

Report: adder and subtractor

Lorenzo Ramella, Alessandro Matteo Rossi, Marco Tambini

June 18, 2021

Contents

1 Basics concepts	2
1.1 Boolean algebra	2
1.2 Classical operations in boolean algebra	2
1.2.1 Addition	2
1.2.2 Subtraction	2
1.2.3 Multiplication	3
2 Logic gates	3
2.1 N-MOS	4
2.2 NOT gate	5
2.3 NOR gate	6
2.4 NAND gate	6
2.5 Other logic gates	6
2.6 Bistable circuit	8
3 Processing components	9
3.1 Half adder	9
3.2 Full adder	9
3.3 Ripple carry adder	10
4 16-bit calculator	13
4.1 Input with 16-bit	13
4.2 Keyboard	13
4.3 4-bit encoder	14
4.4 Memory and successive inputs	14
4.4.1 Rising edge	15
4.4.2 Falling edge	16
4.4.3 Full adders	16
4.5 Sign bit	17
4.6 Memory	18
4.7 Clear	18
4.8 Decoder	19
4.8.1 About the number sign	19
4.8.2 Double dabble	20
5 Appendix	23

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

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

1.2.2 Subtraction

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

$$-A = \text{NOT}(A) + 1$$

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 3.3. 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

$$0110 + \text{NOT}(0011) = 0110 + 1100 = (1)0010$$

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.

1.2.3 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.

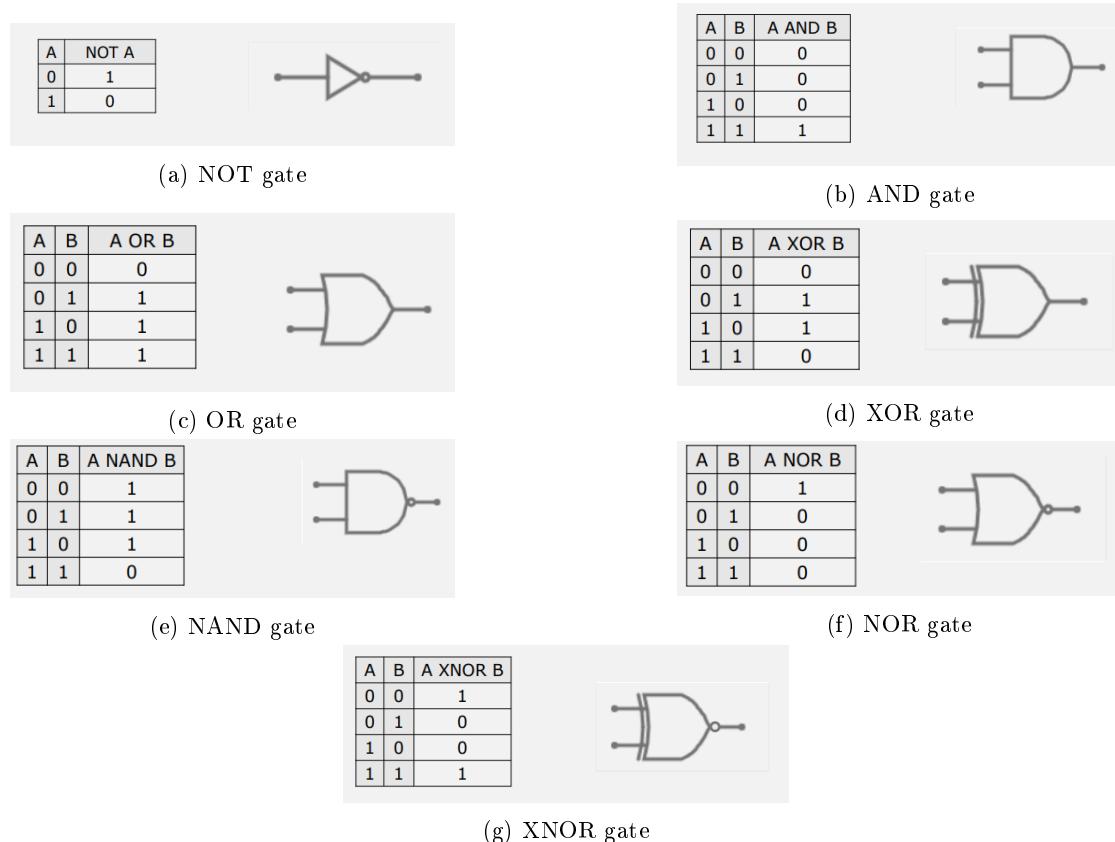


Figure 1: Image of the main logic gates used in digital electronics

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 depletion 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 ourselves in the saturation region.

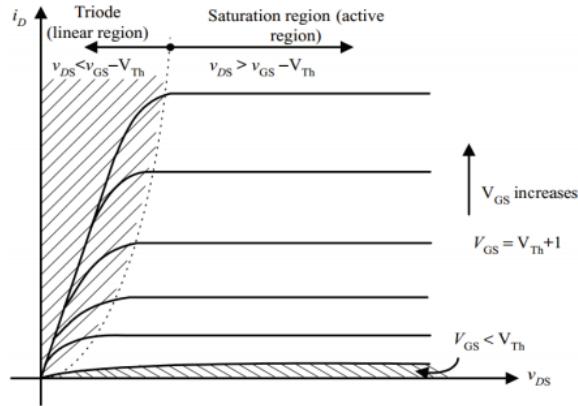


Figure 2: Graph of the I-V N-MOS function

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 V – 5 V. The transistor we used is the *IRF822*, whose threshold voltage has been measured in lab, and it is approximately . We decided to use 5 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.

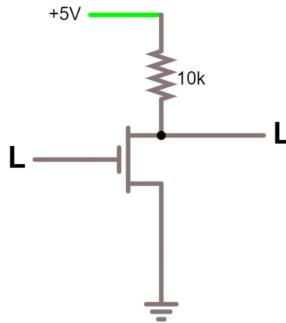


Figure 3: NOT logic gate realized using a N-MOS and a resistor

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.

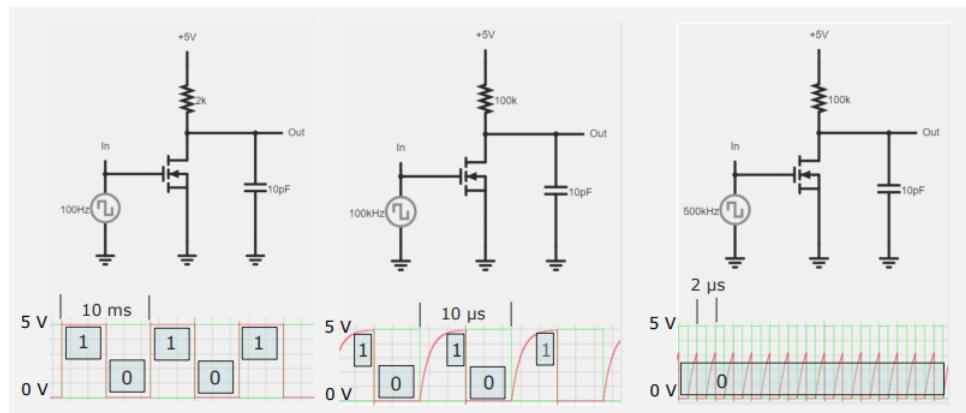


Figure 4: Inverter N-MOS circuit with a resistive load

For the picture of the NOT gate realized in laboratory check figure 24 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.

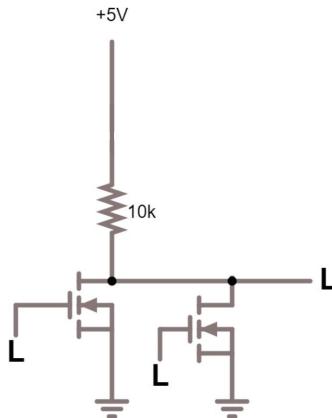


Figure 5: NOR logic gate realized using two short-circuited NOT gates

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 25 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 26 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 22 and 23 in appendix).

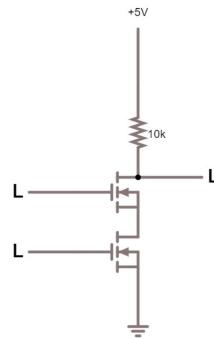


Figure 6: NAND gate realized with one resistor and two N-MOS transistors

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

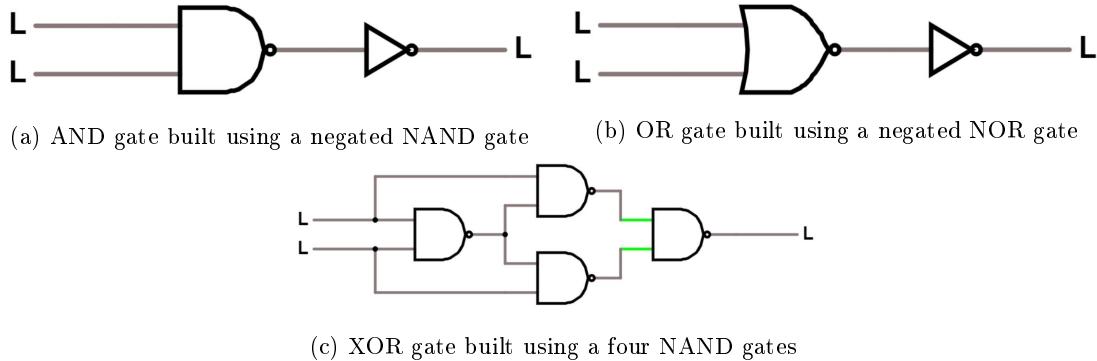


Figure 7: The logic gates used in the calculator

2.6 Bistable circuit

For the 16-bit calculator, that will be discussed in section 4, 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).

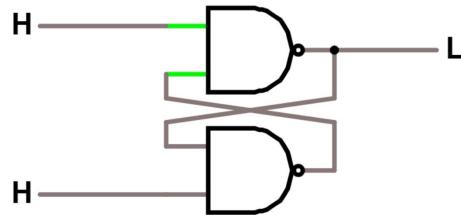


Figure 8: The circuit used to create a flip-flop memory unit

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 Processing components

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.

3.1 Half 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 2.

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

Table 2: 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.

3.2 Full adder

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 3

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 3: 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 9. It requires only 5 logic gates.

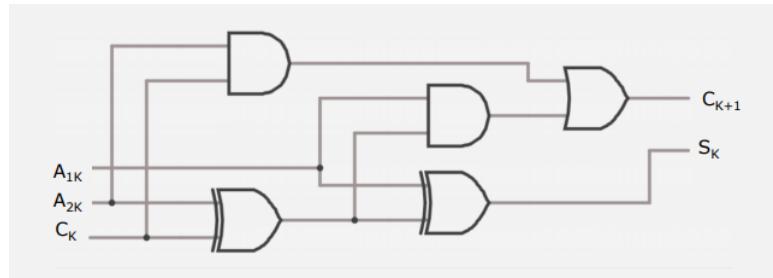


Figure 9: Most efficient way to create a full adder, by using two XOR gates, two AND gates and one OR gate

3.3 Ripple carry adder

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

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 10 you can see a ripple carry adder with a 3-bit input (where one of the bits is the sign bit).

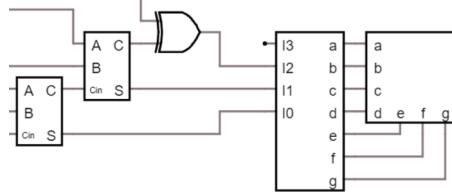


Figure 10: 3-bit ripple carry adder

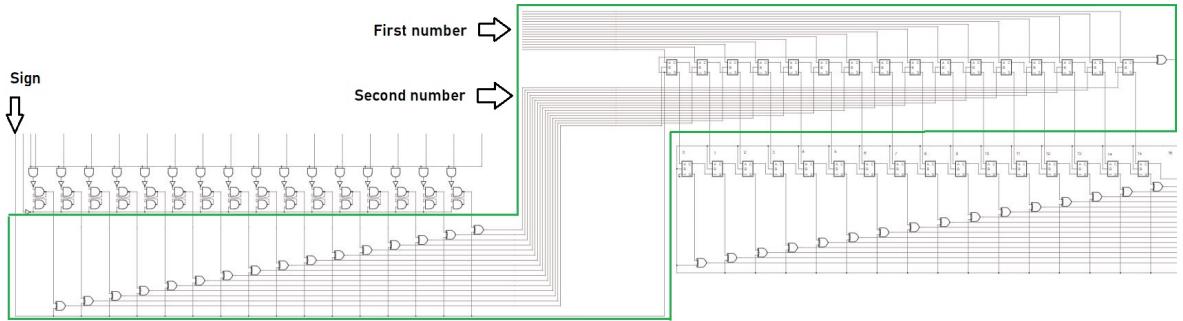


Figure 11: Picture the circuit processing part (highlighted in green)

The processing phase is split in two subphases:

1. During the first subphase the second number, memorized in the flip-flop memory (left side of figure 11), gets inverted if the user picked the subtraction operation, otherwise it is not get modified.
2. Then the algebraic sum happens in the top right side of the circuit.

Regarding the first point, the first mandatory step is to invert the number. We remember that the opposite of a binary number A is

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

and we also remember that in the Boolean Algebra factors can commute during a sum.

So, first of all, we perform the NOT operation in figure 12 thanks to XOR gates. They compare the sign of the second number with the bits, one at a time, and if the sign is negative (the sign line in figure 12, highlighted in red) they invert the bit. Otherwise nothing happens.

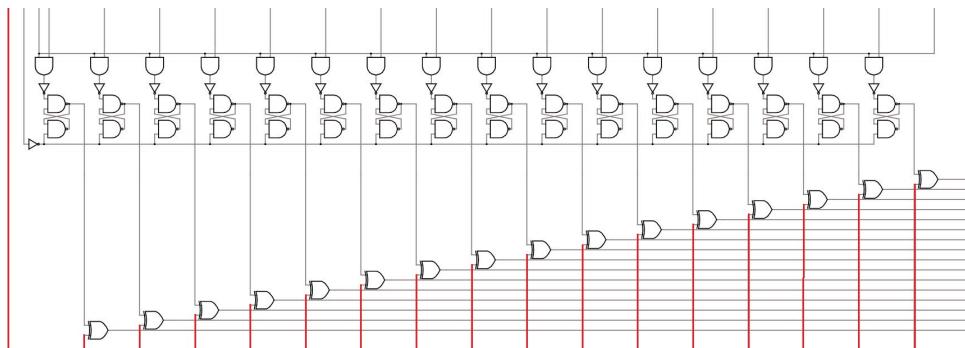


Figure 12: Picture of the inverting part, with the sign wire highlighted in red

Regarding the second point, this is not only where a unit gets added to the second number (equation 1) but also where the algebraical sum between the two inputs happens.

This sum is performed by a *ripple carry adder* (shown in figure 13 in green), which is composed by a series of full adders, highlighted in blue. The sign wire keeps being red.

The ripple carry adder (shortened in RCA) is an B -bit adder circuit composed of B full adders.

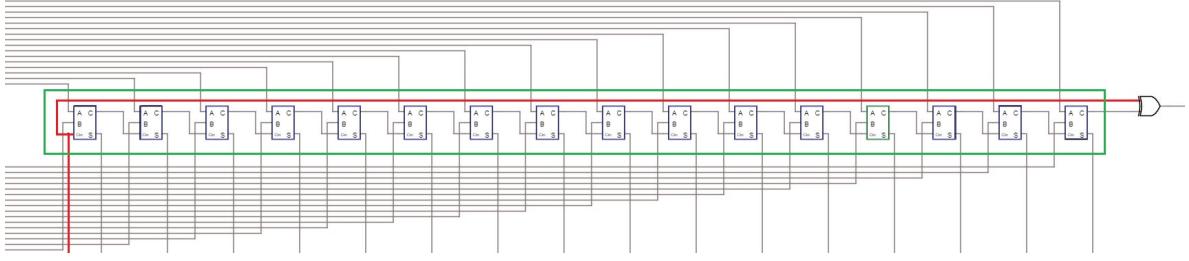


Figure 13: Picture of the ripple carry adder part, with the sign wire highlighted in red and the full adders highlighted in blue

A single full adder is a 3 input bits adder, with a 2-bit output. Its inputs are a number A , a number B and a carry-over produced in a previous operation, whereas the outputs are a result S and a carry-over C_{out} . Its truth table is figure ??, where the input carry-over is C_{in} and the output one is C_{out} .

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

By aligning B full adders is possible to sum (or subtract) two B -bit numbers. Considering that, as already mentioned, if the number is negative it is needed to invert the number and sum 1, the ripple carry adder allows the circuit to:

- Sum two numbers if they are both positive, with no action performed from the inverting part of the processing circuit.
- Subtract one number from another by inverting the second number, as already explained, and summing 1 as the first C_{in} of the first full adder of the RCA.

In conclusion, the output moves to the decoding part of the circuit, that will allow the circuit to show the result on displays. Moreover, a XOR gate checks the relation among the sign bit and the carry-over of the last full adder. This happens because the sign of the final result depends on the relation among the modules of the two numbers, and if the first one is bigger than the other, the result will be surely positive, otherwise negative.

4 16-bit calculator

4.1 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 increase-
ment 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.

4.2 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.

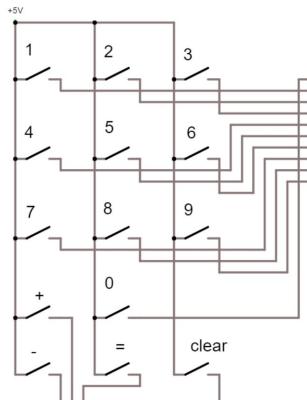


Figure 14: Picture of the simulated circuited keyboard

4.3 4-bit encoder

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

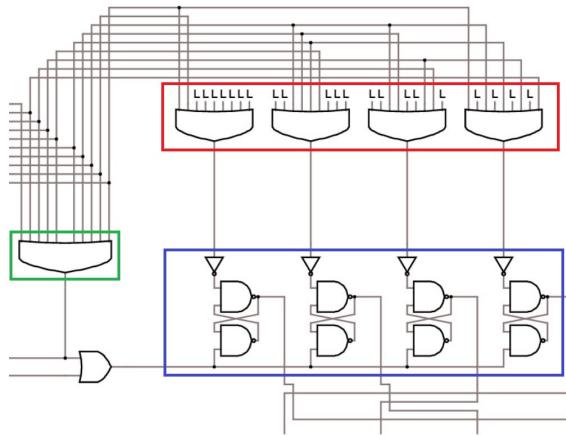


Figure 15: 4-bit encoder

As you can see in figure 15 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.

4.4 Memory and successive inputs

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

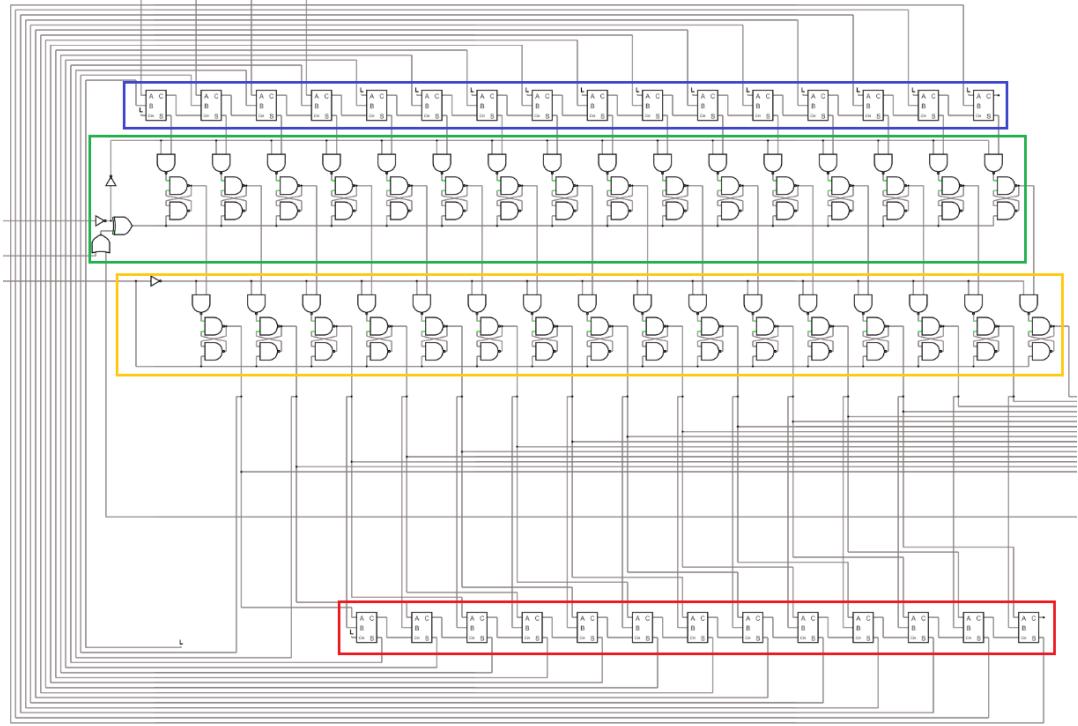


Figure 16

4.4.1 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 20), 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.

4.4.2 Falling edge

After 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.

4.4.3 Full adders

This circuit works with two full adders.

During a calculation, the full adders do not work at the same time. The first full adder, 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 full adder. 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 numbers is performed by the blue full adder.

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

Moreover, we can notice that after multiplying 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 full adder is needed because these binary digits are influenced by the 10x multiplication.

4.5 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 17 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.

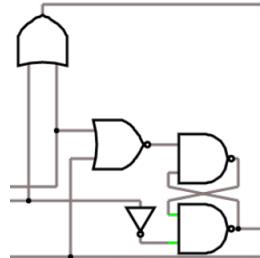


Figure 17: The component used to check a number sign, created with a flip-flop

4.6 Memory

The final part of the input is the two memories in figure 18.

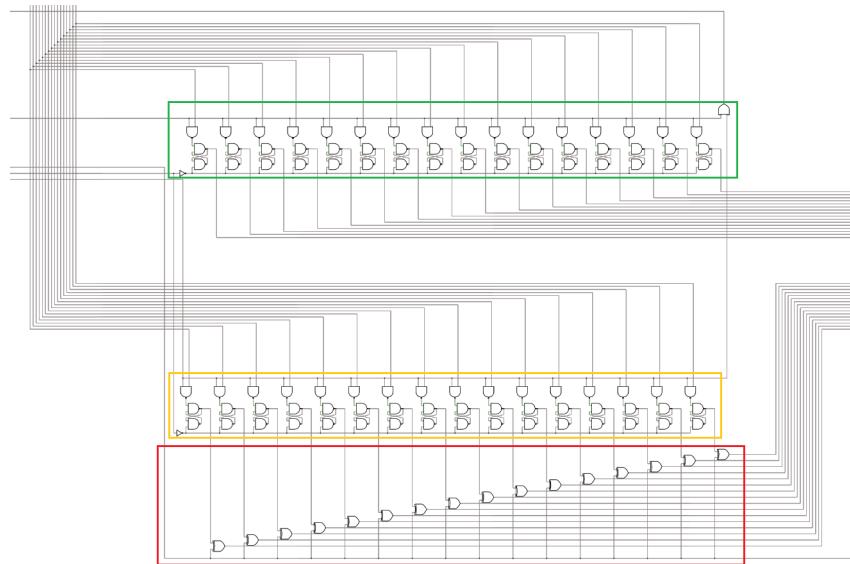


Figure 18: Final part of the input section, composed of two memories, one for each number that needs storage

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 AND wire that is controlled by the wire exiting the OR in figure 17.

With this, the memory will store information only when + or – are pressed. The wire is also connected to the right, side of the memory to a line that clear the memory from immagemaking it possible to input the second number.

The second memory, highlighted in yellow, works in the same way as the first one but take the check at the input is done with the = sign instead.

One difference between the first and second memory are the XOR gates beyond the second memory. A XOR takes as first input a bit and as second the sign bit. Then the XOR output will be the same as the first input if the sign bit is positive, and inverted if negative. This is done to perform an addition between a negative and a positive number.

To see the complete process about the addition of positive and negative number check section ??

4.7 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.

4.8 Decoder

After the inputs have been processed (added or subtracted), it is needed a decoder (see figure 19) in order to convert the output from binary number to its decimal form. Considering that this 16-bit calculator allows the user to insert numbers within the range (INSERIRE IL RANGE CAZZO), the operative boundary of this calculator is (INSERIRE).

The largest number that can be shown as result is around $\pm \text{INSERIER}$ ($+\text{INSERIER}$ or $-\text{INSERIER}$, to be exact), both positive and negative. So the decoding circuit needs to have 6 led displays for the digits and one extra display for the sign.

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. The major issue was to adapt this circuit to work with both positive and negative numbers.

The representation of the negative binary numbers follows the "two's complement" rule. It means that taken a B number of bits, all the numbers that can be represented with a 0 in the $B - 1$ position are considered positive, otherwise they are negative. This allows to split the 17 bits available in two sets of 16 bits, and so it is possible to have the range described above.

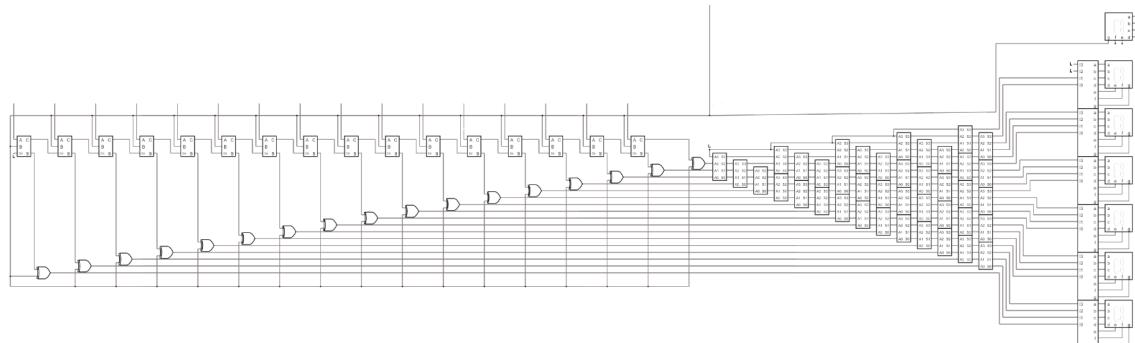


Figure 19: Picture of the decoder

4.8.1 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, the 16th bit.

This part of the circuit (represented in figure 20) uses full-adders and XOR logic gates, components that got already discussed in the previous sections. After the processing of the operation between the two inputs, the first 16 bits reach the A input of a specific full adder (the green lines in figure 20), whereas the sign bit follows the red path.

This 16th bit reaches every B input of all the full adders, and also the XOR gates, that compare the result of the single full adders with the 16th 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

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

give the final result, which will go to the double dabble. Considering the full adder output as "A" for the XOR, and the sign bit as "B", it is possible to deduct that when the sign is false (so the processing output is positive) the XOR gates do not modify the full adder output (which do not work too, considering that their "B", the sign bit, is false as well). When the XOR "B" is true (so the processing output is negative), the XOR gives the opposite of its "A" as output.

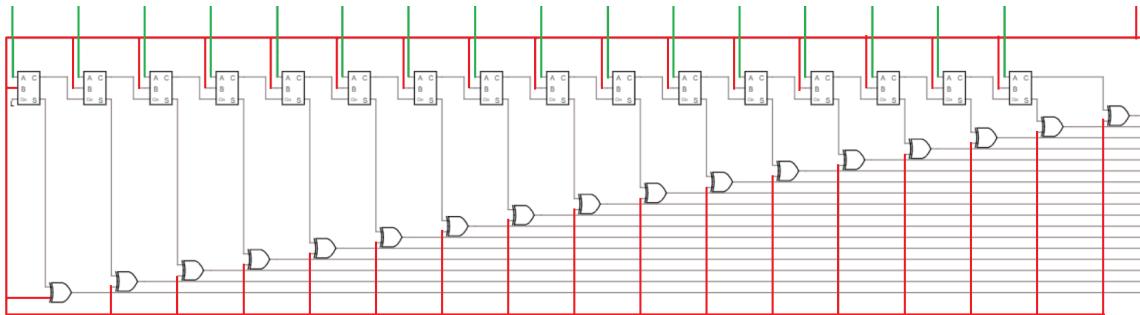


Figure 20: Picture of the first part of the decoder

4.8.2 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 21.

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 21.

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.

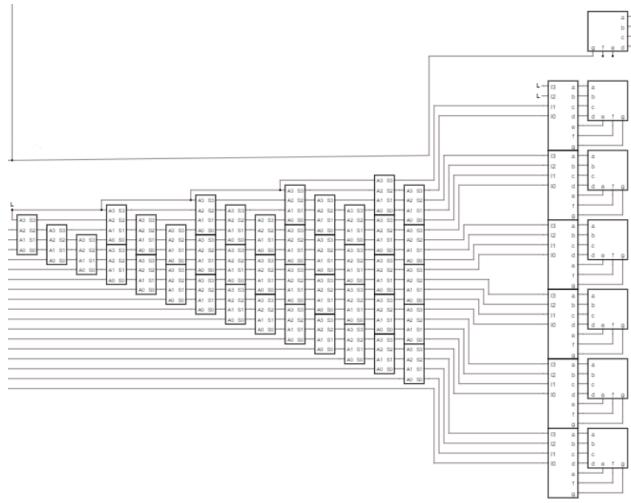


Figure 21: Picture of the double dabble, the second part of the circuit

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 21. If we consider the input A0 of a single Add3 component, when we see that its output, S0, is the input A1 for the next component.

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

#	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" \geq 5)
6	0000	0001	0101	11110000	Shift4 (every 4-bit slot < 5)
7	0000	0001	1000	11110000	Add-3 to "Units" ("Units" \geq 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" \geq 5)
11	0001	0010	0111	10000000	Shift7 (every 4-bit slot < 5)
12	0001	0010	1010	10000000	Add-3 to "Units" ("Units" \geq 5)
13	0010	0101	0101	00000000	Shift8 (every 4-bit slot < 5)

Table 4: 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 5: "Shift and add 3" truth table

5 Appendix

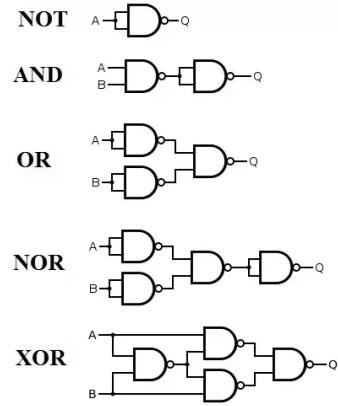


Figure 22: NAND gate is functional complete, every other logic gate can be a circuit made of NANDs only

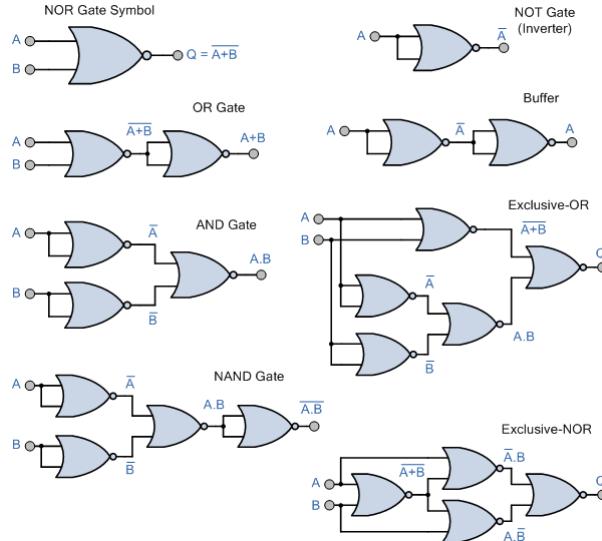


Figure 23: NOR gate is functional complete, every other logic gate can be a circuit made of NORs only

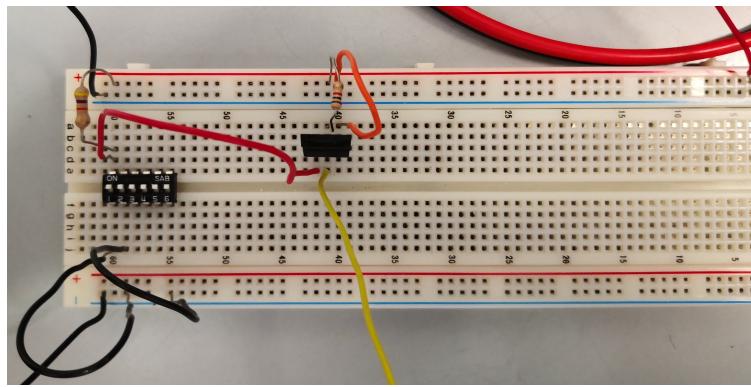


Figure 24: NOT gate realized in laboratory

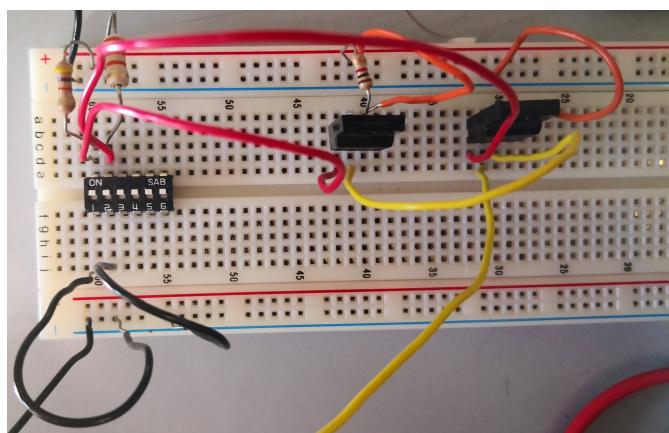


Figure 25: NOR gate realized in laboratory

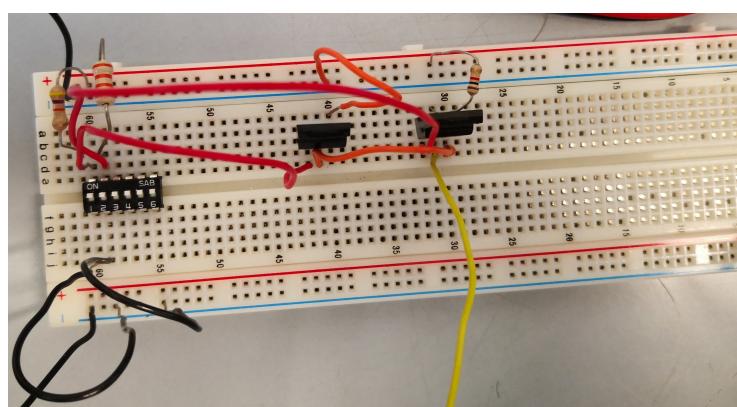


Figure 26: NAND gate realized in laboratory