Task 1

The program is called this way 'run a b c'

The parameters are:

- a: tells whether the input is unsigned or signed (0/1)
- b: size (in bits) of the integer part of the number (≥ 0)
- c: size (in bits) of the decimal part of the number (≥ 0)

All inputs should be numbers. Also b and c can't be 0 at the same time. All these things are validated.

The program logic uses simple formulas to solve the problem separately in signed and unsgined case. The inituition beyond the formulas was obtained by looking at the composition of binary number.

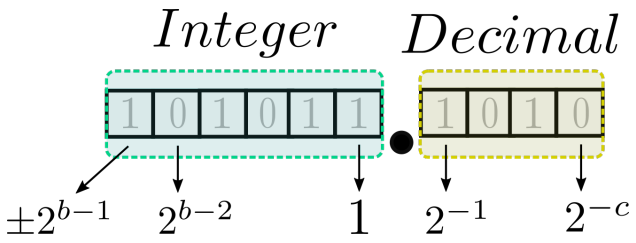

Figure 1: View of the value of each bit

As lowest significant bit worths 2^{-c} then the resolution is 2^{-c} and this works in both signed and unsigned cases

Thought, to compute the range we need to separate in two cases, looking at which are the greatest (we call it r) and lower (we call it l) numbers we can make in each case In the unsgined case the lower number is 0 and the highest will be

$$r = \underbrace{2^{-c} + 2^{-c+1} + ... + 2^{-1}}_{1-2^{-c}} + \underbrace{1 + 2^1 + ... + 2^{b-1}}_{2^b-1} = 2^b - 2^{-c}$$

So the resolution is just $r - l = 2^b - 2^{-c}$

With the signed case we note the smallest number is when only most significant bit is 1

$$l = -2^{b-1}$$

And the highest when all bits are 1 except the most significant one

$$r = 2^{-c} + 2^{-c+1} + ... + 2^1 + 1 + 2^1 + ... + 2^{b-2} = 2^{b-1} - 2^{-c}$$

Thus the resolution is $r - l = 2^{b-1} - 2^{-c} + 2^{b-1} = 2^b - 2^{-c}$ That is the same, that is reasonable because we can express the same amount of numbers with the same amount of bits and that doesn't depend on whether the number is signed or not

Task 2

Starting with the following function expressed in maxterms:

$$f(d,c,b,a) = \prod (M_0, M_1, M_5, M_7, M_8, M_{10}, M_{14}, M_{15})$$

Taking d, c, b, a as input variables. For simplify, the same function is expressed in minterms to operate later:

$$f(d,c,b,a) = \sum (m_2, m_3, m_4, m_6, m_9, m_{11}, m_{12}, m_{13})$$

Starting with it, we build the function without simplifying:

$$f(d,c,b,a) = (\overline{d}\cdot\overline{c}\cdot b\cdot\overline{a})+(\overline{d}\cdot\overline{c}\cdot b\cdot a)+(\overline{d}\cdot c\cdot\overline{b}\cdot\overline{a})+(\overline{d}\cdot c\cdot b\cdot\overline{a})+(d\cdot\overline{c}\cdot\overline{b}\cdot a)+(d\cdot\overline{c}\cdot b\cdot a)+(d\cdot c\cdot\overline{b}\cdot\overline{a})+(d\cdot c\cdot\overline{b}\cdot a)$$

We grouped by common factor in a convenient way:

$$\begin{aligned} f(d,c,b,a) \quad &= \quad \underbrace{(\overline{d}\cdot\overline{c}\cdot b\cdot\overline{a})+(\overline{d}\cdot\overline{c}\cdot b\cdot a)}+\underbrace{(\overline{d}\cdot c\cdot\overline{b}\cdot\overline{a})+(\overline{d}\cdot c\cdot b\cdot\overline{a})}+\underbrace{(d\cdot\overline{c}\cdot\overline{b}\cdot a)+(d\cdot\overline{c}\cdot b\cdot a)}+ \\ &\longrightarrow \quad \underbrace{(d\cdot c\cdot\overline{b}\cdot\overline{a})+(d\cdot c\cdot\overline{b}\cdot a)} \end{aligned}$$

$$f(d,c,b,a) = [\overline{d}\cdot\overline{c}\cdot b\cdot\underbrace{(\overline{a}+a)}_{1}]+[\overline{d}\cdot c\cdot\overline{a}\cdot\underbrace{\overline{b}+b)}_{1}]+[d\cdot\overline{c}\cdot a\cdot\underbrace{(\overline{b}+b)}_{1}]+[d\cdot c\cdot\overline{b}\cdot\underbrace{(\overline{a}+a)}_{1}]$$

$$\boxed{f(d,c,b,a) = (\overline{d}\cdot\overline{c}\cdot b)+(\overline{d}\cdot c\cdot\overline{a})+(d\cdot c\cdot\overline{b})+(d\cdot\overline{c}\cdot a)}$$

Analogously, starting with the expresssion in minterms we reduce the function through a map of Karnaugh:

dc \ ba	00	01	11	10
00	0	0	1	1
01	1	0	0	1
11	1	1	0	0
10	0	1	1	0

From the first group (first row) we have that d, c and b are constantes, thus the first factor stays the way $\overline{d}\overline{c}b$.

From the second group (second row) we have d, c and a as constants, thus this factor stays as $\overline{d}c\overline{a}$.

From the third group (third row) remain constant d, c and b, thus this factor stays as $dc\overline{b}$.

Finally, from the last row, in the group d, c and a stay constant, thus this factor stays the way $d\overline{c}a$.

Adding the partial termns we get the simplified function:

$$\boxed{f(d,c,b,a) = (\overline{d}\cdot\overline{c}\cdot b)+(\overline{d}\cdot c\cdot\overline{a})+(d\cdot c\cdot\overline{b})+(d\cdot\overline{c}\cdot a)}$$

Wich is the same obtained by simplification by boolean algebra.

Taking this function, it was implemented in a logical circuit by AND, OR and NOT gates, as shown below.

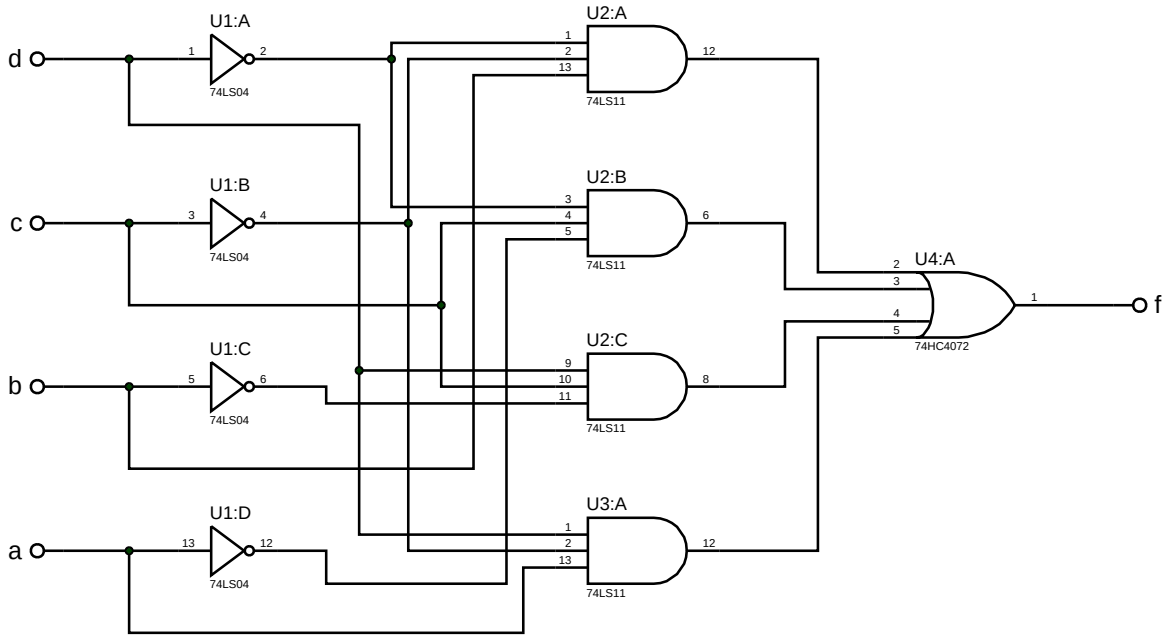

Figure 2: Logical circuit that implements $f(d,c,b,a)$ - Made in Proteus 7.8

For implementation using only NOR gates, first we have to work with the obtained function applyin boolean algebra properties. Taking the function:

$$f(d,c,b,a) = (\overline{d}\cdot\overline{c}\cdot b) + (\overline{d}\cdot c\cdot\overline{a}) + (d\cdot c\cdot\overline{b}) + (d\cdot\overline{c}\cdot a)$$

We twice denied the terms separated by sums, for keeping the equal:

$$f(d,c,b,a) = \overline{\overline{(\overline{d}\cdot\overline{c}\cdot b)}} + \overline{\overline{(\overline{d}\cdot c\cdot\overline{a})}} + \overline{\overline{(d\cdot c\cdot\overline{b})}} + \overline{\overline{(d\cdot\overline{c}\cdot a)}}$$

Next we deny the factors only once, for turning the products into sums (property of De Moivre):

$$\boxed{f(d,c,b,a) = \overline{(d+c+\overline{b})} + \overline{(d+\overline{c}+a)} + \overline{(\overline{d}+\overline{c}+b)} + \overline{(\overline{d}+c+\overline{a})}}$$

Having the expression in terms of sums, it is possible to implement the circuit using only NOR gates, as shown below.

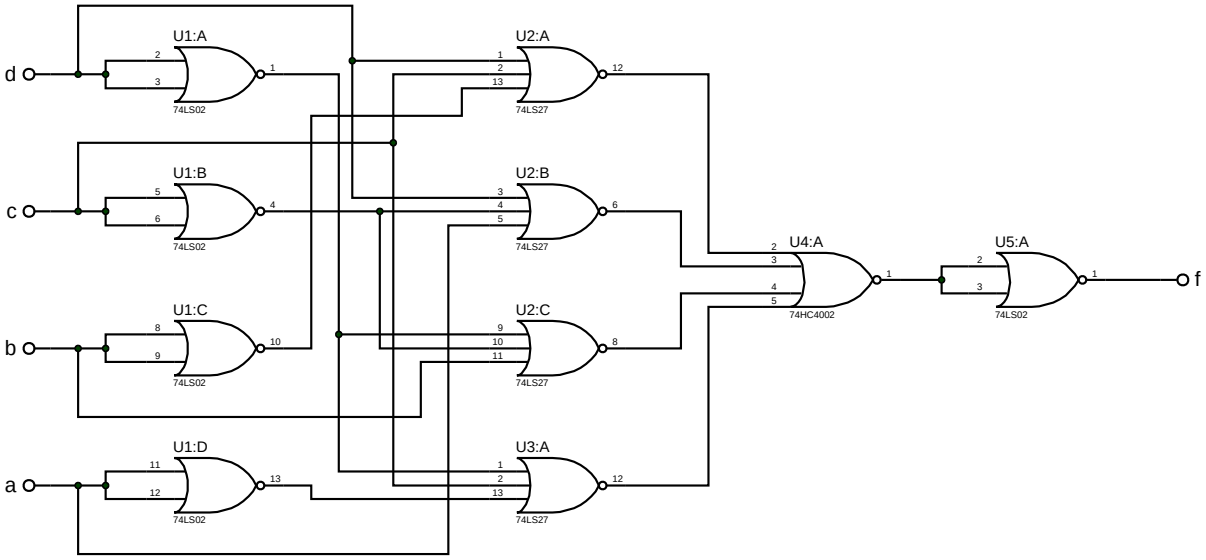

Figure 3: Logical circuit that implements $f(d,c,b,a)$ with NOR gates - Made in Proteus 7.8

Task 6

Compilation

In order to compile the alu and all test cases to bin folder this command must be run

make all

Usage

Exceute each program inside bin folder to run test cases of each module

Module overview

The program module dependencies are organized this way

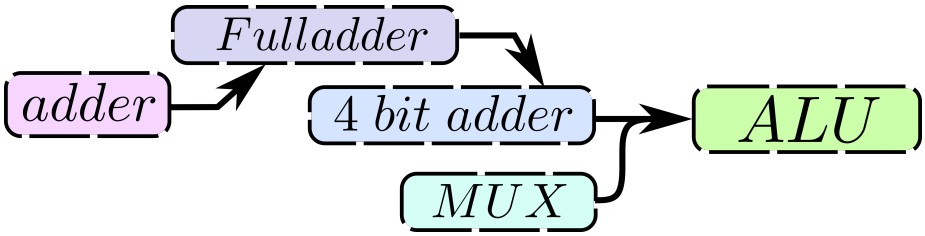

Figure 4: Module dependencies

Each module has its own executable file to run their tests. The modules are

- Mux: It connects the corresponding operation wire with corresponding output. It is an 8 input mux with inputs of size 4.
- 4 $bit\ adder$: Its function is to add 4 bit numbers and compute carry and overflow flags of that operations
- $Fulladder$: Adds 1 bit numbers with carry in, and carry out functionality
- $Adder$: Basic one bit number adder with carry functionality

Currently sum and substract are built from scratch, other operations use verilog built-in fucntions, altough, it would not cause any compatibility problem to have these functions manually coded.

Now we will analize in more depth how the program works

ALU overview

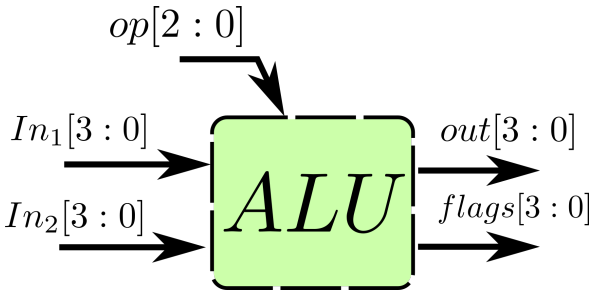

Figure 5: Alu in-out

The ALU makes sure the output and flags are correct accoring to opcodes and inputs.

Operations

There are 8 operations

- **addition** (000) : Implemented using 4 bit full adder
- **difference** (001): Implemented using 4 bit full adder using carry in and built-in not operator
- **and** (010): Implemented with built-in function
- **or** (011): Implemented with built-in function
- **not** (100): Implemented with built-in function
- **xor** (101): Implemented with built-in function
- **2'th complement** (110): Implemented with built-in function
- **shift left** (111): Implemented with built-in function

All 8 operations are connected into a wire array, then Mux comes into action and select according to opcode which action write to output Also another important detail is that operations that use only one number use input 1, and ignore input 2

There are four flags

- **carry/overflow** (Managed by 4 bit adder modules)
- **zero/negative** (Easily managed directly by ALU)

It is important to note that operations that do not use carry and overflow make these bits to be 0.

Mux

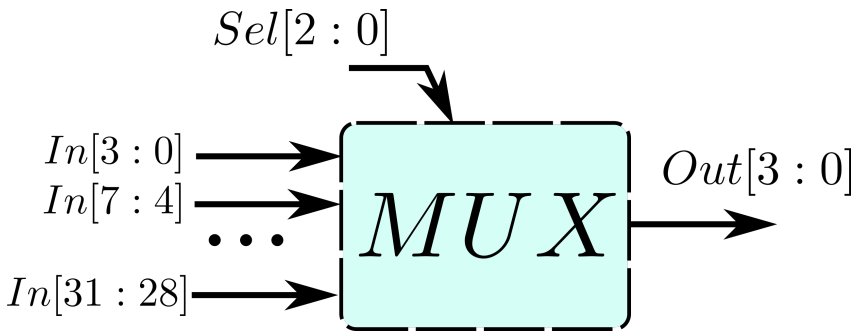

Figure 6: Mux in-out

This is a very simple module, implemented using vectorized expresions. It justs decides which input is wired to output according to selector input

Full 4bit Adder

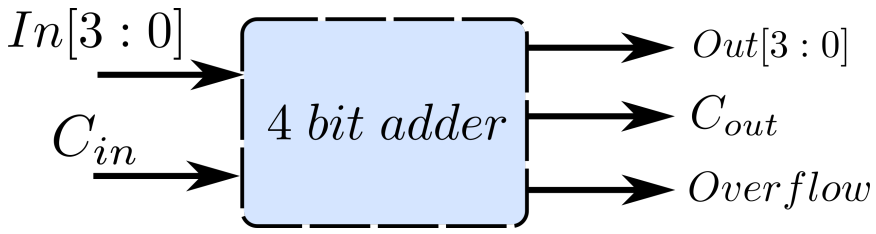

Figure 7: Full 4bit Adder in-out

This module adds two binary 4 bit number with carry-in, carry-out functionality. Also it computes the overflow flag by making the xor of the 'last bit' carry and the carry-out Also , to function this module uses 4 single full adders connected in cascade. We won't describe these modules in this file because they are standard and the sources are enough.