

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 unsigned cases. The intuition beyond the formulas was obtained by looking at the composition of a binary number.

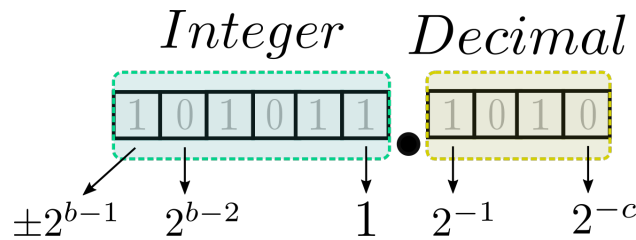


Figure 1: View of the value of each bit

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 is the greater (we call it r) and the lower (we call it l) number in each case. In the unsigned case the lower number is 0 and the highest will be

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

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

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

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:

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

We grouped by common factor in a convenient way:

$$\begin{aligned} f(d, c, b, a) &= \underbrace{(\bar{d} \cdot \bar{c} \cdot b \cdot \bar{a}) + (\bar{d} \cdot \bar{c} \cdot b \cdot a)}_{(d \cdot c \cdot \bar{b} \cdot \bar{a}) + (d \cdot c \cdot \bar{b} \cdot a)} + \underbrace{(\bar{d} \cdot c \cdot \bar{b} \cdot \bar{a}) + (\bar{d} \cdot c \cdot b \cdot \bar{a})}_{(d \cdot c \cdot \bar{b} \cdot \bar{a}) + (d \cdot c \cdot \bar{b} \cdot a)} + \underbrace{(d \cdot \bar{c} \cdot \bar{b} \cdot a) + (d \cdot \bar{c} \cdot b \cdot a)}_{(d \cdot c \cdot \bar{b} \cdot \bar{a}) + (d \cdot c \cdot \bar{b} \cdot a)} + \\ &\rightarrow \underbrace{(d \cdot c \cdot \bar{b} \cdot \bar{a}) + (d \cdot c \cdot \bar{b} \cdot a)}_{(d \cdot c \cdot \bar{b} \cdot \bar{a}) + (d \cdot c \cdot \bar{b} \cdot a)} \end{aligned}$$

$$f(d, c, b, a) = [\bar{d} \cdot \bar{c} \cdot b \cdot \underbrace{(\bar{a} + a)}_1] + [\bar{d} \cdot c \cdot \bar{a} \cdot \underbrace{(\bar{b} + b)}_1] + [d \cdot \bar{c} \cdot a \cdot \underbrace{(\bar{b} + b)}_1] + [d \cdot c \cdot \bar{b} \cdot \underbrace{(\bar{a} + a)}_1]$$

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

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

		ba			
		00	01	11	10
dc	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 $\bar{d}\bar{c}b$.

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

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

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

Adding the partial termns we get the simplified function:

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

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.

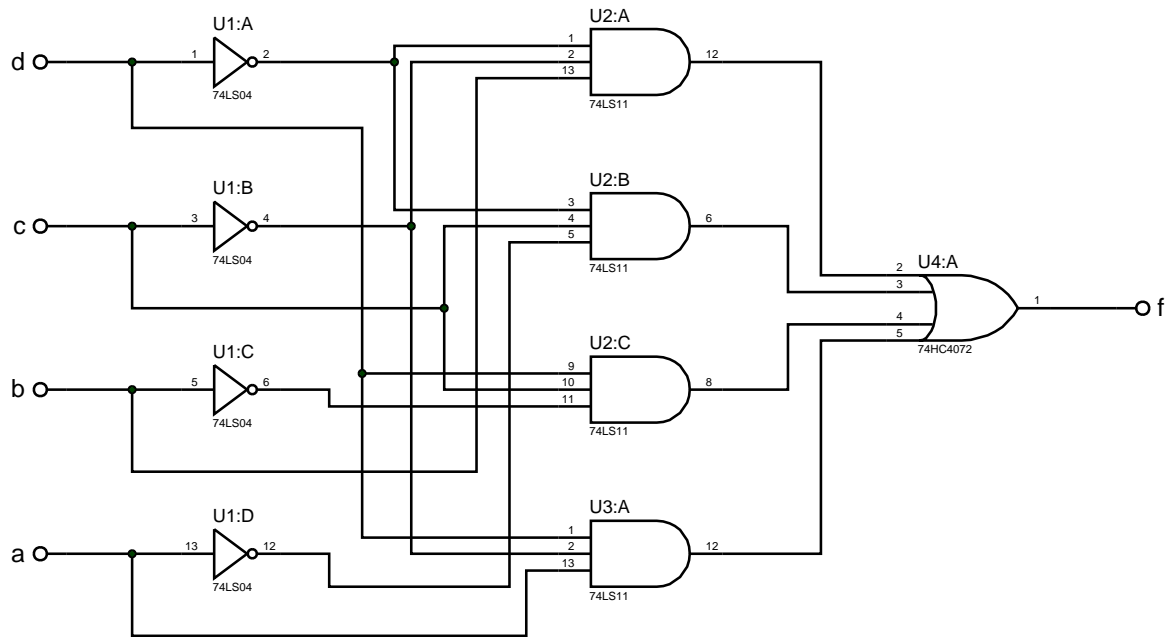


Figure 2: Logical circuit that implements $f(d, c, b, a)$ - Made in Proteus 7.8

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

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

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

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

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

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

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

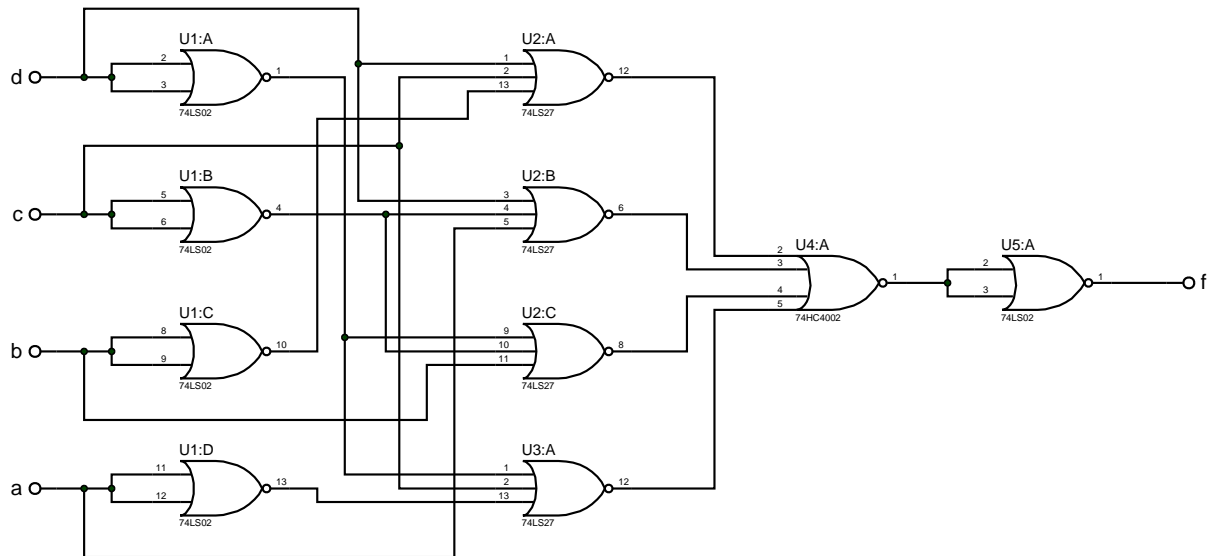


Figure 3: Logical circuit that implements $f(d, c, b, a)$ with NOR gates - Made in Proteus 7.8

Task 3

Decoder

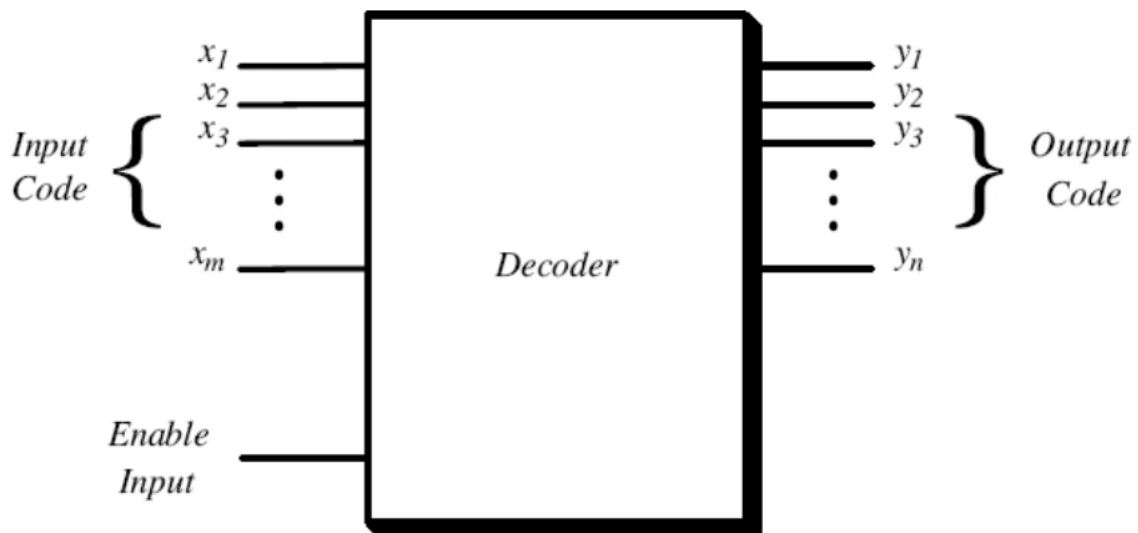


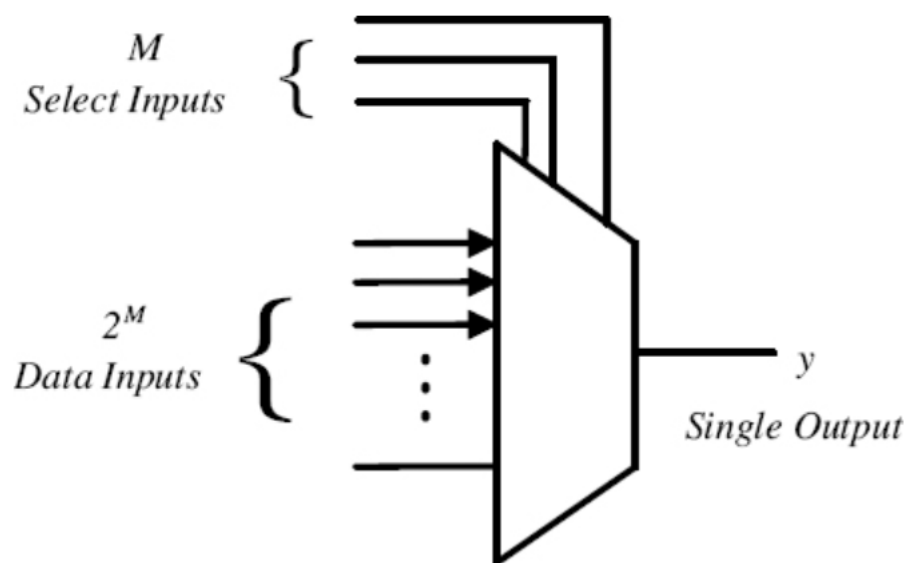
Figure 4: Block Diagram of a $N : 2^N$ Decoder

The binary decoder is a combinational logic device with N input lines and 2^n output lines, one particular combination of inputs activates one output while the remaining ones are disabled. The decoder only works when the enable input is on. Below you can find the truth table of a 2 input decoder:

En	x_2	x_1	y_4	y_3	y_2	y_1
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

Figure 5: Truth table of a 2:4 Decoder

Multiplexer

Figure 6: Block Diagram of a $2^M : 1$ Multiplexer

A multiplexer, also known as 'mux', is another combinational circuit that has 2^M inputs, M select lines and one single output. The select input lines control which data input is connected to the output. The function of a 2 input Mux is described by the truth table shown below:

s	x_2	x_1	y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 7: Truth table of a 2:1 Mux

About the code

There are two different implementations of the mux and the decoder.

In one hand, both mux.v and decoder.v, use conditional statements to describe their behavior. This way is easier for the user to understand how the devices work. Each module has its own test bench which considers all possible inputs for a complete test.

On the other hand, newmux.v and newdecoder.v, have a variable parameter which changes the number of inputs the devices could have. The way that they are implemented is more difficult to understand than the behavioral modeling, but it is shorter. For the decoder, the shift operator resumes perfectly its function and for the mux, the index of the input array chooses the element that has to be in the output. In order to simplify the code, instead of N input/output lines of 1 bit, it has an array of all the input's bits together. In this case the test considers all possible inputs according to parameter chosen.

Task 4

In this case we need to convert a 4-bit number into its complement to two. A truth table is built first with four outputs corresponding to the four input bits of the number complemented, as shown below.

4-Bit In				2-Comp. Out			
d	c	b	a	f_d	f_c	f_b	f_a
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

Table 1: Outputs with complement to two.

The output functions are expressed based on the minterms. They are simplified using Karnaugh's Maps. Starting with f_d function:

$$f_d = \sum (m_1, m_2, m_3, m_4, m_5, m_6, m_7, m_8)$$

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

Solving the map with the indicated groups, the simplified function remains as:

$$f_d = (d \cdot \bar{c} \cdot \bar{b} \cdot \bar{a}) + (\bar{d} \cdot c) + (\bar{d} \cdot \bar{c} \cdot a) + (\bar{d} \cdot b \cdot \bar{a})$$

Now, taking the f_c function, we do the same:

$$f_c = \sum (m_1, m_2, m_3, m_4, m_9, m_{10}, m_{11}, m_{12})$$

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

With the indicated groups, we get the simplified function:

$$f_c = (c \cdot \bar{b} \cdot \bar{a}) + (\bar{c} \cdot a) + (\bar{c} \cdot b \cdot \bar{a})$$

Next, with the f_b function:

$$f_b = \sum (m_1, m_2, m_5, m_6, m_9, m_{10}, m_{13}, m_{14})$$

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

Solving the map, we get:

$$f_b = (\bar{b} \cdot a) + (b \cdot \bar{a})$$

For the last function f_a :

$$f_a = \sum m_1, m_3, m_5, m_7, m_9, m_{11}, m_{13}, m_{15}$$

In the table it can be observed that the output depends directly from input a . We can write the simplified function without making the Karnaugh's map:

$$f_a = a$$

Having already the four output functions, the implementation can be carried out in a circuit with AND, OR and NOT logic gates, as shown in the next page.

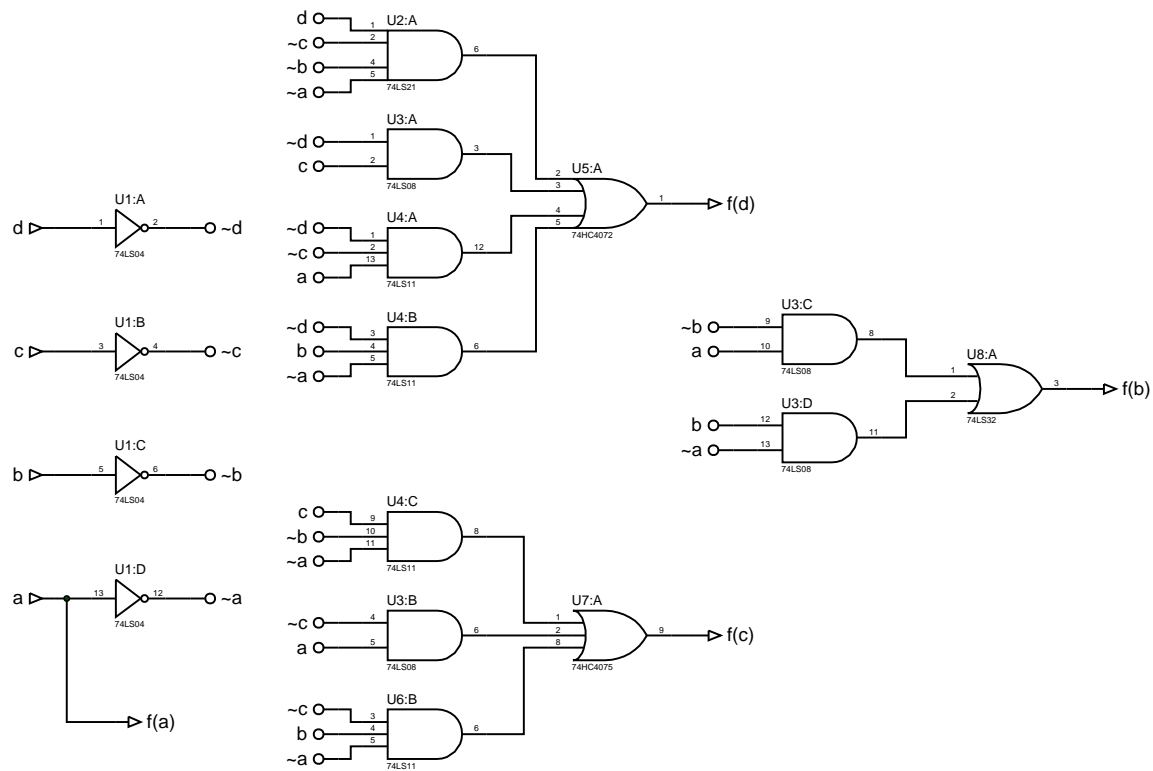


Figure 8: data/Implementation of 2-complement circuit for a 4-bit input number - Designed in Proteus 7.8

Task 5

1 Two BCD addition

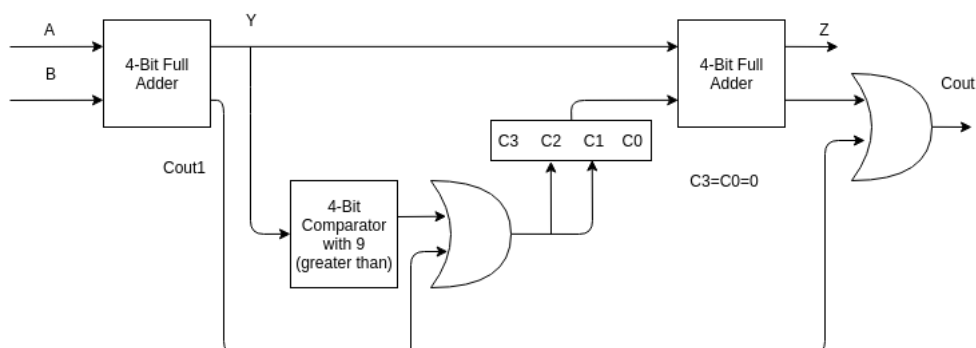


Figure 9: Block diagram of the 4-Bit BCD sum

Given two numbers in BCD format, return the two-digits value of the sum of these two numbers. In order to implement this task, two 4-bit Full Adders were instantiated from the same module. Assuming that inputs are valid, there are just two cases to analyze. First of all, if the sum of the two numbers in decimal is greater than 9 or the carryout bit from the first Full Adder, the sum of them showed in binary code is not the actual BCD. To get the right answer it's necessary to add 6 in decimal value to the sum of those two numbers in order to get the BCD equivalent. Otherwise, there is not addition to the sum of those two numbers.

1.1 Input

Binary to BCD Conversion A Two BCD digits must be entered with the following format: ./a.out +a=XXXX +b=YYYY Where a.out is the name of output executable and both XXXX and YYYY are two 4-bit binary numbers.

Task 6

Module overview

The program module dependencies are organized this way

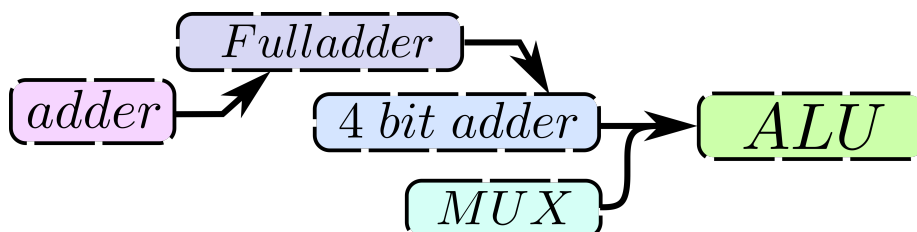


Figure 10: Module dependencies

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 subtract are built from scratch, other operations use verilog built-in functions, although, it would not cause any compatibility problem to have these functions manually coded.

Now we will analyze in more depth how the program works

ALU overview

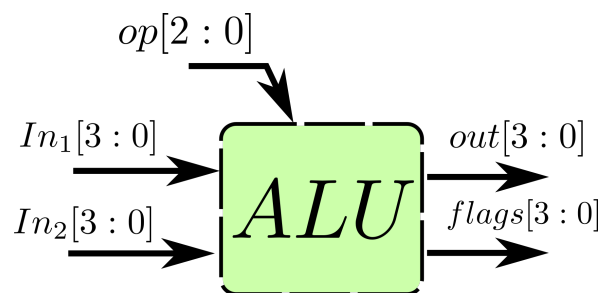


Figure 11: Alu in-out

The ALU makes sure the output and flags are correct according to opcodes and inputs. Note: the carry flag in the difference operation is the result of the two's complement sum, it is not the borrow.

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 know that operations which do not use carry and overflow set these bits to be 0.

Mux

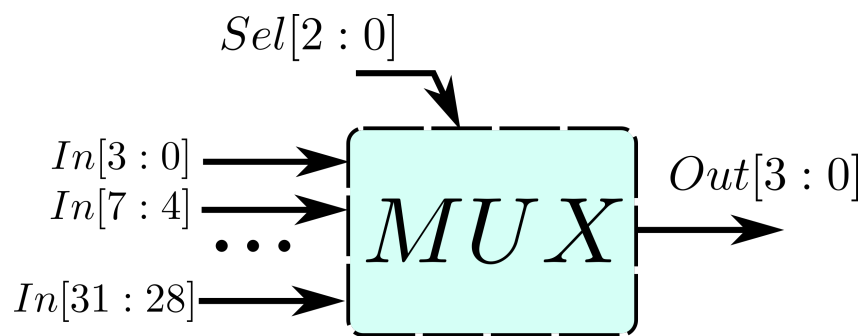


Figure 12: Mux in-out

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

Full 4bit Adder

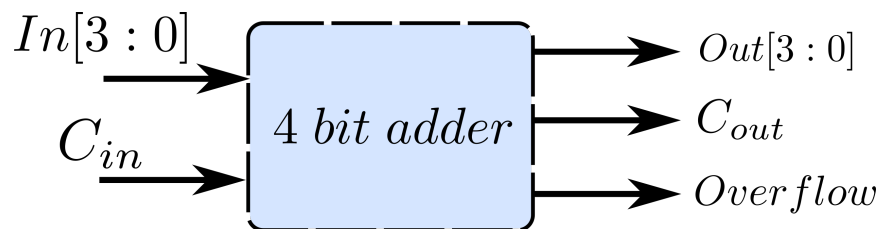


Figure 13: Full 4bit Adder in-out

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.