

Blasco Neural Logic Array

A Hardware-Based Computational Model Using Weighted Neuron-Inspired Circuits

Alejandro Blasco Griñán

Email: alexflop0799@gmail.com

04/08/2025

“Preprint submitted to Zenodo – August 2025”

ABSTRACT

This work presents an alternative computing architecture called the **Blasco Neural Logic Array (BNLA)**, inspired by biological neural networks and implemented using analog electronic components. Unlike the traditional von Neumann architecture, BNLA employs modular "neurons" built with MOSFETs, operational amplifiers, and Zener diodes to create logic gates, memory units, and arithmetic functions such as adders. The design enables distributed and parallel processing, analog signal modulation, and dynamically defined activation paths based on geometric configurations. A functional prototype was built and tested, demonstrating the system's viability both theoretically and physically. The architecture supports scalability, dynamic reconfiguration, and opens new possibilities for alternative computational models grounded in physical logic.

1. INTRODUCTION

Nearly 70 years have passed since the conception of computing as we know it today, and throughout that time, we have continued to use essentially the same system and logic. We may have added subsystems to improve efficiency, we've miniaturized, redistributed, and optimized—but always around the same core.

I refuse to believe this is the most efficient way to build a computing machine, and even less that it's the architecture of the future. All we need to do is look inside our own skulls to see that there are many ways to reach the same result—and many other ways to perform operations and emulate capabilities.

In this study, I present my discovery: a different architecture, its potential applications, and of course, a demonstration of how it works.

This system resembles the way a brain works much more than a conventional ALU, and the most interesting part is the demonstration itself—a moment that, personally, left me speechless.

2. OBJECTIVE

To demonstrate that computing can be achieved through divergent methods, and to put forward new approaches with the potential to improve upon current technologies. To that end, this work aims to show that the proposed system functions as intended and that it is viable using today's existing hardware.

3. THEORETICAL FOUNDATIONS

The brain is an incredibly impressive machine. If we compare its minimal computational units to those of a conventional CPU, we find radically different and vastly superior capabilities. Not only that—it's mechanical. Which basically makes everything we've invented look ridiculous. Imagine: a creature that eats pizza and has a head full of mechanical switches can think better than a near-perfect machine.

But seriously—if we truly believe that the current architecture is the right one to keep pushing forward, then I believe we're mistaken.

This architecture is based entirely on how the biological neuron works, so let's start there.

A neuron is nothing more than a trigger-comparator. It has inputs and outputs, and at its core there's a trigger that activates once a specific threshold is reached—a value that's common to all neurons. However, if neurons only worked like that, the brain wouldn't exist. They'd be useless. With N non-modulated inputs, the neuron would always fire regardless of which input axioms were active—making the whole system, at best, inefficient, and at worst, impossible.

In reality, neurons have another—far more important—property: they modulate the *weight* of each axiom for every output that connects to the next neuron. That weight is analog.

This principle—**input modulation**—is one of the fundamental concepts behind this new architecture.

Now let's imagine a mesh of these theoretical neurons, manufactured by us (i.e., a two-dimensional system). The first question that comes to mind is: *what's the layout?* But the better question is: *what's the optimal layout?* To answer that, we define three constraints that simplify the design and help us shape the structure:

1. The flow of information must move only in the positive X-axis (Y is irrelevant for now).
2. The structure must be two-dimensional.
3. The number of connections is fixed and limited.

After a lot of thinking and experimenting with combinations, I arrived at the following distribution:



Impressive, right? Okay—joking aside, defining a specific shape to base an entire architecture on is harder than it looks. Of course, other viable layouts could exist. This is simply the one I found to work, but I'm sure many more combinations remain unexplored.

We'll use this configuration to demonstrate how the system functions.

In the diagram, the points represent individual neurons, each of which behaves as follows:

4. ARCHITECTURE DESIGN

4.1. Electronic Neuron

1.- Fires/passes voltage when an input voltage threshold (V_{in}) is exceeded, constant across all neurons.

This is what defines the neuron's logic behavior.

2.- Has the same number of inputs and outputs.

This is crucial. Just because connections exist doesn't mean they are in use.

We need a symmetrical and consistent mesh on the plane.

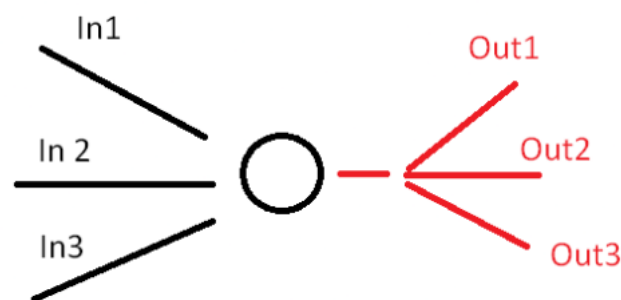
3.- Inputs must be voltage-regulated (analog weight).

The inputs to each neuron must be adjustable so the system can process information as described earlier.

4.- Inputs must be fully switchable (off-capable).

This is a consequence of rule #3: if we can regulate input voltage, we can also drop it to absolute zero—effectively turning the input off.

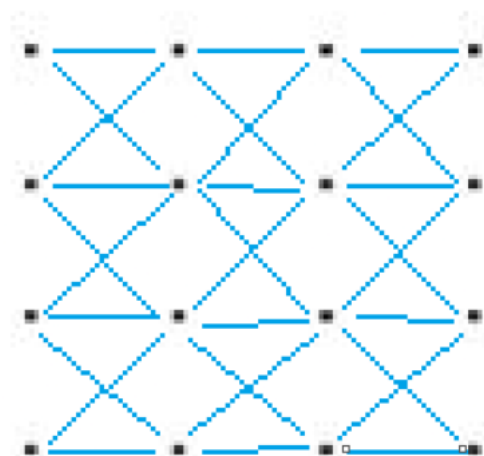
This feature is essential and grants the system **distributional plasticity**, enabling reconfiguration, grouping, and on-the-fly learning.



In the previous diagram, we also see a detail that hadn't been defined yet:

the number of input axioms—in this case, **three**, which is the **minimum required** to execute operations using the proposed layout.

Alright then—if we put all of this together... what do we get?



As shown in the previous image, the distribution of neurons and their associated axioms forms a symmetrically interconnected mesh along both axes.

We will now define each component of the neuron individually, along with its position within the system's functional diagram. We'll proceed from left to right, following the logical order of processing.

4.2 Control Unit: "The PUPPETEER"

As shown in the previous image, the distribution of neurons and their axioms forms a symmetrically interconnected mesh along both axes.

Now we'll define each component of the neuron separately, based on its position and role in the processing scheme. We'll proceed from left to right, in logical order.

An essential feature of the BNLA system is the ability to dynamically configure the activity of each neuron within the mesh. To achieve this, an external control unit is required—one that determines which axioms (inputs) are active and what weights they contribute during processing. This unit has been dubbed “**The Puppeteer**”, as it acts like a controller pulling the strings from outside the logical system.

The Puppeteer is **not part of the computational circuit itself**. Rather, it operates as a **programming layer** over the neuronal mesh. Its primary role is to assign modulation values to the inputs of each neuron, thereby enabling the configuration of specific operations such as logic gates or arithmetic units.

In the practical implementation, this unit is represented by an **Arduino microcontroller** in combination with a **PCA9685 PWM controller**, which allows independent duty-cycle regulation for multiple PWM outputs. These PWM signals are used to control the gate of each modulating MOSFET, ultimately determining the effective voltage that reaches the resistive summation unit of each neuron.

In this way, the Puppeteer serves as the **programming interface** for the neuronal mesh: it doesn't perform computations itself, but it **decides when and how** each part of the network engages, enabling anything from simple configurations to complex operations composed of multiple logic layers.

In future iterations, this unit could evolve into a more autonomous and adaptive control system—potentially capable of learning and managing structural reconfiguration within the network.

5. FUNCTIONAL IMPLEMENTATION OF THE LOGIC NEURON

This section defines the functional implementation of each neuronal unit within the BNLA system, based on the physical principles of modulation, summation, triggering, and transmission. This logical structure allows the construction of complex computational networks by repeating a single modular base unit.

Now that we've established the layout and constraints, let's examine the real-world viability of the neuron (the individual points in the mesh). If we can demonstrate that the neuron works in practice, we can reasonably assume that the overall structure is also viable.

So let's define what the neuron must do—and how we can achieve it physically:

1. INPUT MODULATION

Each input axiom comes from a different neuron and converges at a common point—one of the inputs of a new neuron. These connections have three main components: the conductor, the regulating MOSFETs, and resistors.

The MOSFETs serve as **current modulators**. Their role is not to determine whether a line carries a 1 or a 0, but rather to **multiply** the value that passes through by another value, also between 0 and 1, with near-infinite precision—thus assigning **an analog weight** to the controlled axiom.

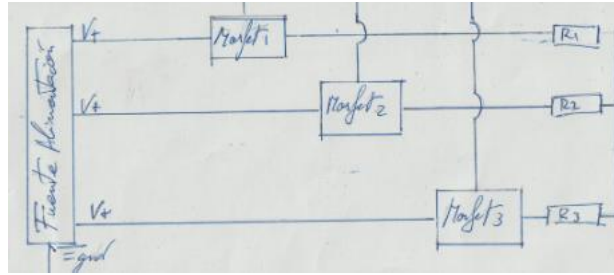
In short:

Imagine that an input carries 9V (representing a logic 1). The MOSFET, controlled by another circuit, multiplies this 9V by a value between 0 and 1 depending on the instruction being executed. This regulates the input voltage (i.e., the weight) of that particular line. This is achieved using MOSFETs.

MOSFETs are controlled via their gate (base), while the collector and emitter form the circuit path for the incoming signal.

As we know, if the gate receives no voltage, **no current flows**—meaning we can completely deactivate certain axioms. This enables the creation of **isolated parallel computing zones** within the network, and helps us meet one of our architectural constraints.

After modulation, we have resistors—identical across all axioms—whose function is to participate in a **resistive summation** circuit, which follows next.



2.-ADDING ENTRY VOLTAGE

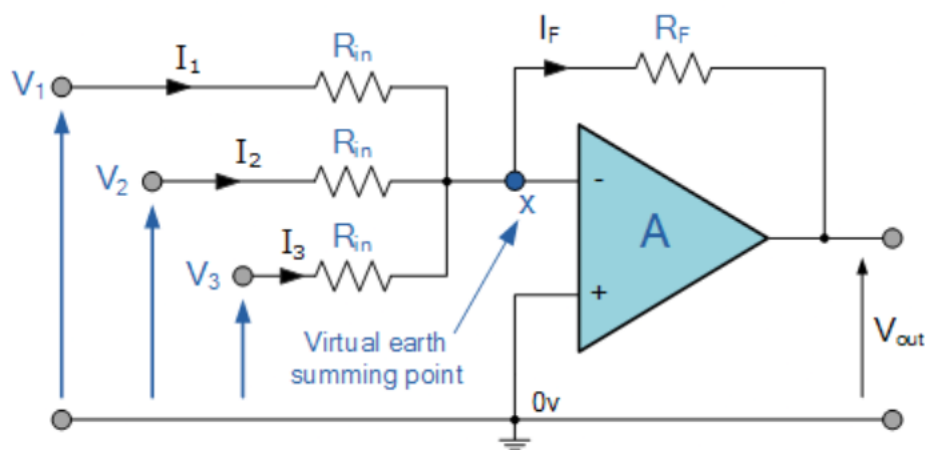
Voltages from parallel-connected lines cannot be directly summed, so we must rely on an integrated circuit known as an **operational amplifier (op-amp)**. The function of an op-amp varies greatly depending on how it's connected, but in our case the use is clear: we will employ a **resistive summing configuration**.

$$-V_{out} = \left[\frac{R_F}{R_{in}} V_1 + \frac{R_F}{R_{in}} V_2 + \frac{R_F}{R_{in}} V_3 \right]$$

As shown by the behavior of the circuit, when the **feedback resistor (Rf)** is equal to each **input resistor (Rin)**—and all input resistors are equal—the result is a **summation of voltages**, each multiplied by a fixed ratio.

$$-V_{out} = \frac{R_f}{R_{in}} (V_1 + V_2 + V_3)$$

each multiplied by a fixed ratio,



This circuit is extremely useful, as it allows us to kill two birds with one stone: we can sum voltages from parallel lines, **and** adjust the overall gain through the resistor ratio.

$$R_f/R_{in}$$

3. TRIGGERING

This part is simple: a **Zener diode**, matched to the gain of the op-amp, is more than enough to serve as a trigger mechanism at a specific voltage level.

4. FEEDING THE NEXT NEURON WITH REGULATED VOLTAGE

Now, this part is a bit more complex. The key lies in feeding the next neuron with a **controlled voltage** that represents a logic 1.

Due to gain variation and the fact that the input MOSFETs modulate the voltages being summed, the output voltage after the Zener will range from its breakdown voltage **up to an unknown maximum**. That's a problem—we need the **next neuron to receive the same stable value** as the one originally input.

For now, we assume the use of **electromechanical switches** to directly allow power from the supply to pass through based on the Zener's triggering.

I'm aware that this could be done with transistors, but due to my limited knowledge of electronics, I haven't yet found a viable combination.

6. THE PROTOTYPE

After defining the theoretical behavior of the neuron and its components, I decided to build a prototype of this model—reinforcing the system's functionality as described.

1.- SCHEMATICS

Here we see the input schematic for the neuron:

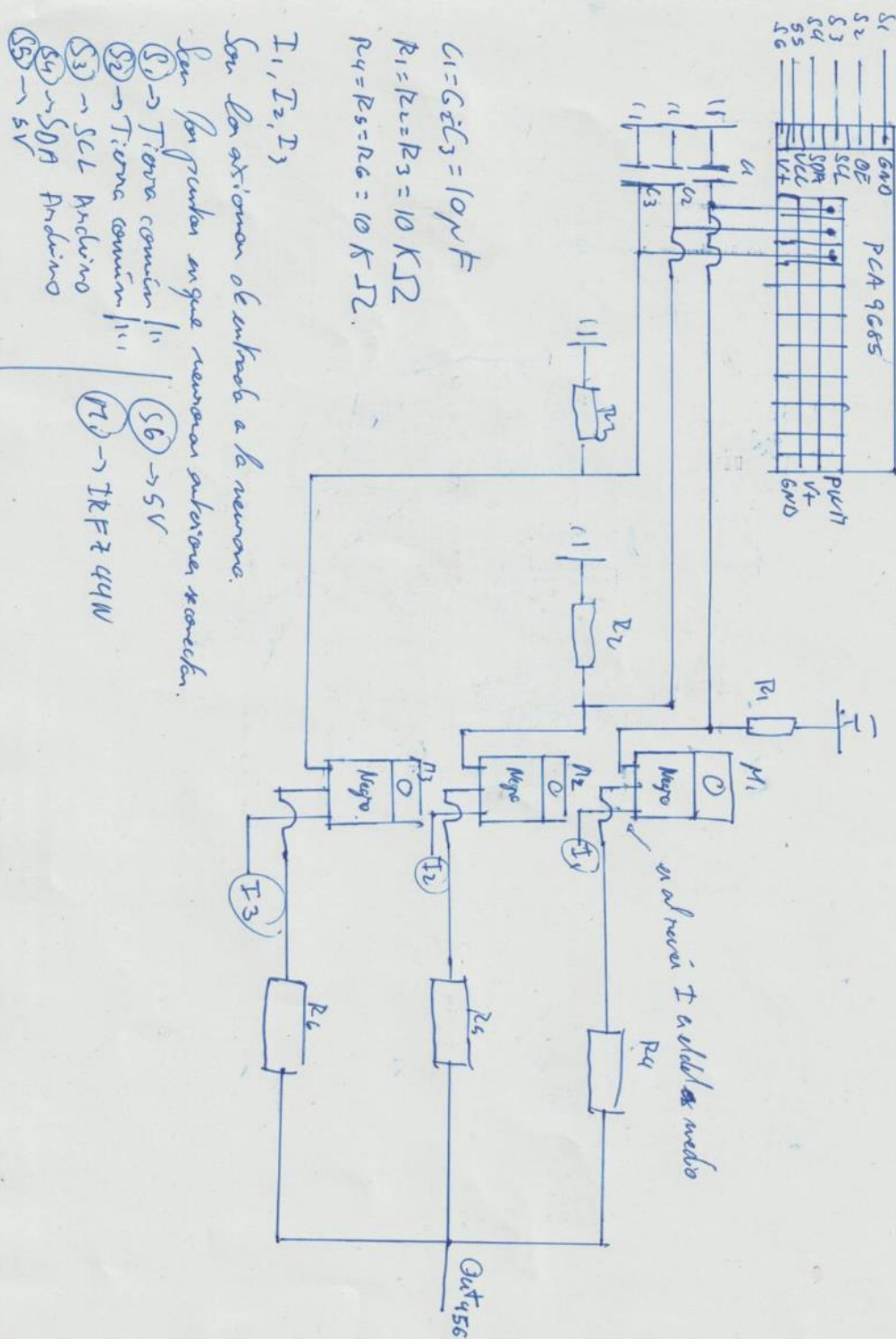
To control the MOSFETs—and activate each one independently with minimal Arduino wiring—we use a **PCA9685**, a 16-channel servo controller (expandable), which gives us individual control over each axiom. This controller outputs a fixed supply voltage and varies only the **duty cycle**, which effectively adjusts the average voltage. By placing capacitors on the control lines (C1, C2, C3), we smooth the PWM signal and stabilize the output.

Each control line also includes a **pull-down resistor** to prevent false positives at near-zero voltage.

Now for the MOSFETs: we use **IRFZ44N** (ideally, logic-level versions), which modulate the summed voltage.

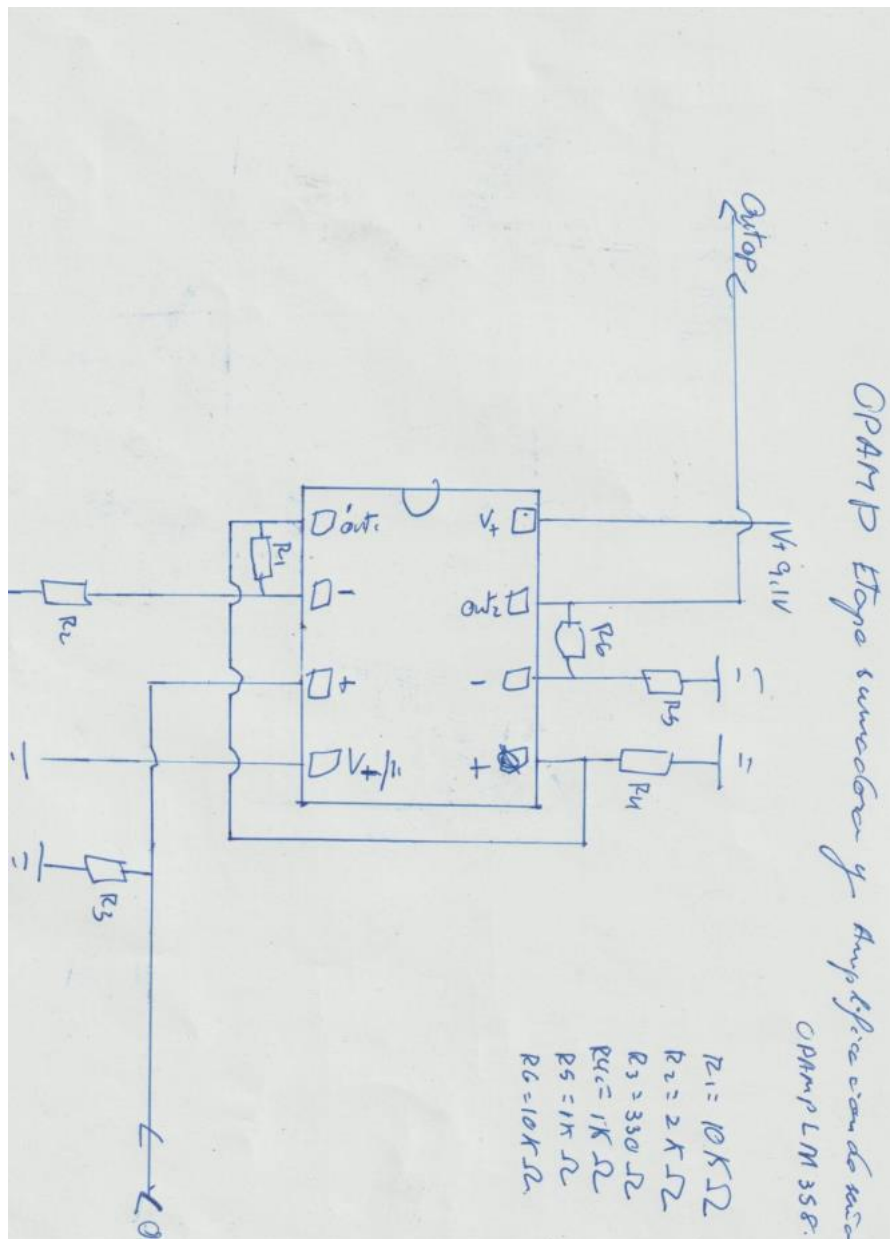
Inputs **I1, I2, I3** come from previous neurons, and the lines connected to resistors **R4, R5, R6** represent the outputs that will be summed—**OUT456**.

PCA 9685 + Controlador de velocidad



Now we move on to the OP-AMP summation stage. In this case, I use a **dual LM358**, employing both op-amps: one as the summing amplifier, the other as a standard operational amplifier.

(There are definitely better ways to do this.)

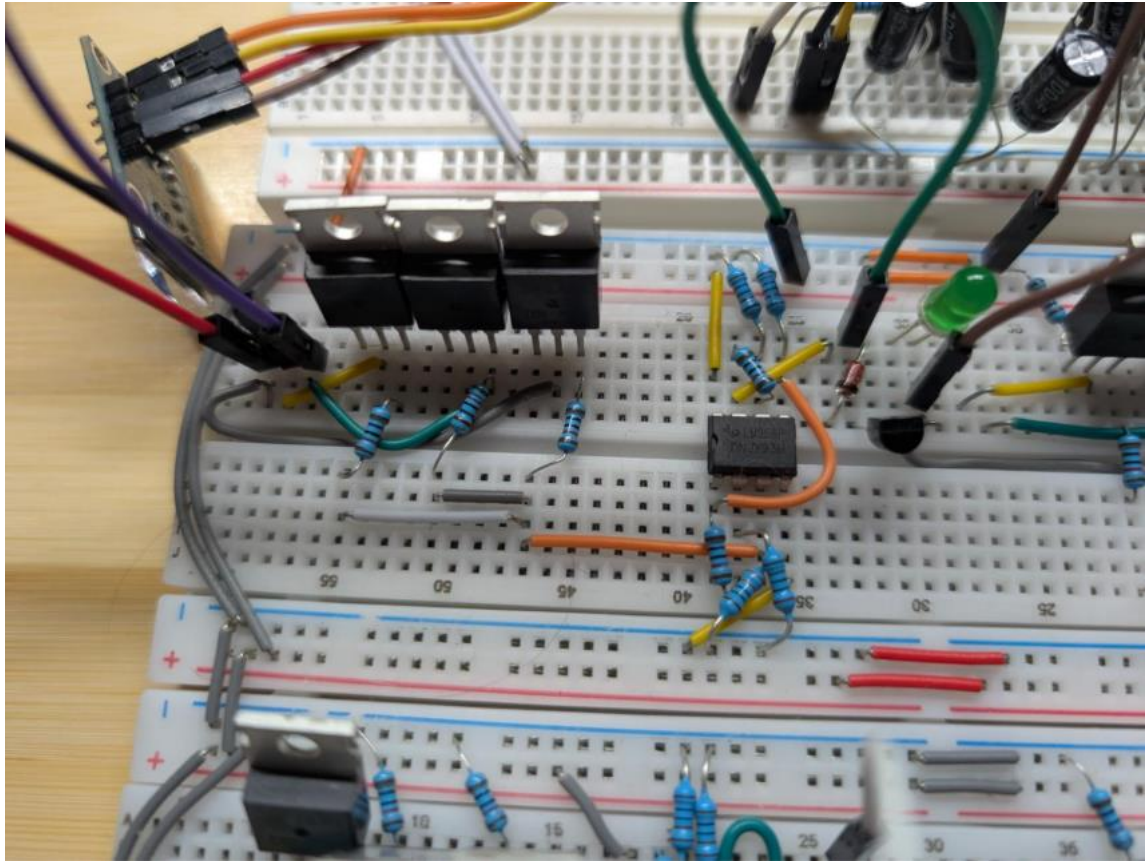


You have to look closely, but the op-amp is connected in a **non-inverting summing configuration**, followed by an amplification stage.

After that comes the **triggering and output stage**, which feeds the next neuron. This is handled by a **Zener diode** (in my case, 4.7V) connected to a switch.

Now let's get to the physical build.

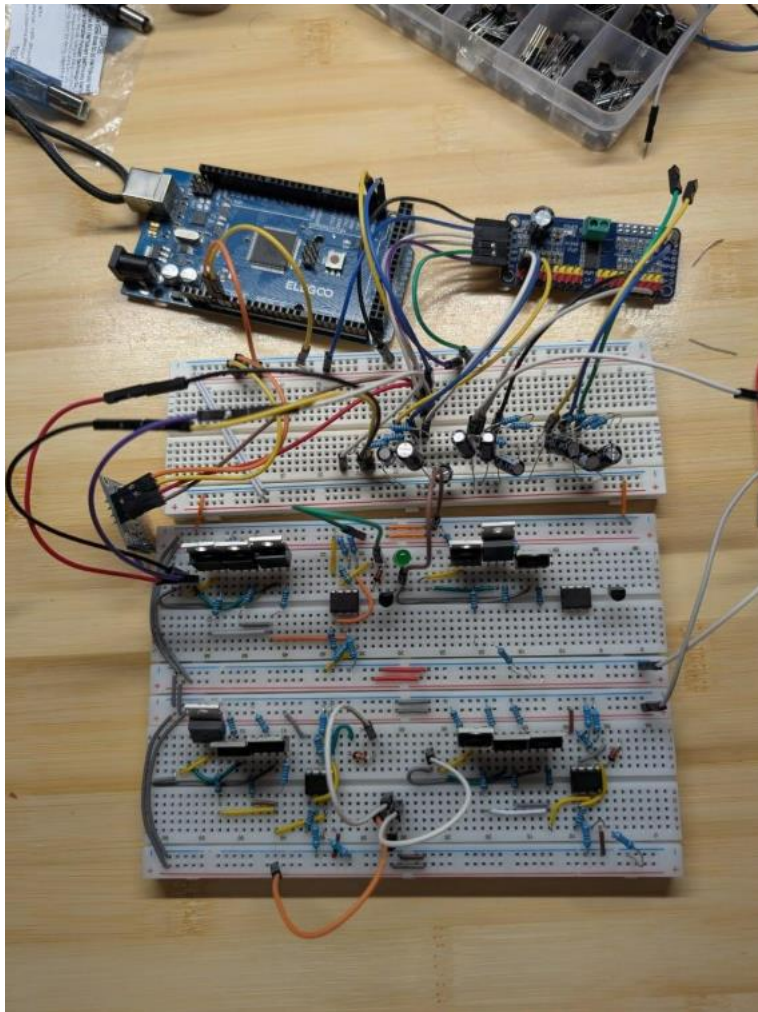
Below, you can see a **discrete neuron on a protoboard**.



And in the following image, a **4-neuron array** alongside the **PCA9685** and the **Arduino "Puppeteer"**.

In each neuron, I've experimented with different types of final switches to power the next neuron.

So far, no clear success—**unless using electromechanical switches**.



What **has** been achieved is successful triggering when expected, using a basic control script.

```
→ Arduino Mega or Mega 2...
NEURON_WORKS.ino
33   delay(10);
34
35   // Activar pines 0, 1 y 2 a 100% duty (ON c
36   pwm.setPWM(0, 0, 4000);
37   pwm.setPWM(1, 0, 4000);
38   // Activar pines 3, 4 y 5 a 100% duty (ON cons
39   pwm.setPWM(3, 0, 4000);
40   pwm.setPWM(4, 0, 4000);
41   pwm.setPWM(5, 0, 4000);
42   // Activar pines 6, 7 y 8 a 100% duty (ON cons
43   pwm.setPWM(6, 0, 4000);
44   pwm.setPWM(7, 0, 4000);
45   pwm.setPWM(8, 0, 4000);
46
47
48   display.clearDisplay();
49   display.setCursor(0, 0);
50   display.println("Pines activos:");
51   display.println("PCA9685 CH 0 -> ON");
Output
```

The code used varies the **duty cycle** by modifying the final digit of the signal.

Results:

- For **duty cycle = 2500**, the neuron fires when **two inputs = 1**
- For **duty cycle > 3000**, the neuron fires with **one input = 1**
- For **duty cycle = 2000**, the neuron fires only when **three inputs = 1**

Therefore, the experiment is considered a success.

There are more complications when several neurons are connected in series, but the experiment's objective was to **demonstrate that this single point on paper is viable**—that the neuron works both in theory and in practice, fulfilling the required and defined properties.

As a result, **that point works**—and we can design logic circuits with it, just like we would with an AND gate.

And this is where the real revolution begins.

7. COMPUTATIONAL LOGIC

Now that we've clearly defined the structure of our mesh, explained how the neuron works, and demonstrated—both theoretically and experimentally—that it behaves as expected...

Why not show how this system operates before moving forward?

There are several ways to demonstrate that an architecture is valid—such as proving it is **Turing-complete** or equivalent. However, that would involve significant conjecture and effort, requiring us to design memory modules, decoders, and other auxiliary circuits.

Instead, we'll demonstrate the **viability of the architecture** (note: not the neuron itself—we've already validated that) by **emulating operations** that traditional computing machines are known to perform well.

In this case, and perhaps the most interesting and representative of computational science, we'll **emulate an ALU** and one of its core functions.

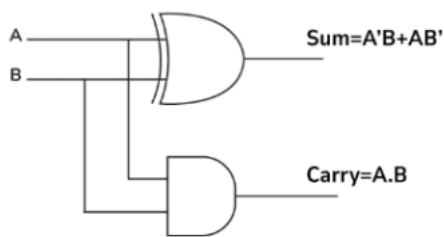
In short, we will **emulate an ALU whose purpose is addition**.

So, let's get to it.

How, using a network like this, can we perform the **simplest possible addition**?

7.1. EL HALF ADDER DE 1 BIT (Convencional computing)

Half Adder



Truth Table

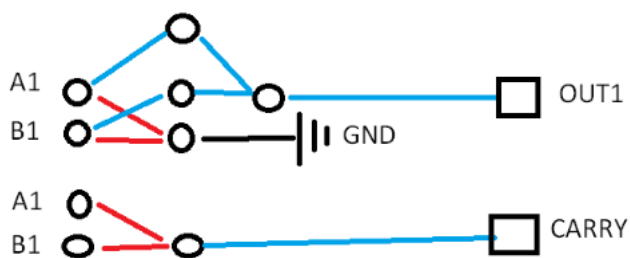
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

This is probably the **simplest adder that exists**: a **1-bit Half Adder**, executed by an **AND gate** and an **XOR gate**, which together produce the logic table shown in the previous image.

Simple, right?

Okay—let's prove that it can be done in our system.

In the next image, we'll see a schematic showing how we can add two 1-bit numbers, **A + B**.



We know that the axioms, being analog, are **infinitesimally adjustable**.

However, for the sake of precision and simplicity, we'll reduce them to **three states or multipliers**:

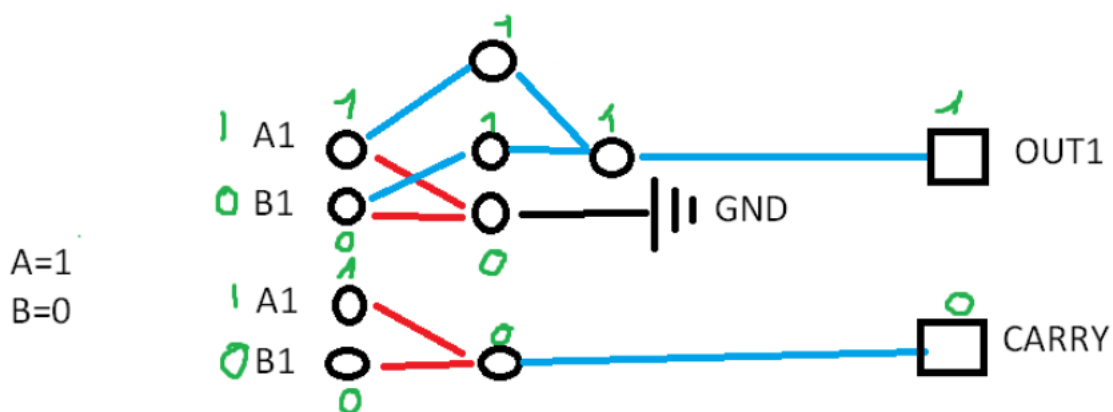
- **Off** → **x0** (*NO LINE PRESENT*)
- **Medium** → **x0.5** (*RED LINE*)
- **On** → **x1** (*BLUE LINE*)

Each axiom's voltage is multiplied by this value to apply the corresponding weight. Therefore, when designing or modeling layouts, we don't need to deal with actual voltages—we can work directly with these multipliers.

The neuron's **trigger value is set to 1**.

So, at each neuron's input, the sum of all axiom multipliers produces a single value, which may be **greater or less than 1**, thereby **firing the neuron or not**.

This is illustrated in the previous diagram using an example with **A = 1** and **B = 0**.



Here we see that **blue paths always activate** if there is a logic 1 in the previous neuron, while **red paths require both preceding neurons to be active** in order to conduct and let the signal pass.

In the upper part of the circuit, this results in a connection to GND, producing **OUT1 = 0** and **CARRY = 1**.

Wow—an actual, functioning half-adder.

I encourage you to test more input values yourself—just draw it out on a whiteboard.

Now let's explore the design and its implications.

First, we can see that we are dividing the operation **A + B** into simpler or more fundamental sub-operations.

Unlike conventional ALUs, which execute these in **sequential clock cycles**, here they are executed **in parallel**.

Each instruction is completed in a single "cycle"—dramatically increasing computational efficiency.

We also observe that we need to **multiply the number of inputs** to enable this parallel processing.

This is not difficult to achieve if we dedicate certain neurons prior to the computation layer to **split and organize** the input signals accordingly.

By the way—you're welcome to try performing additions yourself using this system.

" THE PUPPETEER"

As we've seen, we've configured a neural network to execute a specific operation—and doing so requires control over **which axioms are activated and to what extent**.

This task falls to "**The Puppeteer**": a specialized control unit responsible for activating and modulating sectors of the mesh for each instruction (and, as we'll see later, for learning as well).

We can therefore assume that this control unit allows us to **direct the flow of information** and **segment the mesh into sub-operations** in order to execute a given instruction.

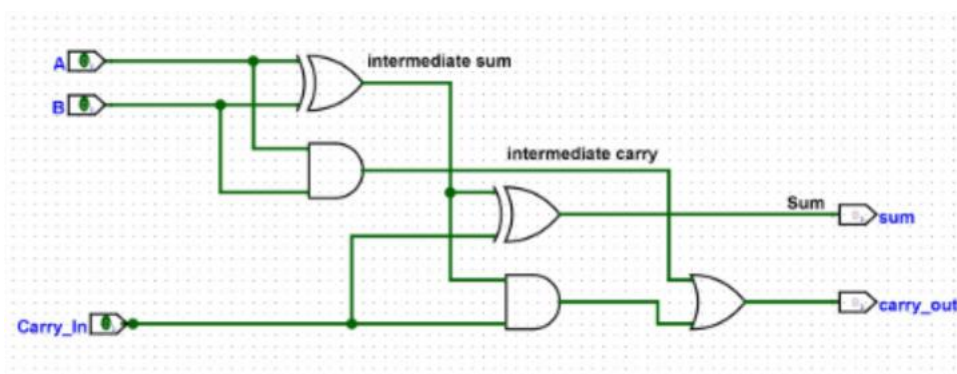
Alright—but even that is not enough.

And here's where things get *really* interesting.

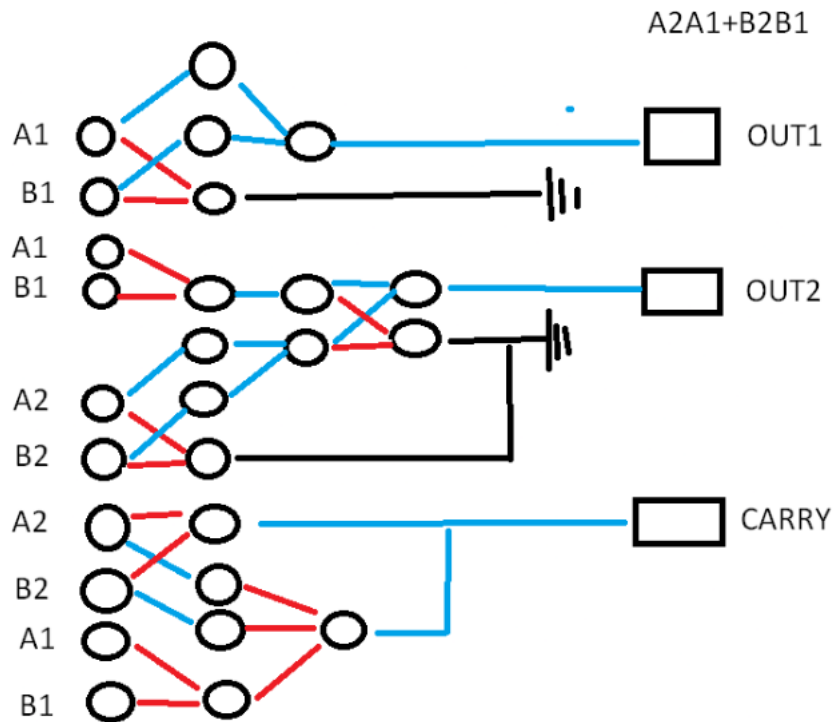
7.2. TWO-BIT FULL ADDER DEMONSTRATION

Now we're getting serious: a **full adder**, and not just any full adder—a **2-bit** one.

CONVENTIONAL COMPUTING



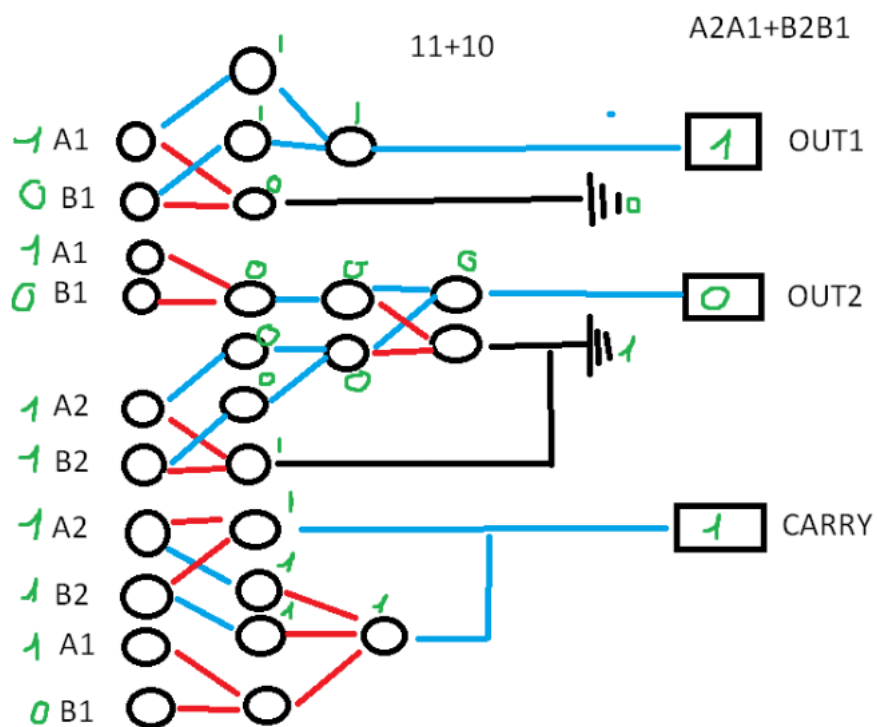
BNLU ARCHITECTURE



Where **red** represents 0.5 and **blue** represents 1—
please try it for yourself, it's incredible.

Let's run a demonstration:

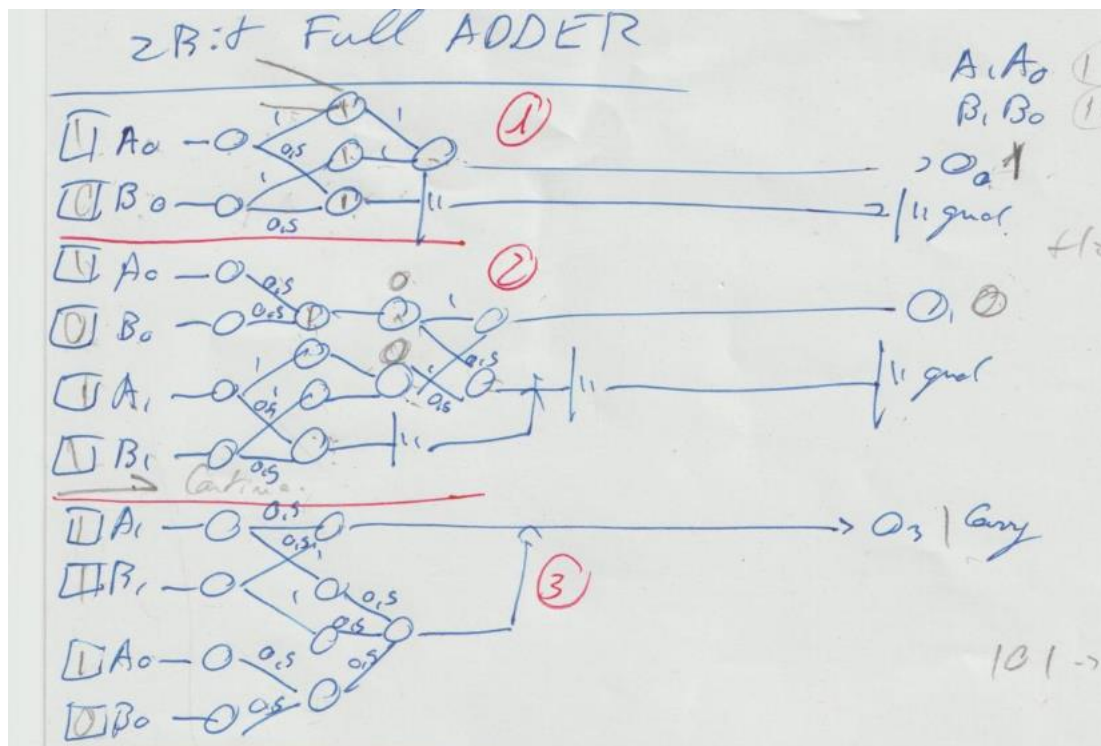
A = 11, B = 10 → That's 3 + 2



IT WORKS!

Seriously—try it, it adds.

Here we see a different schematic.

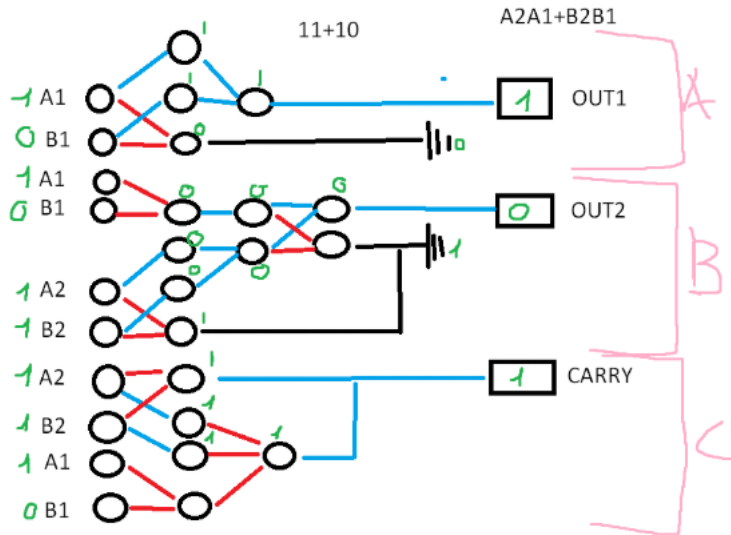


Let's explore it a bit further.

We notice that the number of **inputs has grown exponentially**, from 4 to 10. This makes things a bit more complex, but it's important to observe these details to determine the **minimum mesh required** to perform such an operation.

Let's dive deeper.

If we look closely, since this is a **full adder**, we can identify **three main sections**.



Section A is responsible for summing the **first bits only**.

Section B handles the **carry** from Section A, **repeating the addition internally** while summing the next bits along with the carry from the previous result.

Section C calculates the **final carry**, influenced by the sum in Section B and its own carry.

What this means is: for **A + B** where the number of bits is **greater than 2**, we simply **repeat Section B** as many times as there are additional bits.

In other words, there will always be one **Section A** and one **Section C**, but **Section B** can be repeated.

This allows us to define the number of inputs required for additions where digits > 2

$$Row = (X + (4 \cdot (X - 2)))$$

And for $X < 2$

$$Row = (2^x + (x - 1)) \cdot 2$$

With this, we can **define the minimum mesh size** required for the operation (assuming a square configuration).

Hemos demostrado que esta arquitectura y consigue igualar e incluso superar las funciones de cálculo de la computación tradicional.

EXPERIMENT CONCLUSION

We have demonstrated that this architecture not only **matches**, but may even **surpass** the computational capabilities of traditional systems.

Feasibility & Current Challenges

We can clearly see that this structure is more than viable—but it still presents certain complications for real-world execution:

- **Parallel Input Duplication**

This is the most noticeable issue—not just the number of inputs, but the need to replicate them at specific positions in the mesh.

This is solvable: I’ve developed a couple of **encoder and decoder** designs, which we’ll review later in the “Other Circuits” section.

- **Load on the Puppeteer**

This remains to be fully tested. It has to control many inputs.

For instance, in a mesh of 70 neurons, the controller must handle three times as many axioms—**210 in total**, which could present **performance limitations**.

- **Potential Speed Issues**

The **switching speed, storage, and processing time** for operations remain uncertain.

We are, at the very least, adding **an extra step** compared to conventional computation: the **reconfiguration of the mesh’s morphology**—and that takes time.

Advantages & Emerging Properties

For any **$N \times A$** mesh, this architecture allows for **indefinite parallel operations**.

For example, if adding two 2-bit digits requires a height of 10, and our mesh is **50×50**, we can execute **five additions in parallel**, effectively **doubling** computational power and **scaling linearly** with mesh size.

Moreover, the system can be programmed so that the same operation (e.g. **1 + 1**) occurs **in two different locations**, with one giving a deterministic result based on predefined instructions, and the other using a **slightly randomized configuration**.

This allows for **comparisons of energy efficiency, spatial use**, or other variables.

In essence, the mesh can **learn to self-regulate physiologically—just like a brain**.

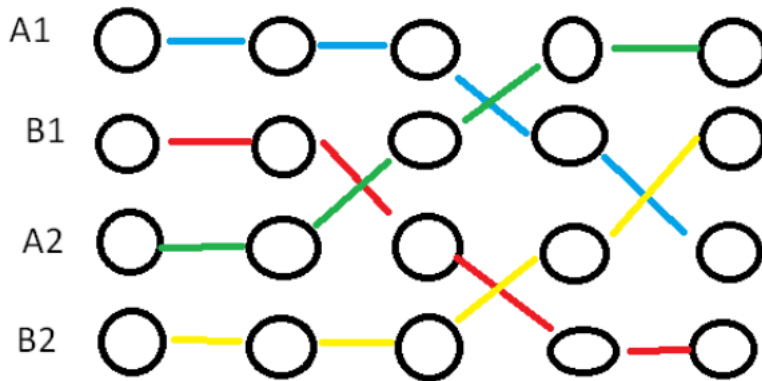
The design of additional circuits is not strictly necessary, but it could **enhance performance**, particularly in areas like memory.

Internally, the system can **emulate cache** by keeping certain neurons **always active or disabled**, thus behaving like **dynamic, delocalized memory bits**.

8. MEMORY AND AUXILIARY STRUCTURES

This section presents **other useful and relevant structures** that enhance the operation of the neural network.

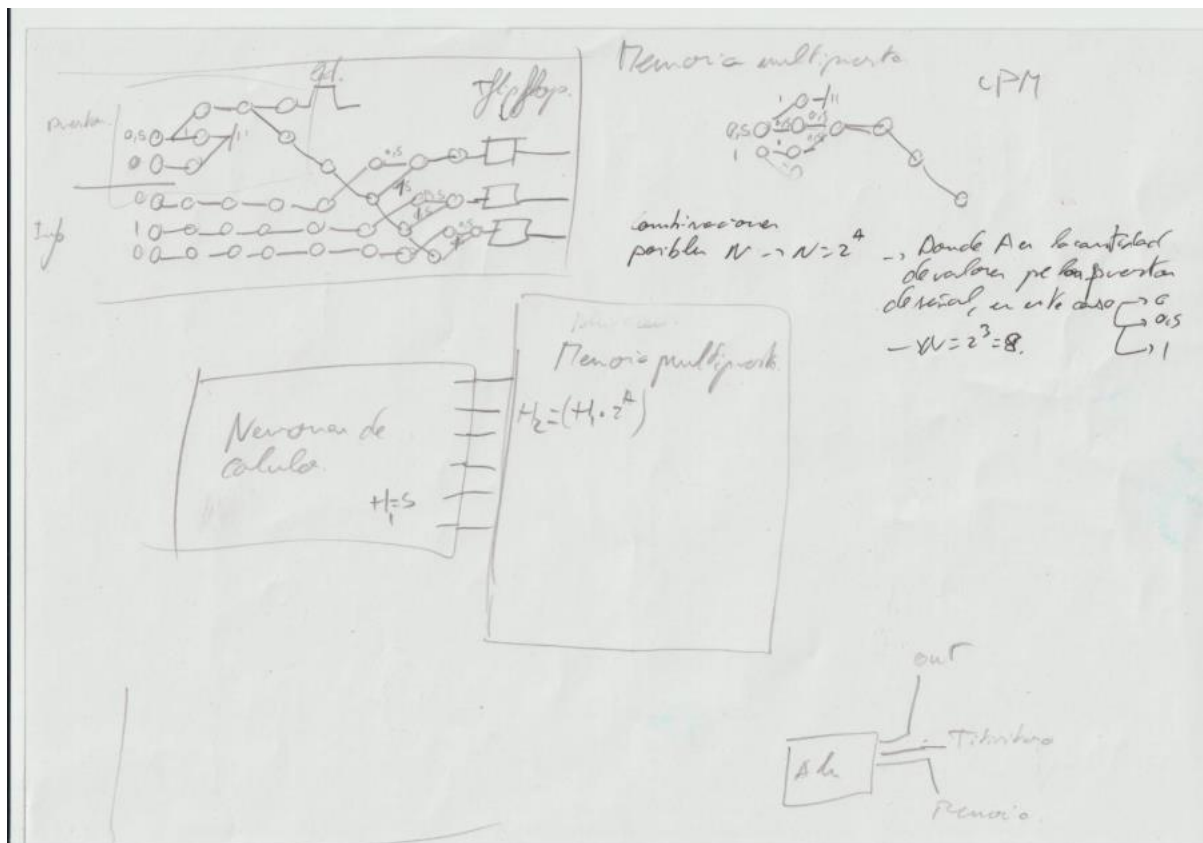
- Blasco Bridge



Its function is to enable the **transfer of information along the Y-axis**, especially when **more than one bit** needs to be transmitted.

In such cases, **bridges of this type** are created to allow smooth data flow through the network.

- MEMORY CONCEPT



As you can see, I've also designed a **memory system**.

Its operation is straightforward:

We have **two gate channels** and **three data channels** for storage—though the number of data channels can be **theoretically infinite**.

The gate channels use combinations of **0, 0.5, and 1** to activate specific gates.

For example, there is one gate that enables **write access** to a memory cell **only** if the combination is **0.5 and 1**.

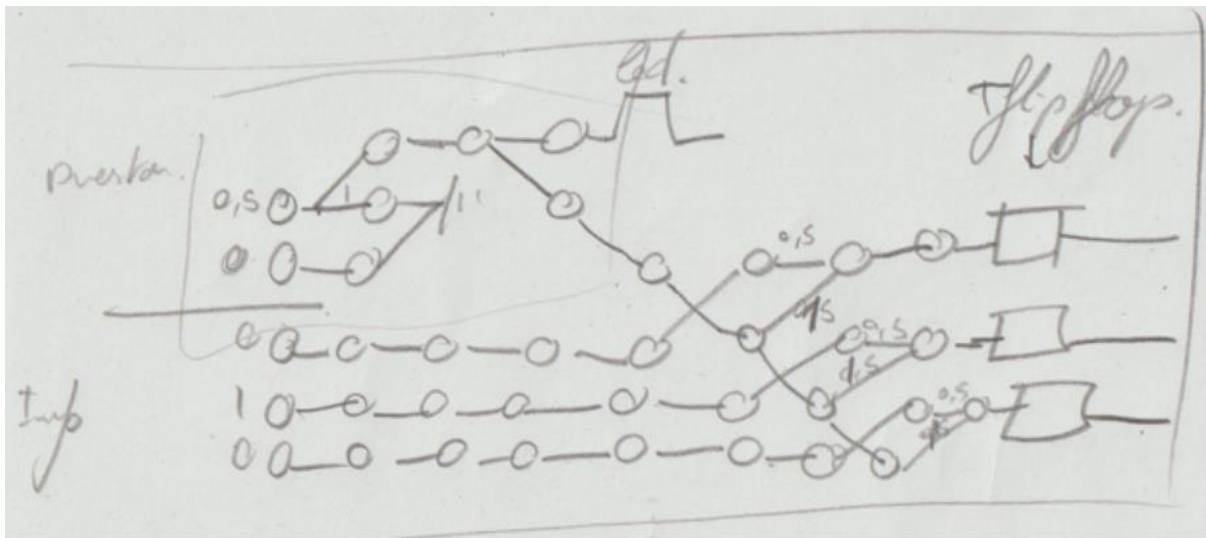
This gives us:

$2^3 \times 2^3 = 64$ possible combinations, or destinations.

In contrast, a conventional memory design would only offer:

$2^2 = 4$ destinations.

This makes the current design **significantly more versatile**.



For example, this particular gate only opens if the inputs are **0.5 and 0**.

Note: This model can **only store binary values (1 and 0)**—not 0.5.

9. CONCLUSIONS

This architecture presents **real computational potential** and could serve as the foundation for a **new generation of processors**.

We have demonstrated that it's not only viable in theory, but also **physically implementable**.

As for the prototype—it **works**,

but this document primarily aims to **define the architectural foundations** and make the system public.

The rest remains **a work in progress**.

Feel free to **contact me** if you'd like more information or to discuss the topic further—we need **collaboration to advance** in this field.

I also invite researchers to either **develop this system** or to **disprove it**.
This is the culmination of **months of work**,
and as its creator, I would like to name the system:

Blasco Neural Logic Array, or **BNLA**.

10. REFERENCIAS

1. *Texas Instruments*. (2000). "LM358 Operational Amplifier – Datasheet."
Retrieved from: <https://www.ti.com/lit/ds/symlink/lm358.pdf>
2. Blasco, [Tu nombre completo o pseudónimo]. "BNLA: Blasco Neural Logic Array – A Proposal for Alternative Computing Architecture." (2025).
Unpublished manuscript.
3. *Arduino Documentation*. (2024). "Using the PCA9685 PWM Driver."
Retrieved from: <https://docs.arduino.cc>

Licencia:

Este trabajo está licenciado bajo la licencia Creative Commons Atribución 4.0 Internacional (CC BY 4.0).

<https://creativecommons.org/licenses/by/4.0/>