Electronics laboratory report - calculator

Lorenzo Ramella, Alessandro Matteo Rossi, Marco Tambini

June 11, 2021

Abstract

The goal of this project is creating a calculator able to perform addition and subtractions by using logic gates.

We created two different circuits: a 3-bit real calculator created in laboratory, and a 16-bit simulated calculator, to show how it is possible to solve the issue of the increasing complexity due to the size growth. Most of the used circuits have been both simulated and created in laboratory.

The circuital simulator used is "circuitjs1".

Contents

1 Basics concepts

1.1 Boolean algebra

To make calculations, a circuit needs to be able to perform logical operations. In particular, we usually use boolean algebra in digital electronics.

To be able to create a circuit like this, first of all, we need to define the various components. The number 0 and 1 need to be properties of an electrical circuit that can be "moved" and they can be realized using voltage. We can assign the number 0 to a low voltage and the number 1 to a high voltage.

For example, if we define 0 V as low, negative or 0 and 5 V as high, positive or 1, we can define a threshold voltage exactly in the middle, so that any voltage under 2.5 V will be considered 0, and any voltage above it will be considered 1.

Once 1 and 0 are defined, we need to define the operations that can be performed:

- "!" is the negation, and it can be represented by a NOT gate.
- " + " is the addition, and it can be represented by an OR gate.
- " ∗ " is the multiplication, and it can be represented by an AND gate.

1.2 Classical operations in boolean algebra

1.2.1 Addition and subtraction

In order to sum up two binary numbers, we perform a classical column sum, keeping track of the carry-overs.

Subtracting two binary numbers is slightly more difficult. We point out that, if we consider a binary number A, its opposite is

$$-A = NOT(A) + 1$$

This means that a subtraction can be performed by summing up a first number X_1 and the opposite of the second number $-X_2$.

In our project we will use a ripple carry adder, that will be presented in section 4.2. This component is able to sum X_1 with the opposite of X_2 and then it will add "1". This is possible thanks to the commutative property.

Now let's perform a subtraction among 6 and 3, which respectively correspond to 0110 and 0011. First of all we do

$$0110 + NOT(0011) = 0110 + 1100 = (1)0010$$

We leave the last carry-over, if present like in this case, because inputs and output must have the same amount of binary digits.
Then we add "1"

$$(1)0010 + 1 = (1)0011 = 0011$$

And this is equal to 3 in decimal form.

1.2.2 Multiplication

In order to multiply two binary numbers, you need to subdivide the operation in different phases. The second factor has to be split into numbers with a single "1" bit, and the first factor is multiplied to every single division. Then the results are summed up. The multiplication is a classical multiplication among real numbers.

For example let's take $6 \cdot 10$, which is $0110 \cdot 1010$ in binary form. If we subdivide the multiplication in $0110 \cdot 1000$ and $0110 \cdot 0010$, we can simply add up all these results. The result is $110000 + 001100 = 111100$ which corresponds with 60 in decimal form. We specify that the multiplication by 10 is seen as a multiplication by 8 summed to a multiplication by 2.

2 Logic gates

When we talk about a logic gate, we are describing a circuit that can take a certain number of inputs and gives a single output, depending on the input received. The output needs to be readable by another logic gate of the same family.

The main logic gates are the following, represented with their circuital symbol in figure 1.

A	NOT A
0	1
1	0

(a) NOT gate

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

(b) AND gate

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

(c) OR gate

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

(d) XOR gate

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

(e) NAND gate

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0

(f) NOR gate

A	B	A XNOR B
0	0	1
0	1	0
1	0	0
1	1	1

(g) XNOR gate

Figure 1: Image of the main logic gates used in digital electronics

2.1 N-MOS

In order to create these logic gates, we need to know how to use a MOSFET. For this project we used N-MOS transistors only.

The N-MOS is a transistor that gets as input a gate voltage, a drain voltage, a source voltage and a body voltage. In most cases the source and the body are internally connected, since the body needs to be at the lowest voltage and the source is usually grounded.

When a positive voltage is applied between the drain and source, a deplition layer that blocks the current flow is formed. If we then start applying a positive voltage between gate and body, the electrons will start "balancing" the gaps in the P substrate, but the current will still be zero.

After V_{GS} (the potential difference among gate and source) overcomes a certain threshold voltage, the current will start to flow from drain to source. At the beginning of this flow, the ratio between the current and V_{DS} (the potential difference among drain and source) is linear but, when V_{DS} becomes big enough, the function becomes almost constant (see figure 2) and we find ourself in the saturation region.

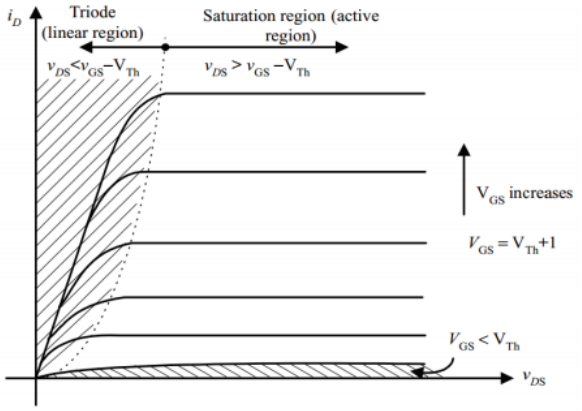


Figure 2: Graph of the I-V N-MOS function

We do not need to distinguish linear section and saturation section, as we will check only the voltage at the drain of the N-MOS for our gate. The threshold voltage is different for every transistor, and it usually is within the range $0.5\,\mathrm{V} - 5\,\mathrm{V}$. The transistor we used is the $IRF822$, whose threshold voltage has been measured in lab, and it is approximately $3\,\mathrm{V}$ (calculated from a laboratory measure with no statistical analysis done, but backed up by the values reported on the datasheet). We decided to use $5\,\mathrm{V}$ for V_{GS} to make input and output approximately the same, as we will see later.

2.2 NOT gate

The easiest logic gate to realize is the NOT gate. We remember that the NOT gate negates the only input it receives. We need a high enough resistance and one N-MOS (as represented in figure 3) to realize it.

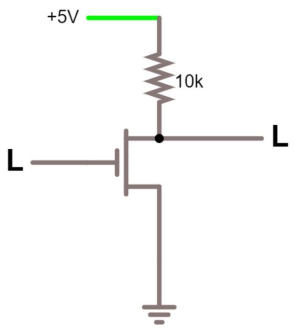


Figure 3: NOT logic gate realized using a N-MOS and a resistor

When the input is 0, the N-MOS will not let the current flow and, thanks to Ohm First Law, we know that the voltage drop across the resistor should be 0, so we get the same supply voltage. The read output will be around 5 V, so we get a 1 as output.

When the input is 1, the N-MOS will let the current flow with a small resistance value. Since there is a higher resistance before the N-MOS, almost all the voltage drop will occur on the previous one, and the drain will be almost zero. The read output will be around 0 V, so we get a 0 output.

With this configuration the signal will not be properly negated for high frequencies, and it will be like in the last picture in figure 4 (it will never reach a 1 state).

A possible solution consists of a combination of N-MOS and P-MOS but, since our calculator does not have to work at such high frequencies, we dismissed this problem.

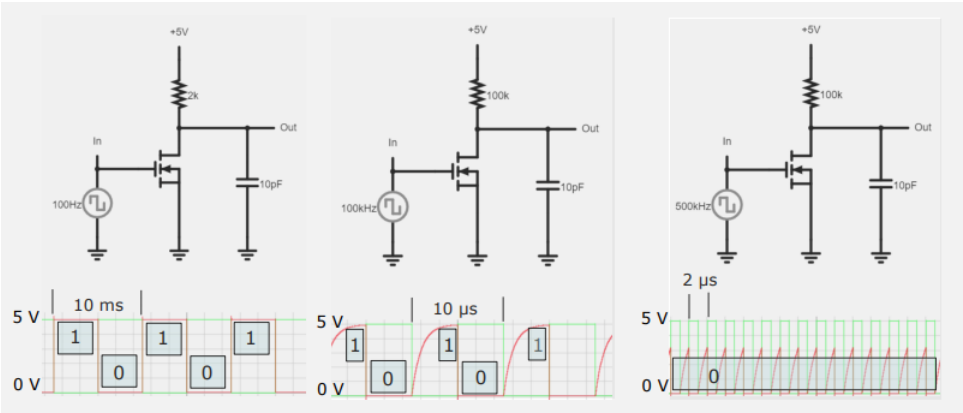


Figure 4: Inverter N-MOS circuit with a resistive load

For the picture of the NOT gate realized in laboratory check figure 32 in Appendix.

2.3 NOR gate

Once we created the NOT gate, we proceeded to realize the NOR gate since, as we will explain later in Section 2.5, the NOR and the NAND gate are both functional complete.

The NOR gate is composed, as seen in figure 5, by two NOT gates short-circuited at their output.

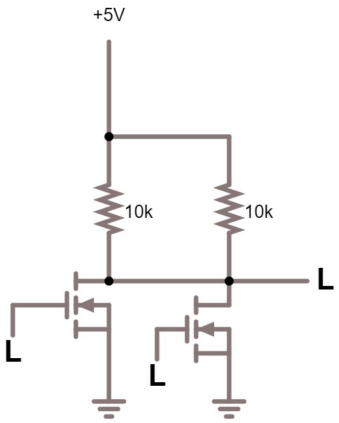


Figure 5: NOR logic gate realized using two short-circuited NOT gates

When both inputs are 0, the two N-MOS will not let any current pass and, just like in the NOT gate, the output will be 1.

When one or both inputs are 1 the N-MOS will let the current flow and there will be a voltage drop across the resistor. So the output will be 0.

The role of the short circuit is ensuring that whether an input is positive, the current will pass through the "open" N-MOS.

For the picture of the NOR gate realized in laboratory check figure 33 in Appendix.

2.4 NAND gate

As we said for the NOR gate, the NAND gate is also functional complete. But it can be created without the shortcircuit needed by the NOR gate, so it is slightly cheaper.

The NAND gate is realized with a series resistor followed by two N-MOS, where the source of the first is plugged in the drain of the second, as seen in figure 6.

When both inputs are 1, the current will be able to flow and the output will be 0.

When one of the inputs is 0, one of the N-MOS will be "closed" and, since the current cannot pass, the output will be 1.

For the picture of the NAND gate realized in laboratory check figure 34 in Appendix.

2.5 Other logic gates

Being functional complete means that a single logic gate can be the only component in a circuit, and that circuit will be equivalent to any other logic gate. NAND and NOR gates have this property (see figure 30 and 31 in appendix).

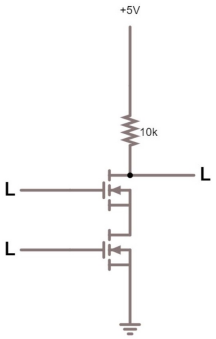
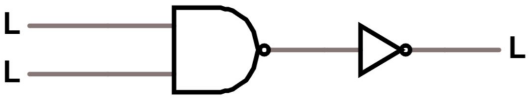


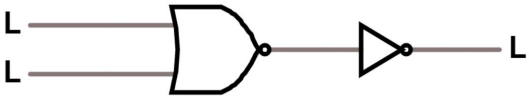
Figure 6: NAND gate realized with one resistor and two N-MOS transistors

The only gates needed for the calculator are shown in figure 7, and they are:

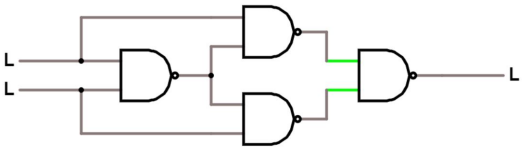
- The AND gate, made by the NAND gate, where the output is negated by a NOT.
- The OR gate, made by the NOR gate, where the output is negated by a NOT.
- The XOR gate, made by four NAND gates as shown in figure 7c



(a) AND gate built using a negated NAND gate



(b) OR gate built using a negated NOR gate



(c) XOR gate built using a four NAND gates

Figure 7: The logic gates used in the calculator

2.6 Bistable circuit

For the 16-bit calculator, that will be discussed in section 7, we also needed to be able to store information.

This means that we need a component that will work as memory. It has to be able to get an input and keep it saved as long as necessary, even if the input signal has already expired. Since the memory has to be both writable and readable, it needs a second input line to clear it.

The circuit that corresponds to the memory we need is the bistable circuit, also known as flip-flop (shown in figure 8).

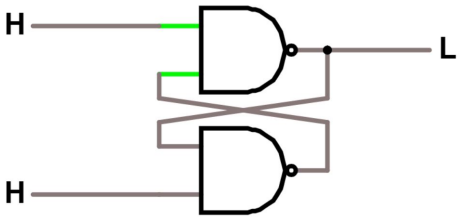


Figure 8: The circuit used to create a flip-flop memory unit

The flip-flop can be created by both NOR gates and NAND gates but, since the NAND gates are cheaper, we decided to use those. This meant that the input has to be negated in order to work properly. Flip-flop truth table can be found in table 1.

#	*Set*	*Clear*	*OldQ*	*NewQ*	*NewQ'*
1	0	0	0	1	1
2	0	0	1	1	1
3	0	1	0	1	0
4	0	1	1	1	0
5	1	0	0	0	1
6	1	0	1	0	1
7	1	1	0	0	1
8	1	1	1	1	0

Table 1: Flip-flop truth table. Please note that the first two rows are not ideal for the storage of data, as Q and Q' have the same value. Rows 3-6 represent the moment when an input is given, whereas 7 and 8 is when data is stored.

3 Encoder

While calculators work using binary numbers, this kind of numeration is unfamiliar and difficult to use for most humans. This is why it is worth to develop an encoder, a circuit that converts decimal inputs in binary numbers.

3.1 Simple encoder

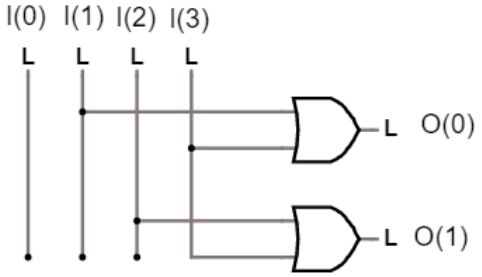


Figure 9: A simple encoder

A simple encoder is the easiest possible design for an encoder. There are 2^n input lines, and n OR gates. When one of the input lines gets high voltage, it activates the OR gates corresponding to its binary representation.

I_0	I_1	I_2	I_3	O_1	O_0
0	0	0	0	0	0
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

Table 2: Truth table for the 2-bit simple encoder

This circuit implementation is very simple, but it does not return a correct result if more than one input line is high. In such cases, it is advisable to use a priority encoder.

3.2 Priority encoder

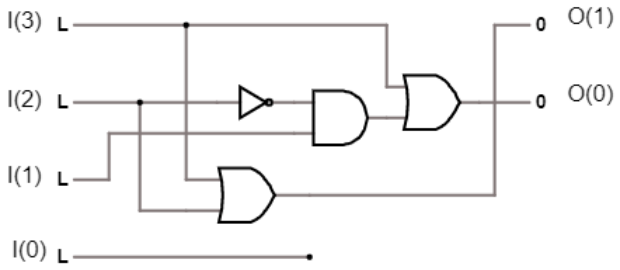


Figure 10: A priority encoder

A priority encoder is similar to a simple encoder, but it works even if more lines are high at the same time, giving priority to the highest number. It achieves this results by using AND gates: the OR gate inputs are no longer directly connected to the input lines, but they need to be "checked" by an AND gate. The AND gate receives as inputs the corresponding input line and the negation of any input line which:

- has an value higher than the first input line;
- its binary representation is 0 in that position.

This way, each input line not only activates the OR gates corresponding to its binary representation, but also de-activates the "wrong" OR gates that a lower value input line could activate.

An example of how this works is provided in figure 10. When $I(2)$ is low, the AND gate has the same value of $I(1)$, and the circuit works like a simple encoder; when $I(2)$ is high, the AND gate is always low, no matter the value of $I(1)$.

I_0	I_1	I_2	I_3	O_0	O_1
x	0	0	0	0	0
x	1	0	0	1	0
x	x	1	0	0	1
x	x	x	1	1	1

Table 3: Truth table for the 2-bit priority encoder

4 Processing

In order to create a calculator that could perform additions and subtractions, we needed to have a component able to sum (or subtract) two binary numbers.

4.1 Half adder and full adder

The easiest component that can do this task is the half adder. It is a small circuit with 2 inputs and two outputs, and it is capable of summing up two binary digits. The input are the two digits, and the outputs are the sum S, performed by a XOR gate (which truth table is in figure 1d), and the carry-over C, calculated by an AND gate (which truth table is in figure 1b).

Half adder truth table can be found in table 4.

A_1	A_2	S	C
0	0	0	0
1	0	1	0
0	1	1	0
1	1	0	1

Table 4: Half adder truth table, where A_1 and A_2 are the input bits, S is the result, and C the carry-over

The major issue with this component is not being useful during a sum of two binary numbers. We specify that a sum of two binary numbers is a classical column sum. For this reason the half adder is not the best choice, because it cannot track the carry-over of the sum performed on the previous bit, if present.

If we want to sum two positive n-bit binary numbers, we need to keep track of the carry-overs. The right component for this duty is a full-adder, whose truth table can be found in table 5

A_1	A_2	C_{in}	S	C_{out}
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

Table 5: Full adder truth table. It is possible to see the third input that the half adder does not have, the carry-over of the previous column sum

The most efficient way to realize a full adder is shown in figure 11. It requires only 5 logic gates.

4.2 Ripple carry adder

If we align a series of full adders we obtain a ripple carry adder (shortened in RCA).

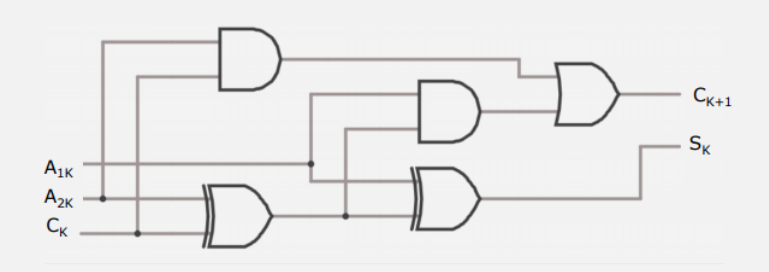


Figure 11: Most efficient way to create a full adder, by using two XOR gates, two AND gates and one OR gate

This circuit is useful because it can perform not only sums among positive numbers, but also algebraic sums. We point out that in our circuit only the second addend could be negative.

This happens because, in our circuit, both numbers are considered positive, but the user can pick the subtraction operation, which makes the second addend negative. It is not possible to modify the sign of the first number.

In figure 12 you can see a ripple carry adder with a 3-bit input (where one of the bits if the sign bit).

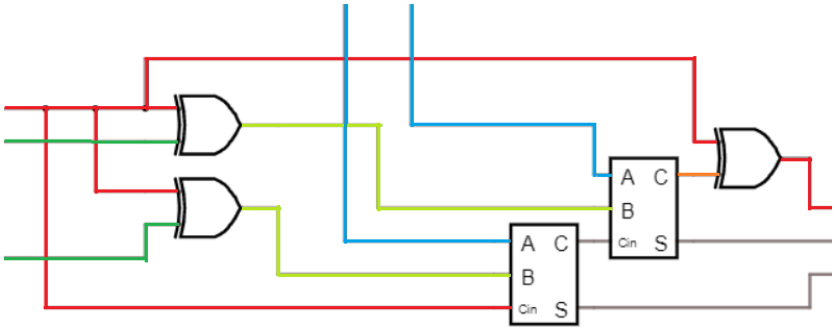


Figure 12: 3-bit ripple carry adder

We can list its working steps:

1. The digits of the first number (blue wires, one per digit) enter the full adders.
2. The digits of the second number (green) get compared to the sign bit (red) by two XOR gates. They will exit these gates on the lime wire, and they will enter the full adders too. If the second number is negative, the red wire will be on, and the digits will be inverted, following the rule on how to obtain the opposite of a binary number.
3. Considering that the opposite of a binary number is $-A = NOT(A) + 1$, the "1" is added in the first full adder as a C_{in}.
4. The sum is performed and the result goes to a decoder. The XOR gate on the right side of the image compares the sign bit with the carry-over of the last sum performed (orange).

Let's consider the possible cases of this last XOR gate:

(a) If the sign bit is 0 and the carry-over of the last full adder is 0, this means we are summing two positive numbers, which result is positive, as confirmed by the XOR.

(b) If the sign bit is 0 and the carry-over of the last full adder is 1, this means we are experiencing a memory overflow. The result goes beyond the range the calculator can handle.

(c) If the sign bit is 1 and the carry-over of the last full adder is 0, this means we are subtracting two numbers, and the second has a greater module. So the result is negative.

(d) If the sign bit is 1 and the carry-over of the last full adder is 1, this means we are subtracting two numbers, and the first has a greater module. So the result is positive.

5 Decoder

The decoder is a device complementary to the encoder, as it "translates" a binary input into a more readable decimal output.

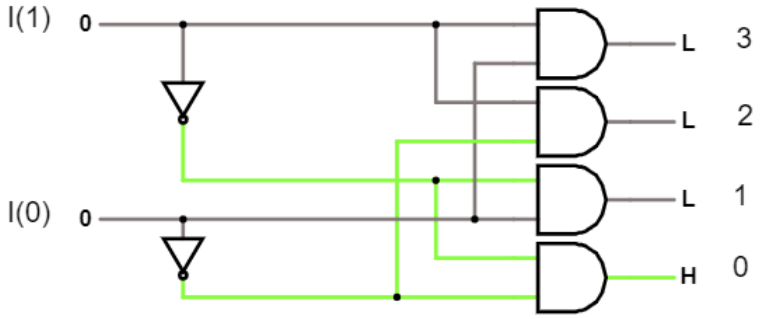


Figure 13: Here's a basic 2-bit decoder

Each output line is connected to an AND gate. Each AND is connected to every single input line, either directly or through an inverter. This way, the AND gate returns a high voltage only if the inputs match the binary representation of its corresponding output line.

I_1	I_0	O_0	O_1	O_2	O_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Table 6: Truth table for the 2-bit decoder

6 Real calculator (3-bit)

Now that the 3 key parts of a basic calculator have been introduced, we can try to assemble one in lab. This real calculator will be quite small and simple, for both time limitations and materials availability.

6.1 The blueprint

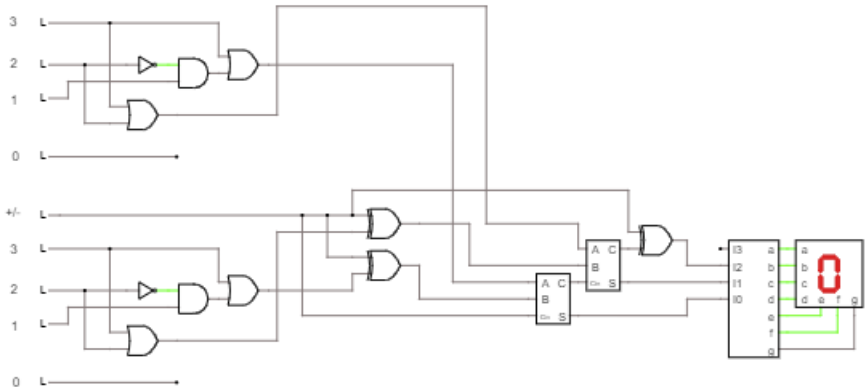


Figure 14: Here's the blueprint of our calculator

In Figure 14 is showcased the blueprint of our calculator. It receives 2 2-bit numbers as inputs, encodes them in binary and then adds them up using a 2-bit ripple carry adder. A decoder then converts the result in decimal and displays it.

There is also an additional input line useful to perform the subtraction. When this input line is low, the 3 XOR gates can be ignored as they return the unchanged input.

When the subtraction line is high, however, those gates return the opposite of said input lines. This is very important, as the first 2 XOR gates have the specific purpose of converting the second input into its opposite.

Effectively, when the subtraction line is high, we are adding two signed 3-bit numbers, a negative number and a positive number. To do that, we would need a 3-bit ripple carry adder, but:

- The first input is always positive, so its sign bit its always 0 (in fact, it doesn't even exist). So, we only need an additional half-adder;
- We are not intrested in the carry bit of the last half-adder, as we are working with 3-bit numbers.

But a half-adder circuit without the carry output is a XOR gate. This is precisely the purpose of the third XOR gate. With this configuration, the subtraction line does 3 things:

- It endodes the second input into its opposite;

- Works as the sign bit of the second input;
- It is directly connected into the C_0 input of the first full-adder to add 1 to the final result.

As for the decoder, both in our blueprint and in lab we are going to use a pre-made decoder. Assembling a decoder using AND gates would require a significant amount of time and resources which we do not have access to.

6.2 Breadboard 1

The calculator has been assembled using 3 different breadboards. The first breadboard handles the input part and the encoding process.

The input part is created using a "keyboard", which is a series of 6 switches built into a compact board.

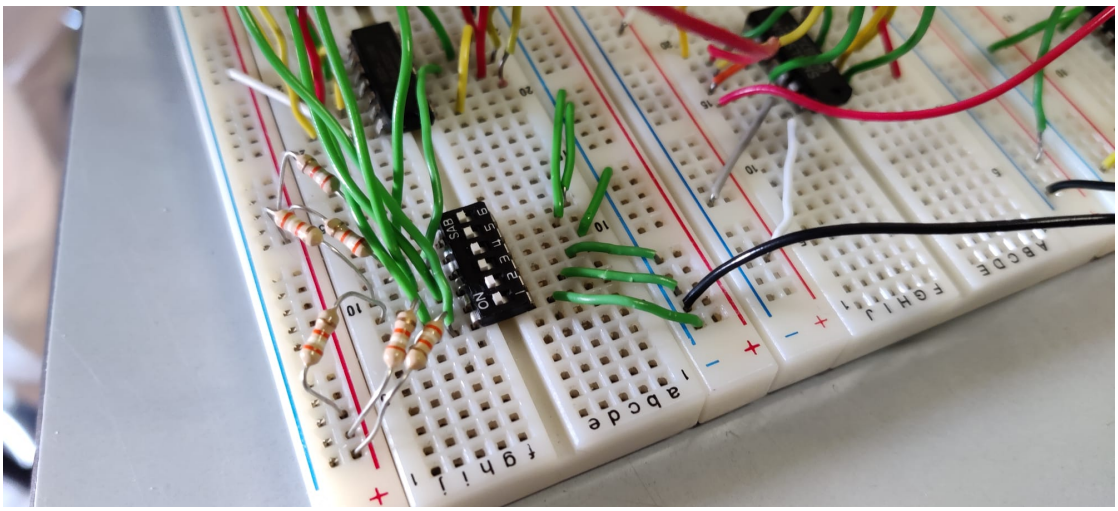


Figure 15: The "keyboard"

One side of each switch is grounded, while the other one is connected to the power supply through a resistor ($\approx 30 - 50\ k\Omega$).

When the switch is open, there is no current flow through the resistor and therefore no voltage difference across the resistor. Any high impedance input device can then be connected to the second end of the resistor to read a high voltage.

When the switch is closed, the current is free to flow through the resistor down to earth. The second end of the resistor is at the same voltage as the ground, and the input device reads a low voltage.

This device allows us to conveniently manipulate 6 different input lines. We are going to use the switches 1, 2 and 3 to write the first input, and the switches 4, 5 and 6 to write the second (4 being 1, 5 being 2 and 6 being 3). To write 0, the user just needs to leave all the switches open.

The rest of the first breadboard is dedicated to the encoding process. The process works exactly as described in section 3.2, with the right part of the board dedicated to the encoding of the first input and the left part dedicated to the second.

We are not going into the details of the wiring. We will just mention that the white wires are used to connect the gates to the power supply, and the black and brown wires are used to connect the gates to earth.

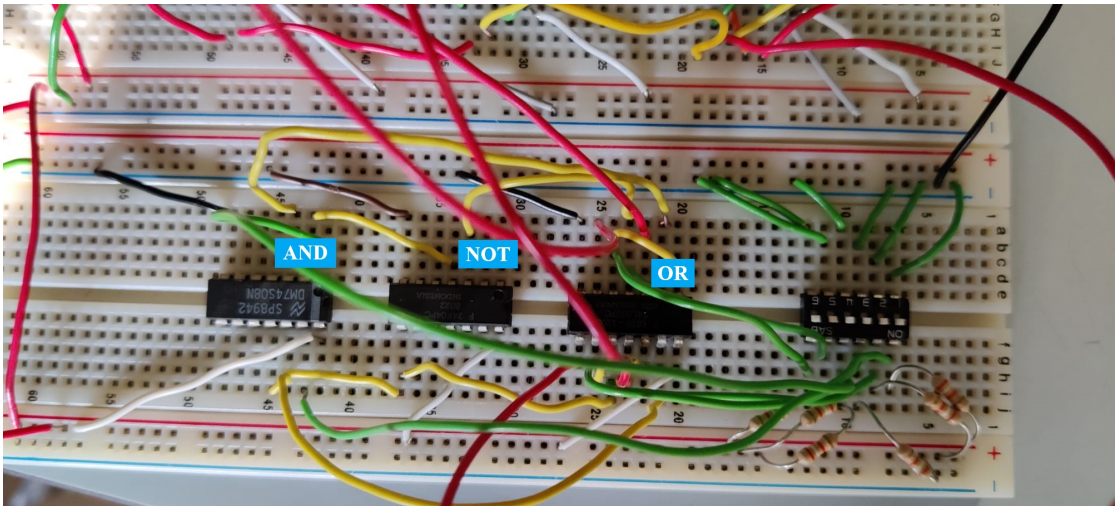


Figure 16: Breadboard 1

6.3 Breadboard 2

The second breadboard handles the calculation process. It is a 2-bit ripple carry adder, composed of 2 full adders. The first full adder is on the left side of the board and the second adder is on the right side. The process works exactly as described in section 4.2.

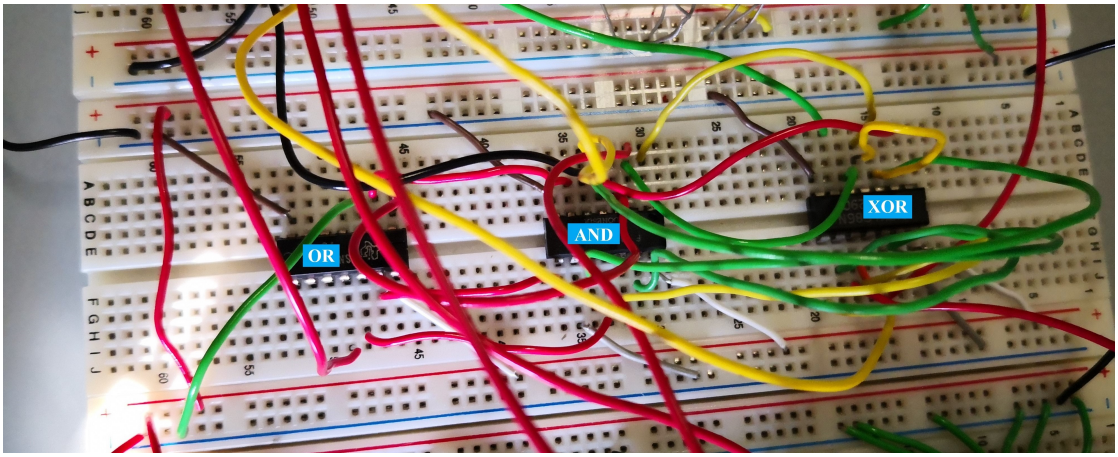


Figure 17: Breadboard 2

6.4 Breadboard 3

The third breadboard handles the decoding process and houses everything that could not fit into the previous 2 breadboards.

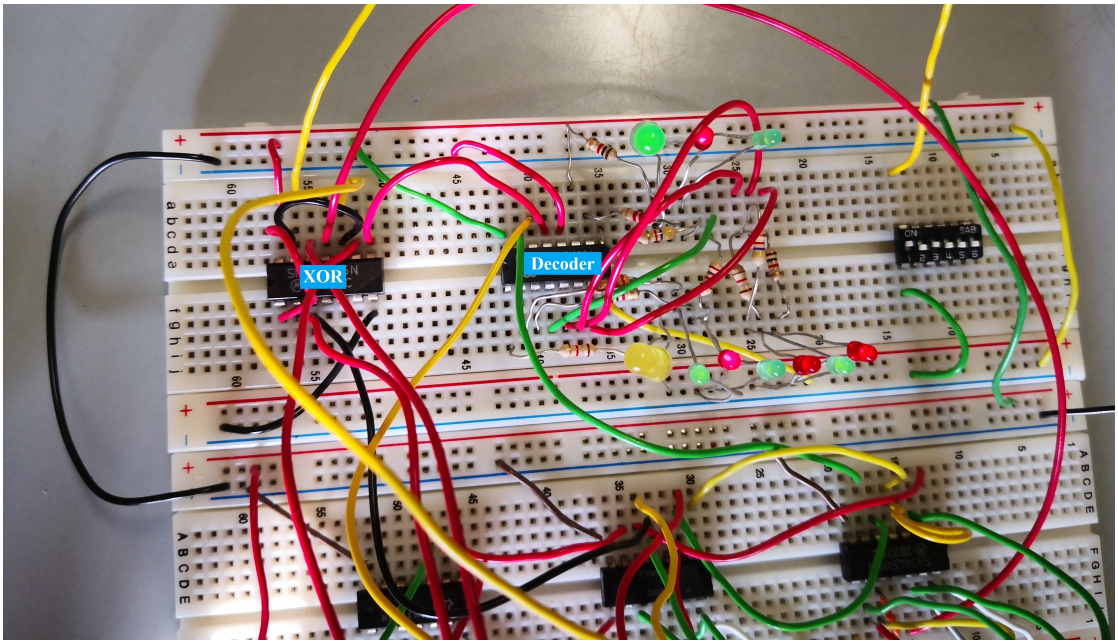


Figure 18: Breadboard 3

The breadboard houses the 3 XOR gates necessary to enable the subtraction process. Those gates are connected to both the previous breadboards.

The decoder works in a similar way to what is described in section 5. The only key difference is that it works using negative logic: a 1 output is low in voltage an a 0 output is high in voltage.

To read our output more easily, we connected the 10 outputs to 10 LEDs. The decoded number corresponds to the least bright LED. The yellow LED corresponds to the number 0, and the value of the others can be deduced counting up going counterclockwise. The 8 and 9 LEDs are there for diagnostic purposes. Ideally, they should never turn off.

There is also a switch, whose purpose is to enable the operation of subtraction. While in our simulator the "subtraction line", and everything connected to it, was at low voltage when not connected to any power supply, in reality an isolated subtraction line finds itself at high voltage.

To enable the addition/subtraction selection, then, we connected this line to earth through a switch. When the switch is open, the subtraction line is at ground potential and the calculator does the addition process. When the switch is closed, the subtraction line is isolated (high voltage) and the calculator does the subtraction process.

6.5 The complete project

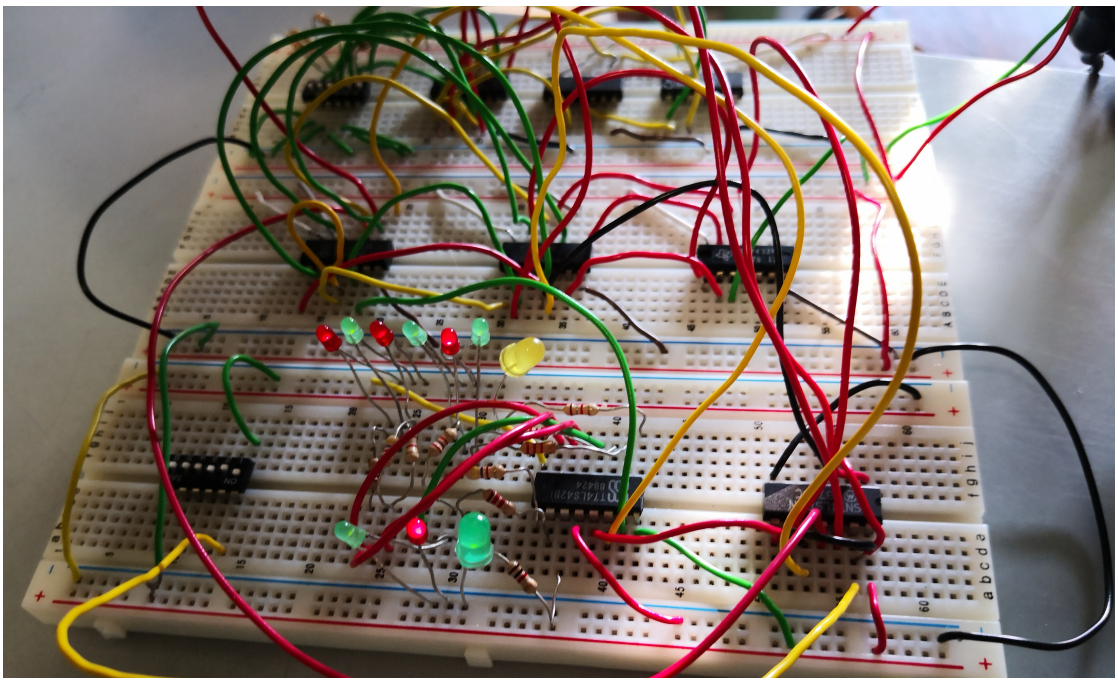


Figure 19: The complete 3-bit calculator

After assembling the circuit, we then verified its truth table (Table 7).

I_A	I_B	sub	O	I_A	I_B	sub	O	I_A	I_B	sub	O	I_A	I_B	sub	O
0	0	0	0	0	2	0	2	0	0	1	0	0	2	1	6
1	0	0	1	1	2	0	3	1	0	1	1	1	2	1	7
2	0	0	2	2	2	0	4	2	0	1	2	2	2	1	0
3	0	0	3	3	2	0	5	3	0	1	3	3	2	1	1
0	1	0	1	0	3	0	3	0	1	1	7	0	3	1	5
1	1	0	2	1	3	0	4	1	1	1	0	1	3	1	6
2	1	0	3	2	3	0	5	2	1	1	1	2	3	1	7
3	1	0	4	3	3	0	6	3	1	1	2	3	3	1	0

Table 7: Truth table for the 3-bit calculator, where sub=0 means addition and sub=1 means subtraction

Given that, in a 3-bit signed configuration, $7 = -1$, $6 = -2$, $5 = -3$ and $4 = -4$, we can conclude that our calculator works as intended.

7 16-bit calculator

7.1 Input with 16-bit

Our goal is to create a 16-bit calculator. It has to receive 16 bits as input and that produces 17 output bits (the 17th is the sign bit).

One of the problems that arise when we want to use such a big memory space is the increasement in complexity; this makes the circuit more expensive and also takes a lot of physical space, that could be used for other purposes. In a small scale, we could use a 2-bit priority encoder, but, for our project, it would be unpractical and unnecessarily harder.

Another problem regarding the priority encoder, is that this component would require more physical space. We would need an input line for every single number we could insert. Considering that the possible inputs are all the natural numbers in the interval $[0, 2^{16})$, we would need 2^{16} input lines.

To solve both problems we decided to use a keyboard. It takes the input through some buttons, instead of the levers we used in the laboratory. This method solved both the problem with the number of inputs and the problem of the complexity of the decoder.

7.2 Keyboard

The keyboard part connects a point at high voltage to the rest of the circuits. It is composed by:

- 10 buttons for the digits from 0 to 9;
- An addition button labeled +;
- A subtraction button labeled −;
- A button to have the result displayed, labeled =;
- A clear button that resets the entire circuit.

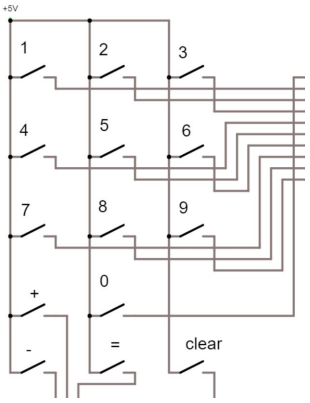


Figure 20: Picture of the simulated circuited keyboard

7.3 4-bit encoder

The first part of our encoder is a 4-bit encoder, without priority.

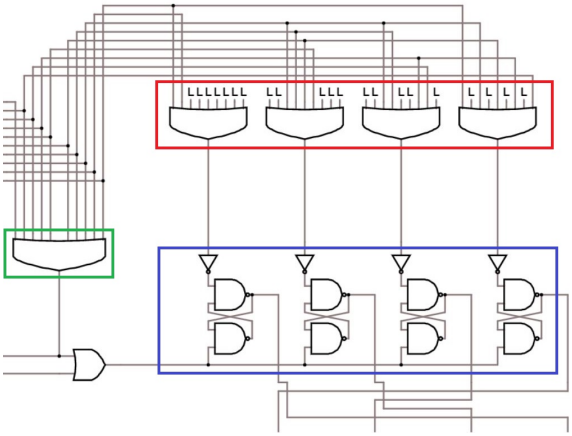


Figure 21: 4-bit encoder

As you can see in figure 21 this part can be split in three different sections:

- The red part is a 4-bit encoder without priority. As we were saying in the paragraph above, the advantage of using a keyboard is that we will, under normal condition, only get 1 input at a time making a priority encoder useless. The encoder works by checking, with an OR gate, which bits the inserted number activates.

  For example, if we press the button "6" the binary input will be 0110, so only the second and the third bit will be activated.

- The green part is an OR gate connected to all the input lines, in order to check whether a button corresponding to a number is pressed. This choice has been made because if 0 was pressed, it would not result in any binary input, but it would be needed to multiply the previous number by 10.

- The blue part is a small "flash memory", which gets cleared when the input button is no longer pressed.

  This part, whose functioning relies on the concept of bistable circuit, is a simple "security" method, useful to ensure that the inputs arrive correctly to the next memory. This part could be removed if the circuit timing was perfect, but we preferred to keep it to ensure no problem would arise.

7.4 Memory and successive inputs

This part allows the circuit to receive consecutive inputs, and it is composed by two memories and two ripple carry adders.

In this section the input gets stored and, after a new button is pressed, the stored number is multiplied by 10 and then added to the new input. By doing this we can obtain every number within the memory limit.

It is important to point out that this part works by using the concepts of "rising edge", where the input passes from a low state to a high one, and "falling edge", where the input changes from high to low. This concept makes possible to process a single input in two phases.

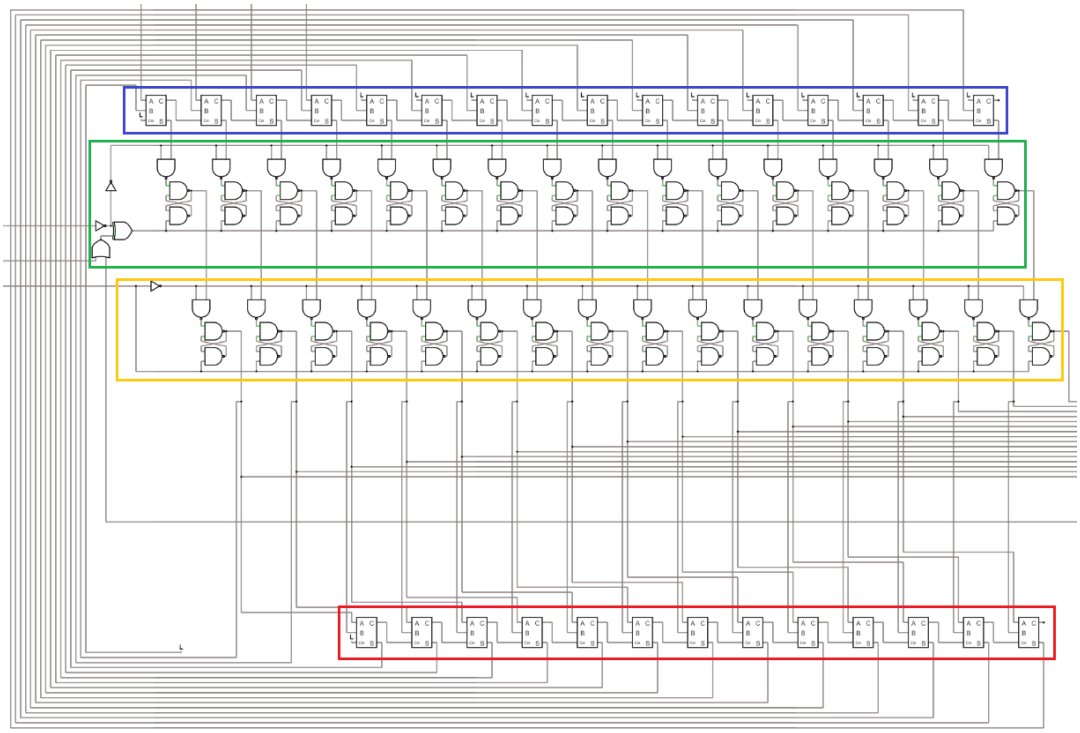


Figure 22: The two memories (green and yellow), the 10x multiplier (the red RCA) and the adder (the blue RCA)

7.4.1 Rising edge

When the first button is pressed, the circuit is in a state of "rising edge". In this moment, the horizontal wire that goes to all the NANDs of the second memory (highlighted in yellow in figure 22), switches to low to prevent the NANDs from letting any signal pass from the first to the second memory.

The second wire involved is the horizontal one under the first memory (highlighted in green), that also turns off. This wire is responsible for clearing the memory. This happens when this wire switches from high to low.

After a little while, the last wire that changes its state is the one on top of the first memory. This allows a new input to be memorized in the cleared memory.

It is important to note that, during the data storage, the reset wire (the horizontal one at the bottom of the green box) is in a low state, since it cleared the memory right before the storage of new data.

This last fact does not cause any trouble when the input is 0, since we are in the case 5 or 6

of the flip-flop truth table (table 1) and, as you can see, the output is 0.

The problem arises when the input data is 1, since we are in case 1 or 2 of the flip-flop truth table, where the two inputs are 0. As we can see in the truth table, in this case, both outputs are set at 1.

After one of the two inputs switches from low to high, the flip-flop returns in a normal state (state 3-6). But, to store a bit "1", we need to go from states 1 or 2, to state 4 of the truth table. In other words, as long as the clear line is the first one switching from low to high, we easily pass from state 1-2 to state 4.

Note that, it is not possible to reach state 3 in the case explained before, because state 1-2 "NewQ" becomes state 4 "OldQ" while state 3 "OldQ" is 0.

The problem we have just described has been solved in the simulated calculator by using the delay given from the logic gates. In reality, this could be a problem since this relies on physical properties that could depend from temperature, making this kind of timing, whether applied to a real circuit, not as precise as in the simulator.

The timing problem also includes the fact that the operation on the second memory should be done before the one on the first memory. In our circuit, this delay is performed by the "clearing part" (the XOR gate on bottom left corner of the green box).

Another possible solution could be using an external clock to define when the action should be performed. But, in order to keep our simulated circuit as easy as possible, we decided to simply address the problem in this report.

7.4.2 Falling edge

After the the number button is released the circuit is in the "falling edge" phase.

The process is similar to the previous one. The first wire to change its state is the horizontal one below the second memory, this time. Its task is the memory reset, like the one in the first memory. After this, the horizontal wire on top of the second memory changes its state, to allow the second memory to store information coming from the first.

The last connection that switches is, as before, the horizontal one on top of the first memory. It switches from high to low and prevents any further memory modification.

The problem described at the end of the rising edge is, in fact, a problem that begins with the rising edge and ends during the falling edge. Since the first memory works during the rising edge phase, and the second one works during the falling edge phase, the second memory will suffer the same problem, and so it will have the same possible solutions explained before.

7.4.3 Ripple carry adders

This circuit works with two ripple carry adders.

During a calculation, the RCAs do not work at the same time. The first RCA, highlighted in blue lets the inserted digit head towards the first memory highlighted in green. Then, this digit passes through the second memory (yellow), during the falling edge phase, and gets multiplied by 10 from the red RCA. This multiplied digit is not used unless a new digit is inserted by the user.

If a new number button is pressed, this new value gets summed to the previous data, already multiplied by 10, and stored in the green memory. The sum of the two number is performed by the blue RCA.

If a third digit is inserted, the process repeats again.

Moreover, we can notice that after multipling by 10, the first bit is always set to 0. The second and the third bits depend only from the first and second bit of the original number, because they are only affected by the multiplication by 2 (0010) and not by the multiplication by 8 (1000).

From the fourth bit the addition performed by the red RCA is needed because these binary digits are influnced by the 10x multiplication.

7.5 Sign bit

To do the subtraction we could have used the full subtractor, but we decided to use the addition between positive and negative numbers in binary. To perform the subtraction we needed a way to read whether the second number was positive or negative . As visible in figure 23 we simply used a flip-flop. From the top (coming from the left side):

- The first horizontal line is the input +;
- The second is −;
- The third is the clear signal, that also resets the sign bit.

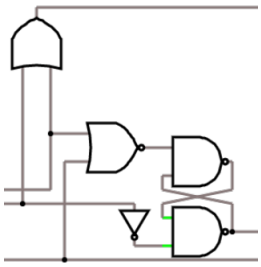


Figure 23: The component used to check a number sign, created with a flip-flop

7.6 Memory

The final part of the input is composed by the two memories in figure 24.

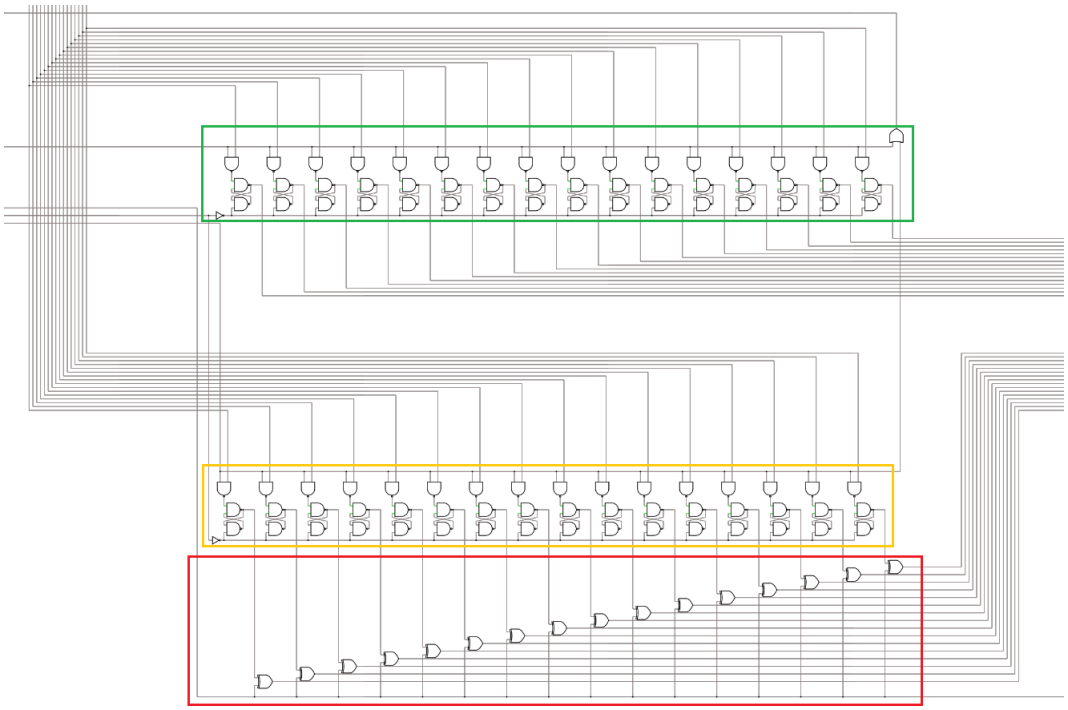


Figure 24: Final part of the input section, composed of two memories, one for each number that needs storage

The first memory, highlighted in green, is where the first number gets stored. Considering that, the first number is always positive, we can save it and give it to the processing section, just as it is.

Since we want this memory to be modified only when it has to register the first number, we added the "NAND wire" that is the wire exiting the OR in figure 23. This horizontal wire is located over the green memory.

With this, the memory will store information only when + or − are pressed. The vertical wire that exists the OR gate on the top right corner of the green memory is responsible of clearing the green memory in figure 22, making it possible to insert the second number.

The second memory, highlighted in yellow, works in the same way as the first one, but it stores the number when the "=" button in pressed.

The XOR gates beyond the yellow memory invert the bits of second number, as already explained in section 4.2, when the user is doing a subtraction.

7.7 Clear

The last input, not discussed yet, is the clear button. This button is connected to the reset line of all the memories and the sign bit, and it switches every flip-flop to 0.

7.8 Processing

The processing phase of the 16-bit calculator is done as explained in section 4.2.

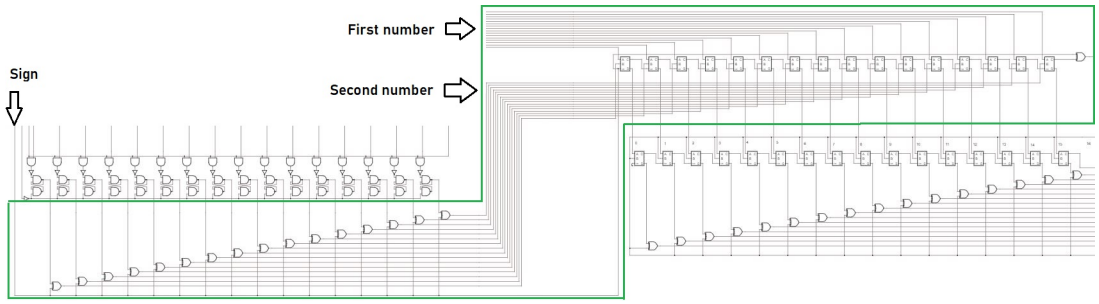


Figure 25: Processing section of the 16-bit calculator. The XOR gates in the bottom left zone are the ones highlighted in red in figure 24

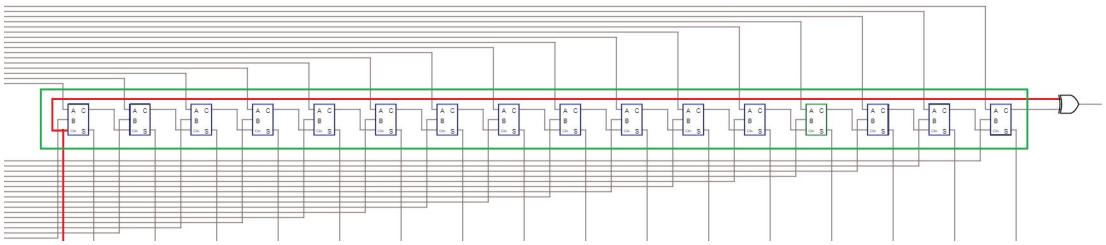


Figure 26: Ripple carry adder to sum the first and the second number. The red wire is the sign bit

The results that exit the RCA will go to the decoding part.

7.9 Decoder

After the inputs have been processed, it is needed a decoder (see figure 27) in order to convert the output from binary to its decimal form.

The theorical largest number, in absolute value, that can be outputted by 17 bits (one of them is the sign bit) is 2^{16}. So the decoding circuit needs to have 5 led displays for the digits and one extra display for the sign. It is possible to note that the image shows a 6th led display for the digits, because we originally thought, by mistake, we could have a 18 bit output.

We decided to operate the binary-decimal conversion with the so called "double dabble" circuit, that will be explained later in the report. Nevertheless this was not the main problem. The double-dabble converts positive numbers perfectly, but not negative ones. So we had to convert a negative number, if outputted, to a positive one and send the sign to the sign led display.

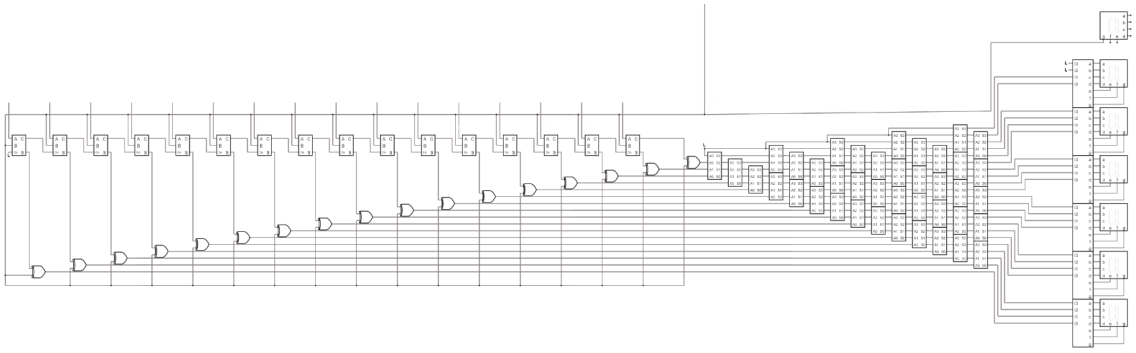


Figure 27: Picture of the decoder

7.9.1 About the number sign

The number sign has to be taken into account before converting the number into its decimal form. The number is a 16-bit binary, as already said, and it has one extra bit for the sign.

This part of the circuit (represented in figure 28) uses a ripple carry adder and XOR logic gates, components that got already discussed in the previous sections. After the operation has been done between the two inputs, the first 16 bits reach the A input the full adders (the green lines in figure 28), whereas the sign bit follows the red path.

This sign bit reaches every full adder B input, and also the XOR gates, that compare the result of the single full adders with the sign bit. The sign bit is true (or 1) when the number is negative and 0 otherwise. This allows the full adders to sum 1, following the formula above, if the processing output is negative, whereas if it is positive the number just stays the same.

After this, the XOR gates, which truth table is give the final result, which will go to the double dabble.

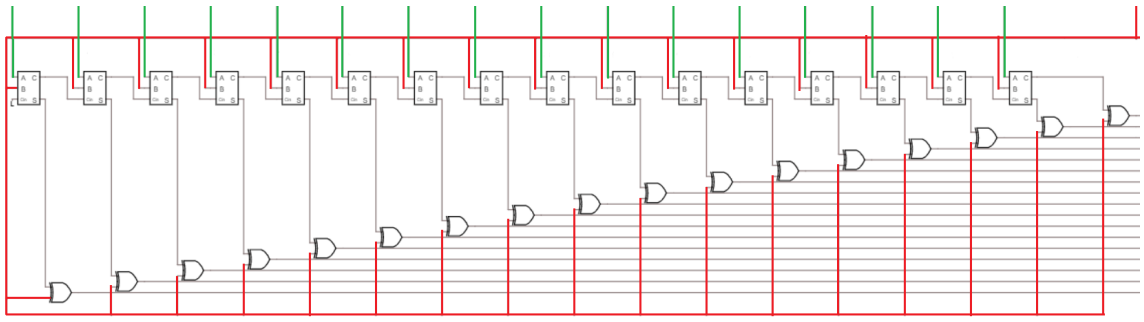


Figure 28: Picture of the first part of the decoder

7.9.2 Double dabble

This second part of the decoder is reached by the number that needs to be converted into decimal form.

The entire circuit relies on an algorithmic process, based on the concept of "shift and add 3", which is the name of the component that mostly populates figure 29.

This algorithm takes a binary number and after having processed it gives an output divided into smaller parts composed of 4 bits each. Everyone of these parts will be then elaborated by 7 segment decoders and represent a single digit of the decimal number. The 7-segment decoders are obviously connected to 7-segments led displays, that can be seen on the right side of figure 29.

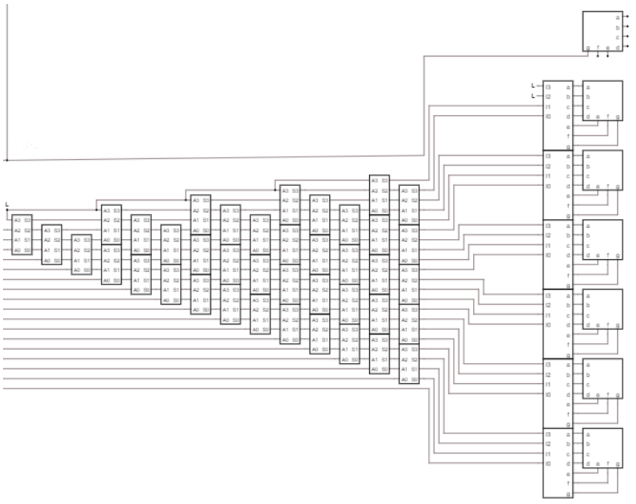


Figure 29: Picture of the double dabble, the second part of the circuit

The algorithm works as follows (graphic representation in figure ??):

1. Let's consider a 8-bit binary number, but the same argument works for n bits.
2. Let's consider the units, as long as that binary value is lower or equal to 4, the binary input can keep shifting and increasing the units.
3. When the units value is an integer greater than 4, 3 is added to them and the shifting process continues.

It is mandatory to add 3 because during the conversion the weight of the 4 bits of the unit is 16, at maximum, whereas those for digits represent a maximum of 10 in decimal form. So to compensate this loss, we add a half of the lost weight.

The goal of the "Shift and add 3" component, programmed using the customizable logic of the simulator, is to operate this shift or addition depending on the 4 inputs given. Its truth table is figure ??.

It is clear, thanks to this figure, that when the number is greater than four it gets added three to it. The "shift" part can be seen in figure 29. If we consider the input A0 of a single Add3 component, its output, S0, is the input A1 for the next component.

#	Hundreds	Tens	Units	Binary	Operation
1	0000	0000	0000	11111111	Start
2	0000	0000	0001	11111110	Shift1 (every 4-bit slot < 5)
3	0000	0000	0011	11111100	Shift2 (every 4-bit slot < 5)
4	0000	0000	0111	11111000	Shift3 (every 4-bit slot < 5)
5	0000	0000	1010	11110000	Add-3 to "Units" ("Units" ≥ 5)
6	0000	0001	0101	11110000	Shift4 (every 4-bit slot < 5)
7	0000	0001	1000	11110000	Add-3 to "Units" ("Units" ≥ 5)
8	0000	0011	0001	11100000	Shift5 (every 4-bit slot < 5)
9	0000	0110	0011	11000000	Shift6 (every 4-bit slot < 5)
10	0000	1001	0011	11000000	Add-3 to "Tens" ("Tens" ≥ 5)
11	0001	0010	0111	10000000	Shift7 (every 4-bit slot < 5)
12	0001	0010	1010	10000000	Add-3 to "Units" ("Units" ≥ 5)
13	0010	0101	0101	00000000	Shift8 (every 4-bit slot < 5)

Table 8: Double dabble algorithm applied to the decimal number 255

Input	Output
0000	0000
0001	0001
0010	0010
0011	0011
0100	0100
0101	1000
0110	1001
0111	1010
1000	1011
1001	1100

Table 9: "Shift and add 3" truth table

After all this process the number is divided into 4 bit groups that enter the 7-segment decoders and than display the result on led displays.

8 Appendix

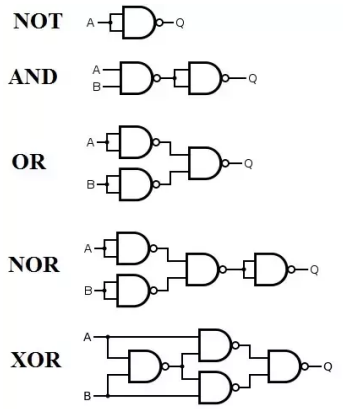


Figure 30: NAND gate is functional complete, every other logic gate can be a circuit made of NANDs only

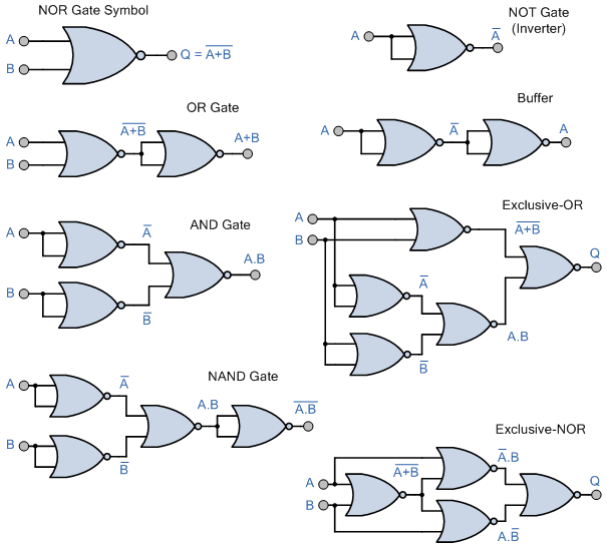


Figure 31: NOR gate is functional complete, every other logic gate can be a circuit made of NORs only

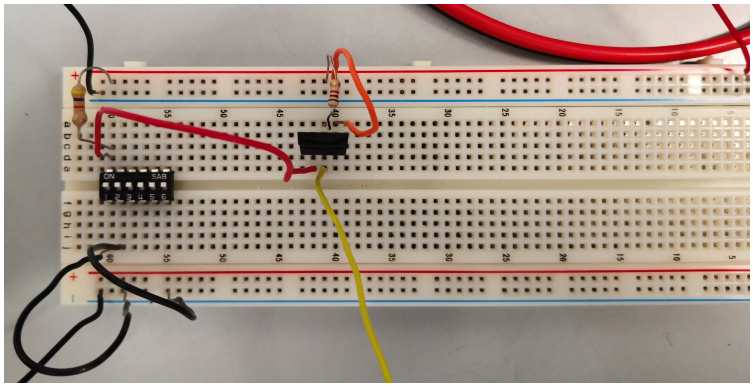


Figure 32: NOT gate realized in laboratory

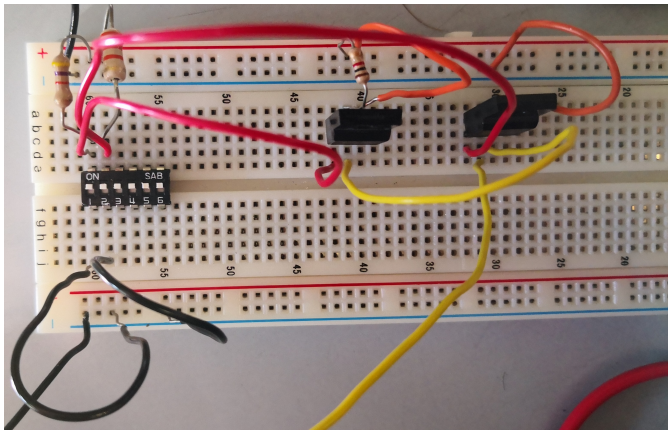


Figure 33: NOR gate realized in laboratory

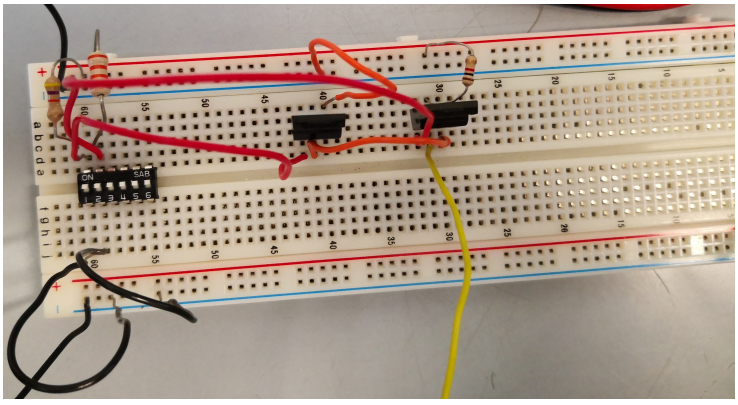


Figure 34: NAND gate realized in laboratory