# Report: adder and subtractor

Lorenzo Ramella, Alessandro Matteo Rossi, Marco Tambini

June 17, 2021

## Contents



# 1 Basics concepts

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 have to be properties of an electric circuit that can be "moved"; the easiest waof this proprieties to use is the voltage so 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 a OR gate.
- " ∗ " is the multiplication, and it can be represented by an AND gate.

## 2 Logic gates

When we talk about a logic gate, we are talking about a circuit that can take a certain number of inputs and give 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 NMOS

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

The NMOS 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 earthed.

When a positive voltage is applied between the drain and source, a deplition layer block the passage of current is formed and there is no passage of current. If we then start applying a positive voltage between gate and body, the electrons will start "balancing" the gaps in the P substrate, but there will still be no current flow.

After $V_{GS}$ (the potential difference among gate and source) surpasseses 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 ratio stops its growth and becomes almost linear as seen in figure and we find ourself in the region of saturation.

We didn't really need to differentiate in linear section and saturation section of the NMOS since what we will check is the voltage at the drain of the NMOS for our gate. The threshold voltage is different for every transistor, and usually is within the range $0.5\,\mathrm{V} - 5\,\mathrm{V}$. The transistor we used is the $IRF822$, and we found in lag that the threshold voltage is approssimatively , we decided to use $5V$ for $V_{GS}$ to make, as we will see later, input and output aproximativaly the same.

## 2.2 NOT gate

The easiest logic gate to realize is the NOT gate; we remember that the NOT gate negates the input, and so it works with a single input. We need a high enough resistance and one NMOS (as represented in figure 2) to realize it.

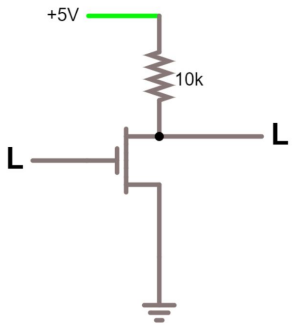



Figure 2: NOT logic gate realized using a NMOS and a resistor

When the input is 0, the NMOS 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 $5\,\mathrm{V}$, so we get a 1 output.

When the input is 1, the NMOS will let the current flow with a small resistance value; since

there is a higher resistance before the NMOS, almost all the voltage drop will happen 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 figure 3 instead and instead pass almost all the time in a 0 state.

A possible solution consists of a combination of NMOS and PMOS but, since our calculator does not have to work at such frequencies, we dismissed this problem.

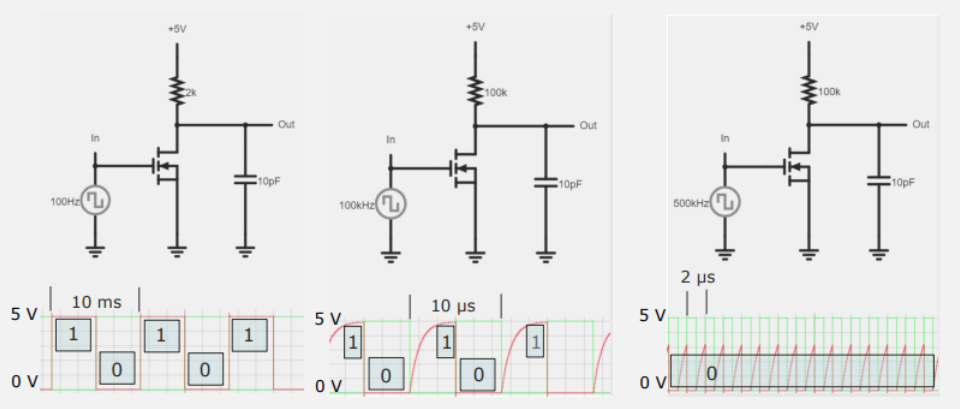



Figure 3: Inverter NMOS circuit with a resistive load

For the immage of the NOT gate and some result we got in lab check the end of the report at

## 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 4, by two NOT gates short-circuited at their output.

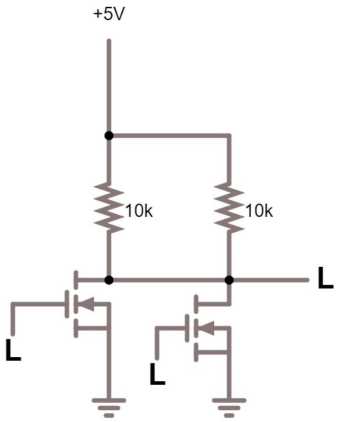



Figure 4: NOR logic gate realized using two short-circuited NOT gates

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

When one or both the inputs are 1 the NMOS 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 to ensure that whether an input is positive, the current will pass throught the "open" NMOS.

For the immage of the NOR gate and some result we got in lab check the end of the report at

## 2.4 NAND gate

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

The NAND gate is realized by a series resistor followed by two NMOS, where the source of the first is plugged in the drain of the second, as seen in figure 5.

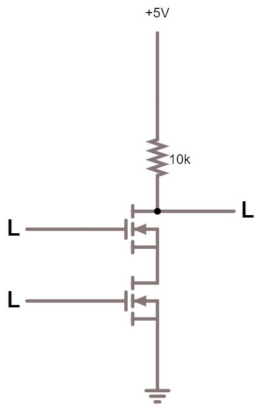



Figure 5: NAND gate realized with one resistor and two NMOS transistors

Whether 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 NMOS will be "closed" and, since the current cannot pass, the output will be 1.

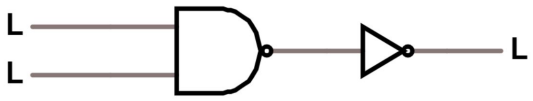
For the immage of the NAND gate and some result we got in lab check the end of the report at

## 2.5 Other logic gates

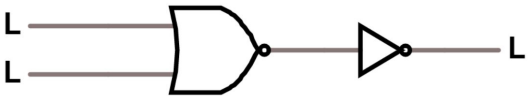
Being functional complete means that a single logic can be the only component in a circuit, and that circuit will be equivalent to every other logic gate. NAND and NOR gates have this property. We could have created the NOT gate with the circuit shown in figure **??** in Appendix, instead of using a customized one.

The only gates needed for the calculator are shown in figure 6, and they are:

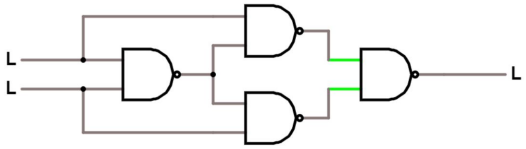
- The AND gate, made by the NAND gate, where the output is negated by a NOT.
- The OR gate, made by the NOR gate, where the output is negated by a NOT.
- The XOR gate, made by four NAND gates.




(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 6: The logic gates used in the calculator

## 2.6 Bistable circuit

For the 16-bit calculator, that will be discussed in section 3, 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 7).

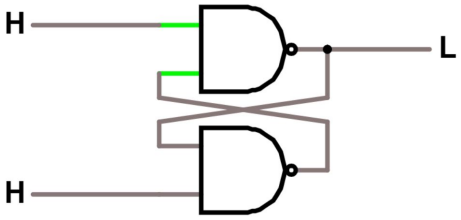



Figure 7: The circuit used to create a flip-flop memory unit

The flip-flop can be created by both NOR gates and NAND gates but, since the NAND gate 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 memory, as $Q$ and $Q'$ have the same value. Rows 3, 4, 5 and 6 represent the moment when an input is give, whereas 7 and 8 is when data is stored.

# 3 16-bit calculator

## 3.1 Input with 16-bit

Our goal is to create a 16-bit calculator. A calculator that could 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, it that this componentthat would araise in that of the phisical input since priority encoder would require as many input as the number of decimal number we want.

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.

## 3.2 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.

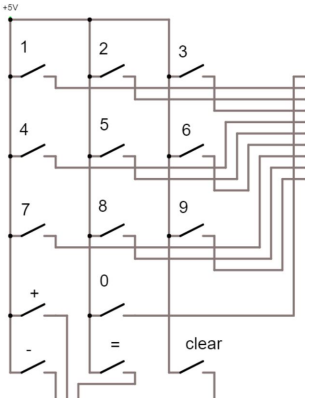



Figure 8: Picture of the simulated circuited keyboard

## 3.3 4-bit encoder

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

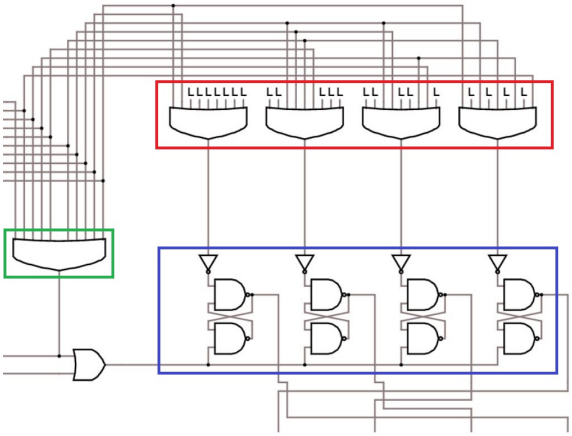


Figure 9

As you can see in figure 9 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 were pressed, it should not result in any binary input but it would be read as the number "10".

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

  This part, whose task will be discussed in the next subsection, 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.

## 3.4 Memory and successive inputs

This part allows the circuit to receive that make possible to get 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. Using this concept it is possible to process a single input in two phases.

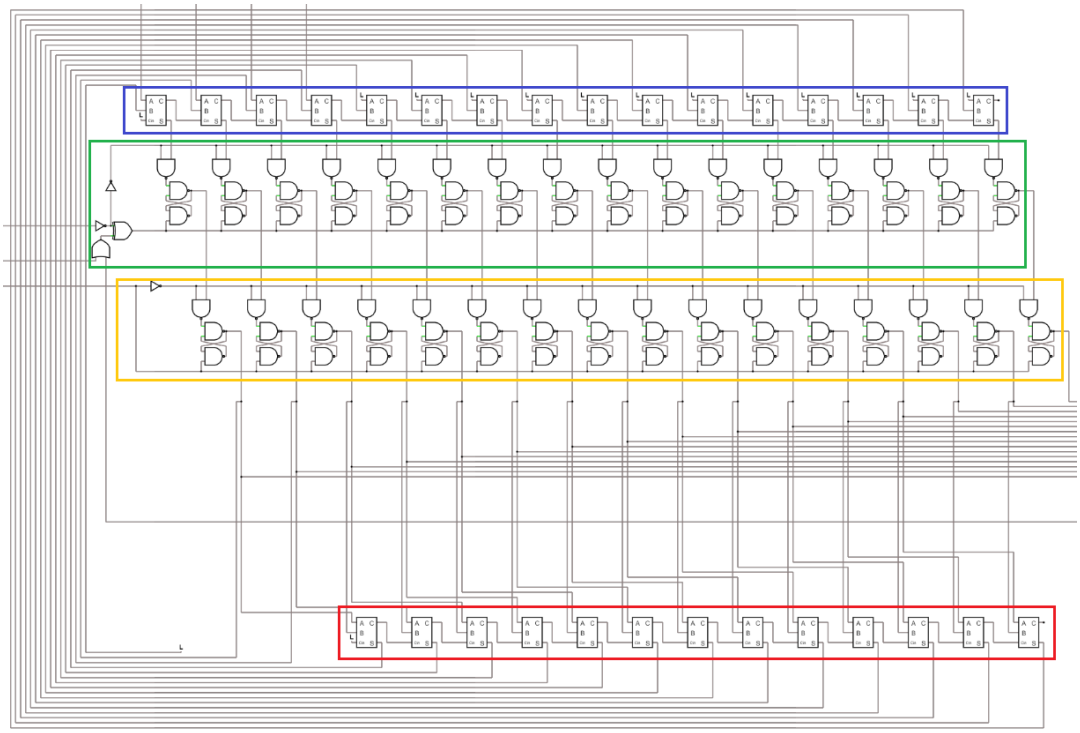


Figure 10

### 3.4.1 Rising edge

When the first button is pressed, the circuit is in a state of "rising edge". In this moment the first memory horizontal wire, that goes to all the NANDs of the second memory (highlighted in yellow in figure 10), switches to low to prevent the NAND 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 line is responsible for clearing the memory when it switches from high to low.

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

It's important to note that, during the data storage, the reset button is in a low state, since it cleared the memory right before the storage of new data.

This statement don't pose problem when the input is 0 since we find ourself 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 arise when the input data is 1 since we find ourself in case 1 or 2 where the two input are 0; as we can see in the truth table both output are set at 1.

After one of the two input switch from low to high the flip-flop return in a normal state; what we want is to go from the state above to the state 4 of the table of truth; this mean that, as long as the first line to return to high is the one for the reset, we see no problem.

The problem we 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 the first memory delay is performed by the clearing part on the low left side of the first memory.

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.

### 3.4.2 Falling edge

After the button is released the circuit finds itself in falling edge phase.

The process is similar to the previous one: the first wire to change this time is the one below the second memory. Its task is the memory reset, like the one in the first memory. After this, the wire on top of the second memory change its state to allow the last one (CHE SAREBBE?) to store information coming from the first memory.

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 during the rising edge and ends at 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 solution explained before.

### 3.4.3 Multiplication

The full adders work between the falling edge and the rising edge.

The second full adder, highlighted in red, is the one responsible of multiplying 10 to the previous number and, as already said, it works among falling edge and rising edge.

To do a multiplication in binary you need to shift the number as much as the bit of the multiplier and add all the result. For example let's take $6 \cdot 10$, writing it in binary we obtain $0110 \cdot 1010$. if we subdivide the multiplication if $0110 \cdot 1000$ and $0110 \cdot 0010$ we can simply shift the number. after this we add the two result $101000 + 001010 = 110010$ and the result that we obtain corresponds with 60 in decimal form.

To do this operation we simply let the input go to the second and fourth entrance of the adder and take in output the result multiplied by 10. Since the multiplier is 10 we can see that the first bit of the result will always be low and that the first three number of the result don't need any addition so we can directly take result without passing by a full adder.

### 3.4.4 Adder

The first adder, highlighted in blue, has the duty of adding the previous number, multiplied by 10, to the new number in input. This second adder works between the rising edge and the falling edge.

### 3.5 Sign bit

To do the subtraction we could have used the full subtractor, but we decided to use the addition between positive and negative number in binary. To do the subtraction we needed a way to read whether the second number was positive or negative. As visible in figure 11 we simply used a flip flop. From the top:

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

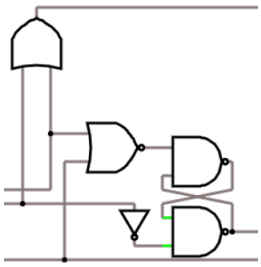


Figure 11: The component used to check a number sign, created with a flip flop

### 3.6 Memory

The final part of the input is the two memories in figure 12.

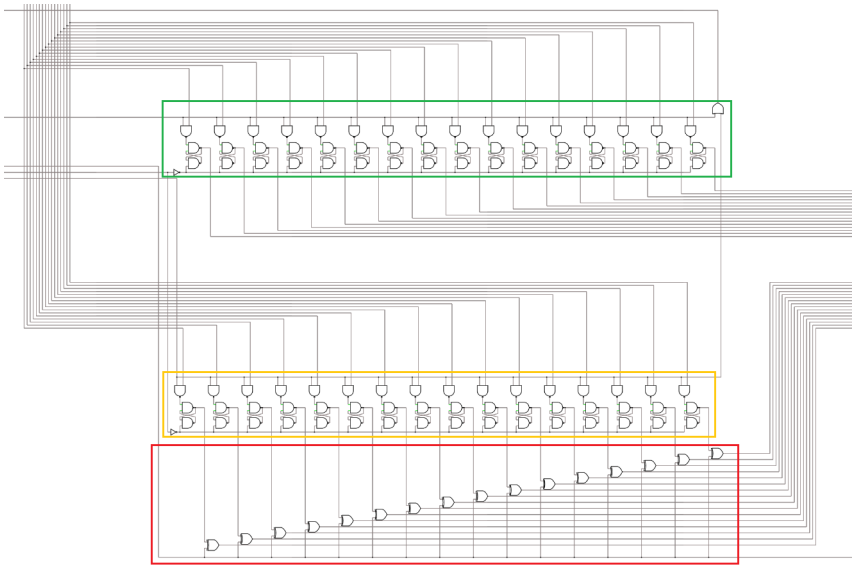


Figure 12: Final part of the input section, composed by two memories, one for each number that needs storage

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 11.

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 ??

## 3.7 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.