INSTITUTO TECNOLÓGICO DE BUENOS AIRES

Assignment $N^o 1$

Electrónica 3 - 2018

Group 2

September 5, 2018

1 Excercise 1: Resolution and Range of a Fixed-Point Binary Representation

1.1 What is the Fixed-Point Binary Representation

A fixed-point number has an integer part and a fractional part separated by a decimal point with a fixed position, as shown below:

$$(IntegerPart).(FractionalPart)$$

The integer part is formed by n bits and the fractional part is formed by m bits.

$$(bit\#1 \quad bit\#2 \quad \ldots \quad bit\#n).(bit\#1 \quad bit\#2 \quad \ldots \quad bit\#m)$$

1.2 What is Resolution and Range

1.2.1 Resolution

The resolution of a number using the fixed point representation is the smallest unit that can be handled with it. Given a fixed-point number with m fractional bits, the resolution is 2^{-m}.

1.2.2 Range

The range is the difference between the biggest value that can be obtained with the fixed-point representation of a number with n bits in the integer part and with m bits in the fractional part, and the smallest number that can be represented.

1.3 Using this Program

1.3.1 Input

Three arguments must be entered through Command Line, separated by one or more spaces:

1. 1 (indicating that the numeric representation of the binary number is signed) or 0 (indicating that the representation is unsigned).
2. n: A possitive integer (indicating the number of bits that correspond to the integer part of the number, which appears before the decimal point).
3. m: A possitive integer (indicating the number of bits that correspond to the fractional part of the number, which appears after the decimal point). Important: n and m are restricted to be less than or equal to 1000.

For example: "0 1 1".

1.3.2 Output

The result of this program is the resolution and range of the number that has n digits in the integer part and m digits in the fractional part. If any of the input requirements is not respected, an error message is printed to the user.

1.4 Testing the Program

The program was tested with the following cases and worked as expected:

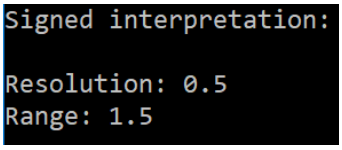


Figure 1.1: Output corresponding to the example input "0 1 1".

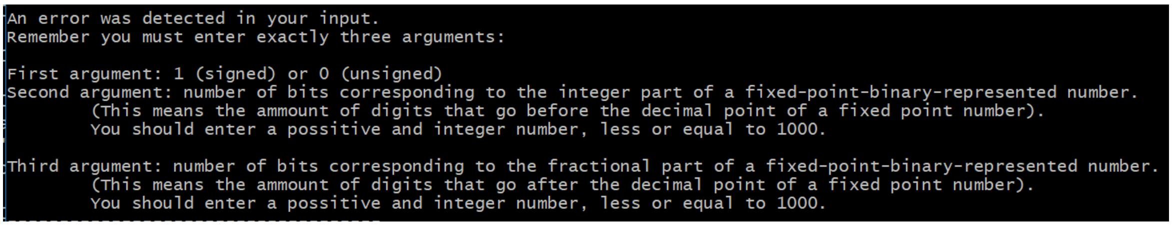


Figure 1.2: Output corresponding to an input error.

1.4.1 Inputs that should work

- 1 1 0
- 1 0 1
- 0 0 1
- 1 0 0
- 0 0 0
- 1 12 0
- 1 1   1
- 1 3   1
- 1   3   1
- 0 1000 1000
- 0 1000 00000001
- 0 000 00000001

1.4.2 Inputs that should not work and print error

- 1 1 -1
- 1 1
- 0 1000 1001
- 00 100 1
- 01 100 1
- 0 100M 1

2 Excercise 2: Simplification of a Maxterm Expression and its Corresponding Logical Circuit

Having the function in maxterms

$$f_1(A,B,C,D) = \prod(M_0, M_1, M_5, M_7, M_8, M_{10}, M_{14}, M_{15})$$

equivalent to

$$f_2(A,B,C,D) = \sum(m_2, m_3, m_4, m_6, m_9, m_{11}, m_{12}, m_{13})$$

using minterms, can be simplified by different ways and represented using logic gates.

2.1 Simplification by Boolean Algebra

The function can be simplified by using the Boolean algebra property (1):

$$(A+B).(A+\overline{B}) = A \ (1)$$

or

$$(AB) + (A\overline{B}) = A \ (2)$$

the function can be simplified using (1):

$$\begin{aligned} f_1(A,B,C,D) &= (A+B+C+D).(A+B+C+\overline{D}).(A+\overline{B}+C+\overline{D}).(A+\overline{B}+\overline{C}+\overline{D}). \\ &\quad (\overline{A}+B+C+D).(\overline{A}+B+\overline{C}+D).(\overline{A}+\overline{B}+\overline{C}+D).(\overline{A}+\overline{B}+\overline{C}+\overline{D}) \\ &= (A+B+C).(A+\overline{B}+\overline{D}).(\overline{A}+B+D).(\overline{A}+\overline{B}+\overline{C}) \end{aligned}$$

Using the Boolean algebra property (2):

$$\begin{aligned} f_2(A,B,C,D) &= (\overline{AB}C\overline{D}) + (\overline{AB}CD) + (\overline{A}B\overline{CD}) + (\overline{A}BC\overline{D}) + \\ &\quad (A\overline{BC}D) + (A\overline{B}CD) + (AB\overline{CD}) + (AB\overline{C}D) \\ &= (A\overline{B}D) + (\overline{A}B\overline{D}) + (\overline{AB}C) + (AB\overline{C}) \end{aligned}$$

2.2 Simplification by Karnaugh Map

A Karnaugh map is an easier way to simplify logic expressions when the function is too complex or too large to handle, because the Karnaugh map gives a more representative view for a faster analysis.
If the simplification is done with minterms, groups are formed by 1's that are together. The ammount of ones that can be present in a group, is a power of two. Each group is represented in the equation by ANDS between the independent variables it contains (the ones that change). The groups are put together by using an OR between them.

CD \ AB	00	01	11	10
00	0	1	1	0
01	0	0	1	1
11	1	0	0	1
10	1	1	0	0

Now grouping by colours we get that the function expressed in minterms is:

$$
\begin{aligned}
f_2(A,B,C,D) = {} & (A\overline{B}D)\ (Red) \\
& +(\overline{A}B\overline{D})\ (Blue) \\
& +(\overline{AB}C)\ (Orange) \\
& +(AB\overline{C})\ (Green)
\end{aligned}
$$

The same method can be done with maxterms; grouping zeroes (instead of ones) in powers of 2, making an AND between groups in case there is more than one, and each group is represented with an OR of its independent variables.

2.3 Logic Circuit: AND, OR and NOT

Using the logic gates AND, OR and NOT, the simplified version of the function is represented by the following figure:

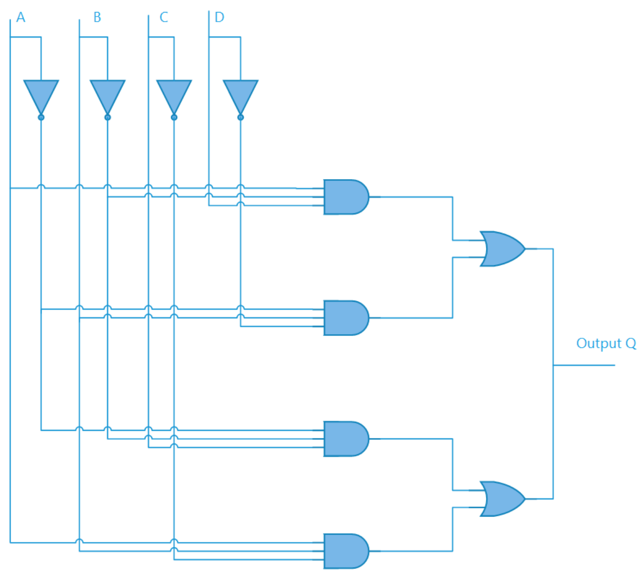


Figure 2.1: Logic circuit using AND, OR and NOT gates

2.4 Logic Circuit: NAND

All the gates used above could be equivalent to a combination of NAND or NOR gates. Therefore, the simplified function can be drawn as the next figure:

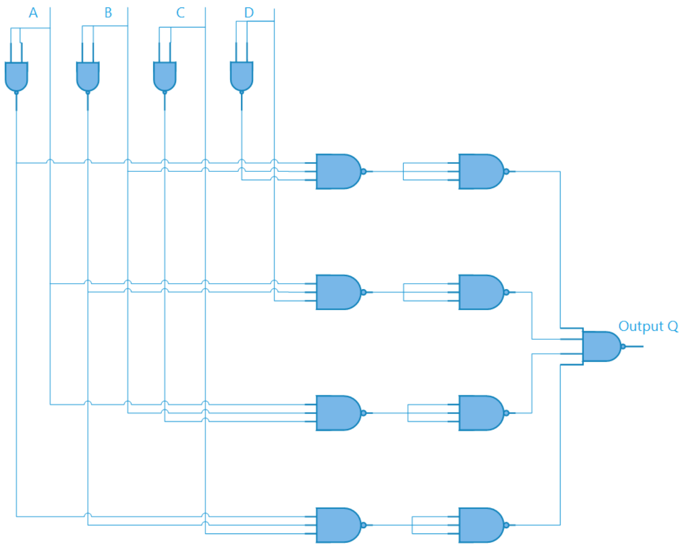


Figure 2.2: Logic circuit using NAND gates

3 Excercise 3: Four-Entry Encoder and Four-Output Demux using Verilog

Implement the following modules in Verilog:

- 4 inputs ENCODER
- 4 outputs DEMUX

3.1 4-Input ENCODER

3.1.1 Description

An encoder is an Application-Specific Integrated Circuit (ASIC) that converts information. In this case it receives a signal from a 4-bit input and returns the position of the Most Significant Bit that is currently on.

3.1.2 Code Implementation

The Code Implementation of both the Module and its testbench can be found in the directory E3TP1/src/encoder.

3.1.3 Module Tests

Testbench results can be found in Table 3.1.3

Input	Output	Value
0001	00	0
0010	01	1
0100	10	2
1000	11	3
0011	01	1
0101	10	2
1001	11	3
0110	10	2
1010	11	3
1100	11	3

Table 3.1.3 ENCODER Testbench Results

3.1.4 Conclusions

The module is working as expected, where it is taking only the Most Significant Bit as the value to be encoded.

3.2 4-output DEMUX

3.2.1 Description

A DEMUX is an ASIC which receives an input signal and a selector signal. The selector signal determines through which output port the input signal is sent.

3.2.2 Code Implementation

The implemented code for the DEMUX and its testbench can be found in the directory E3TP1/src/demux.

3.2.3 Module Tests

Input	Selector	Out_0	Out_1	Out_2	Out_3
1	0	1	0	0	0
1	1	0	1	0	0
1	2	0	0	1	0
1	3	0	0	0	1

Table 3.2.3 DEMUX Testbench Results

3.2.4 Conclusions

The module works as expected.

4 Excercise 4

If we write every signle input bit according to the minterms, we get the following equations:

$$f_1(m_i) = m_1 + m_2 + m_3 + m_4 + m_5 + m_6 + m_7 + m_8$$

$$f_2(m_i) = m_1 + m_2 + m_3 + m_4 + m_9 + m_{10} + m_{11} + m_{12}$$

x_1	x_2	x_3	x_4	f_1	f_2	f_3	f_4
0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	1
0	0	1	0	1	1	1	0
0	0	1	1	1	1	0	1
0	1	0	0	1	1	0	0
0	1	0	1	1	0	1	1
0	1	1	0	1	0	1	0
0	1	1	1	1	0	0	1
1	0	0	0	1	0	0	0
1	0	0	1	0	1	1	1
1	0	1	0	0	1	1	0
1	0	1	1	0	1	0	1
1	1	0	0	0	1	0	0
1	1	0	1	0	0	1	1
1	1	1	0	0	0	1	0
1	1	1	1	0	0	0	1

Figure 4.1: Two's Complement truth table for 4 bits

$$f_3(m_i) = m_1 + m_2 + m_5 + m_6 + m_9 + m_{10} + m_{13} + m_{14}$$

$$f_4(m_i) = m_1 + m_3 + m_5 + m_7 + m_9 + m_{11} + m_{13} + m_{15}$$

Replacing the values of each minterm, we get the following:

$$f_1(x_1; x_2; x_3; x_4) = \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_2x_3x_4 + \bar{x}_1x_2\bar{x}_3\bar{x}_4 + \bar{x}_1x_2\bar{x}_3x_4 + \bar{x}_1x_2x_3\bar{x}_4 + \bar{x}_1x_2x_3x_4 + x_1\bar{x}_2\bar{x}_3\bar{x}_4$$

$$f_2(x_1; x_2; x_3; x_4) = \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + \bar{x}_1\bar{x}_2x_3x_4 + \bar{x}_1x_2\bar{x}_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3x_4 + x_1\bar{x}_2x_3\bar{x}_4 + x_1\bar{x}_2x_3x_4 + x_1x_2\bar{x}_3\bar{x}_4$$

$$f_3(x_1; x_2; x_3; x_4) = \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3\bar{x}_4 + \bar{x}_1x_2\bar{x}_3x_4 + \bar{x}_1x_2x_3\bar{x}_4 + x_1\bar{x}_2\bar{x}_3x_4 + x_1\bar{x}_2x_3\bar{x}_4 + x_1x_2\bar{x}_3x_4 + x_1x_2x_3\bar{x}_4$$

$$f_4(x_1; x_2; x_3; x_4) = \bar{x}_1\bar{x}_2\bar{x}_3x_4 + \bar{x}_1\bar{x}_2x_3x_4 + \bar{x}_1x_2\bar{x}_3x_4 + \bar{x}_1x_2x_3x_4 + x_1\bar{x}_2\bar{x}_3x_4 + x_1\bar{x}_2x_3x_4 + x_1x_2\bar{x}_3x_4 + x_1x_2x_3x_4$$

By simplification methods and properties, we can achieve these four formulas to describe each output bit according to the input bits:

$$f_1(x_1; x_2; x_3; x_4) = x_1\bar{x}_2\bar{x}_3\bar{x}_4 + \bar{x}_1(x_2 + x_3 + x_4)$$

$$f_2(x_1; x_2; x_3; x_4) = x_2\bar{x}_3\bar{x}_4 + \bar{x}_2(x_3 + x_4)$$

$$f_3(x_1; x_2; x_3; x_4) = x_3\bar{x}_4 + \bar{x}_3x_4$$

$$f_4(x_1; x_2; x_3; x_4) = x_4$$

The previous formulas can be represented in logic gates' graphs as shown in figures 4.2, 4.3, 4.4 and 4.5. Finally, this logic was implemented on verilog as shown in the figure 4.6 and by testing the code with test.v, we got the output shown in figure 4.7, confirming that the code was executed correctly.

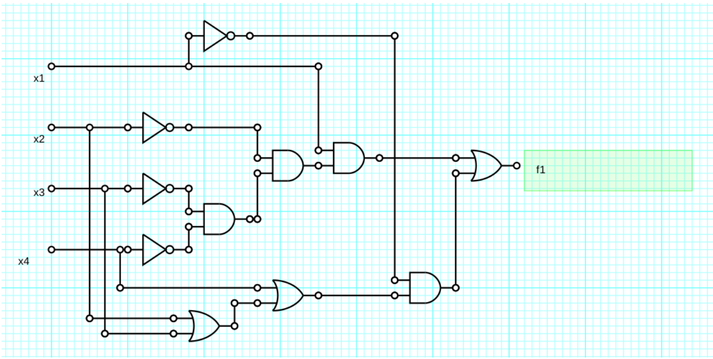


Figure 4.2: 1st Bit's logic gates graph

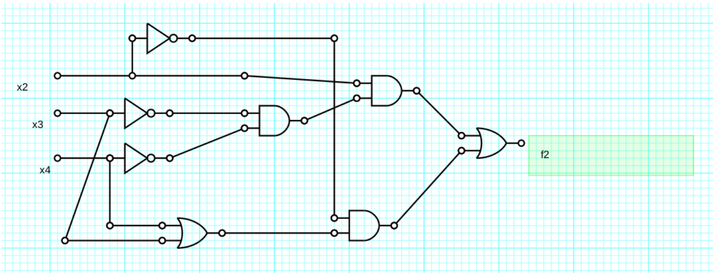


Figure 4.3: 2nd Bit's logic gates graph

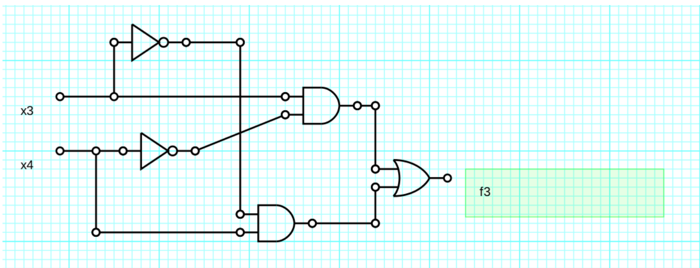


Figure 4.4: 3rd Bit's logic gates graph

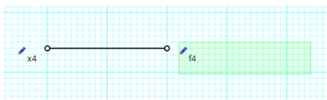


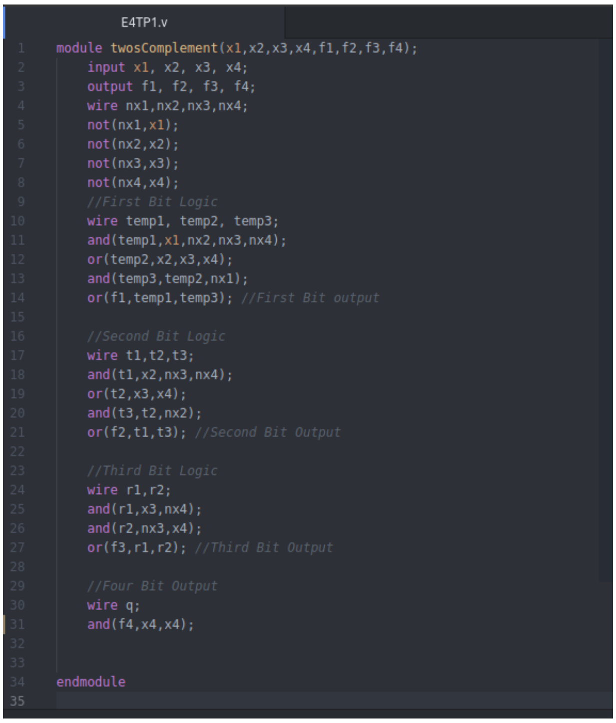
Figure 4.5: 4th Bit's logic gates graph

5 Excercise 5: BCD Format Adder

A module that receives as inputs two one-digit numbers in Binary-Coded-Decimals (BCD) format and outputs a two-digit number in BCD format, is implemented.

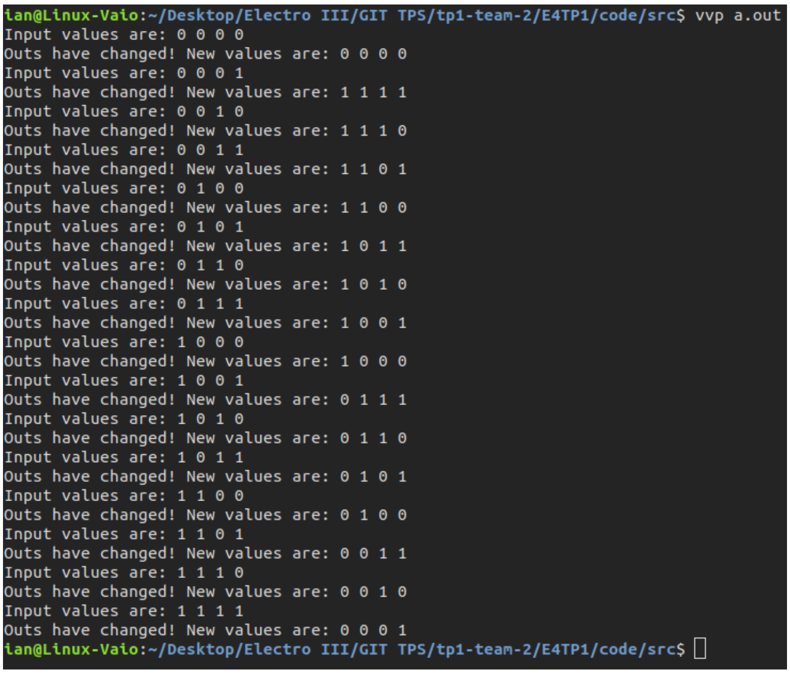
5.1 Design Considerations

- BCD digits are formed by 4 bits with a range of integer values between 0 and 9. Any value outside that range should be considered an error.
- It needs to make a simple addition. Given that the maximum value of the sum is 18, the result will be a 5-bit integer.



```verilog
E4TP1.v
module twosComplement(x1,x2,x3,x4,f1,f2,f3,f4);
    input x1, x2, x3, x4;
    output f1, f2, f3, f4;
    wire nx1,nx2,nx3,nx4;
    not(nx1,x1);
    not(nx2,x2);
    not(nx3,x3);
    not(nx4,x4);
    //First Bit Logic
    wire temp1, temp2, temp3;
    and(temp1,x1,nx2,nx3,nx4);
    or(temp2,x2,x3,x4);
    and(temp3,temp2,nx1);
    or(f1,temp1,temp3); //First Bit output

    //Second Bit Logic
    wire t1,t2,t3;
    and(t1,x2,nx3,nx4);
    or(t2,x3,x4);
    and(t3,t2,nx2);
    or(f2,t1,t3); //Second Bit Output

    //Third Bit Logic
    wire r1,r2;
    and(r1,x3,nx4);
    and(r2,nx3,x4);
    or(f3,r1,r2); //Third Bit Output

    //Four Bit Output
    wire q;
    and(f4,x4,x4);


endmodule
```

Figure 4.6: Verilog implementation



```
ian@Linux-Vaio:~/Desktop/Electro III/GIT TPS/tp1-team-2/E4TP1/code/src$ vvp a.out
Input values are: 0 0 0 0
Outs have changed! New values are: 0 0 0 0
Input values are: 0 0 0 1
Outs have changed! New values are: 1 1 1 1
Input values are: 0 0 1 0
Outs have changed! New values are: 1 1 1 0
Input values are: 0 0 1 1
Outs have changed! New values are: 1 1 0 1
Input values are: 0 1 0 0
Outs have changed! New values are: 1 1 0 0
Input values are: 0 1 0 1
Outs have changed! New values are: 1 0 1 1
Input values are: 0 1 1 0
Outs have changed! New values are: 1 0 1 0
Input values are: 0 1 1 1
Outs have changed! New values are: 1 0 0 1
Input values are: 1 0 0 0
Outs have changed! New values are: 1 0 0 0
Input values are: 1 0 0 1
Outs have changed! New values are: 0 1 1 1
Input values are: 1 0 1 0
Outs have changed! New values are: 0 1 1 0
Input values are: 1 0 1 1
Outs have changed! New values are: 0 1 0 1
Input values are: 1 1 0 0
Outs have changed! New values are: 0 1 0 0
Input values are: 1 1 0 1
Outs have changed! New values are: 0 0 1 1
Input values are: 1 1 1 0
Outs have changed! New values are: 0 0 1 0
Input values are: 1 1 1 1
Outs have changed! New values are: 0 0 0 1
ian@Linux-Vaio:~/Desktop/Electro III/GIT TPS/tp1-team-2/E4TP1/code/src$
```

Figure 4.7: Terminal's output

- The value of the sum must be returned in BCD format, so the 5-bit integer needs to be separated into 2 BCD digits.

Given these conditions, the module needs:

- 2 4-bit input ports
- 2 4-bit output ports
- 1 ERROR register

5.2 Code Implementation

The BCD Adder was formed by the following modules:

- 2 BCD format "filters"
- 1 4-bit numbers' adder
- 1 BCD format "decoder"

The code implementation for each of the modules can be found in their respective folders in E5TP1/src.

5.3 Module Testbench

Testbench results for each submodule can be found in the directory /E5TP1/tests.txt.
Testbench results for the main module can be found in the directory /E5TP1/final.txt.

5.4 Conclusions

Each sub-module is working as intended, and so the final module works as intended: returning the sum of the BCD format numbers and indicating whether the received input is valid or not.

6 Excercise 6: ALU Implementation

For this exercise we were asked to implement a 4 bit Aritmetic Logic Unit (ALU). The operations we had to develop were SUM, SUBSTRACTION, AND, OR, NOT, XOR, two's complement and shift left. In order to do this, we decided to create a module responsible of adding 2 bits, and as output, it returned 2 bits, one bit for the answer, and another for the carry bit. By using this module, we then decided to create a secondary module, responsible for adding 3 bits. This decision gave us al lot of simplification in the development of the module SUM for 4 bits. As you can see in the code "sum.v" found in the folder src, we commented the previous development without the module sum3Bits, and the new development with the module sum3Bits. For the substaction, we decided to re-use the module created on exercise 4 of two's complement, and utilizing it correctly with the module SUM, we had our SUBSTRACTION module. For the operations AND, OR, NOT, XOR we chosen to use the predefined modules provided by verilog and utilize them bitwise.

6.1 Definitions

In this Aritmetic Logic Unit, we implemented with two accumulators (that we will call A and B), each one of four bits, three operational bits, four bits for the output accumulator (that we will call accumulator C) and one carry bit, ordered in the way they were mentioned.
To select the operation you want to make, you should turn the three operational bits in the following way:

- AND: Performs an AND operation bitwise between acummulators A and B and returns it on accumulator C, meanwhile, the carry bit is left to zero.
- NOT: Performs a logic NOT operation bitwise between acummulators A and B, and returns the answer in the accumulator C. The carry bit stays as zero
- OR: Performs a logic OR operation bitwise between accumulators A and B, and returns the answer in the accumulator C. The carry bit stays as zero.

	Bit 0	Bit 1	Bit 2
AND	0	0	0
NOT	0	0	1
OR	0	1	0
XOR	0	1	1
SHIFT LEFT	1	0	0
SUM	1	0	1
SUBSTRACTION	1	1	0
TWO'S COMPLEMENT	1	1	1

Table 6.1: Representation of operational bits

- XOR: Performs a logic XOR operation bitwise between accumulators A and B, and returns the answer in the accumulator C. The carry bit stays as zero.
- SHIFT LEFT: It manages to move every bit in accumulator A, one space left, and insterts a logic zero to the less significant bit. The answer is given in the C acccumulator and the carry bit will become 0 or 1 dependig on the most significant bit of A.
- SUM: Performs a numeric sum of the binary values of accumulator A and B and the answer is given in accumulator C. Depending on the overflow, the carry bit will become 1 or 0.
- SUBSTRACTION: Performs a numeric substraction of the binary values of accumulator A and B and the answer is given in accumulator C. The carry bit will become 1.
- TWO'S COMPLEMENT: Performs a two's complement of the binary value of accumulator A and the answer is given in accumulator C. The carry bit will become 0.