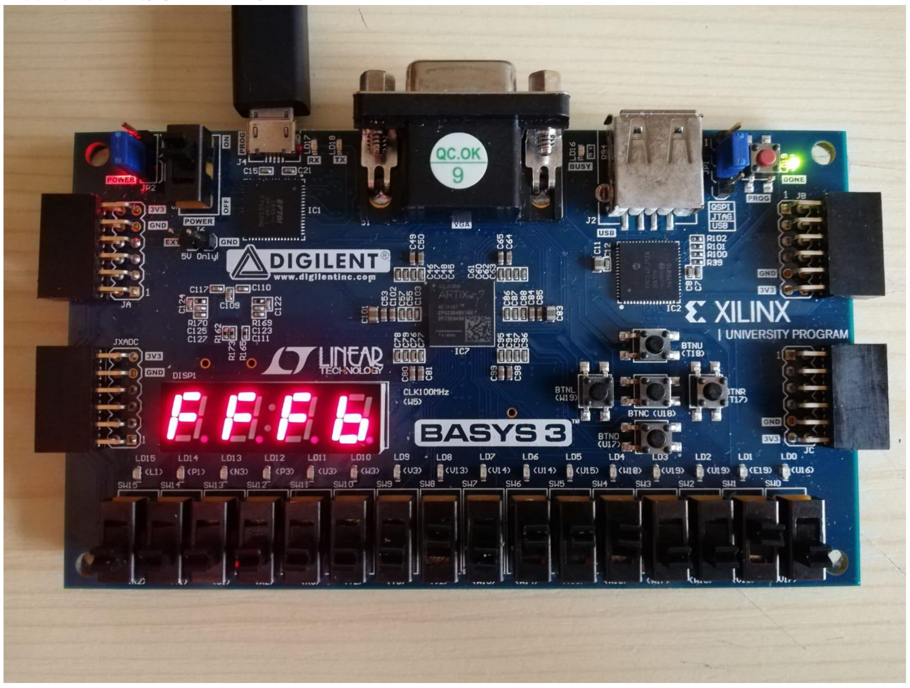
For better visibility and more details, follow the table below.

data_1	**data_2**	**operation**	**result** (after a clock pulse)	**Explanation**
85	51	data_1 + data_2	136	-
51	85	data_1 - data_2	-34	-
85	3	increment data_1	86	-
5	-128	increment data_2	-127	-
85	3	decrement data_1	84	-
3	-128	decrement data_2	-129	-
-128	3	data_1 AND data_2	0	10000000 and 00000011 = …00000000
85	51	data_1 OR data_2	119	01010101 or 00110011 = …01110111
5	3	NOT data_1	-6	-
5	3	NOT data_2	-4	-
5	3	Negate data_1	-5	-
5	3	Negate data_2	-3	-
85	3	right rotate of data_1	-86	01010101 -> …10101010
5	51	right rotate of data_2	-103	00110011 -> …10011001
85	3	left rotate of data_1	-86	01010101 -> …10101010
5	51	left rotate of data_2	102	00110011 -> …01100110
-16	-128	data_1 A * data_2	2048	-
-100	12	data_1 / data_2	-2044	11111000 00000100 -> q = -8; r = 4

For the **implementation on the board**, a couple of operations for each operation were tested and presented below. The *data_1, data_2* and *op* will be accessed using the switches.

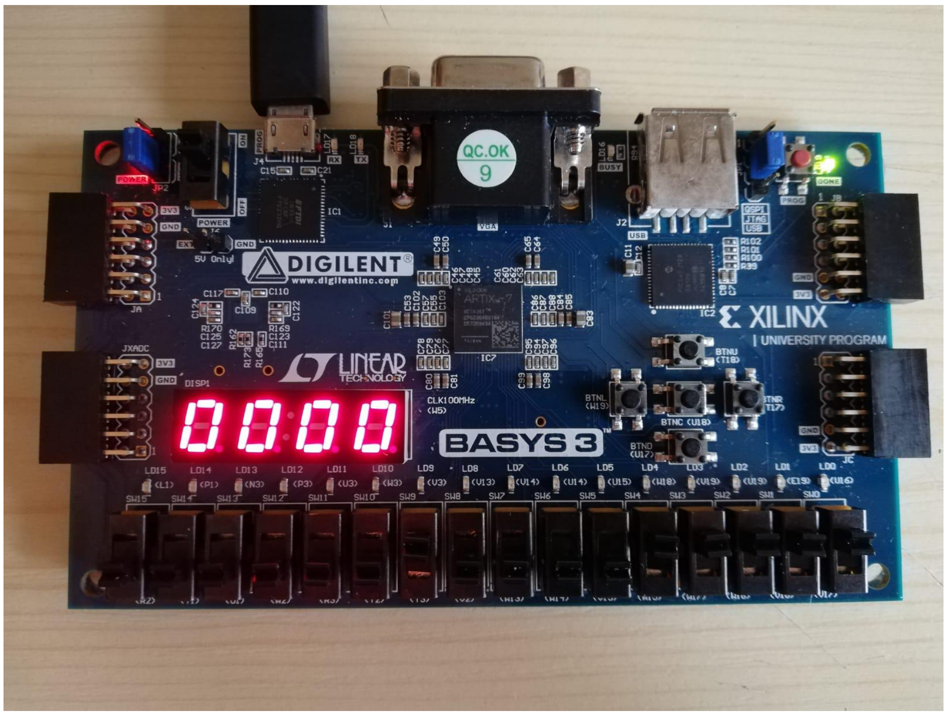


Test case 1: SUBTRACT



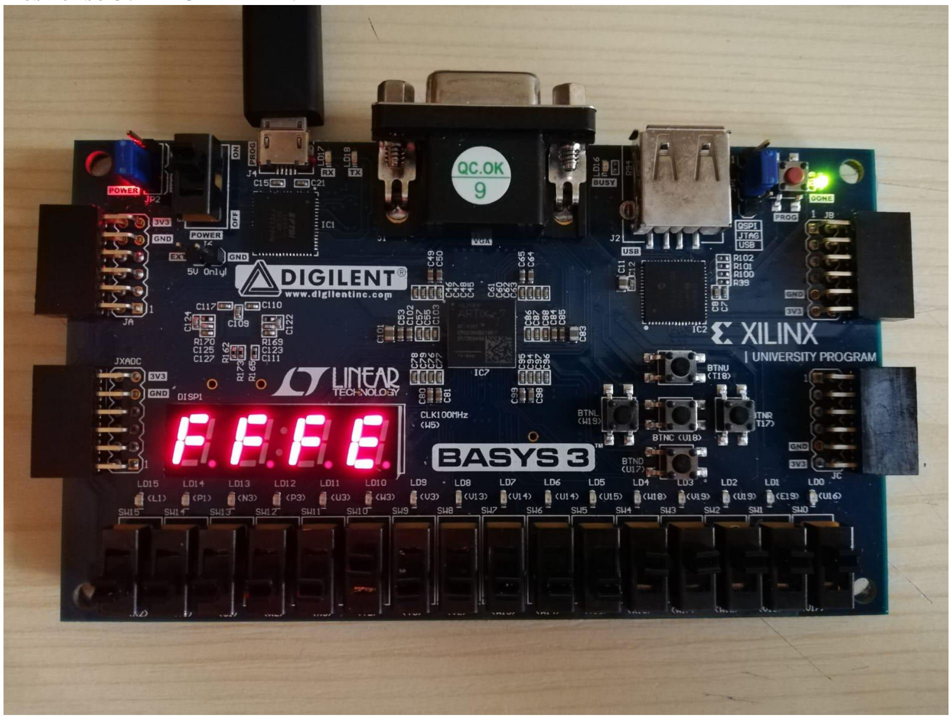
Data_1: B"0010" (decimal: 2)
Data_2: Data_2: B"0111" (decimal: 7)
Operation: 2 - 7 = -5 (x"FFFb")

Test case 2: INCREMENT



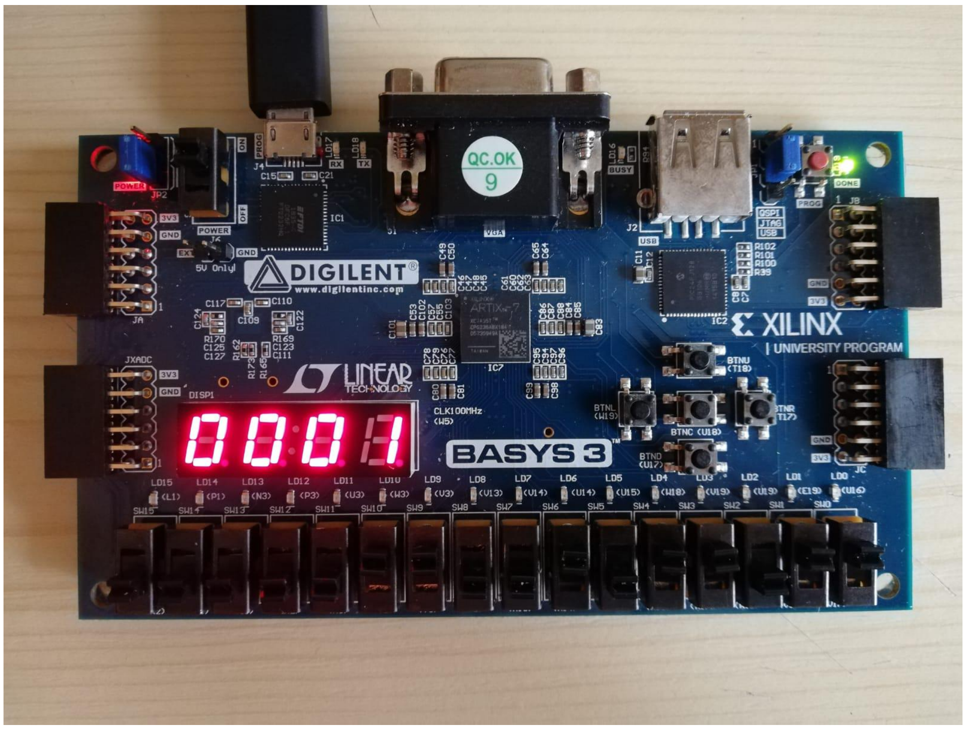
Data_1: B”1111” (decimal: -1)
Data_2: B”0001” (decimal: 1)
Operation: -1 +1 = 0 (x”0000”)

Test case 3: DECREMENT



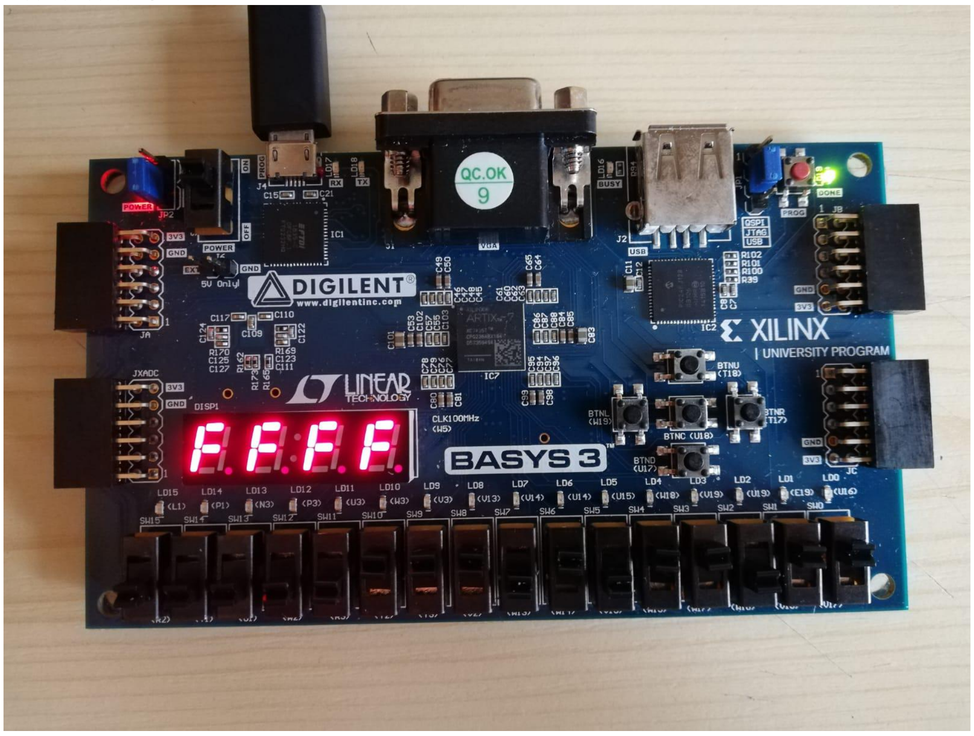
Data_1: B”1111” (decimal: -1)
Data_2: B”0111” (decimal: 7)
Operation: -1 -1 = -2 (x”FFFE”)

Test case 4: AND



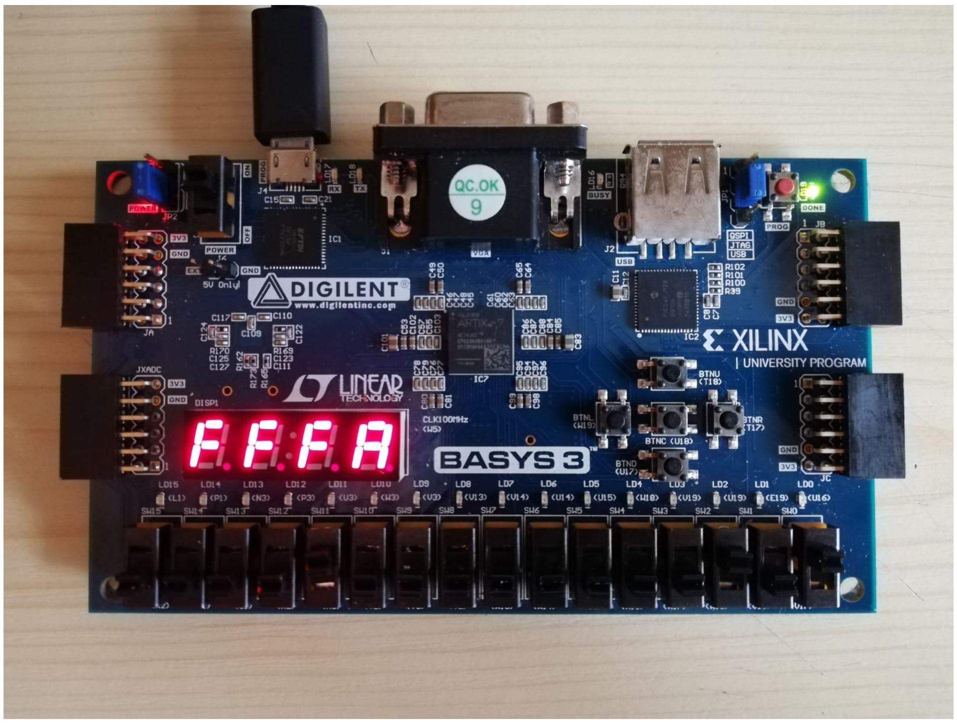
Data_1: B”1011” (decimal: -5)
Data_2: B”0101” (decimal: 5)
Operation: 1011 and 0101 = 0001 (x”0001”)

Test case 5: OR



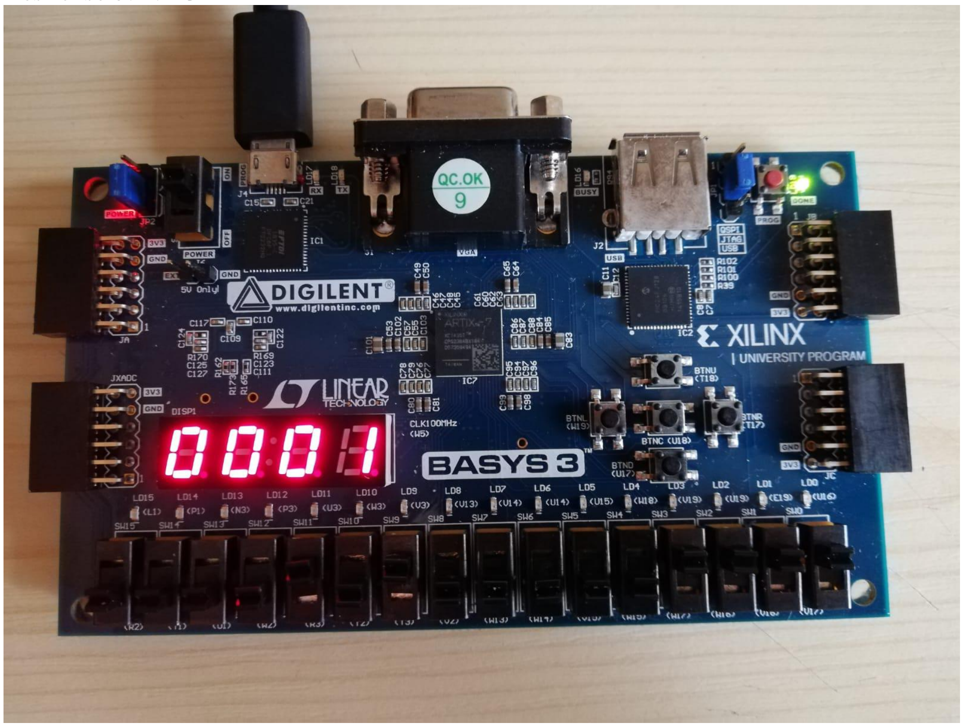
Data_1: B”1011” (decimal: -5)
Data_2: B”0101” (decimal: 5)
Operation: 1011 or 0101 = 1111 (x”FFFF”)

Test case 6: NOT



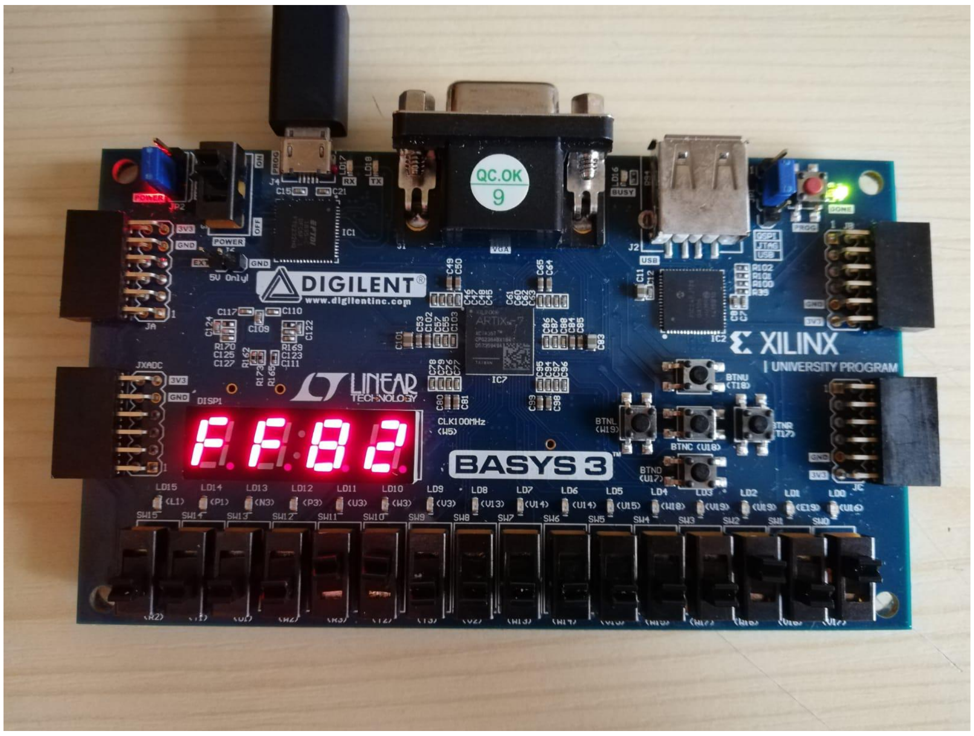
Data_1: B”0101” (decimal: 5)
Data_2: B”0000” (decimal: 0)
Operation: not 0101 = 1010 (x”FFFA”)

Test case 7: NEGATE



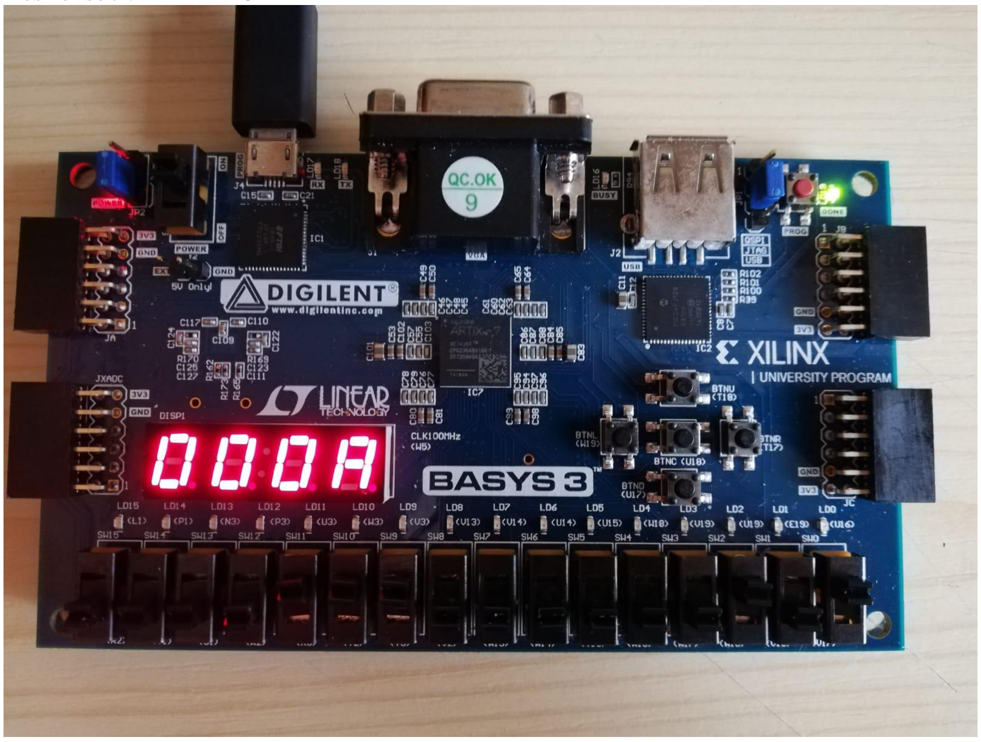
Data_1: B”1111” (decimal: -1)
Data_2: B”0000” (decimal: 0)
Operation: (not 1111) + 1 = 0001 (x”0001”)

Test case 8: RIGHT ROTATE



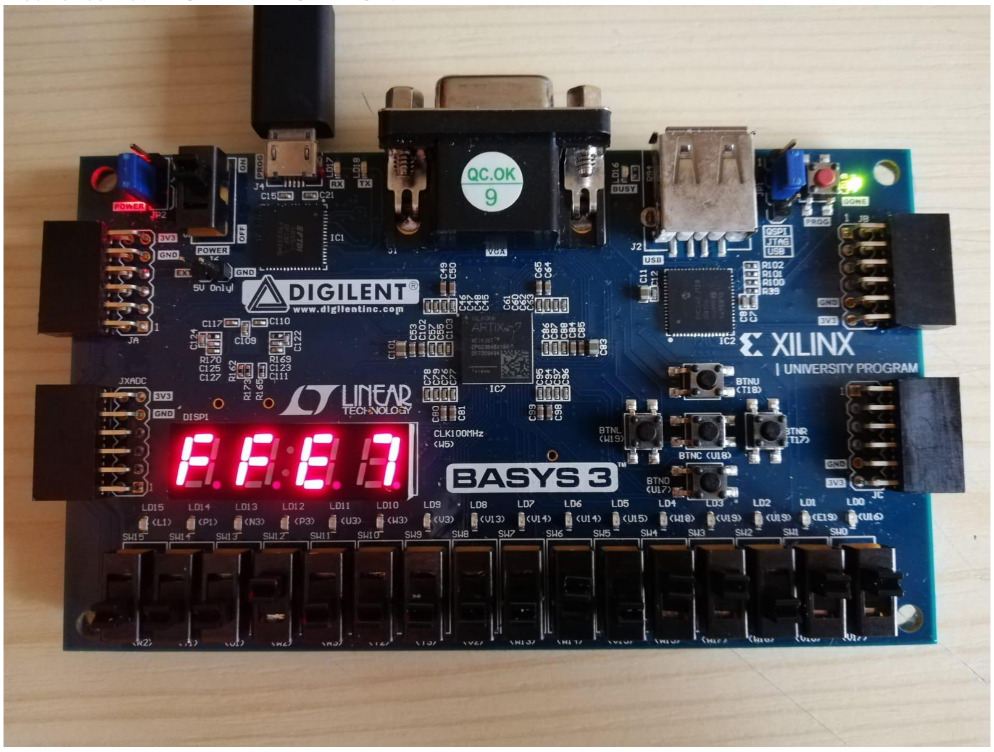
Data_1: B"00000101" (decimal: 5)
Data_2: B"0000" (decimal: 0)
Operation: 00000101 => 10000010 (x"FF82")

Test case 9: LEFT ROATATE



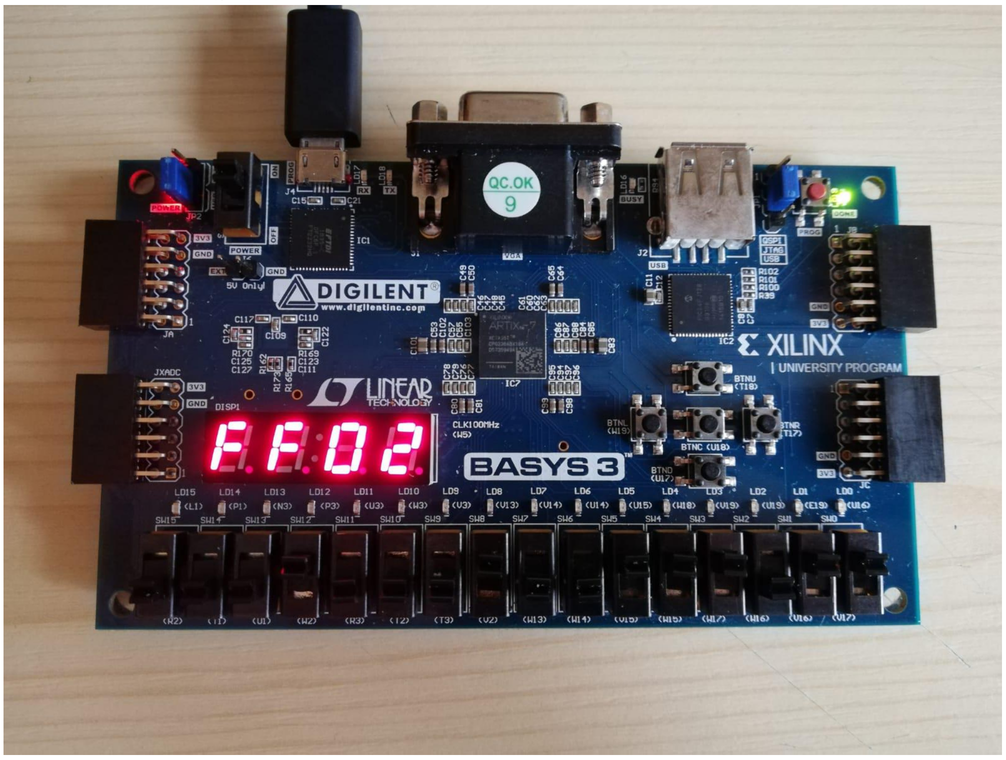
Data_1: B"00000101" (decimal: 5)
Data_2: B"0000" (decimal: 0)
Operation: 00000101 => 00001010 (x"000A")

Test case 10: MULTIPLICATION



Data_1: B”1011” (decimal: -5)
Data_2: B”0101” (decimal: 5)
Operation: (-5) * 5 = -25 (x”FFE7”)

Test case 11: DIVISION



Data_1: B”1011” (decimal: -5)
Data_2: B”0011” (decimal: 3)
Operation: (-5) / 3 = -1 rest 2 (x”FFFb”), where “FF” is the quotient and “02” is the reminder.

7. Conclusions

Each operation was implemented successfully in the simulator and on the board. Further improvements can be done to this project. For example, implementing more operations (like average value, arithmetical/logical shifts) or cutting off the redundant operations (not, negate, increment, decrement) would reduce the number of bits for the *op* variable and the data could be extended on more bits on the board. Moreover, there could have been implemented a decimal representation for the 7-segment display for better visibility.
However, the project fulfills all the requirements and the simulator provided by the Vivavo facilitates changing the visual representation of the numbers and the result according to the necessity of each operation applied.

8. Bibliography

[1] https://www.ques10.com/p/19357/draw-and-explain-the-flowchart-of-add-and-shift-me/
[2] https://www.csee.umbc.edu/portal/help/VHDL/stdpkg.html
[3] https://www.geeksforgeeks.org/restoring-division-algorithm-unsigned-integer/
[4] http://users.utcluj.ro/~negrum/wp-content/uploads/2020/01/ca/Lab02.pdf