Neuron & Neural Network:

Simulation Using Cadence Virtuoso & Python

Wali Afridi (wua3), Asma Ansari (ara89), Demetrios Gavalas (deg273), Simeon Turner (smt259)

Abstract

Artificial intelligence (AI) has become increasingly integrated into our daily lives through interactive tools, such as ChatGPT. With the introduction of more AI-integrated services everyday, demand for computing resources dedicated solely to driving neural networks, a key component powering AI, has skyrocketed. This project aims to evaluate one neuron's design in Cadence Virtuoso and implement a neural network in Python using this neuron's characteristics to understand its performance as it processes a simple task.

Introduction

Large language models (LLMs) are a resource-intensive task that cost data centers significant power and computing resources. Designing efficient neural networks requires efficient chip design, especially on-chip neurons. Our objective is to understand what design considerations and limitations go into implementing a neuron in hardware. This project involves implementing a MAC unit, ReLU, and state machine in Virtuoso to represent a simple neuron with up to three inputs.

Furthermore, we demonstrated driving two layers of two neurons into one layer of a single neuron in schematic to evaluate its correctness and performance. We supplement these results by developing a software-based representation of the neuron and a more complex neural network that allows back propagation to indicate how our neurons perform with a real task.

Top-Level Design

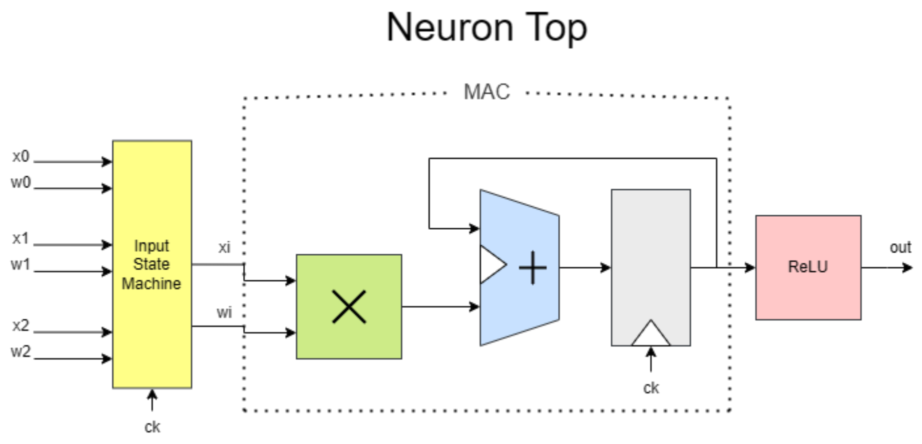


Figure 1. Top level neuron block diagram.

Figure 1 is the block diagram for our top level neuron circuit. This circuit includes the multiply accumulate unit with Wallace tree multiplier and ripple carry adder and register, as well as the input state machine and combinational ReLU. The input state machine selects which inputs to direct to the MAC for one iteration of multiplying and accumulating, switching inputs each cycle. The multiply-accumulate unit computes the product of two inputs and iteratively sums their products into a single register. The ReLU outputs an unsigned integer depending on the output of the MAC: if the output is negative, then the output is zero, otherwise we simply output the value of the MAC. Figure 2 illustrates our workflow for implementing the neuron circuitry in Cadence and supplementing our design with software simulations.

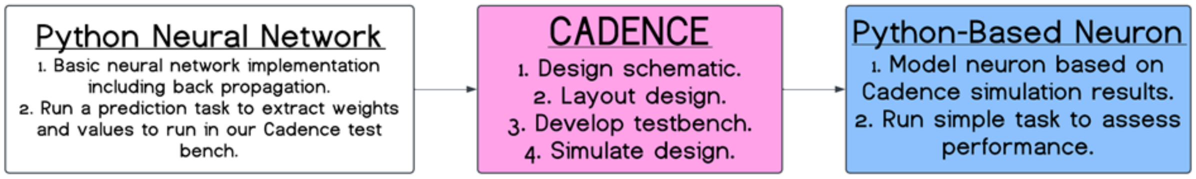

Python Neural Network
1. Basic neural network implementation including back propagation.
2. Run a prediction task to extract weights and values to run in our Cadence test bench.
CADENCE
1. Design schematic.
2. Layout design.
3. Develop testbench.
4. Simulate design.
Python-Based Neuron
1. Model neuron based on Cadence simulation results.
2. Run simple task to assess performance.

Figure 2. An outline describing how the hardware and software components interact.

Theory of Operation

Our neuron is composed of multiple sub-components: a Multiply-Accumulate (MAC) unit, a Rectified Linear Unit (ReLU), and an input state machine. The below equation describes the full functionality of our specific neuron.

$$out = ReLU \sum_{i=1}^{N} \left\lfloor \left(\frac{(x_i \cdot w_i)}{8} \right) \right\rfloor$$

Figure 3. The output of the neuron described by a mathematical expression.

The neuron takes in three 4-bit unsigned input values, x_0, x_1, and x_2, as well as three 4-bit two's complement signed weight values. Sampling of these inputs are controlled by the input state machine, which has three 1-bit control signals that are set one-hot style as depicted in Figure 2. Sampling each input and weight pair takes a full clock cycle, so we take 3 full clock cycles to sample all of our inputs. This means that our input data only needs to be valid for the clock cycle that it is sampled, i.e. x_0 and w_0 only need to be valid on the first clock cycle, x_1 and w_1 only need to be valid on the second clock cycle, etc.

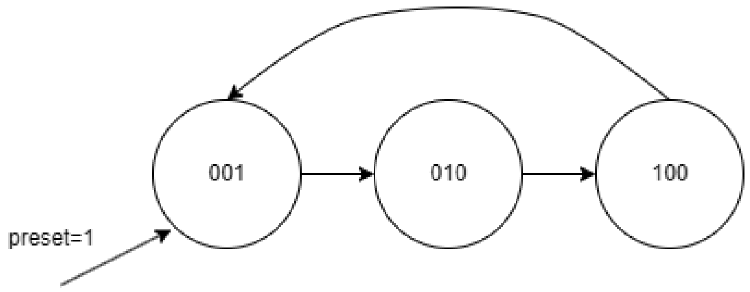


Figure 2. A FSM diagram of the neuron input state machine. The state $S_2S_1S_0$ encodes the control signals with $S_i = 1$ indicating we are sampling the i'th input and weight value pair.

Once we have our sample input and weight, we input our data into a Wallace tree multiplier modified to perform multiplication with one input being signed and the other unsigned. This multiplier takes in a 4-bit unsigned integer and a 4-bit signed two's complement integer and outputs an 8-bit signed two's complement integer. Our neuron truncates this number by throwing out the bottom-most 3 bits. This truncation is described by the division by 8 and flooring of each product in figure 1. The resulting 5-bit value is input to a ripple carry adder taking in the current accumulated value and adding the newly multiplied value, later stored in a 5-bit register. After each clock cycle that the register is updated, the combinational ReLU determines whether the 5-bit two's complement integer is negative or not, and outputs the value if it is non-negative, or outputs a zero if it is negative. The output would not include the sign bit, which allows us to chop the top-most bit off of our 5-bit number and output a 4-bit value.

Specifications

Hardware

The MAC unit and ReLU are crucial components for our neuron's design, but to enhance performance further, we chose to increase the number of inputs that each neuron receives. This design choice required us to include a state machine that cycles through the inputs on the rising edge of the clock. Therefore, after three clock cycles, the neuron has finally finished processing information. The specific functions for each sub-circuit component are as follows:

Component	**Function**
Multiplier	Combinational Wallace Tree multiplier that takes one 4b signed input, and one 4b unsigned input
Full Adder	Standard CMOS full adder implementation
Half Adder	Standard CMOS half adder implementation
Ripple Carry Adder	5 bit ripple carry adder which adds value in accumulation register and multiplication product.
Register	Stores accumulated value, asynchronous resets.
Ring Counter	One-hot ring counter state machine which selects inputs as valid in the order 1→2→3→1…. Uses one asynchronous set and two asynchronous reset flip flops to wake up in 1-0-0 state.
ReLU	Passes the 5 bit input and outputs an unsigned 4 bit number. If the input is positive the value is passed, otherwise the output is all zero.

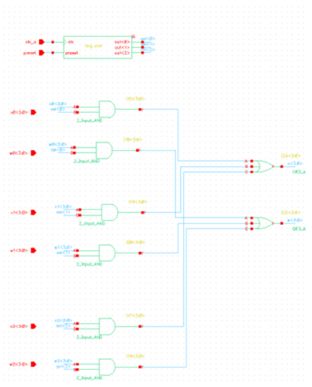
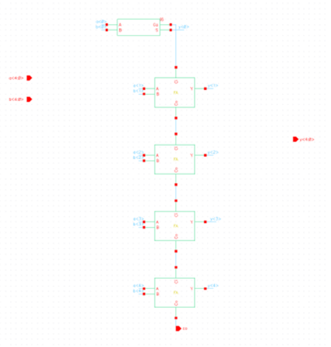
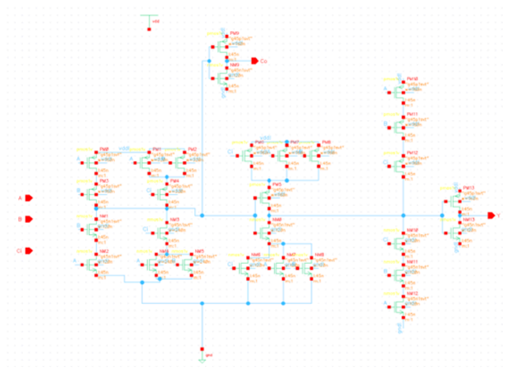
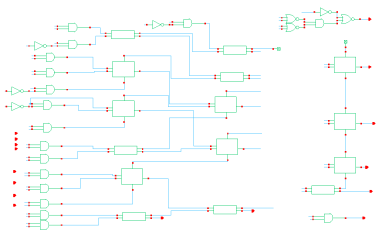
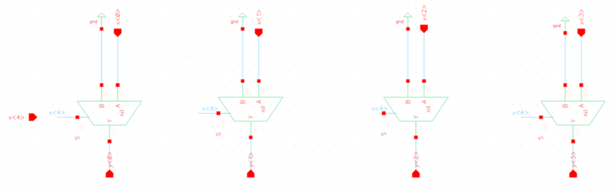
Table 1. Summary of each sub-circuit's use within the neuron's design.

Software

Our software analysis is based on a functional-level (FL) model for a neural network along with a Python class defining a neuron based on the Virtuoso simulation results for power and delay. The FL model is meant to be a form of black-box testing that provides us key information to guide our hardware design. Most importantly, we are not including back propagation in our neuron's design, so we are implementing that in our FL model to help us develop our testbench for the hardware neuron.

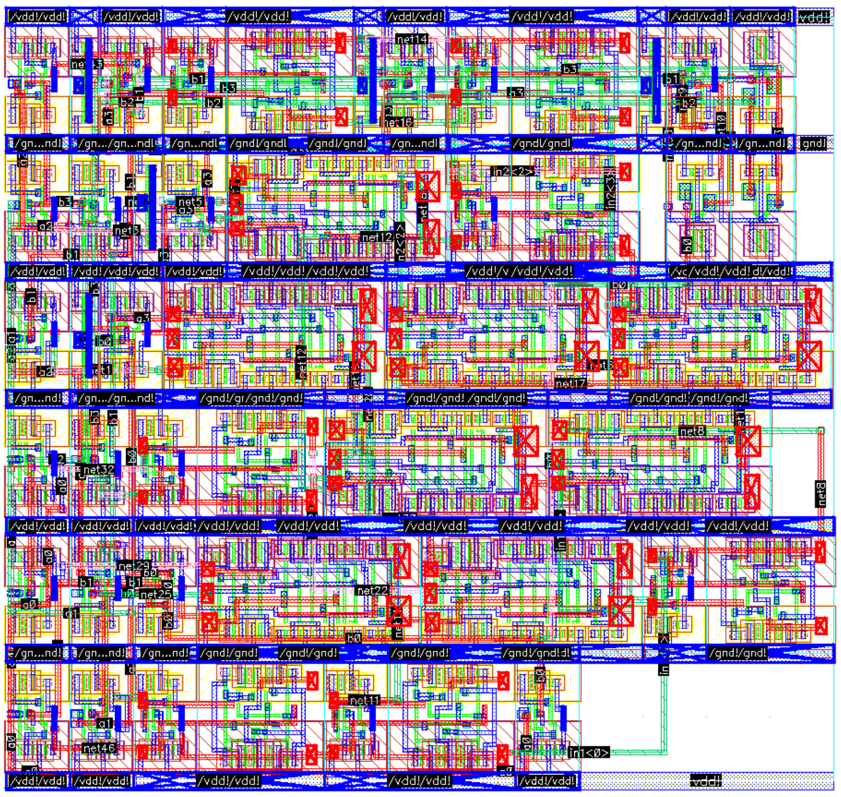
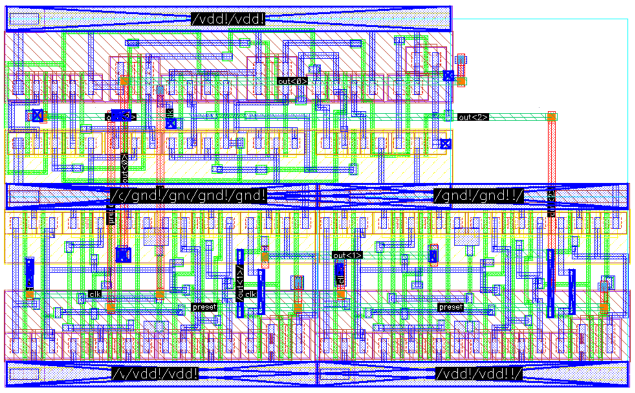
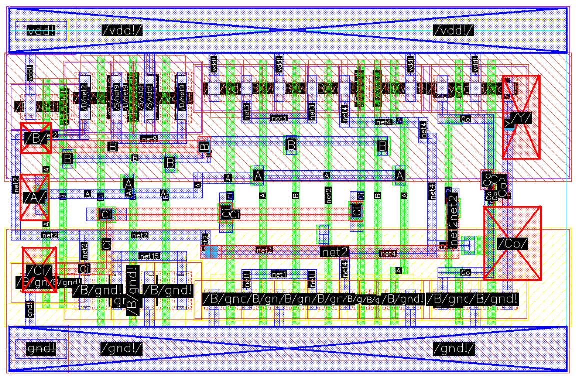
The Python class representing a neuron allows us to analyze performance as the neuron processes a task and back propagates to correct its error. Essentially, this component of the analysis is to extrapolate how well our neuron performs in more realistic conditions as opposed to simply driving layers of neurons.

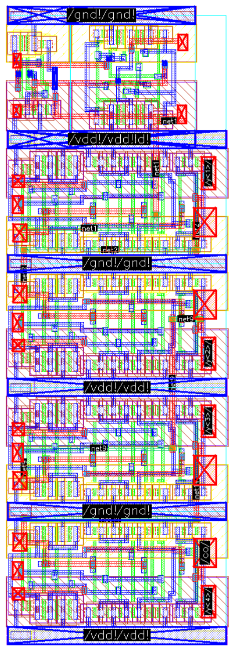
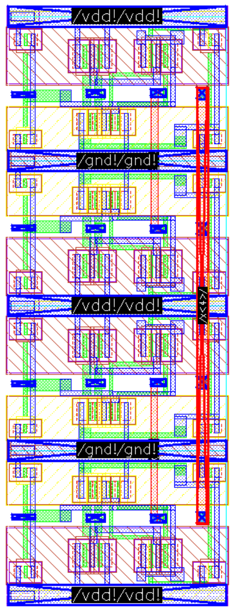
Sub-Circuit Schematics

 **Figure 3. Input State Machine Schematic.**	 **Figure 4. Ripple Carry Adder Schematic**
 **Figure 5. Full Adder Schematic.**	 **Figure 6. Wallace Tree Multiplier Schematic.**
 **Figure 7. ReLU Schematic.**	

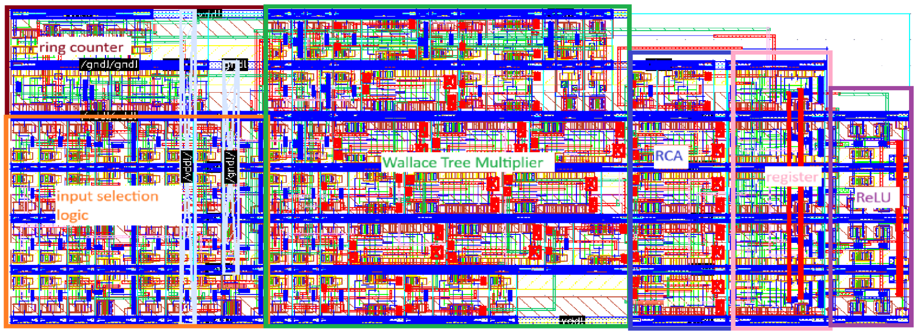
Layout

Component	Layout

Wallace Tree Multiplier 12 µm x 13 µm	
Input State Machine 7 µm x 4 µm	
Full Adder 2 µm x 3.5 µm	

Ripple Carry Adder 10μm x 3.5μm	 /gnd!/gnd! /vdd!/vdd! /gnd!/gnd! /vdd!/vdd! /gnd!/gnd! /vdd!/vdd!
ReLU	 /vdd!/vdd! /gnd!/gnd! /vdd!/vdd! /gnd!/gnd! /vdd!/vdd!

Neuron Top Level Layout

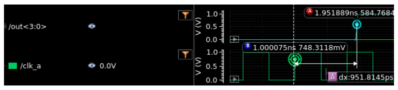
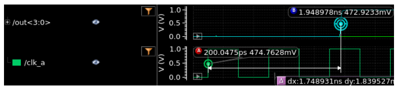


Results

Hardware

Key Specs:

Using schematic and extracted transient simulation. Inputs are ideal voltage sources with 10 ps rise and fall times. Each output drives x4 inverter in series with 10 fF capacitor.

parameter	schematic	extracted
Clk → Q Delay [ps]	951.814  /out<3:0> /clk_a 0.0V V (V) 1.951889ns 584.7684 1.000075ns 748.3118mV dx:951.8145ps	1748.931  /out<3:0> /clk_a V (V) 1.948978ns 472.9233mV 200.0475ps 474.7628mV dx:1.748931ns dy:1.839527
Energy [fJ / computation] (One computation is 3 cycles)	116	164
Area [μm^2]	314[1]	388.8

Functional Results:

Functional simulation was performed to evaluate the numerical output of the neuron for different combinations of input. Input source and output load are the same as above. The clock period used was 1 ns. Testing was first performed with schematic and then validated with extracted sim.

input 1 (x_0, w_0)	input 2 (x_1, w_1)	input 3 (x_2, w_2)	Output
3 * 3	0 * 0	0 * 0	9
3 * 3	3 * 3	3 * 3	24[2]
1 * -1	1 * -1	1 * -1	0
5 * 5	2 * -2	2 * -2	16
4 * 6	4 * 4	8 * -8	24
5 * 4	5 * -5	5 * -5	0
10 * 4	10 * 4	10 * 4	120

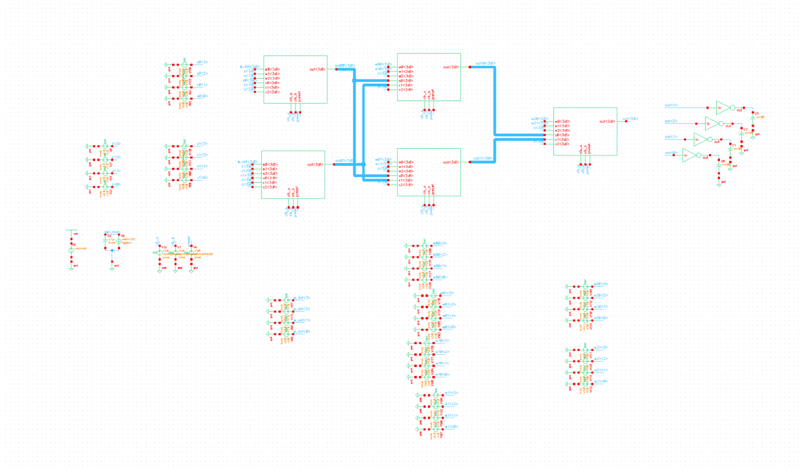
Neural Network

In order to demonstrate our neurons functioning as part of a network, we simulated 5 neurons in a 2-2-1 configuration. That is, we have 2 neurons in our first layer, 2 neurons in our second layer, and 1 neuron in the last layer of our network. Using the software discussed below, we conducted a bit level simulation of the neurons configured as a network. Using this, we were able to train

[1] Estimate based on standard cell count. Some circuitry was added after this estimate, so the difference in actual area represents both packing inefficiency as well as some extra standard cells.

[2] $9+9+9 \neq 24$! But each product is truncated before being accumulated, so the actual computation is $8 + 8 + 8 = 24$, so this behavior is expected.

the network (selecting weight values) to perform a simple classification problem, XOR of two bits. All the simulation conditions were the same as above, and all performed using *schematic* level simulation.



Testbench for neural network

inputs	output
00	0000
10	0001
01	0001
11	0000

With correctly trained weights, our neuron was able to imitate a basic logic gate with 100% accuracy! This gives us confidence that it could be used in a larger network.

Performance

parameter	network simulation (schematic)
Clk → Q Delay [ps]	2142.884
Energy [fJ / computation] (One computation is 3 cycles)	204
Area [μm^2]	1944

The delay between the rising clock edge to the output increases significantly, even in schematic simulation. This is due to the higher output capacitance of the other neurons compared to the testbed, and suggests that we would have to refine our sizing further to improve performance in a real use case. The energy per computation also significantly increases, which is to be expected since there are many more circuits being run. Since we did not layout the network, the area figure is just the area of the laid-out neuron times five.

Software

Our FL model for the neural network essentially XORs all the inputs and passes them through the activation function before moving on to the next layer. This provided us intuition into how our "hardware" neural network should be implemented in software. Thus, we created a Python class that extends our Neuron class such that it can calculate the total power and delay associated with the neural network attempting to predict the next output (based on the truth table for XOR).

Our neural network is trained over several thousand iterations to improve the error associated with the network's prediction. Since our task is so simple, the lowest error as the network makes its final prediction was zero percent using only five "physical" neurons. However, this could be the result of overfitting to the training model which is not a major concern given the task. Running the neural network code several times shows variance in the error ranging from 25% to 50% but never more. The simulation results are as follows:

Total Energy (fJ)	1093.40
Total Delay (ps)	34978.62
Total MAC Operations	40
Total ReLU Activations	9

Table 2. Software simulation results for a five-neuron neural network.

Discussion

The software portion of this project provided us intuition regarding how the neurons are activated as it processes a simple task. Changing the layer sizes was our main method of achieving consistently low errors with our neural network's predictions, but this came at the cost of energy and delay. The best combination seemed to be more layers with fewer neurons rather than fewer layers with more neurons. Neurons within layers activate given specific conditions, but its propagation through the layers of the neural network are crucial to the final prediction. More layers means there are more neurons in succession that are activated or deactivated whereas layers with more neurons but fewer layers can remain activated which does not necessarily improve errors in the final prediction.

In addition, we learned that the design for our neuron can be improved with respect to the handling of our truncation with our multiplier output. Initially, we assumed that neural networks with 3 input neurons would frequently have output values accumulating to very high values, but this was not the case when working with 1 and 0 inputs for an XOR network. Because we were not using all 3 inputs to each neuron and because of our truncation, our inputs had to be scaled by a factor of 8 to register an output of 1 from our neural network. This is even with larger-than-one weight values for all of our neurons in the network. In the future, it is generally better to include the lower bit values outputted from the multiplier in the neuron rather than truncate them off to prevent the reducing scale factor between layers in our neural network, i.e. the factor of 8 seen back in Figure 3.

This project gave us the opportunity to design a key component of a neural network down to a layout level, and then simulate our neuron behaving as a neural network. This simulation gave us more confidence in our design, but also exposed some issues with both our layout as well as our high level design choices. A real network would likely need more advanced control logic. We simply read out the output after the correct number of cycles, but a more robust and efficient process would use a “ready” signal. This would make the network more robust and also provide an opportunity to reduce dynamic power consumption via clock gating, since the neuron should not be doing anything until the previous layer’s output is valid. We also saw how our performance decreased significantly when using the network, showing that to make this work with faster clocks, we would need to further optimize for drive strength and minimize wiring parasitics in layout.

Our “physical” neural network provided us key insights into how software design interacts with hardware design. Oftentimes, smart programming choices can make up for poor hardware performance, but in the realm of AI, optimizing both areas is crucial. Data centers that process information for AI tools and services are demanding more and more power, so being conscientious of power in both hardware and software design is the most significant task as the world integrates AI further.

Citations

[1] Ahmed, Rekib Uddin, and Prabir Saha. “Implementation Topology of Full Adder Cells.” *Procedia Computer Science*, vol. 165, Jan. 2019, pp. 676–83. *ScienceDirect*, https://doi.org/10.1016/j.procs.2020.01.063.

[2] All About Electronics. “Binary Multiplication of Signed Numbers | 2s Complement Binary Multiplication” *Youtube*, https:/www.youtube.com/watch?v=NxBRKkNIpwY.

[3] “Hardware Neurons: A Building Block for Deep Learning Neural Network Accelerator.” *Socionext America*, https://socionextus.com/blogs/hardware-neurons-a-building-block-for-deep-learning-neural-network-accelerator/.

[4] HogoNext. “How to Build a Simple Neural Network From Scratch (Without Libraries).” *HogoNext*, 2024, https://hogonext.com/how-to-build-a-simple-neural-network-from-scratch-without-libraries/.