A MEMRISTOR-BASED NEUROMORPHIC COMPUTING APPLICATION

by

Adrian Rothenbuhler

A thesis

submitted in partial fulfillment

of the requirements for the degree of

Master of Science in Electrical Engineering

Boise State University

May 2013

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the thesis submitted by

Adrian Rothenbuhler

Thesis Title: A Memristor-based Neuromorphic Computing Application

Date of Final Oral Examination: 12 December 2012

The following individuals read and discussed the thesis submitted by student Adrian Rothenbuhler, and they evaluated his presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Elisa H. Barney Smith, Ph.D.	Chair, Supervisory Committee
Kristy Campbell, Ph.D.	Member, Supervisory Committee
Vishal Saxena, Ph.D.	Member, Supervisory Committee

The final reading approval of the thesis was granted by Elisa H. Barney Smith, Ph.D., Chair, Supervisory Committee. The thesis was approved for the Graduate College by John R. Pelton, Ph.D., Dean of the Graduate College.

ACKNOWLEDGMENTS

First and foremost, I would like to thank Dr. Barney Smith for her encouragement, guidance, and support. For the opportunities she has given me as a professor, advisor, and mentor, I am truly grateful. Furthermore, I would like to express my gratitude for Dr. Saxena and Dr. Campbell who patiently helped me understand many aspects of my research and enabled me to succeed and grow as a student and professional.

Special thanks go to Thanh Tran for the many productive discussions and her help that benefitted my work in many ways. Thanks are also due aplenty to Jeanette Brooks, who spent many hours soldering PCBs. Her experience and insight on PCB fabrication greatly influenced the overall quality of this work. I would also like to express my gratitude to Kolton Drake for many helpful discussions, Stephen Stauts for bonding and testing memristors, Arlen Planting and Vikram Patel for support with the NIOS II soft-core processor and Quartus software environment. Special thanks go also to Sakkarapani Balagopal and Rajaram Mohan Roy Koppula for helping me with ADC/DAC fundamentals and board design.

Last but not least, I am indebted to my wife Audrey for her constant encouragement, support, patience, and willingness to listen to my theories. Receiving such great support and understanding during times I was extremely busy is not a given and for that I am forever grateful.

ABSTRACT

Artificial neural networks have recently received renewed interest because of the discovery of the memristor. The memristor is the fourth basic circuit element, hypothesized to exist by Leon Chua in 1971 and physically realized in 2008. The two-terminal device acts like a resistor with memory and is therefore of great interest for use as a synapse in hardware ANNs. Recent advances in memristor technology allowed these devices to migrate from the experimental stage to the application stage.

This Master's thesis presents the development of a threshold logic gate (TLG), which is a special case of an ANN, implemented with discrete circuit elements using memristors as synapses. Further, a programming circuit is developed, allowing the memristors and therefore the network to be reconfigured and trained in real-time. The results show that memristors are indeed viable for use in ANNs, but are somewhat hard to control as a lot of intrinsic device characteristics are still under investigation and are currently not fully understood. A simple threshold logic gate was built and can be reconfigured to implement AND, OR, NAND, and NOR functionality. The findings presented here contribute towards improvements on the device as well as algorithmic level to implement a memristor-based ANN capable of on-line learning.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ADC – Analog to digital converter

AI – Artificial intelligence

ANN – Artificial neural network

CMOS – Complementary metal oxide semiconductor

CPU – Central processing unit

DAC – Digital to analog converter

DC – Direct current

DRAM – Dynamic random access memory

DSP – Digital signal processing

ESD – Electrostatic discharge

ESR – Effective series resistance

FPGA – Field programmable gate array

FSM – Finite state machine

GBW – Gain bandwidth

GBWP – Gain bandwidth product

HSMC – High speed mezzanine card

HW – Hardware

LED – Light emitting diode

LUT – Look-up-table

MAC – Multiply and accumulate

MRII – Madaline rule II

NDR – Negative differential resistance

PCB – Printed circuit board

PGA – Programmable gain amplifier

PSD – Power spectral density

SNR – Signal to noise ratio

SPA – Semiconductor parameter analyzer

STDP – Spike-timing dependent plasticity

TLG – Threshold logic gate

TLU – Threshold logic unit

UART – Universal asynchronous receiver/transmitter

VLSI – Very-large-scale-integration

CHAPTER 1

INTRODUCTION

This chapter introduces the memristor, upon which this work is based, and explains current technological challenges prompting the need for neuromorphic computing. The introduction culminates with the motivation of this work and concludes with an ethical analysis.

1.1 The Memristor

In 1971, Leon Chua realized that the three basic circuit elements - the resistor, the capacitor, and the inductor - share some similarities that are symmetric. In fact, the symmetry suggested that there should be a fourth basic circuit element. Chua reasoned that from a circuit's point of view there are four fundamental variables, namely the current i, the voltage v, the charge q, and the flux linkage φ. The resistor is defined by the relationship between v and i, the capacitor between q and v, and the inductor between φ and i. Only the relationship between φ and q remained unexplained. For the sake of completeness and symmetry, Chua argued that a fourth basic two-terminal circuit element defined by a φ-q curve should exist [8], providing the fourth and missing basic circuit equation linking φ and q, which is mathematically described as

$$d\varphi = M dq, \tag{1.1}$$

where M is the memristance, the proposed device's parameter.

This fourth basic circuit element is termed memristor, which is a contraction for memory resistor. The memristor is like a potentiometer, allowing us to continuously change its resistance. However, a memristor has only two terminals and the resistance is changed by applying different writing and erasing potentials across the device. Since it can remember its previous state, the device has memory. In 1976, Chua and Kang defined the memristor as a general nonlinear dynamic system, called a memristive system [10]. The difference between a memristive system and a regular nonlinear dynamic system is that the output of a memristor is zero, whenever the input is zero. This is evident by the state equations

$$\frac{dx}{dt} = f(x, u, t), \tag{1.2}$$

$$y = g(x, u, t)u, \tag{1.3}$$

where u and y are the input and output of the system and x is its state. While Equation 1.2 governs the state of the memristor's memory, we can clearly see that it has no effect on the output when the input is zero. Therefore, the i-v curve of a memristor looks like a Lissajous figure, which always crosses the origin [10]. Chua further described a memristor as a voltage-controlled, one-port device, which can be described similarly to Equations 1.2 and 1.3 as

$$\frac{dx}{dt} = f(x, v, t) \tag{1.4}$$

and

$$i = G(x, v, t)v, \tag{1.5}$$

where i is the device current, v the voltage across the device, G the conductance of the memristor, and x is its state. Chua and Kang furthermore stated that several physical devices such as the thermistor, the Hodgkin-Huxley circuit model, or the discharge tube exhibit an i-v curve, which can be described by the general memristor model [10]. However, the mathematical memristor concept as given by Equations 1.2 and 1.3 has not been connected to any practical physical systems for almost 40 years. Strukov et al. were the first to publicly announce that they had built a device that behaved like a perfect memristor [30].

1.2 Current Technological Challenges

Though we still seem to follow Moore's law and computers are still getting more and more powerful, processor speeds seem to have staggered and peaked-out at the lower GHz level. There are several different reasons that prohibit technology from advancing without bound to satisfy our ever-growing hunger for more computing power. On one side there are fabrication challenges, which require new semiconductor technologies to be developed in order to fabricate digital hardware on an even smaller scale. On the other side the laws of physics will not allow us to keep increasing processor clocking speeds. Researchers have begun looking into using light as a high-speed data carrier for on-chip communication. While this might help advance digital processing technology further, there is another shortcoming of modern computer architecture. Conventional digital computers rely upon a sequential processing scheme (fetch, decode, execution). The Von Neumann or Princeton computer architecture, proposed in 1945 by John Von Neumann, is still prevalently used, but because it uses a shared bus between data and program memory, the so called Von Neumann bottleneck arises

[35]. Since both data and instructions are stored in the same memory space, the CPU can only fetch either a new instruction or data at any one time, limiting the throughput. With increasing CPU speeds and bigger memory sizes, the Von Neumann bottleneck becomes more of a problem, because the throughput between CPU and memory doesn't increase as fast as the rest of the technology. Mechanisms like cache and branch prediction can help alleviate the issue to some extent, but the bottleneck still appears to be a severe problem for modern processor generations [35]. The Harvard computer architecture doesn't have this limitation, as program memory and data memory are separated; the processor uses a separate bus for each memory. It can therefore access both memories concurrently. Processors based on the Harvard architecture often use the so called *modified Harvard architecture*, which relaxes the strict Harvard architecture a little bit. The CPU operates on two separate caches (one for instructions and another for data), therefore making it look like a pure Harvard machine. However, both caches are fed by the same memory where instructions and data are located, therefore also implementing the Von Neumann architecture. Modern processors such as the ARM architecture and X86 processors use this modified Harvard architecture [34].

Faster and more powerful processors come at a high cost, because higher switching frequencies consume more power. However, processor speeds have recently staggered at the lower GHz range because of some circuit-related issues. As Casper points out, legacy backplanes, which are used to connect several different peripheral devices like memory and other I/O devices, have a severe dip in their frequency response at around 5 GHz [7]. This means that faster CPUs would not necessarily increase overall computing speeds, because the underlying hardware is not capable of supporting such high clocking rates. While semiconductor processes need to be refined to actually make CPUs capable of much higher clocking frequencies, researchers are looking at using different backplane interconnects supporting higher multi-Gb/s I/O. One solution is to use light and fiber-optic technology as it not only supports much higher speeds, but drastically reduces insertion losses compared to conventional copper-based backplane interconnections [7].

Looking at modern processors from another perspective, we can see that they are all based on finite state automatons, which transition between a finite number of predefined states. These states are defined as digital logic in silicon and can't be reconfigured once produced. Field-programmable-gate-arrays (FPGA) on the other hand are reconfigurable and have the capability of performing the same operations at much higher throughput than a conventional CPU. This is because the user-defined functions are all fully implemented in hardware and run in parallel. Besides massive parallelism offered by FPGAs, their clocking rates are much lower, often in the 100 MHz range, which enables them to drastically reduce power consumption. Since FPGAs implement all of the user-defined functionality directly in hardware and not in software, they are way more efficient than conventional processors. While the majority of low-level software functionality can be implemented in digital hardware, it would require a very large amount of FPGA resources for higher-level functionality and is therefore not a viable alternative. Exceptions are applications that only need to perform a few select tasks.

1.3 Bioinspired Circuits

Nature has adapted to many changes and found solutions for many problems. Instead of *reinventing the wheel*, why not look to nature for solutions to issues or needs we have in our everyday life? This questions prompts many researchers to get inspired by nature, causing a *Bioinspired Engineering* area to emerge. Many products we're using on a daily basis are based on processes found in nature. With the computing challenges outlined in Section 1.2, researchers are looking to nature to improve computing power and lower energy consumption. For example, to perform certain cortical simulations of a cat brain, a team from IBM uses a supercomputer with 147,456 CPUs and 144TB of main memory. The simulation runs about 83 times slower than in nature [17]. While computers are fast enough to emulate brains of spiders, mice, and even cats, the energy requirement and dissipation grows exponentially with the intelligence of the animal [17]. The example above clearly depicts the vast differences between our current computing technology and nature's computing capability. The main difference is found in the architecture. While our CPUs employ mainly the Von Neumann architecture, nature achieves high efficiency with high connectivity between neurons, offering highly parallel processing power [17]. It is not hard to see that bioinspiration and neuromorphic research are leading the way towards revolutionizing current computation technology. Researchers even expect that this kind of research will give artificial intelligence a boost.

1.4 Motivation and Objective

So far we have seen current technological computing challenges and that researchers are turning to nature to find answers and new ideas for improvement. While neural networks and other artificial intelligence functionality can be relatively easily implemented in software, hardware implementation of such systems is relatively scarce. However, with the discovery of the memristor there is new hope on the horizon. It is no coincidence that the memristor is being used in neuromorphic computing. Back in 1976, Chua and Kang discovered that the Hodgkin-Huxley model that describes how action potentials in mammalian neurons are initiated and propagated, follows the general mathematical memristor model [10]. The memristor is therefore perfect for use as a synapse in a hardware-based artificial neural network (ANN).

The work presented here is motivated by the current technological challenges outlined above and by the recent developments in memristor technology. While many research groups have already published their simulation-based work on advanced memristor networks capable of learning and functioning like a brain, there has been very little work involving an actual hardware-based ANN. The vast majority of actual hardware-based results originates from memristor characterization and is not tied to an actual application. The work presented here is focussed on building a memristor-based neuromorphic computing application in order to show the viability of memristors as synapses in ANNs. It needs to be noted that the implemented network is a threshold logic gate (TLG), which is a special case of an ANN. Furthermore, the implemented network is not driven by a learning algorithm, but is rather controlled manually to adjust the memristors. Chapter 5 outlines current shortcomings of memristors and learning algorithms and presents possible solutions that can be used to add learning to the network at a later point. The following list shows the objectives of this work:

1. Develop memristor programming and reading circuit.
2. Develop synapse and neuron circuit.
3. Build a fully hardware implemented TLG.
4. Evaluate and characterize memristors for use in a synapse.

5. Find the effect of the training environment on training algorithms.

As can be seen, the majority of the objectives are hardware oriented and involve a fair amount of circuit development. Chapter 2 presents some general background information pertinent to this work. Chapter 3 describes a method to program and read the state of a memristor, which is an integral part of the final fully hardware implemented TLG. Furthermore, the neural network, consisting of synapse, neuron, and other supporting functionality, needs to be built in hardware such that it supports the use of memristors. Chapter 4 elaborates on that. Last, but not least, we need to characterize and evaluate memristor behavior in an ANN type of application, such that necessary adjustments can be made for the final application. Also, finding the effects of the training environment on training algorithms is necessary in order to come up with a new training algorithm that is hardware friendly. Chapter 5 presents the characterization and TLG performance results, while Chapter 6 concludes this work with a summary and recommendations for future work.

1.5 Ethical Issues

It is the author's opinion that the ethical implications of any type of work should be carefully considered. It is good practice to act according to the *NSPE Code of Ethics for Engineers* whose first fundamental canon suggests "to hold paramount the safety, health, and welfare of the public" [27]. Especially when dealing with new technologies that overlap with and influence artificial intelligence, it is important to seek ethical advice that goes beyond the fundamental ethical canons for engineers.

When reading literature about ethical implications and issues of artificial intelligence, the term *super-intelligence* often appears. While super-intelligence does not yet exist and experts cannot foresee when and if super-intelligence will be invented, several authors have argued that super-intelligence might emerge because of advances in hardware performance and the ability to implement algorithms and more importantly architectures similar to those used by mammalian and human brains [4]. Whether it is our intention to work towards super-intelligence or not, neuromorphic research is working towards its development. Super-intelligence would indeed allow us to build more powerful computers as stated above, but it would also enable advanced weaponry [4]. While the former can easily be seen as an advantage, the latter could be a threat to humanity if this technology falls into the wrong hands. Especially when bearing in mind what Nick Bostrom stated about the creation of super-intelligence: *Super-intelligence may be the last invention humans ever need to make.* This is because the intellectual superiority of super-intelligence would be much better and faster at doing scientific research than any human being, possibly even better than all humans together [4].

Another aspect to keep in mind is that super-intelligence might not have human-like psyches. Even if it is built based on human-like emotions, it has the capability of evolving and changing its own source code. On the other side, defining a top-level goal is of utmost importance, but if this goal later turns out to be a false utopia, things essential to human flourishing get irreversibly lost [4]. Bostrom states: "We need to be careful about what we wish for from a super-intelligence, because we might get it." It is hard to decide whether the development of super-intelligence should be delayed or advanced, but experts agree that humankind will sooner or later be faced with some kind of super-intelligence [4]. Therefore, we all need to remind ourselves of the moral implications super-intelligence could bring.

CHAPTER 2

BACKGROUND

This chapter gives the background information needed to understand and reproduce the work presented here. It begins with a basic introduction to artificial neural networks and mentions two common learning algorithms used for training. Furthermore, the functionality and intrinsic operation of memristors are presented. Although not necessarily part of the work presented here, the theory and physical relationships to memristors are important parts that need to be understood in order to be able to control the device accurately. The chapter will end with a brief review of opamps, which are integral to the circuit development presented later.

2.1 Artificial Neural Networks

The neuromorphic application built for this Master's thesis is based on the basic structure of artificial neural networks (ANN). The following sections will make clear that ANNs coupled with a learning algorithm can be used for learning applications. It needs to be pointed out that the work presented here does not include a learning algorithm and the threshold logic gate (TLG) built for proof-of-concept purposes does not exhibit learning, because current learning algorithms are not hardware friendly enough to be used with memristors (see Chapter 5 for more details). However, the following sections on ANNs are important for this work as they constitute the foundation for the TLG and heavily influenced the circuit design presented in Chapters 3 and 4.

2.1.1 A Brief History Lesson

In 1943, McCulloch and Pitts published a paper called “A logical calculus of the ideas immanent in nervous activity” [23]. This paper is regarded as the beginning of artificial intelligence, as it elaborated on the mammalian neuron as well as the combination of multiple neurons through synapses [24]. In 1949, Donald Hebb published his theory in neuroscience, explaining how neurons in the brain adapt during the learning process [15]. The theory, commonly known as Hebbian theory, describes a synaptic plasticity mechanism that causes a pre-synapse to increase its efficacy on a post-synapse by repeatedly stimulating it. These findings spurred research in neuroscience and most importantly artificial intelligence. In 1957, Frank Rosenblatt invented the perceptron, a classification algorithm based on a linear predictor function [24]. More specifically, the perceptron is an automaton demonstrating that adaptive neural networks with a rich interconnectivity and synapse-like nonlinearities could mimic cognitive functions [12, 24].

Rosenblatt demonstrated that his generalized perceptron using a variable and fixed layer of weights could solve any dichotomy. This generalized perceptron is also called a single-layer perceptron or neural network [24]. In 1967, Minsky and Papert concluded that a single-layer perceptron is not able to implement nonlinear decision boundaries [24]. Unfortunately, many misunderstood this shortcoming as a general problem of neural networks, bringing the first era of artificial intelligence to an end. Neural network research was at a dead end until multi-layer networks (i.e., multi-layer perceptrons) were more thoroughly investigated and training methods were found, enabling the network to solve more complex problems.

2.1.2 Basic Structure

As already mentioned, ANNs are motivated by the biological structure of mammalian brains. The structure and functionality is best explained with a graphical depiction as seen in Figure 2.1. The depiction starts with the input zone of the cell (neuron). Input signals from other neurons or stimuli are received through the dendrites and passed on to the cell body. The summation zone is where the signals are summed up and possibly trigger a signal that is carried through the conduction zone to the output. The synaptic knobs in the output zone are connected to additional dendrites. Current theories of memory in mammalian brains state that the synapses play a key role in how memories are stored in the brain. These theories are backed up by the Hebbian theory and are evidenced by the chemical reactions in the pre-synaptic cells, which affect the amount of neurotransmitters emitted and received by the post-synaptic cell [15, 29]. The neurotransmitters are responsible for transporting information across the synaptic cleft [29].

Figure 2.1 depicts the synapses and neurons as the active parts and the dendrites and axons as passive conduction elements. We can now make the jump to an artificial neural network depicted in Figure 2.2, which shows a simple single-layer ANN. The synapses can be thought of as weights, either amplifying or attenuating the signal, while the neuron can be broken down into signal summation and activation function. As with the biological neuron, the activation function is what triggers an output signal depending on the strength of the sum. Each layer can have multiple neurons and each input is connected to each neuron through a synapse. The general mathematical

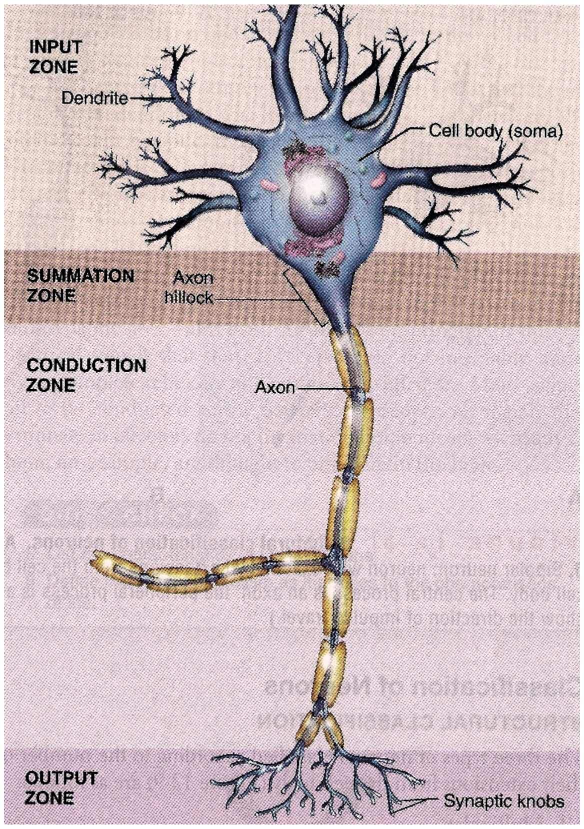


Figure 2.1: Biological structure of a neural network [29]

model of a single-layer can be described as

$$y = f(\sum g_i(x_i, w_i)) = f(\sum x_i w_i), \tag{2.1}$$

where x is the input, w is the weight, and *g(x,w)* is the synapse function adding weight to the signal. The function *f(·)* is called the activation function and will be introduced later. A multi-layer network is essentially just the combination of multiple single-layers by treating the outputs of a layer as the inputs to the next layer. Although Rosenblatt differentiated between series-coupled, cross-coupled, and back-coupled network topologies, series-coupled networks are most common [12, 24].

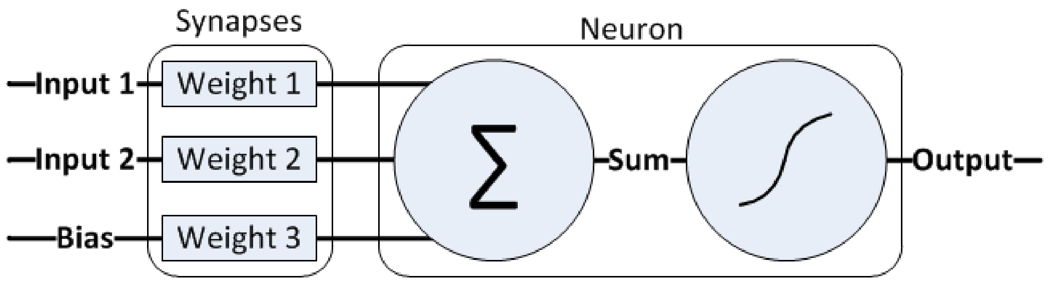
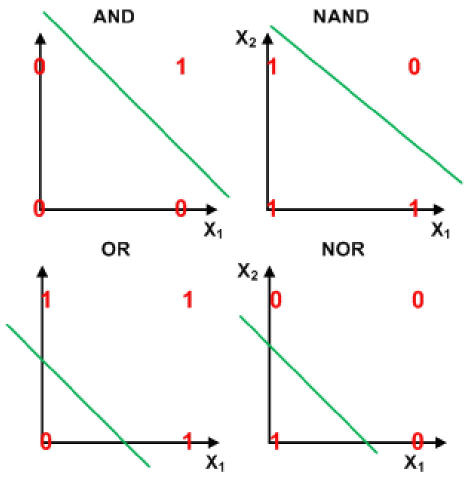


Figure 2.2: Neuron with two input synapses and one bias synapse

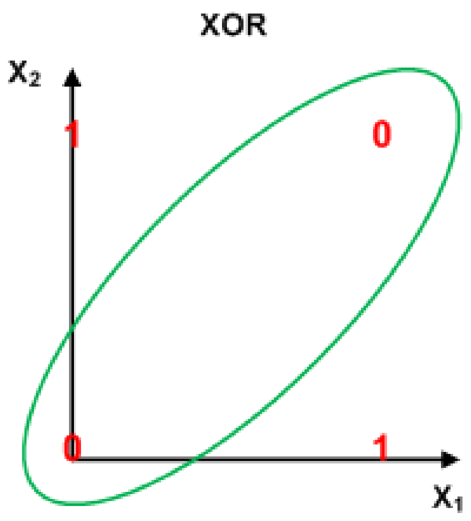
Often, a network is divided into input, hidden, and output layers, where the input layer is the first, the output layer the last, and the hidden layer everything else in-between (i.e., a layer or layers not directly connected to inputs or outputs). Also, the input layer is most commonly only a fixed weight layer used for performing signal conditioning [12]. The neural network seen in Figure 2.2 is often called a feedforward network as the act of classification consists of feeding information forward from one layer to the next. The flow of information is unidirectional.

2.1.3 Linear and Non-Linear Separability

Before we introduce activation functions, it is important to understand the two types of patterns an ANN can encounter. In general, a pattern can be classified as either a linearly or nonlinearly separable pattern. Figure 2.3 shows the difference between the two types on some simple, well-known binary patterns. It is obvious that linearly separable patterns can be solved by a simple linear decision boundary implemented with a rather simple algorithm. More complex, non-linearly separable patterns require a more complex pattern recognition algorithm. Figure 2.3a shows the AND, OR, NAND, and NOR patterns, which are all linearly separable. Figure 2.3b, however, shows the XOR pattern, which is not linearly separable, as evidenced by the *x*-*y* plot. Rosenblatt's as well as Minsky and Papert's findings can be combined to conclude that



(a)



(b)

Figure 2.3: Linear and Non-Linear Separability

a single-layer perceptron can solve any linearly-separable dichotomy, but a multi-layer network is required to solve all nonlinear dichotomies. In short, a "simple" nonlinear decision boundary can be implemented with a single-layer perceptron, but a more "complex" one, like the one seen in Figure 2.3b, can't and requires at least one additional network layer. It should be noted that the XOR pattern seen in Figure 2.3b is often used as a benchmarking problem because of its rather complex, fully enclosed decision boundary. Literature commonly compares performance of ANNs through the XOR problem.

2.1.4 Activation Function

As previously mentioned, the activation function is the part of the neuron responsible for triggering an output signal depending on the strength of the sum of its inputs. In other words, the activation function can be thought of as a decision maker. For example, if the sum is above a certain threshold, the neuron's output either indicates, depending on the activation function, whether an event has occurred (hard threshold) or how likely it is that an event has occurred (soft threshold). Rosenblatt described the activation function as either linear, hard-threshold, or an S-curve [24]. We shall call the linear and S-curve-like activation functions a soft-threshold, as they do not make a definite decisions (1 or 0, yes or no) for every input value.

A linear activation function is essentially just a scaling factor or slope m. Equation 2.1 can then be expressed as

$$y = m \sum x_i w_i. \tag{2.2}$$

While a linear activation function can solve linearly separable problems like the AND, OR, NAND, or NOR problem, it is often not used because of its limitations. Linear activation functions are often used for input layers using fixed weights to perform signal scaling or conditioning [12].

Modern ANNs most often use nonlinear S-curve-like activation functions. Examples are the sigmoid, hyperbolic tangent, and polynomial functions. The sigmoid is probably the most commonly used activation function and can mathematically be expressed as

$$f(x) = \frac{1}{1 + e^{-\lambda x}}, \tag{2.3}$$

where λ determines the steepness of the curve. Figure 2.4 shows the sigmoid activation function based on different values of λ. As λ increases, the curve becomes steeper and essentially turns into a hard threshold as λ approaches ∞. The parameter λ needs to be carefully chosen, as high values are better for making a [0,1] decision, but training algorithms (like backpropagation) work better with lower values due to the smoother derivative of the sigmoid.

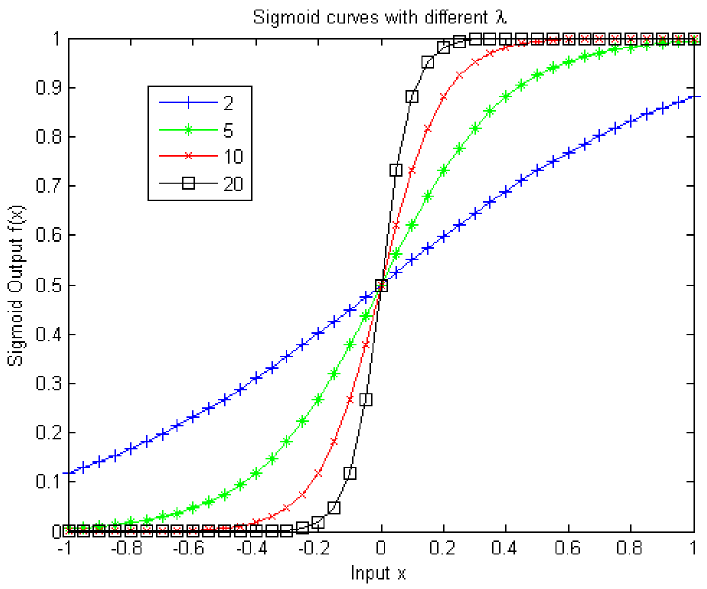


Figure 2.4: Sigmoid activation function

As can be seen from Figure 2.4, the sigmoid is a saturating function, meaning that its output values are always within the interval [0,1]. Figure 2.4 suggests that the input range should be in the range of [-1,1]. While this is generally a good idea, it is especially important for sigmoids. Let's elaborate on this with an example: Assuming that we have two inputs, the first being the weight of cars in pounds and the second being their fuel consumption in miles per gallon. Both of these inputs are most definitely exceeding the ideal input range of [-1,1]. In fact, both numbers are always greater than zero, but most importantly, the weight is usually a much larger number than fuel consumption. There are two problems with this: First, the weight, which for the average car is in the thousands, will easily saturate the sigmoid. Second, the difference in orders of magnitude between the two inputs require the synapse values for the "weight" measurement input to be much smaller than for the "miles per gallon" measurement input. It is very hard for the training algorithm to accommodate such inputs. Therefore, it is recommended that the training dataset is normalized and input scaling is used [12]. While normalization does only get us to a [0,1] input range, this is often all we can do for real-world input values. However, computer simulations in conjunction with the work presented here have shown that an ANN trains much better if binary problems are presented to the network in the form of [-1,1] instead of [0,1].

Last, but not least, a nonlinear hard-threshold can be used as an activation function, which is essentially just a step function but is more commonly referred to as the *signum* or *sign* function. It can mathematically be expressed as

$$f(x) = sign(x) = \begin{cases} -1 & \text{for} \quad x < 0 \\ 0 & \text{for} \quad x = 0 \\ 1 & \text{for} \quad x > 0 \end{cases} . \qquad (2.4)$$

Even though the signum function is piecewise linear, it is considered a nonlinear function, but the point here is that it is not continuous. Discontinuity of an activation function poses a serious problem with the commonly used backpropagation learning algorithm, because the derivative is undefined at *x=0*. However, there is an alternative to the backpropagation algorithm as we will see in Section 2.1.6.

It needs to be noted that a neuron using a hard-thresholding activation function is also called a threshold-logic-unit (TLU). A network using TLUs is therefore called a threshold-logic-gate (TLG), which is a special case of an ANN.

2.1.5 Bias Input

Figure 2.2 depicts a simple ANN with three inputs, one of them being a bias input. There is no general rule that a bias input is required. However, training is facilitated if one is used. This is best explained with a 3-D weight space depiction of the network in Figure 2.2. Assuming that we would like to recognize the patterns for AND, OR, NAND, and NOR, we would need four different sets of distinct weights. These four sets of weights are mutually exclusive, meaning that each pattern can have more than one set of working weights, but all sets of weights for one particular pattern are not a solution to any of the other patterns. Figure 2.5 shows this relationship. Note that the weights were limited over the interval of [-1,1] for simplicity's sake. The two input weights are w_1 and w_2, respectively, and w_0 is the bias weight. If w_0, w_1, and w_2 are chosen to be *-1*, then the NOR pattern can be recognized. Note that changing the bias weight w_0 from -1 to 1 changes the network to recognize the NAND pattern without having to change the actual input weights. Furthermore, setting $w_3 = 0$ (no bias input) causes the network to be uncertain (i.e., from the graph in Figure 2.5 it is not clear which pattern is chosen). For this particular network topology and patterns a bias input is required for proper classification. In essence, the bias input shifts the decision threshold up or down.

2.1.6 Training Algorithms

As mentioned before, the backpropagation algorithm is among the most popular training algorithms used to train complex ANNs. It is based on a gradient descent algorithm, trying to minimize the output error by tracing the error backwards through the network. This backwards propagation is used along with the derivative of each activation function to calculate the weight change for each synapse [12]. The back-propagation algorithm works with virtually any activation function as long as it is continuous and its derivative is defined over the entire interval [12]. For hardware implementations the backpropagation algorithm is not ideal for the following reasons:

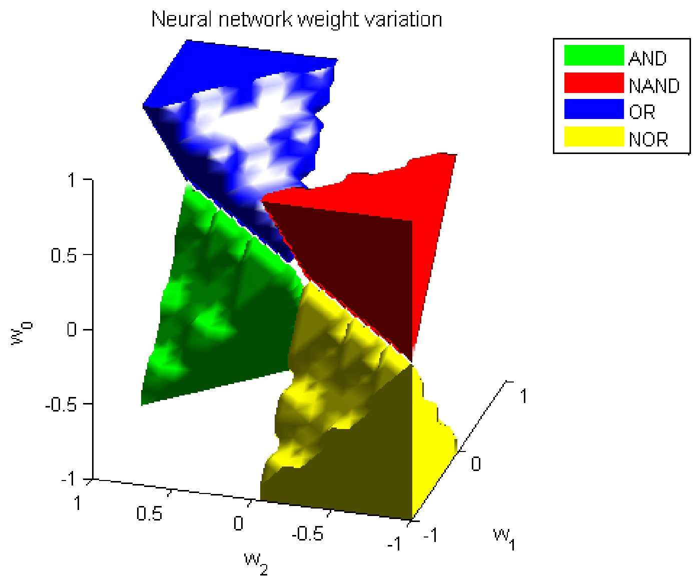


Figure 2.5: ANN weight cube [32]

- A continuous activation function such as the sigmoid is hard to implement in discrete circuits;

- An inverter used as a hard-threshold is the optimal discrete circuit activation function;

- Simulations done by Thanh Tran showed that both single and multi-layer ANNs recognizing AND, OR, NAND, NOR, and XOR patterns train very well using Widrow's MRII training algorithm [32].

Since the objective of this work is to build the hardware TLG that Tran has simulated, we will forego backpropagation and treat MRII as a potential training algorithm candidate. Although a training algorithm will not be implemented here, it will influence the circuit design of the network.

MRII Algorithm

Back in 1960, Widrow used Rosenblatt's perceptron to build what he called an Adaline (ADALINE = ADAptive LINEar) neuron. An Adaline is very similar to the general perceptron, but uses a quantizer for the activation function, making it a TLU, and requires binary inputs of [-1,1]. The neuron forms a linear combination of inputs and weight values. If the linear combination is above a certain threshold, the output is +1, otherwise it is -1. The threshold can be adjusted using the weight w_0, which is the bias weight [33]. In essence, the Adaline is just a special case of the perceptron and is considered hardware friendly because of its simple hard-thresholding activation function. This is perhaps the reason why Adalines are still the subject of research, even though they are over 50 years old [20].

Almost 30 years later in 1988, Widrow and Winter came up with MRII, a simple training algorithm, which can be used to train a network of multiple Adalines [36]. MRII stands for "Madaline Rule II," where "Madaline" means multiple Adalines. The algorithm is based on the minimal disturbance principle and starts working from the output layer backwards to the input layer. Unlike the backpropagation algorithm, MRII always performs feedforward on all training samples during a single training cycle, calculating how many of them are misclassified by the network. If the error is zero, then the algorithm stops without performing weight adaptation, as the network is already trained. If the error is greater than zero, then the algorithm disturbs the weights of the output layer node (neuron) that has the least confidence. Since Adalines are optimized for values in the range of [-1,1], the neuron whose sum is around 0 shows weak confidence, whereas a sum close to or above -1 or 1 shows high confidence. Measuring confidence of a neuron means tapping into its signal flow right

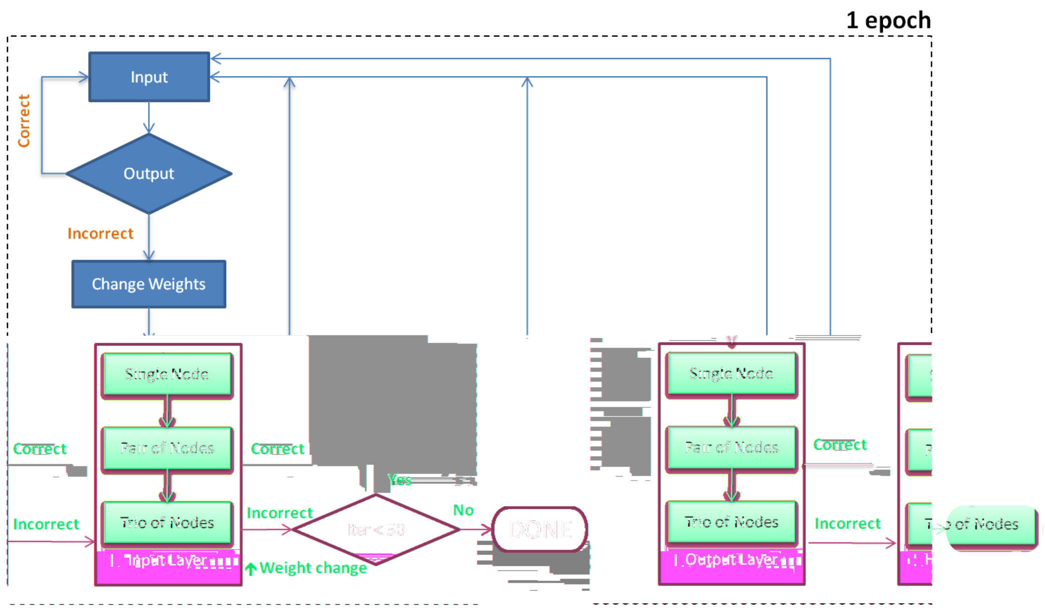


Figure 2.6: MRII training algorithm flow chart [32]

before the sum is passed to the quantizer (thresholding unit) [36].

The weight adaptation is performed by adding a random value to each involved weight. Mathematically this can be expressed as

$$w_i[n] = w_i[n-1] + \Delta w_i, \tag{2.5}$$

where Δw is the change in weight and can itself be expressed as

$$\Delta w_i = \eta * rand(), \tag{2.6}$$

where η is the learning rate, governing the distribution of the noise or in other words, how much the weights are disturbed.

Figure 2.6 shows a simplified flow-chart of the MRII training algorithm. Once the algorithm detects a mismatch between training set and classification output, it starts to adapt the weights. It does so by starting with the output layer node that has the least confidence. If the weight disturbance was not effective, the node will be set back to its old weights. The algorithm then performs weight disturbance on the two lowest confidence nodes, repeating the same procedure. After every disturbance, the network is tested by feeding the entire training set forward. MRII continues to include more nodes and eventually more layers until the total error is reduced. As Figure 2.6 shows, the algorithm moves back towards the input layer until the overall error has been reduced. If that is the case, the algorithm accepts the new weights and starts over. If the error was not reduced, or even increased, the distribution η of the random number generator is increased and another attempt is made. Because weight disturbance relies on adding random values, it might take multiple attempts to get a correct output. Each time the algorithm encounters an error reduction, η is reset [32].

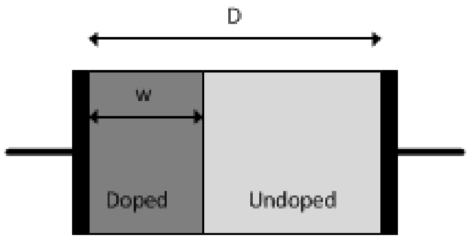
Increasing the distribution of the random number generator each time MRII encounters a failed training loop is a modification Tran made to the base algorithm. She furthermore implemented a maximum loop rule, which says that if the algorithm was not able to reduce the error to zero after N_{iter} iterations (Tran used $N_{iter} = 50$ for XOR and $N_{iter} = 30$ for AND, OR, NAND, NOR), then it is probably hopeless to keep training. The network is reset and reinitialized with fresh, randomized weights. Training for N_{iter} iterations is considered an epoch and it might take multiple epochs to successfully train an Adaline-based ANN. Tran's results proved that these modifications yielded very good results with significantly fewer training iterations compared to backpropagation [32].

2.2 Memristor

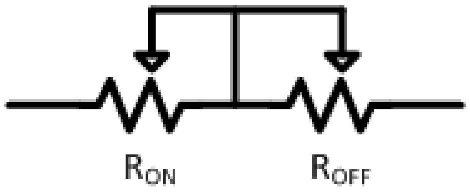
Chapter 1 generally introduced the memristor as a voltage controlled, one-port device relating the flux linkage φ to the charge q. While the general memristor model was introduced in terms of a nonlinear system, we are still missing the connection to an actual physical device.

2.2.1 Physical Model

In essence, the memristor is a device much like a resistor, but can change its intrinsic resistance based on the applied voltage and most importantly remember its resistance when power is removed, making it perfect for non-volatile analog memory applications. While modulated device resistance due to ion migration had been observed for decades, memristor research received much attention since Strukov's paper publicly linked this behavior to Chua's memristor model in 2008 for the first time [26]. Strukov furthermore claimed that a memristor can be thought of as a thin semiconductor film with thickness D sandwiched between two contacts. The semiconductor film has a doped area with width w and an undoped area with width D-w. The total resistance of the device is then simply the series resistance of the doped and undoped region. By changing the width w, we can then adjust the resistance of both the doped and undoped region, therefore changing the overall resistance of the device [30]. Figure 2.7a shows the physical memristor model with the doped and undoped area, while Figure 2.7b shows the circuit model with the two series resistances. Changing the width w affects both R_{ON} and R_{OFF} and therefore the overall device resistance. Mathematically, this can be expressed as



(a)



(b)

Figure 2.7: Physical and circuit model of a memristor

$$V(t) = \left(R_{ON} \frac{w(t)}{D} + R_{OFF} \left(1 - \frac{w(t)}{D} \right) \right) I(t), \tag{2.7}$$

where *V(t)* is an external bias voltage causing the charged dopants to drift, therefore adjusting w. Equation 2.7 is true for the simplest case of ohmic electronic conduction and linear ionic drift in a uniform field with average ion mobility μ_v [30]. The parameter w can then be expressed as

$$w(t) = \mu_v \frac{R_{ON}}{D} q(t). \tag{2.8}$$

Inserting Equation 2.8 into Equation 2.7 yields the memristance of the system, which for $R_{ON} << R_{OFF}$ simplifies to

$$M(q) = R_{OFF} \left(1 - \frac{\mu_v R_{ON}}{D^2} q(t) \right). \tag{2.9}$$

2.2.2 Device Topology

As stated above, a memristor's resistance modulation can be achieved by ion migration in the semiconductor layer of the device. Memristors are therefore sometimes also called ion-conducting devices, as the magnetic flux moves ions into or out of the semiconductor film's insulation layer, affecting the device's conductance

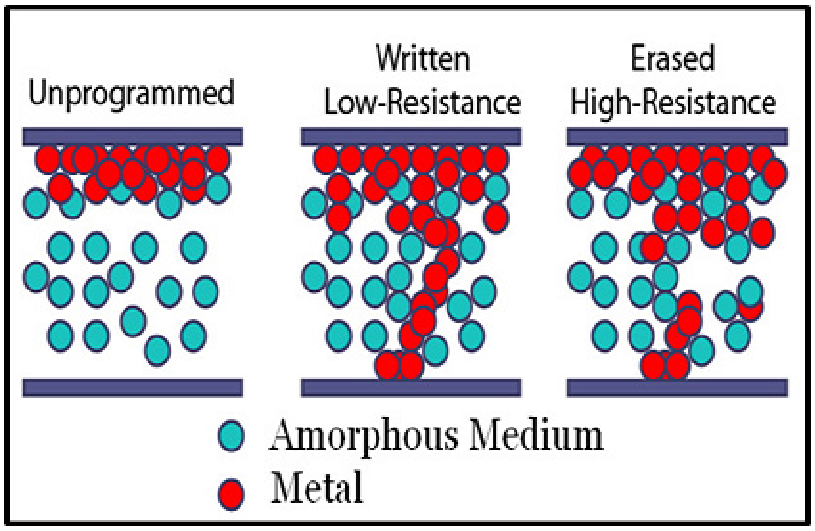


Figure 2.8: Ion conduction [5]

[17, 18, 26, 30]. Literature reveals several different semiconductor materials being used as active regions for memristors, ranging from TiO_2 [30] over Ag-Si [18] to Ag-chalcogenide [26]. The work presented here is based on Ag-chalcogenide (silver) and Cu-chalcogenide (copper) devices manufactured by Dr. Campbell's research group at Boise State University.

In general, the semiconductor film of a memristor consists of a top electrode, a bottom electrode, and an insulating layer. After fabrication, the device is in a high-resistance state ($G\Omega$). Regardless of the switching material used in the active layer, a so-called electroforming process is required to inject metal ions into the insulating layer, causing semi-permanent structural modifications [18]. After this initial procedure, the filament-like structure enables metal ions to easily migrate in and out of the active region. By applying a positive voltage, ions move into the active layer, forming a low resistance path, whereas a negative voltage causes ions to migrate out of the active layer and go back to the top electrode [26]. Figure 2.8 shows a simplified depiction of the initial (unprogrammed), low-resistance, and high-resistance states.

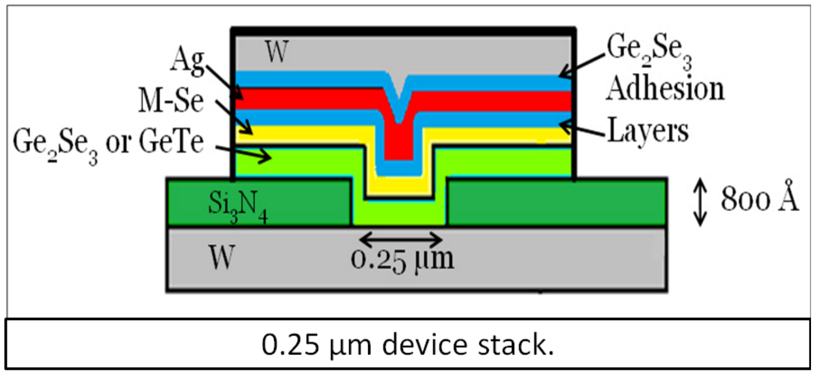


Figure 2.9: Device cross-section [5]

Figure 2.9 shows a cross-section of a typical Ag-chalcogenide memristor fabricated at Boise State University. The Ge_2Se_3 adhesion layers are only used for device processing, as Ag cannot be directly deposited onto the M-Se layer. Furthermore, W does not adhere well to Ag [6, 26]. The yellow middle layer consisting of M-Se is essentially the amorphous insulation layer, where M is a placeholder for various different metals. Using different materials for the M-Se layer, different device characteristics can be achieved [6].

Figure 2.10 shows an optical top down view of a fabricated memristor device on the substrate. The large square on the left is the bond pad for the top electrode and the white metal line on the right is the bottom electrode. The device is located at the intersection of the blue and white metal lines.

2.2.3 Threshold

Literature reports that a certain threshold voltage is required for ion migration to start. It is noteworthy that the threshold applies to both writing and erasing (i.e. positive and negative) voltages, but the threshold levels are not equal [17, 18, 26, 30].

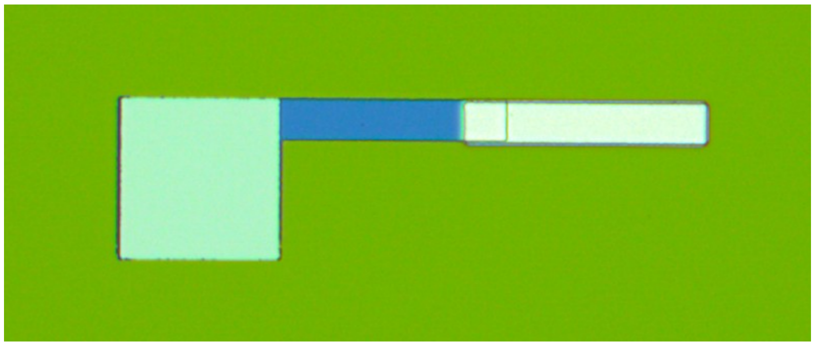


Figure 2.10: Top down view of a memristor device [5]

While Chua's memristor model does not account for a threshold, one might argue that contemporary memristors are not actual memristors, but rather memristive-like devices. On the other hand, having a threshold is very practical to read a device's state. Without a threshold, every read cycle would inevitably change the device's state, which is not desirable. Fortunately, Chua recently published a tutorial on the memristor and explained that its fingerprint is a continuum of pinched hysteresis loops at all frequencies and for all initial conditions [9]. The Ag-chalcogenide memristors used for this work exhibit this phenomenon and by that definition can be called memristors. Figure 3.3 in Chapter 3 shows that the hysteresis loop is indeed pinched.

2.2.4 Controlling a Memristor

Since a memristor is only a two terminal device as opposed to the three terminal potentiometer, one might wonder how the device's resistance can accurately be controlled. The previous section already alluded to the fact that positive and negative potentials cause the device to write and erase. At this point, there is no accurate model that foresees how much the memristor's state will move if a voltage pulse of length x and voltage y is applied. As stated in Section 2.2.3, the writing and erasing voltages have to exceed a certain threshold for the memristor to change its state.

There are both a writing as well as an erasing threshold and they are not necessarily at the same absolute voltage level. Furthermore, thresholds can change as the device's resistance is altered, making it even more difficult to make an accurate prediction of where the state is going to end up [5]. It is therefore important to read the device's state after each writing/erasing pulse so that programming error can be detected and compensated for by applying subsequent pulses. A typical writing/erasing algorithm will therefore send multiple pulses interspersed with reading pulses to the memristor. It is of importance to make sure that the following general rules are followed:

- The reading voltage should be as low as possible to prevent unintentional ion movement. For the Ag-chalcogenide devices used for this work, the reading voltage should not exceed 40mV [5].
- If a large reduction in resistance has been observed after applying a writing pulse, the device should be fully erased before applying another writing pulse. A large ΔR indicates hard device switching, meaning that most of the silver from the top electrode is embedded in the amorphous insulation layer. Applying another writing pulse might irreversibly damage the device [5].
- Erasing pulses of very short duration and high enough amplitude can cause the device to go into negative differential resistance (NDR) mode or even damage the device. Repeated mild erase pulses won't cause long-term damage [5].

2.2.5 Memristor vs. Memistor

As a final note, it should be mentioned that the memristor is not to be confused with the memistor, a three-terminal device invented by Bernard Widrow in 1960. Like the memristor, the name memistor is a contraction for "Memory Resistor" and is based on the phenomenon of electroplating. Resistance can be controlled by chemically depositing metal onto a resistive substrate [33]. Although Widrow's memistor seems to be very similar to Chua's memristor, Widrow invented his device for the sole purpose of creating an adaptive linear neuron, whereas Chua used his device to explain the relationship between charge q and flux linkage φ and therefore defined the missing fourth basic circuit element.

2.3 Opamps

Operational amplifiers, opamps for short, play an integral part of the work shown here. Therefore, it is appropriate to state the general equations as a reference for the reader.

2.3.1 Basic Opamp Equations

An opamp can be used in two different basic configurations: the inverting and non-inverting mode. Figure 2.11 shows the two basic opamp configurations.

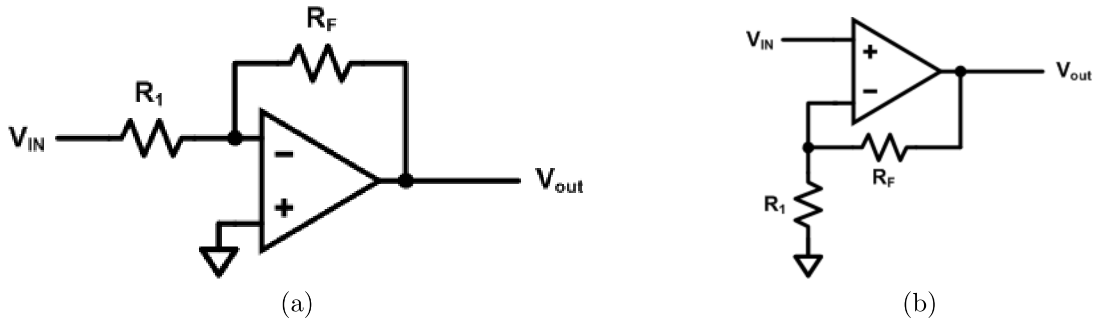


Figure 2.11: Opamp in (a) inverting and (b) non-inverting mode

The inverting input in 2.11a is effectively tied to ground through the virtual ground assumption. Since no current is flowing into the input terminals of an ideal opamp, the current flowing through R_1 (from V_{IN} to ground), is also flowing through R_f. The output of an opamp in an inverting configuration can therefore simply be expressed as

$$V_{out} = V_{ref} - (V_{in} - V_{ref})\frac{R_f}{R_1}, \tag{2.10}$$

where V_{in} is the input voltage and V_{ref} is the reference voltage at the non-inverting node. If V_{ref} is set to zero (i.e., non-inverting node connected to ground), Equation 2.10 reduces to

$$V_{out} = -V_{in}\frac{R_f}{R_1}. \tag{2.11}$$

For an opamp in a non-inverting configuration, the circuit behaves similarly. However, the input signal is what gives us the fixed reference. Through the virtual ground assumption, the same potential appears on the inverting input node, causing a current to flow through R_1. We know that there is no current flowing into or out of either input nodes, therefore the current has to be supplied through R_f from the output node. The output voltage of such a configuration is expressed as

$$V_{out} = V_{ref} + (V_{in} - V_{ref})\frac{R_1 + R_f}{R_1}, \tag{2.12}$$

where V_{ref} is the reference voltage at the bottom of R_1. Again, let's set V_{ref} to ground (zero) as depicted in Figure 2.11b to arrive at the simplified non-inverting opamp output equation

$$V_{out} = V_{in}\frac{R_1 + R_f}{R_1}. \tag{2.13}$$

2.3.2 Frequency Response and Gain Bandwidth Product

Reality tells us that virtually anything has a frequency response. While we can control the gain of an opamp through R_1 and R_f, opamps are limited in frequency, meaning that as the frequency of the applied signal is increased, the device might not be able to add the requested gain. In general, opamps have a low-pass like frequency response and circuit designers need to keep the frequency response in mind. However, the tricky part is that the frequency response changes with the gain. Let's say, for example, an opamp has a *unity gain bandwidth* (GBW) of 1 MHz. This implies that at 1 MHz the opamp can produce a gain of 1. Even though decreasing the frequency will bring us back into the nominal frequency bandwidth, an opamp cannot attain any gain at any frequency. The so called *gain bandwidth product* (GBWP), given by

$$GBWP = A_N * f, \tag{2.14}$$

where f is the desired frequency and A_N is the noise gain, specifies the bandwidth of an opamp at a given gain and frequency. The noise gain is the same for both inverting and non-inverting opamp configurations and is defined as

$$A_N = A_{noninverting} = \frac{1}{\beta_{noninverting}} = \frac{R_1 + R_f}{R_1}. \tag{2.15}$$

Keep in mind that GBW is constant for voltage-feedback opamps. Therefore, one has to pick values for A_N and f that result in a GBWP that is less than or equal to GBW. In short, when designing an opamp-based circuit, designers have to choose an opamp such that the product of the desired frequency and gain does not exceed GBW.

CHAPTER 3

MEMRISTOR PROGRAMMING AND READING CIRCUITRY

Before a memristor-based TLG can be built to perform classification, we first need to be able to program memristors and read their current state. This can in theory be done relatively easy, as we will see in Section 3.1, but it also poses some challenges when precision and accuracy is required. At this point, it should be noted that if the programing and reading hardware is not properly working or the memristor's functionality is compromised, the TLG cannot be trained and is therefore not able to classify patterns.

Thanh Tran simulated a memristor-based ANN including programming/reading circuitry [32]. While her work pioneered neuromorphic computing in terms of using memristors as synapses in ANNs, it did not account for several hardware related challenges due to the nature of a simulation. For example, there are no dynamic range or noise issues in a simulated circuit. Granted that this could be added on, the real problem is that there is currently no accurate memristor simulation model available. This chapter provides a thorough discussion of these challenges in terms of memristor programming and reading circuitry and presents solutions as well as results.

3.1 Initial Attempts

When dealing with novel technology, like the memristor, it is usually a good idea to start with a simple experiment and then improve on that. While a simple but effective programming and reading circuit had already been developed by Drake et al. [11], it was important for this work to start from scratch and first get some hands-on experience with memristors. Using an Agilent 33250A pulse generator, an Agilent MSO7104A oscilloscope, and only a single resistor, a primitive but effective experimental setup was built to demonstrate the memristive behavior of the memristive devices used for this work. This experimental circuit is shown in Figure 3.1 and, although crude, allows us to observe the memristor's behavior while it is being programmed. Recall that a positive potential "writes" the memristor to a lower resistance and a negative potential "erases" it to a higher resistance state. Through the voltage divider configuration, we can therefore expect V_{out} to follow V_{in} when $R_M >> R_1$ and V_{out} to be less than V_{in} when $R_M << R_1$.

Figure 3.2 shows the result of applying a 100Hz sine wave with amplitude of 3V and offset of 1V to the circuit in Figure 3.1. Resistor R_1 was picked to be 5kΩ. Note that the function generator was connected to V_{in}, channel 1 of the oscilloscope to V_{in}, and channel 2 to V_{out}. It can clearly be seen that the output voltage V_{out} (also called

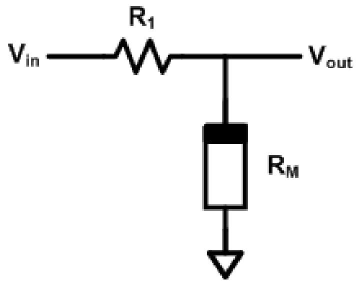


Figure 3.1: Simple memristor experimental circuit

V_{mem} since it depicts the voltage across the memristor R_M) changes depending on the applied potential of the sine wave. The result is as expected: A positive potential writes the memristor into a lower resistance state causing the applied voltage to decrease. Note the red line, roughly indicating the memristor's resistance, going from a higher resistance state to a lower state and back to a higher state when a negative potential is applied (erasing). Applying a sine wave will essentially cause the device to perform hard-switching between its high and low resistance states. Figure 3.2 clearly shows that the memristor doesn't follow the applied sinewave (Vin) linearly, but switches between high and low resitance during the write and erase cycles.

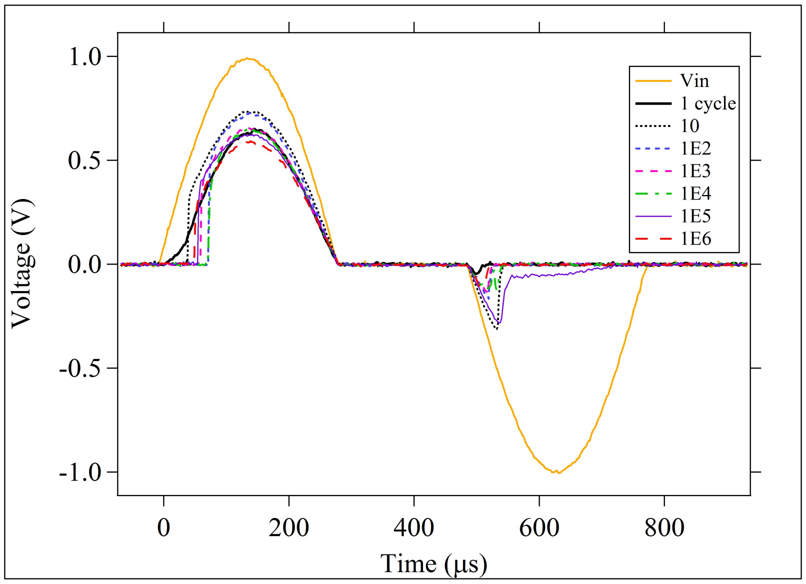


Figure 3.2: Memristive behavior when a sine wave is applied [5]

Figure 3.3 shows the pinched hysteresis loop or Lissajous pattern produced by the memristor using V_{in} for the x-axis and V_{mem} for the y-axis. Note that the hysteresis loop consists of a linear part (around the origin) and a nonlinear part (away from the origin), indicating that the memristor behaves like a linear resistor below the writing/erasing thresholds. The locations where the curve goes from the linear region

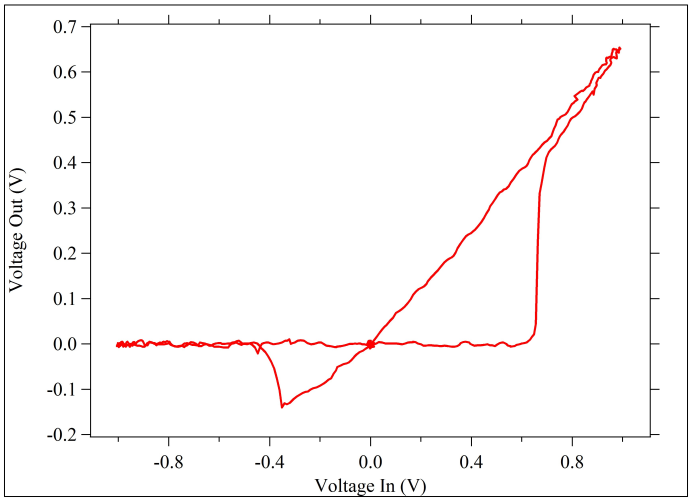


Figure 3.3: Memristor exhibiting Lissajous pattern [5]

into the nonlinear region indicates where the writing/erasing thresholds are located. These results nicely demonstrate the memristor's capability to go back and forth between states. The device could therefore potentially be used to store digital values (high, low). However, it is of no interest to use an analog memory for storing digital data. Instead, the device should be used to store analog information. In order to do so, we have to modify our circuit so that the memristor can be accurately controlled. From Figure 3.2, we can deduce two problems that need to be solved in order to move on. First, accuracy needs to be improved. The memristor's resistance curve shows some fluctuations in the high resistance region, which is due to measurement error. Since the oscilloscope's range was set so the sine wave's 3V$_{pp}$ swing can be observed, it becomes very hard to accurately measure the small changes in voltage on the scope's channel 2 when $R_M << R_1$. Second, a continuous sine wave is not ideal for accurately controlling a memristor (i.e., no hard-switching). Instead, it is better to apply positive and negative pulses in order to better control the movement of the device's state in either direction.

A second experimental setup was created based on Kolton Drake's programming circuit, consisting of an LF411 opamp [11]. Figure 3.4 shows this circuit, which is different from the circuit in Figure 3.1; it is not a voltage divider, but rather a voltage follower with gain $(R_M + R_f)/R_M$. As long as the opamp's bandwidth and range limitations are not exceeded, the pulse applied to V_{in} is also applied to the memristor because of the opamp's virtual ground. Since we're not continuously programming the memristor, but rather applying single pulses, we have to read the memristor's state after each pulse to check how far the state has moved. As explained in Section 2.2.4, the reading voltage across the device used for this work should not exceed 20-40mV. Reading such a low voltage might be hard to measure, so the feedback resistor R_f is set to add sufficient gain for measuring the memristor's state at an adequate voltage level. The advantage of this circuit is that the gain can be adjusted by the feedback resistor R_f such that sending a programming pulse as well as a reading pulse will give us an adequate voltage range at the V_{out} node.

For the initial pulse experiment, the feedback resistor R_f was chosen to be around 11kΩ. Figure 3.5 shows the results of applying several different positive and negative pulses. Once again, positive pulses reduce the memristor's resistance and negative pulses increase it. The graph clearly shows that by applying single pulses, the memristor's state can be changed in a more controlled fashion than with a sine wave. While

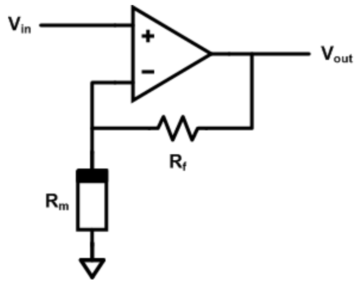


Figure 3.4: Simple opamp based memristor programming circuit

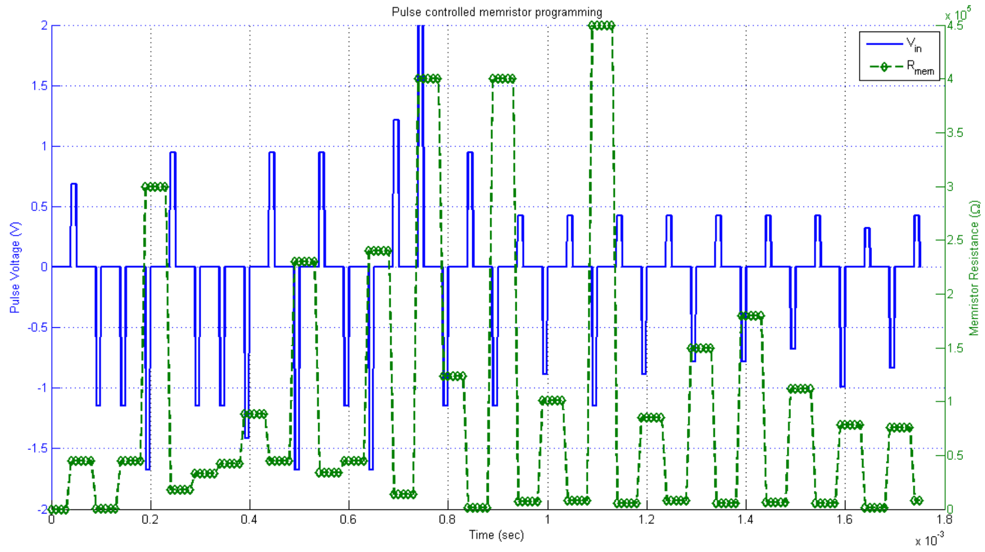


Figure 3.5: Result of memristor pulse programming

this experiment proved to be much more successful and revealing than the previous one, there are still some circuit limitations that need to be overcome in order to get useful and accurate characterization data. The next section will touch upon these limitations and provide solutions that will lead to the final reading/programming circuit.

3.2 Challenges and Solutions

After experimenting with the initial memristor circuits as explained in Section 3.1, it was discovered that there are some challenges that need to be overcome in order to properly program and read memristive devices. So far we have seen several different issues that might affect the accuracy of the programming/reading output voltage, which indicates the memristor's state. The first is measurement accuracy due to the nonlinearity of the output voltage. The second is limitations of the opamp-based circuit. Dealing with a 20-40mV reading voltage poses some additional challenges like suppression of thermal and transient noise. This section will elaborate on these issues and show how they can be overcome.

3.2.1 Dynamic Range, Gain, and Bandwidth

The opamp-based pulse programming circuit in Figure 3.4 is definitely a better approach to memristor programming and reading than the voltage divider approach. However, Section 3.1 already alluded to the fact that opamp limitations might cause the circuit to fail. Before we dive into searching for limiting parameters, let's first look at the nonlinear output problem, which severely affects the accuracy of the reading circuit. Figure 3.6 shows the nonlinear output of the reading circuit seen in Figure 3.4. The curve is based on a 50mV reading voltage and R_f of 100kΩ. Note that the curve is already depicted in a logarithmic format and the output voltage change decreases exponentially as R_{mem} increases. If we were to look at this curve in a non-logarithmic

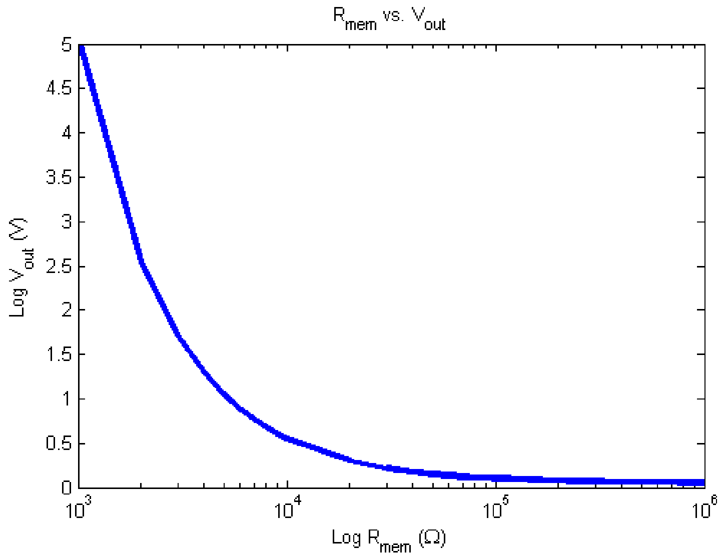
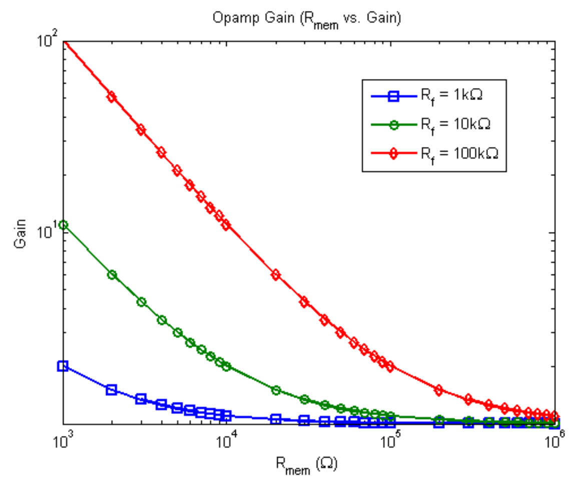


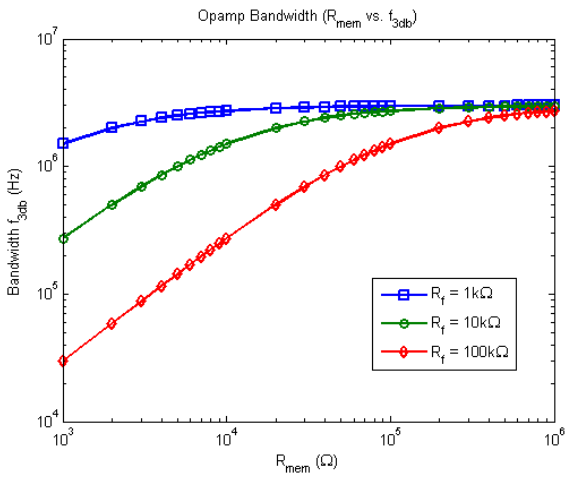
Figure 3.6: Nonlinear output of the reading circuit

format, the drop-off in the low R_{mem} region would be almost vertical, and almost horizontal in the high R_{mem} region. There are two things that we can conclude from Figure 3.6: First, the dynamic range of this circuit is not ideal and needs to be improved. Second, the output swing is about $5V_{pp}$ at a reading voltage of 50mV and R_f of 100kΩ, covering a memristor resistance range of 1kΩ - 1MΩ. Once we start applying programming pulses with amplitudes of up to 3V, the opamp's output would most definitely rail, as the worst case in this configuration would require the opamp to produce an output voltage of about 300V. Clearly, this is not possible and therefore requires a smaller feedback resistor. However, decreasing R_f would also compress the output swing when reading and this is clearly not desirable either. Figure 3.7a shows the gain of the opamp over a wide range of memristor resistances with different feedback resistors. Figure 3.7b shows that the gain bandwidth decreases as the gain in Figure 3.7a increases. Since the programming circuit should be able to send pulses of at least a 1μs pulse width, we have to keep GBWP above 1MHz. Two of the three feedback resistors used in Figure 3.7b violate this constraint. Although the low amplitude reading pulse could essentially be a very slow DC pulse, we don't want to make that trade-off just yet.

One thing we have to keep in mind is that so far we have assumed that we can use the memristor over a range spanning about 6 orders of magnitude ($\Omega - M\Omega$) or maybe even more. Because of that, it is no wonder that dynamic range issues arise. As a first step in solving these issues, we will limit the operational range of the memristor to 2 orders of magnitude and operate between $1k\Omega - 100k\Omega$. As a second step, we realize that even for a decreased operational range of the memristor, one feedback resistor is not sufficient to accurately cover the entire output range. Instead, we will turn the circuit in Figure 3.5 into a programmable gain amplifier (PGA) using multiple



(a)



(b)

Figure 3.7: Opamp gain and bandwidth with different feedback resistors

feedback resistors that can be added to modify the gain such that the output voltage is at a reasonable level for reading.

Figure 3.8 shows the modified circuit with three additional feedback resistors that can be switched in or out to change the opamp's gain. The feedback resistor R_{f1}, which is not connected to a switch, has to have the largest value of all four. Each additional feedback resistor, having a resistance lower than R_{f1}, can be added in parallel to decrease the effective resistance in the feedback path. Table 3.1 shows feedback resistor values of the PGA and Figure 3.9 shows the resulting output. It can clearly be seen that the gain never exceeds 10 and that the output voltage swings nicely between 0.1V and 0.4V for each range. Even though these output voltages seem low, high-performance ADCs usually have an input range of 0-1V, which makes it perfect for this application.

Table 3.1: Programmable gain feedback resistor values

Resistor	**Value**	**Range**
R_{f1}	300kΩ	41kΩ - 200kΩ
R_{f2}	56Ω	7kΩ - 40kΩ
R_{f3}	9kΩ	1kΩ - 6kΩ
R_{f4}	1kΩ	Programming

In Figure 3.9, we can see that the memristor resistance range can be adequately covered up to 200kΩ. Therefore, we can expand our range upwards, which is a benefit for the synapse's weight adjustment, as will be discussed in Chapter 4. Another important fact that needs to be pointed out is that Table 3.1 indicates resistor R_{f4} as a programming only feedback resistor. Since it would not provide much amplification within the 1kΩ - 200kΩ range, it is of no use for reading and is therefore not used to produce the gain curves seen in Figure 3.9. The range could be extended downwards

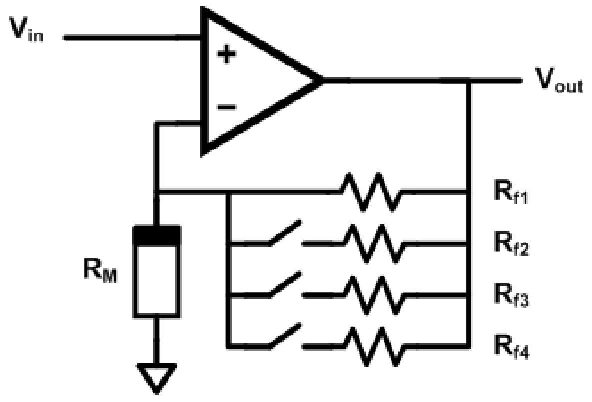


Figure 3.8: Programming/reading circuit with programmable gain

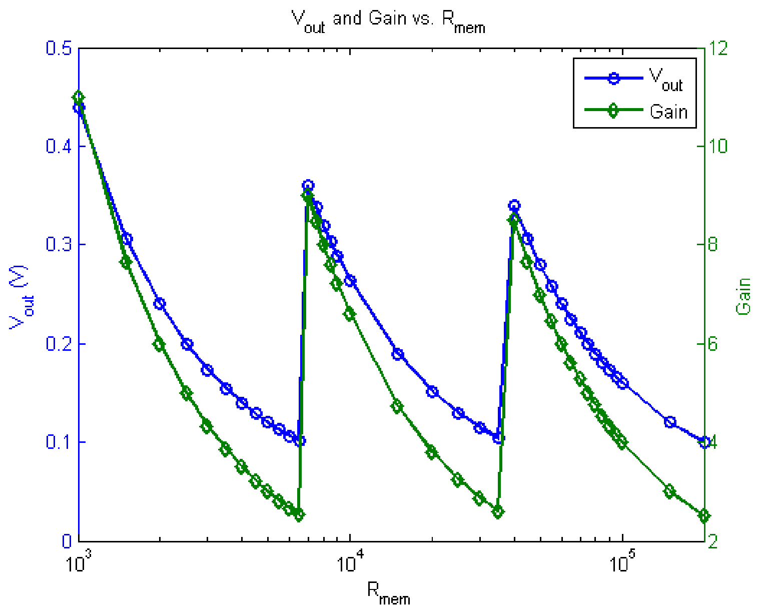
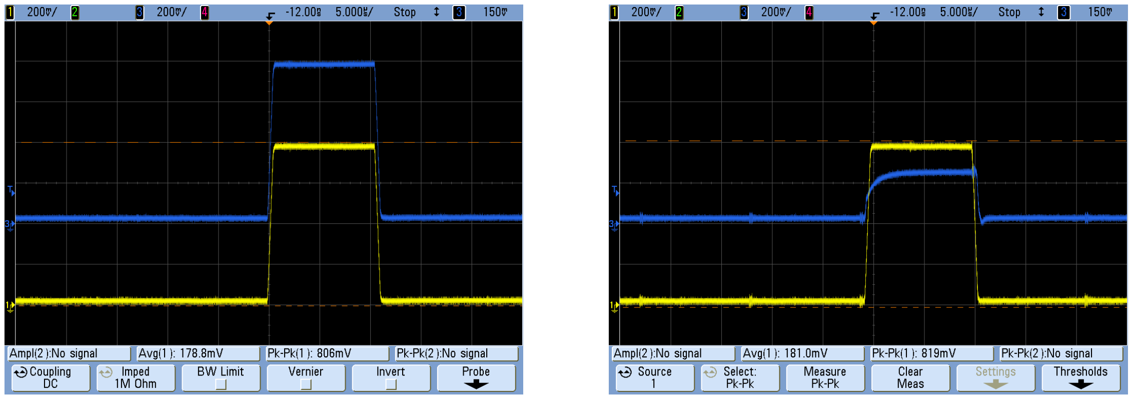


Figure 3.9: Output of programming/reading circuit with programmable gain

and R_{f4} could help cover it, but we don't want to extend our range below 1kΩ, the reason being that higher currents resulting from lower resistances could potentially harm the memristor. When programming, however, we don't want to limit the current too much either, but more importantly, R_{f4} should not add extraneous gain. First, we don't need gain when programming. Second, in order to prevent the opamp from slewing and to stay within the GBWP, R_{f4} needs to be at the low end of the memristance range.

Figure 3.10a shows an applied programming pulse (yellow, bottom) to the opamp's noninverting input and the corresponding signal at the opamp's inverting input (blue, top). The blue signal is driven by the opamp and is what's being applied to the memristor. Note that the inverting input follows the noninverting input exactly, as we can expect from the virtual gain assumption. This behavior can only be observed when R_{f4} is enabled. Otherwise, the GBWP rule might be violated, resulting in



(a) (b)

Figure 3.10: Programming pulse and inverting input behavior

a configuration in which the opamp cannot adhere to the virtual gain assumption. Figure 3.10b shows the result of using R_{f2} instead of R_{f4} as a feedback resistor when programming. We can see that the opamp cannot drive the inverting input such that it follows the noninverting input exactly. Hence, the pulse applied to the memristor is almost four-times less in magnitude than intended.

3.2.2 Noise

As explained in Section 2.2, the reading voltage across the memristive devices used for this work should not exceed 20-40mV. Otherwise, we endanger the integrity of the device as silver might be moved from one side to the other in an uncontrolled and undetectable fashion. As Section 2.2.4 indicates, trying to move silver, whether intentionally during programming or while reading, after the top electrode has no more silver left, might cause permanent damage to the device. Therefore, the reading circuit has to be designed to accommodate reading voltages in the lower mV region. This requires minimization of noise sources, as their amplitude could potentially exceed that of the small reading pulse.

Thermal Noise

When dealing with voltages in the millivolt range, thermal noise needs to be seriously considered and carefully dealt with. Thermal noise in a resistor is primarily the result of random motion of electrons due to thermal effects and is not dependent on the applied voltage. Thermal noise is sometimes also called white noise, because the power spectrum spreads evenly over nearly the entire frequency spectrum [2]. Noise is best expressed as a power spectral density or PSD. Thermal noise can then be expressed as

$$V_R^2(f) = 4kTR, \tag{3.1}$$

where k is Boltzmann's constant, T the temperature in $^\circ K$, and R is the resistor's resistance [2]. Equation 3.1 shows that thermal noise is frequency independent, hence the even spread over the power spectrum. More importantly, it shows that the larger the resistance, the higher the thermal noise. In order to find the actual induced thermal noise voltage, we have to use

$$V_R = \sqrt{V_R^2(f)\Delta f}, \tag{3.2}$$

where Δf is the range of the covered frequency spectrum. Assuming no bandwidth limitation, a resistor with a very high resistance ($> 1G\Omega$) at room temperature can easily induce a couple hundred microvolts of thermal noise [2]. Using a 20-40mV reading voltage is challenging as we get closer and closer to the thermal noise, decreasing the signal-to-noise-ratio (SNR). Limiting the memristor range to a maximum of a couple kΩ now proves to be a good idea, as it limits the amount of thermal noise introduced.

Transient and Periodic Noise

We will categorize all remaining noise signals as those of transient or periodic nature. Periodic noise usually originates from an alternating source, such as an AC voltage or a switching power supply. While by using a capacitor in low-pass configuration, we can easily filter out the 60 or 120Hz ripple voltage left on top of a power supply's DC output voltage, it is harder to filter high frequency noise from switching power supplies. This is because the switch's operating frequency can change depending on the load. On the other hand, transient noise is almost impossible to successfully suppress, as we often do not have conclusive information about its nature and origin. Transient noise signals can have many different origins, ranging from electrostatic discharge over noise from other circuits propagating through the power-lines to parasitic noise introduced in our own circuit. While we can try to block noise from outside by adding filtering capacitors and shielding techniques, it is very important to identify and understand the noise sources in our own circuit.

On-board generated transient noise was found to originate from the PGA's switches. Figure 3.8 shows multiple switches in the opamp's feedback path. Not only do the switches cause glitching due to charge injection, but they also cause the opamp to oscillate due to the rapid change in feedback resistance. Figure 3.11 shows the result of changing the PGA's gain. A glitch is followed by oscillation with a maximum peak-to-peak voltage of about 300mV. Transient noise that high is not only harmful for memristors, but could also cause an unintentional change of its state.

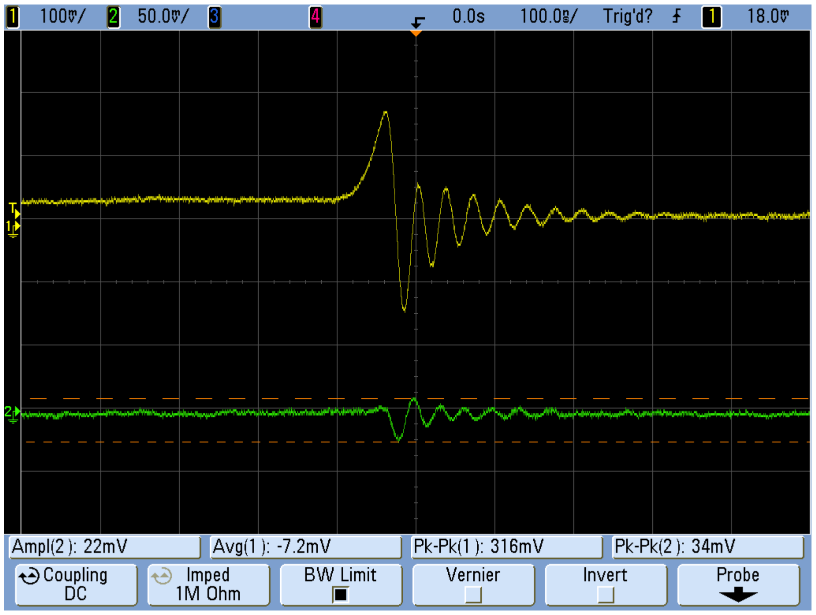


Figure 3.11: Glitch followed by opamp oscillation

3.2.3 Solutions

Now that we are aware of possible sources of noise, let's look at some preventative measures. At this point, we are not really concerned about thermal noise, as we have already ensured its minimization by using relatively low resistances. However, there are still plenty of other noise sources that need to be taken care of.

Decoupling Capacitors and Board Design

As mentioned in Section 3.2.2, decoupling capacitors can be used to decouple power supply induced noise. Decoupling capacitors, also called bypass capacitors, essentially decouple one part of the circuit from another. They act as low-pass filters, but also help to stabilize the supply voltage and reduce ground bounce and voltage droop. Each active circuit component (i.e., opamp, inverter, etc.) should have its own decoupling capacitor located right next to its supply pin. Circuit designers should consult each component's datasheet for suggested decoupling capacitor values, but some commonly used values are in the range of 0.1μF and 0.33μF. The size of the capacitor also depends on the type of material used (i.e., tantalum, ceramic, aluminum, etc.). Ceramic capacitors, for example, have a low effective series resistance (ESR) of less than 1Ω and can therefore respond to fluctuations faster than aluminum capacitors, which have a high ESR [31]. The ESR refers to the capacitor's intrinsic resistance, limiting its charge/discharge time constant and its frequency response.

The design of the printed circuit board (PCB) also plays a very important factor in noise reduction. The following design rules are only a small collection of common PCB design techniques:

- Keep traces as short as possible to keep the resistance of connecting wires low.
- Vias add resistance. Add multiple vias in parallel to reduce the effective resistance to ground- and supply-planes in order to reduce IR drop.
- Add appropriate decoupling capacitors to each supply pin and ensure that the capacitors are as close to each pin as possible.
- In addition to the ground-plane on the bottom layer, add a ground-plane to the top layer and connect the two. Extend both of them past all signal traces to provide shielding.
- Make sure the ground-plane does not get disrupted by traces. The wider the path for the current, the lower the resistance.
- Use wide power-planes instead of thin traces for supply voltages to minimize supply voltage droop.

Reducing Charge Injection and Opamp Ringing

In Section 3.2.2, we have seen that the charge injection of the PGA's switches causes glitching. The glitches along with the fast changing feedback path resistance causes the opamp to oscillate. As Figure 3.11 shows, this is a condition that needs to be resolved.

As a first measure, we can reduce the charge injection. There are two ways to do this. The first one is to simply choose CMOS transistor switches with lower charge injection. The gate of the CMOS transistor forms a capacitor with the channel right underneath. This capacitor needs to be charged and discharged when the gate is turned on and off respectively. Charging this parasitic capacitor is not a big deal as it simply presents a very small time delay. However, since the charge is effectively located in the transistor's channel, it injects right into the drain and source when the gate is turned off [2]. The capacitance of the parasitic gate capacitor is dependent on the length and width of the channel, which in turn affects the on-resistance of the transistor. Low-resistance transistors or switches therefore usually have a relatively high charge injection, since their channel is wider to reduce resistance. The increased area of the channel effectively makes for a larger capacitor. Since we're not concerned about low on-resistances of the PGA's switches, we can pick a CMOS transistor switch with a slightly higher on-resistance and therefore decreased charge injection. The ADG1221 is a good pick as it has a charge injection of less than 0.5pC over the entire signal range.

The second way to reduce charge injection is to slow the digital switching signal. This can be done by adding an RC network to the switch's input, where the resistor is in series with the input and the capacitor is in parallel to ground. The charging and discharging curve of the parallel capacitor will effectively cause the gate of the CMOS transistor to transition from an on to off state over a longer time period. Therefore, the charge should not be injected all at once, but slowly as the gate is turned off. RC networks were therefore added to each signal controlling a CMOS transistor switch, so that each gate can be slowed down if needed.

Last, but not least, we will have to deal with the opamp's oscillation. As Figure 3.11 shows, this is the real problem as it is responsible for the relatively large peak-to-peak voltage swing. The problem is essentially just that of GBW and slew rate limitations. The glitch occurs in a very short time of less than 100ns, which would correspond to a frequency of about 10GHz. Clearly, this is tough to handle, even for high-speed opamps. In fact, several different high-speed opamps have been compared by simulation and experiment to find the one with the least oscillation. Unfortunately, no off-the-shelve opamp is the perfect fit for this application, as this is a very special case. Off-the-shelve products are usually optimized for one particular application. Here, however, we need high-speed, high slew-rate, unity-gain stability as well as stability at gains up to 10, and low input offset. Once the entire circuit is built on a chip instead of using discrete components on a circuit board, a custom made opamp can be designed to fit this application. For the time being, we will have to live with the oscillation issue and compensate for it by slowing the switches' signal down such that the CMOS transistor performs the transition slower.

Environment, Experimental Setup, and Operator Behavior

The environment where the experiment or circuit is set up can also cause noise emissions. For example, the operator might get charged up while walking across the floor, causing electrostatic discharge (ESD). A lab with anti-static floors is therefore recommended for operation of such low-voltage circuitry. Furthermore, an ESD mat can help reduce electrostatic issues as it absorbs high-voltage discharges and acts as a clean ground-plane on which the circuit or device under test as well as measuring equipment can be set up. Wearing an ESD wrist-band and anti-static shoes are also good ESD reduction methods.

In general, any type of movement, especially of ungrounded surfaces, has the potential to get statically charged. Closing doors, moving chairs, people getting up, etc. all showed significant noise spikes in certain situations. Although these spikes only last for a very short time, they could potentially damage sensitive electronics such as memristors. That's why it's so important to install decoupling capacitors throughout a PCB as they can dampen or sometimes even completely filter out these spikes. For example, the circuit seen in Figure 3.4 was initially set up on a bread-board without any decoupling capacitors. Power supply and environmental noise was so heavy that the scope was instantaneously triggered by noise when its trigger was set to slightly less than 40mV to capture the 40mV reading pulse. After building a properly designed PCB, the scope's trigger could be lowered to less than 5mV without being triggered by noise.

3.3 Pulse Generator and Reading Circuit

Before we finish this chapter with the final improved memristor programming circuit, we need to discuss the requirements of the pulse generator and pulse reading circuit and find adequate equipment that can fulfill these tasks.

3.3.1 Requirements

The following list shows the requirements of the pulse generator and reading circuit:

- Variable pulse-width of at least 1μs up to several ms.
- Variable pulse height of 20mV up to 3V.
- Variable rise time profile.
- Able to quickly and easily send pulses of different heights and widths.
- Able to analyze the result of a reading pulse in near real-time.

These requirements could be met with the equipment used for the initial experimental setup (see Section 3.1), but is too cumbersome for automated measurements and network training. A better solution is to use a high speed DAC and ADC powered by an FPGA.

3.3.2 FPGA and DSP Board

In order to build a pulse generator and reading circuit as outlined in Section 3.3.1, the Altera Cyclone III development kit featuring the Cyclone III EP3C120F780 FPGA running at speeds up to 125MHz and a high speed DSP board was chosen. The DSP board connects to the development board through a high-speed mezzanine connector (HSMC) and consists of two 14-bit 275 MSPS DACs and two 14-bit 150 MSPS ADCs.

3.3.3 DSP Board Modifications and Supporting Hardware

The DACs have complementary, balanced outputs with a maximum output swing of 1V$_{pp}$ centered around GND. A balun (BALanced -UNbalanced) transformer is

(a)

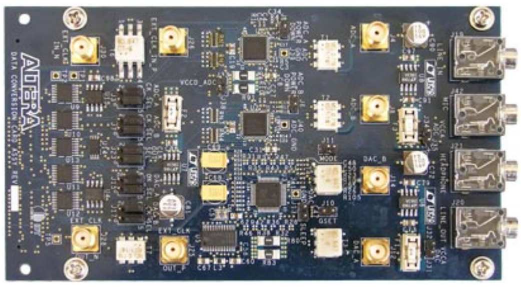

(b)

Figure 3.12: FPGA development board and DSP board

used to decouple each DAC from the output connectors. Since the baluns have a frequency response of 4kHz-300MHz, we cannot produce slow ramps and long pulses. We would expect that the balun would still pass short DC pulses falling within its frequency response. However, as Figure 3.13 shows, a 2V, 100μs pulse is severely damped such that two short pulses at +2V and -2V occur. Clearly, this does not satisfy our requirements as stated earlier. Since most high-speed DSP systems make use of a balun transformer, we would either have to build our own DAC setup or simply modify the DSP board by removing the balun.

It was determined that balun removal was the easier task. However, removing the balun means that the output is now a fully-differential signal, which is not compatible with our opamp-based programming circuit. A differential to single-ended conversion circuit was therefore built to provide the appropriate single-ended pulse signal. Figure 3.14 shows this circuit. Note that the opamp-based differential to single-ended circuit is essentially the same thing as a balun, but does not block lower frequencies (i.e., it passes DC). Figure 3.14 also shows an amplification/attenuation stage after the differential to single-ended conversion stage. This was added to provide

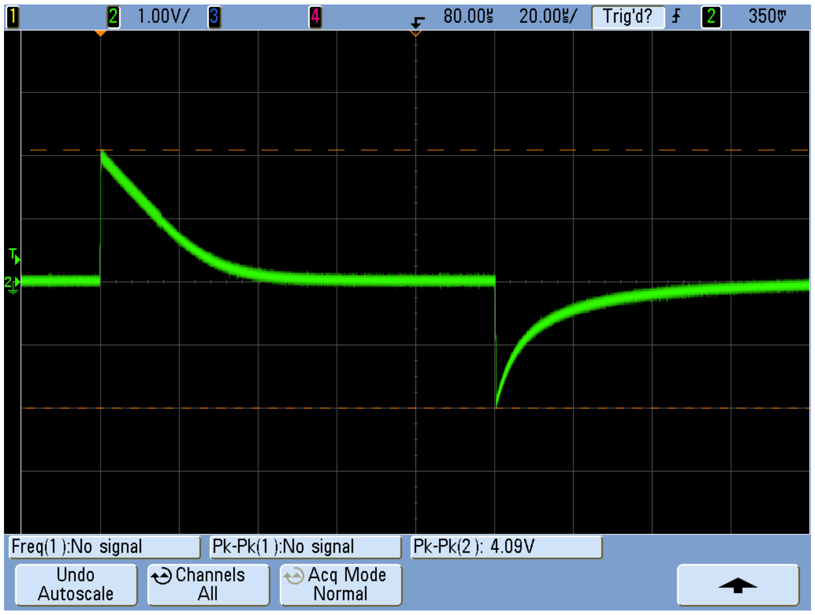


Figure 3.13: Frequency response of a 100μs pulse

amplification when writing/erasing pulses of up to +/- 3V are needed (DAC output is only 1V$_{pp}$). Attenuation is used when small signals like the 20-40mV reading pulses are required. Since the signal is sent from one board to another and then converted from fully differential to single-ended, it was decided to send a larger signal to get out of the noise floor and then to attenuate right before it reaches the programming opamp.

While the ADC is also decoupled from the rest of the circuit through a balun transformer, this is not as big of a deal as it is for the DAC. This is because the reading pulse is usually very short (1-5μs) and does not need to be variable over a range as extensive as the writing/erasing pulse. Also, the reading pulse is sent in the form of a near triangular pulse, reducing the amount of DC component present. Therefore, we do not expect the balun to block any part of the reading response. However, the ADC's maximum allowed input voltage is 1V. While the reading pulse

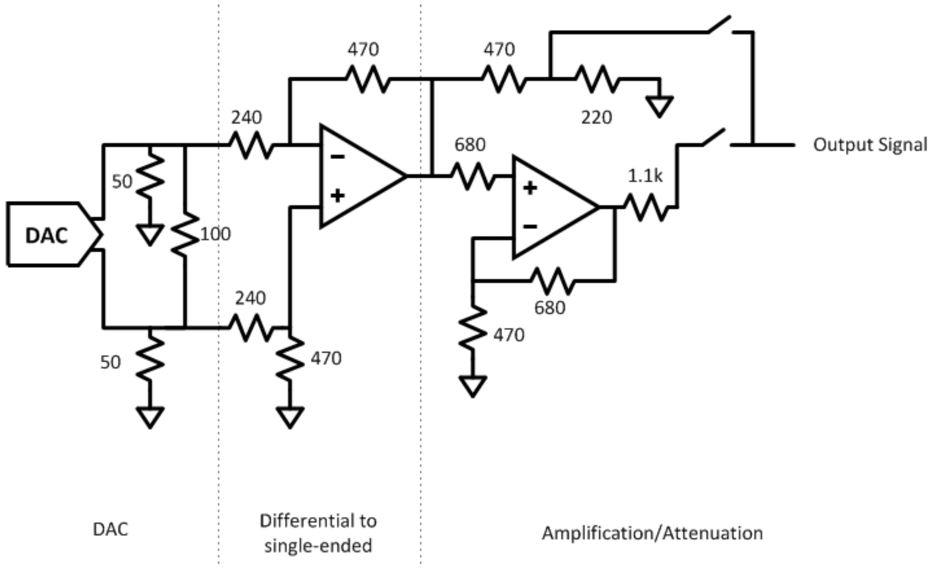


Figure 3.14: DAC output conditioning circuit

response is not expected to exceed 0.4V (see Section 3.2.1), we have to disconnect the ADC from the circuit when sending programming pulses. A CMOS transistor switch is therefore used to disconnect the ADC from the programming opamp's output and will only close when reading a memristor. Since the balun transformer presents a 50Ω load to the programming opamp, it is a good idea to buffer the opamp's signal first and then feed it to the DSP board.

3.3.4 Verilog Pulse Controller

Now that we have the appropriate hardware to create and read pulses, we need to discuss how these pulses can be generated on the FPGA. A pulse controller has been written in Verilog to implement both pulse generation and pulse reading functionality. The general architecture is shown in Figure 3.15. The controller is based on a finite-state-machine (FSM), controls all four of its subcomponents and handles the data communication between the NIOS II soft-core processor (see Section 3.3.5), the DSP board, and other external hardware such as switches and LEDs.

The pulse generator is based on an FSM itself and uses an intricate architecture to allow the generation of pulses with variable height, width, and rise time. While the pulse height is limited to a 14-bit data bus due to the 14-bit DAC, the pulse width features a 32-bit data bus, allowing the pulse generator to generate pulses up to 42s long (@50MHz). There are 8 user selectable rise time profiles available, allowing us to generate anything from square pulses to triangular waves.

The pulse reader is also based on an FSM and is tightly coupled with the pulse generator. Once the pulse controller commands the pulse generator to generate a pulse, the pulse reader locks the initial input value in and waits until the pulse generator reaches the maximum pulse height. That's when the input is read again and stored internally. This procedure is repeated right before the pulse generator starts descending back to zero. Since fast rising pulses could still cause the opamp to overshoot and oscillate a little bit, it is beneficial to measure the response when reaching the top and right before descending back to zero (after the oscillations settled). Furthermore, the pulse reader also keeps track of the highest value measured during the entire width of the pulse. This allows us to not only detect overshoot and

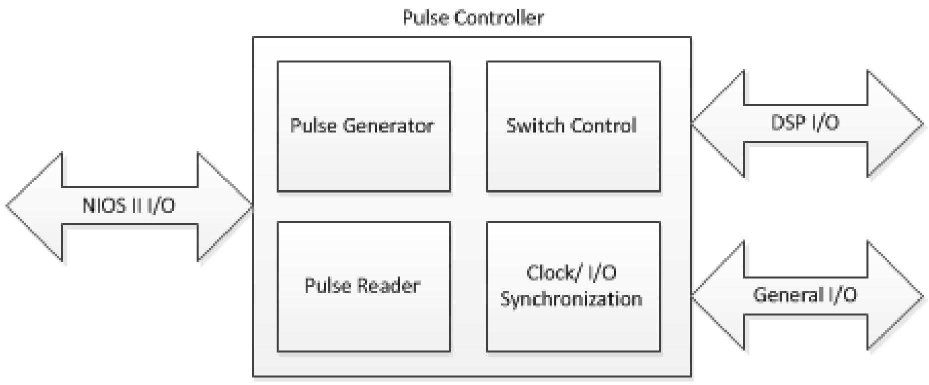


Figure 3.15: Pulse controller architecture

oscillations, but also to calculate an average. Once the pulse generator reaches zero, the input value is read again for comparison's sake with the initial value. All five measured input values are stored internally and can be read by the NIOS II processor at any time until they are cleared right before another read pulse is generated. The pulse reader also monitors the ADC's out-of-range indicator and shuts the pulse generator down immediately after an out-of-range condition is detected to prevent damage to the ADC's input.

The switch control is used to send the right signals to the switches controlling the PGA's gain, the amplification/attenuation stage, and the ADC decoupling switch. This control module implements necessary lock-out conditions, preventing unsafe operation of the circuit (i.e., sending a programming pulse with the ADC still connected). Another feature of the switch control module is to address the memristor array (i.e., connect the desired memristor to the programmer).

The Clock and I/O synchronization module is essentially just the I/O node between the pulse controller and the DSP board. Even though the ADC and DAC are driven by the same 50MHz clock as the FPGA, they both have dedicated data clock outputs used for proper data latching. The synchronization module provides output latches to buffer the data before it's sent to the DAC and doubly buffers the incoming ADC data by first latching it using the ADC's data clock and then using the FPGA's internal clock.

Not shown in Figure 3.15 is the top level project layout including a phase-locked-loop (PLL) module and the NIOS II soft-core processor module. The PLL module is used to provide a fully differential clock signal required by the DSP board. When routing high-speed data lines such as clocks and data buses from one board to the other via a connector, it is good practice to use differential signals to ensure signal

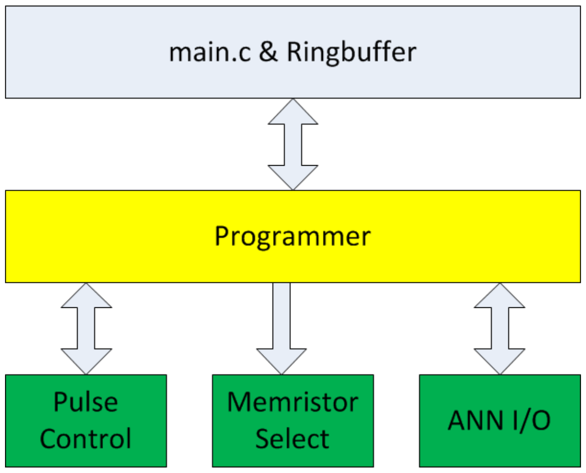


Figure 3.16: Software architecture

integrity and to minimize the impact of noise.

3.3.5 NIOS II Processor and C Control Program

As Section 3.3.4 already mentioned, a NIOS II soft-core processor was implemented on the FPGA to provide a flexible platform to control and interact with the programming circuit. More specifically, the NIOS II soft-core processor is used as a master controller and communication interface between the pulse controller and a computer. The software, written in C, consists of firmware level code, controlling the hardware-based pulse generator and reading modules. The higher level code serves to communicate with a host computer and to perform the network training. Even though the training algorithm is implemented in software, it is considered on-line training as the algorithm runs on the FPGA, which is an integral part of the feed-forward network. The host computer is merely used as a user-interface. All the functionality is implemented either in hardware or software running on the soft-core processor.

Figure 3.16 shows a block diagram of the software architecture. The bottom blocks depict the firmware and are basically just device drivers to provide hardware abstraction for higher level functions. The programmer simply talks to the device drivers without having to interact with the hardware itself. This abstraction provides an environment where the Matlab simulation code that Thanh Tran wrote for her memristor-based ANN simulations [32] can easily be ported over to C.

The top-level is formed by the *main.c* file as the program's entry point and most importantly the ring-buffer. There are many valid scenarios of a communication layer that could have been used for this project. However, it was decided to use an interrupt based ring-buffer. The advantage is that there is very little overhead involved with such a setup because no polling is involved. As long as there is no communication, the processor spends all of its time entirely on training and programming. Once the host starts communicating with the soft-core processor, an interrupt ensures that each sent character is read from the UART's internal buffer and stored in a ring-buffer for processing.

3.4 Improved Programming/Reading Circuit

Throughout this chapter several improvements to the initial programming/reading circuit seen in Figure 3.4 have been made. An FPGA was added to control the PGA and to provide a flexible pulse generator. The improved programming circuit is now connected to the supporting hardware.

Figure 3.17 shows the final and improved programming/reading circuit. The functionality of the DAC, fully differential to single-ended, and amplification/attenuation circuit was already elaborated on in Section 3.3.3. These three stages are considered the signal generation and conditioning part of the programmer. The actual programming circuit is based on the PGA seen in Figure 3.8. Since the memristor

Table 3.2: Opamp details

Symbol	**Opamp Type**	**Max. Gain**
U1	AD8055	2.95
U2	AD8055	2.44
U3	LM6171	Up to 10
U4	AD8055	1

in the PGA's feedback network is not always in place (the programming circuit connects to one of many memristors we wish to program), a 1kΩ resistor was placed in parallel. This is useful when switching a memristor in, because the sudden load in the feedback network would cause the opamp's output to oscillate due to the sudden load change. The parallel load is always connected when no memristor is present in the programming circuit and is only disconnected once a memristor is present. The nodes *X1* and *X2* indicate the connection to the memristor array. This implies that the single memristor seen in Figure 3.17 is in reality an array of memristors. Section 4.6 elaborates on how each memristor is addressed individually.

The buffer and ADC stage consists of a simple buffer in voltage follower mode and is solely intended to decouple the ADC's 50Ω load from the programming circuit's output. Note that the 1kΩ load on the buffer's positive input is not necessarily needed, but was included in the PCB design such that it can be populated if an unstable situation arises. Also note that the ADC is not directly connected to the output. As with the DAC, the ADC's input is fully differential and uses the same balun configuration. As mentioned before, the ADC's balun does not need to be removed due to the short reading pulse duration. Therefore, the single-ended to fully differential conversion circuitry is part of the ADC setup on the DSP board. The opamps used in Figure 3.17 are as in Table 3.2.

Note that *U1, U2,* and *U4* have low gains and therefore do not require an opamp

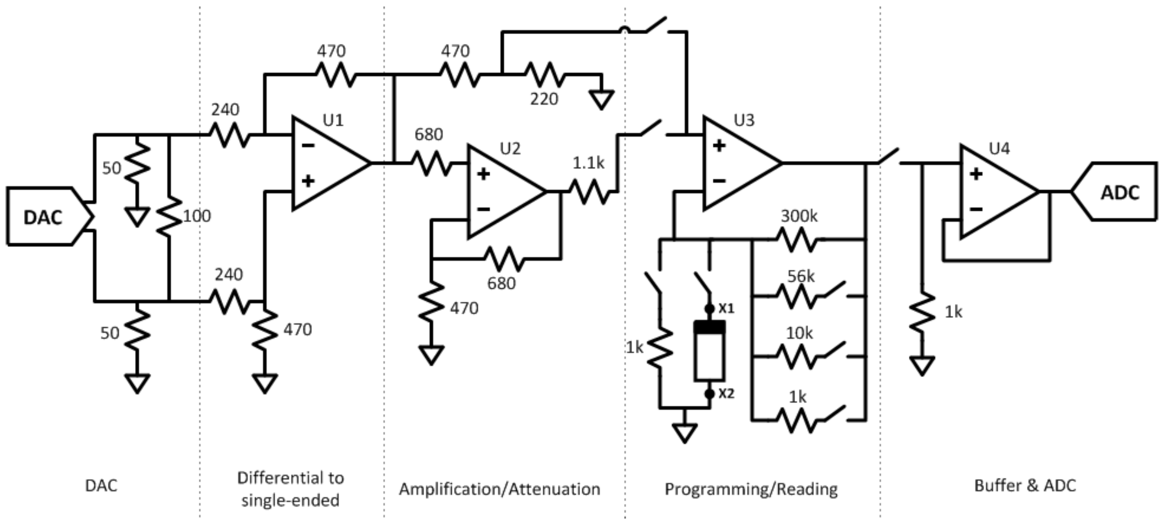


Figure 3.17: Final memristor programming/reading circuit

type with special characteristics. The AD8055 used here is an opamp commonly used for signal buffering and ADC/DAC signal conditioning. However, *U3* is a little different. As Section 3.2.3 mentioned, the programming opamp has to satisfy stringent requirements. Unity gain as well as higher gain (up to 10) stability, high-speed, and low input-offset voltage are all very important parameters that directly influence the performance of the programming circuit. Picking a suitable device is therefore crucial. Unfortunately, it is very hard to find an off-the-shelf opamp that fulfills all requirements, as this application calls for a very specialized opamp. Therefore, 10 different opamps were chosen for further investigation based on bandwidth, gain stability between 1 and 20, input-offset voltage, high input impedance, low output impedance, and high peak-to-peak output range. The chosen devices were tested with different loads and programming pulses. The LM6171 performed best overall and was therefore chosen for *U3*.

CHAPTER 4

FROM ANN THEORY TO HARDWARE

In Chapter 2, the basics of ANNs, memristors, and opamps were introduced. Chapter 3 then showed the development of a circuit that can be used to program memristors. This chapter first shows how the theoretical ANN structure can be implemented in discrete hardware. Then the synapses are built with an array of memristors and connected to the programming circuit so that the network can be trained.

4.1 Historic and Current Attempts of Hardware ANNs

Before diving into the circuit development of hardware-based ANNs, let's first do a brief historical review of what has already been done. While Section 2.1.1 already gave a brief review of the history of ANNs, it did not provide any information about ANNs built in hardware. Despite the fact that computer simulation tools were not as readily available in the 1960's, an elaborate perceptron software simulation program for the IBM 7090/94 system was developed [24]. However, Rosenblatt has always insisted on the parallel, analog implementation of his perceptron model for large-scale experiments. In 1958, he built the Mark I perceptron at the Cornell Aeronautical Laboratory, consisting of a 20x20 photo-cell retina and 512 stepper-motor-controlled potentiometers acting as synapses [24]. Between 1961 and 1967, Rosenblatt built the four-layer *Tobermory* perceptron, consisting of 45 sensory synapses in the input layer, 1600 associative synapses in the first hidden layer, 1000 associative synapses in the second hidden layer, and 12 response synapses in the output layer [24]. The network was intended for speech recognition and was built entirely in hardware. By the time *Tobermory* was completed, it was already outperformed by the rapidly advancing digital computer technology [24].

The continued technological advance of computers caused researchers to "build" their ANNs in software. Today's computers have enough computing power to simulate very large ANNs with thousands, even millions, of synapses and neurons. However, the computational complexity increases dramatically when a network is expanded. Let's recall the example from Section 1.3, where a simulation of a cat's neuro cortex was run on a massive IBM supercomputer. The simulation ran 83 times slower than in nature [17]. The issue is that modern computers suffer from their prevalent serial architecture and the Von Neumann bottleneck. That's why researchers started going back to hardware and tried to implement neural networks using both discrete circuits and CMOS technology in the late 1980s. With the rise of field programmable gate arrays (FPGA), which offer massive parallelism, an alternative for software implementation of ANNs was presented. FPGAs are of interest, because the network can be implemented in hardware and runs fully parallel. Therefore, slower clocking rates can be used, which drastically reduce power consumption. Over the last two decades, many successful attempts have been made to implement ANNs on FPGAs.

In 1991, Holt and Baker determined that the optimal data bus width of an FPGA implemented ANN is 16-bit fixed point, since the tradeoffs of precision vs. area are optimized [16]. While longer bus widths reduce quantization errors in calculations, they are significantly more expensive in terms of utilized hardware. Approaches using floating-point arithmetic have been explored, but were deemed not feasible [25]. A study on the XOR benchmarking problem has shown that a 16-bit fixed point ANN does not only process weight updates faster, but most importantly converges to a solution faster and more consistently than a 32-bit floating point implementation on both an FPGA and Intel Pentium III CPU [21, 25].

The problem with FPGAs, however, is that ANNs are analog in nature, making computations more expensive in digital hardware. The advantage of parallelism now turns into a resource issue. For instance, a 3-layer network with 8 inputs and 1 output implemented on a Xilinx Spartan-3E FPGA requires about 80% of 18x18 multipliers used for the activation functions. Furthermore, the network used 41% of all available LUTs and 27% slice registers [28]. Even though this implementation includes the backpropagation learning algorithm, the majority of resources (18x18 multipliers) is used by the neurons, which use a multiply-and-accumulate (MAC) technique. In addition to that, the activation function itself can be very costly in terms of digital hardware resources. Usually, a sigmoid or hyperbolic tangent is used for the activation function. Obviously, they are complex to compute in digital hardware. Alternatives are to use a look-up table, which requires large amounts of memory, or to use a computationally simpler approximation, which reduces accuracy [3]. Keep in mind that inaccuracy and memory resources are highly dependent on the bus width.

While FPGAs are not a viable solution for neuro cortical simulations, they provide an efficient and low-cost platform for practical applications. It is common to train an ANN off-line (on a computer; not on the actual HW/FPGA) and then download the configured network to the system (FPGA). The disadvantage of off-line training is that possible changes to the pattern recognition parameters requires re-synthesis. In addition to that, if the system is deployed in the field, downloading the new bit-file to the FPGA might add additional difficulty. However, there are several different ways of training the network online on the FPGA. The simplest is to couple the FPGA or its memory with a processor running the training software. Alternatively, a soft-core processor can be directly implemented on the FPGA. Recent research has shown that the common backpropagation algorithm can also efficiently be implemented directly on the FPGA in digital hardware [13, 14].

As Rosenblatt pointed out, ANNs should be implemented in a parallel, analog manner to be most efficient. FPGAs get close, but due to their digital nature they are still too complex. A better way would be to implement the network on an analog-mixed-signal chip. In the early 1990s, Lont and Guggenbühl successfully implemented a neural network on-chip in CMOS technology [22]. The synapse weights were stored in an off-chip digital memory, loaded to the chip by a D/A converter, and then stored in DRAM-like capacitor cells. This approach seems like a good idea, but still suffers from adaptive weights being implemented in a digital architecture. With the discovery of memristors, we are one step closer to the implementation of a fully parallel ANN in a fully analog architecture.

4.2 Neuron

It is best to look at the physical composition of the neuron before going into details about synaptical circuit design. The neuron actually sets some constraints that need to be met by the synapse. Let's first revisit the general synapse-neuron model as discussed in Section 2.1.2. Each neuron sums up all signals from the synapses that are connected to it, before sending the sum through the activation function. The neuron can therefore be broken into two parts: The summation and the activation function. The summation itself is very straightforward and only needs to be discussed in detail because of its implementation in hardware. However, the activation function, as seen in Section 2.1.2, can be described and implemented in many different ways (i.e., sigmoid or hyperbolic tangent).

4.2.1 Summation

Figure 2.2 in Section 2.1.2 shows the general synapse-neuron model and although it doesn't show it explicitly, the summation is essentially performed by combining all synapse connections in a common node. Two observations can now be made, the first being that Kirchhoff's current law governs the behavior of multiple signals being connected to a common node. Thus, the quantity being added is current. The second observation is that the common node should not be floating, but rather be at a fixed reference. An opamp can easily meet this constraint. Using these two observations, an opamp-based circuit that simply adds the input signals can be developed. Figure 4.1 shows this circuit. It needs to be noted that the feedback resistor R_f can add gain. Even though neurons usually do not have gain in their summation stage, we will later see that this is actually a useful feature for setting the synapse's weight range.

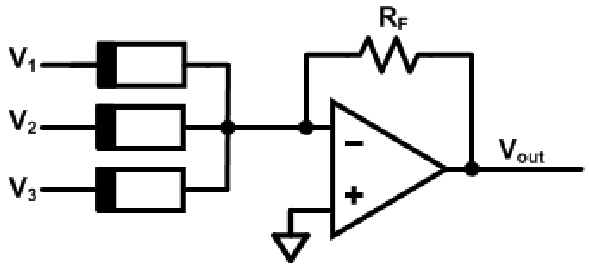


Figure 4.1: Opamp summing the three input synapses

4.2.2 Activation Function

Now that the summation is taken care of, let's look at the activation function. Section 2.1.3 showed that the activation function adds the nonlinearity required to solve nonlinearly separable problems. If the activation function were a simple linear operator, a nonlinear problem could not be solved. Designing a truly nonlinear activation function as described in Section 2.1.2 is not a trivial task. A non-ideal inverter would do the trick, but since semiconductor technology has advanced to the point where inverters behave very closely to the ideal model (i.e., a sharp transition from one state to the other), we can only get a digital output instead of an analog output, which can assume any value between the activation function's two saturation values.

Nonlinear sigmoid-like activation functions can be built using transistors on-chip with relatively little effort. However, this work is based on discrete components on a PCB due to the proof-of-concept phase of the project and designing our own custom activation function on-chip is out of question. Building the transistor-based activation function on a PCB is an alternative, but would be too complicated due to the vast space and resource requirement. Fortunately, Tran showed in her work that the activation function can be simplified to a hard-threshold activation function for implementation of reconfigurable TLGs such as AND, OR, NAND, NOR, and even XOR and XNOR [32]. She furthermore showed that methods to train such networks exist. A hard-threshold activation function can easily be realized using a comparator. As already mentioned in Section 2.1.4, a neuron using such an activation function is referred to as a threshold logic unit (TLU). A network using TLUs is then called a threshold logic gate (TLG).

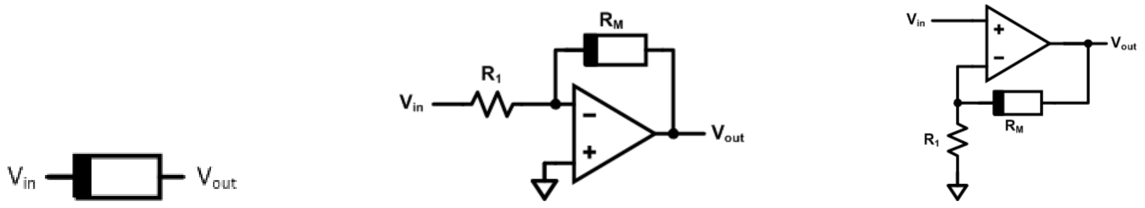
Threshold Logic Unit

In Section 2.1.3, we have seen that some functions can be solved with a linear decision boundary. These linearly separable functions, like AND, OR, NAND, or NOR do not require the synapse to have a sigmoidal activation function. While the nonlinearity is still needed for the network to function properly, we can simplify the problem by reducing the activation function to a piecewise linear function, which is still nonlinear. A piecewise linear function in the form of a step, for example, is sufficient to successfully build a network capable of recognizing linearly separable functions. A step function can easily be implemented with a comparator or a logic inverter. A comparator is in general the better choice, as inverters have a wide hysteresis (i.e., they switch from logic low to high at a lower point than switching from logic high to low; the switching point is not necessarily midway between logic low and high). Since such an activation function causes the neuron's output to be either logic 1 or 0, it is essentially a neuron with a logic output.

As mentioned in Section 2.1.6, networks using TLUs cannot be trained with the backpropagation algorithm, as the derivative of the activation function is infinity at the switching point and zero otherwise. Instead, the MRII algorithm is a viable option to train such a network. Tran showed that a multi-layer TLG can be used to solve simple nonlinearly separable functions like XOR and XNOR using MRII for training [32].

4.3 Synapse Supporting Negative Weights

In general, a synapse can loosely be defined as an amplifier or attenuator, as it adds a weight to the input signal. In physical circuitry, this can be done in many different ways, but the simplest and most intuitive approach is to use a resistor. However, resistors are fixed and therefore do not allow us to adjust the weight or gain of a synapse. This is where the memristor comes in, because it offers adjustable resistance while behaving like a linear resistor as long as the signal does not exceed a certain threshold value. We might find that our ANN requires some of its weights to be negative. Since resistance is a magnitude and therefore an absolute value, a synapse cannot simply be composed of a resistive device. Instead, a couple more circuit elements are needed to give the synapse the capability to be set to a negative weight value.



(a) (b) (c)

Figure 4.2: (a) Simple synapse using only a memristor, (b) Synapse using an inverting opamp for gains in the range of [-M,0], (c) Synapse using a noninverting opamp for gains in the range of [0,M].

We can assume that the synapse is just a black box, adding weight to its input. The weight can be thought of as a gain, meaning that the synapse is theoretically adding a gain from -∞ to +∞ to the input signal. In practice, the gain is bounded by a lower bound M and an upper bound N because of physical limitations. The synapse's gain is therefore finite but continuous over the region *[M,N]*. Figure 4.2 shows some possibilities for synapse circuits that fulfill this general model. However, all of them have some limitations that we wish to overcome. The simple synapse shown in Figure 4.2a consists of a single memristor and is at best able to set the gain in the range of [0,1]. Figures 4.2b and 4.2c are not as restricted as Figure 4.2a, but can only add negative or positive gain, respectively. Depending on the application, type of ANN topology, and decision boundary, synapses only supporting positive weights (i.e., Figure 4.2c) might be sufficient. Since we're trying to develop a general synapse hardware model, we have to find a better solution that can support both positive and negative weights with the same circuit.

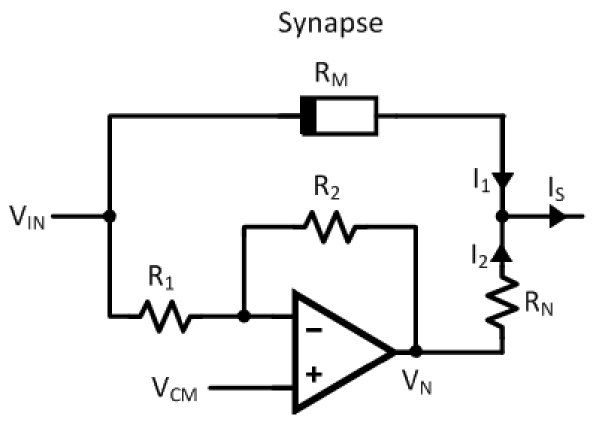


Figure 4.3: Synapse supporting both positive and negative weights

Accepting the constraints outlined above, we realize that a fixed negative offset in parallel with a positive gain stage can achieve gains in the range of *[M,N]*, where *M<0* and *N>0*. This can be done relatively easy with discrete circuit components by using an inverting opamp with fixed gain to give us a constant negative gain offset. A noninverting opamp with adjustable gain using a memristor is put in parallel to compensate for the fixed negative gain. Since we want to keep the design as simple as possible, we can simplify this synapse model by realizing that the adjustable noninverting opamp stage can be replaced by a single memristor. Figure 4.3 shows the final synapse circuit design, which supports both positive and negative weights.

Now that a hardware synapse supporting negative weights has been developed, a general model that allows conversion from conceptual weights to physical parameterization of the circuit needs to be formulated. Let's start by reiterating that the neuron to which the synapse is connected sums current. Therefore, the synapse in Figure 4.3 converts a voltage to current and the neuron converts the summed currents back to a voltage. The current I_s in Figure 4.3 will only flow when a load - a neuron - is connected. Therefore, the synapse needs to be connected to a neuron for circuit analysis. Figure 4.4 shows this setup. Note that the input to the synapse is a voltage (V_{in}) and so is the neuron's output (V_{out}). Multiple synapses can be added, each producing its own current signal I_{s_i}, which are all summed by the neuron.

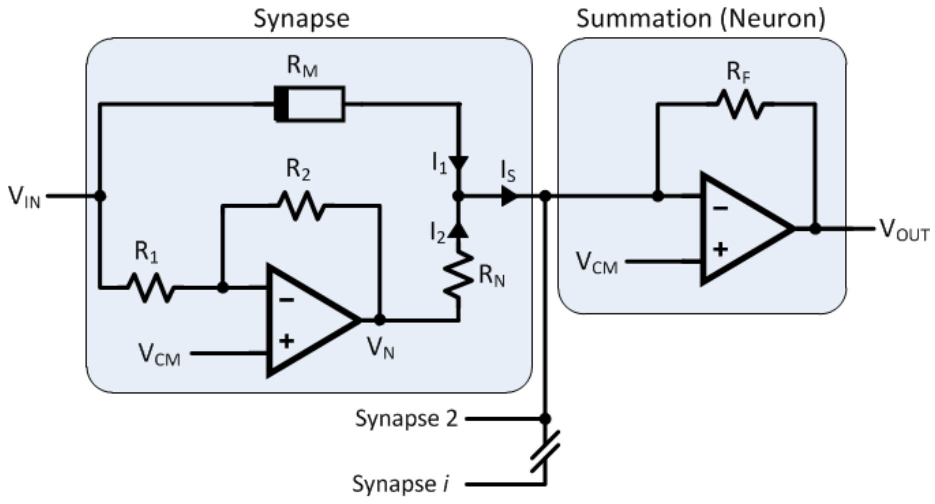


Figure 4.4: Synapse connected to a neuron

The summing opamp's noninverting input is tied to V_{CM}, resulting in a fixed inverting terminal to which all synapse currents flow. The currents I_1 and I_2 can then be expressed in terms of the synapse's circuit elements. The output voltage V_N of the negative offset opamp is described by

$$V_N = V_{CM} - R_2 \frac{V_{IN} - V_{CM}}{R_1}. \tag{4.1}$$

The output current I_S of the synapse can then be expressed as

$$I_S = I_1 + I_2 = \frac{V_{CM} - V_{IN}}{R_M} + \frac{V_N - V_{CM}}{R_N}. \tag{4.2}$$

Using both Equations 4.1 and 4.2 as well as the general inverting opamp equation (Equation 2.10), the transfer function H of the synapse-neuron circuit can be derived to be

$$H = \frac{V_{OUT} - V_{CM}}{V_{IN} - V_{CM}} = -R_f\left(\frac{1}{R_M} - \frac{1}{R_N}\right) = -R_f(G_M - G_N). \tag{4.3}$$

Note that both the input and output voltages of the circuit are with reference to V_{CM}. Hence, the term shows up in the generalized definition of the transfer function H. Another important observation we can make when looking at Equation 4.3 is that the transfer function H can also be expressed in terms of the conductances G_M and G_N. In fact, the conceptual weight of a synapse is best expressed physically as its conductance, not its resistance. Since both R_f and G_N are fixed, the conductance G_M of the memristor is what affects the output. However, the conductance of the memristor is not the weight of the synapse. The synapse's gain is a function of R_M, R_f, and R_N. The transfer function H in Equation 4.3 is also the value of the conceptual weight used in computer simulations. In order to convert from conceptual weights to physical memristor resistance, Equation 4.3 can be solved for R_M

$$R_M = \frac{R_F R_N}{R_F - HR_N}. \tag{4.4}$$

The summing opamp's feedback resistor R_f is responsible for converting the summed current back to a voltage. This is only possible because the opamp's inverting input is held constant at V_{CM} due to the virtual ground. As stated before in

Section 4.2, this resistor can be used to add gain. As Section 4.4 shows, R_f is actually used, among other parameters, to set the gain range of the synapse.

As a concluding statement of this section it should be noted that the input offset voltage and noise of opamps can affect the output of the circuit seen in Figure 4.4. However, the noise added by the input offset voltage through the opamps's noise gain is merely a constant measure and therefore a characteristic of the circuit itself. Therefore, it can easily be compensated for, especially with a learning algorithm.

4.4 Finding Optimal Values for the Synapse-Neuron Circuit

At this point we have a solid base for the synapse as well as the neuron circuit and have seen the general mathematical formulation for both. Equation 4.4 allows us to convert a conceptual weight to physical memristor resistances. This section focuses on the optimization of resistor values for R_f and R_N as well as choosing the right memristor resistance range so that the desired synapse gain range can be realized.

The fine tuning of the circuit described in Sections 4.2 and 4.3 starts by first defining the range of the synapse gain. Let's define that gain to be in the range [-2,2]. In Chapter 3, the operating range of the memristors was set to lie between 1kΩ and 100kΩ. The boundary parameters are therefore as seen in Table 4.1.

Table 4.1: Boundary parameters

Parameter	**Symbol**	**Value**
Highest gain	H_{High}	2
Lowest gain	H_{Low}	-2
Highest memristor value	R_{MHigh}	100kΩ
Lowest memristor value	R_{MLow}	1kΩ

The value of R$_N$ is computed using

$$R_N = R_{MHigh} - \frac{R_{MHigh} * H_{High}}{R_{MHigh} * H_{High} - R_{MLow} * H_{Low} * (R_{MHigh} - R_{MLow})}, \tag{4.5}$$

which makes sure that the synapse's output current at R$_{MLow}$ is twice the negative value of the input and at R$_{MHigh}$ twice the positive input value. With R$_N$ found, the neuron's feedback resistor R$_F$ can be computed using

$$R_F = R_N \frac{R_{MHigh} * H_{High} - R_{MLow} * H_{Low}}{R_{MHigh} - R_{MLow}}. \tag{4.6}$$

Using the boundary parameter values from Table 4.1, R$_N$ evaluates to 1.98kΩ and R$_F$ to 4.04kΩ. Figure 4.5 shows the simulated gain of the synapse-neuron circuit using these values.

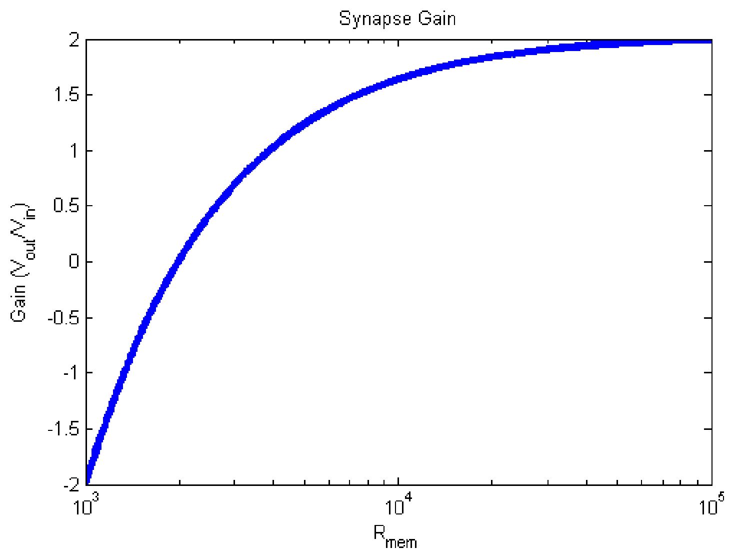


Figure 4.5: Simulated synapse gain

As Figure 4.5 shows, the gain curve is highly nonlinear, meaning that the change in synapse gain is far greater for lower memristor resistances than it is for higher resistances. This implies that the weight to resistance conversion is nonlinear. Recall the weight cube in Figure 2.5 depicting where the AND, OR, NAND, and NOR patterns are located in the weight space. Figure 4.6 shows the same in terms of memristor resistance and it becomes clear that the NOR pattern is essentially the hardest to train while the OR pattern is the easiest. This is because the OR pattern spreads over a much larger resistance range as opposed to the NOR pattern.

Since we know that R_F affects the gain (i.e., [-10,10] vs. [-2,2]) and R_N causes the gain-resistance curve to shift horizontally, we can alleviate the nonlinear weight distribution issues by optimizing the values for R_F and R_N. Figure 4.7 shows different resistance-gain curves. Note that the x-axis is not in log-scale as opposed to Figure 4.5 to emphasize the nonlinearity issue and how it can be solved. As a trade-off, we might have to alter the memristor resistance range for adequate gain coverage.

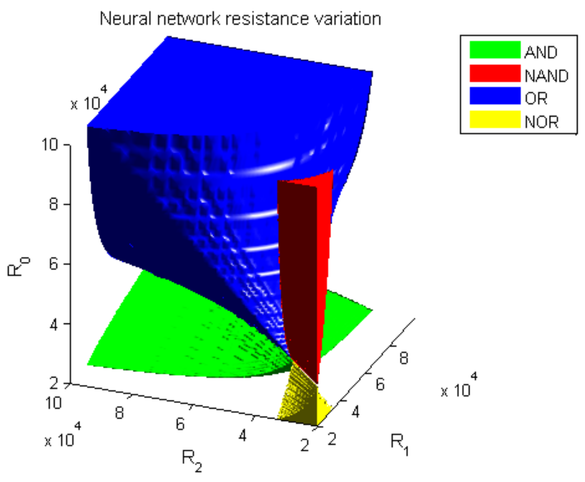


Figure 4.6: Network resistance variation and its outcome

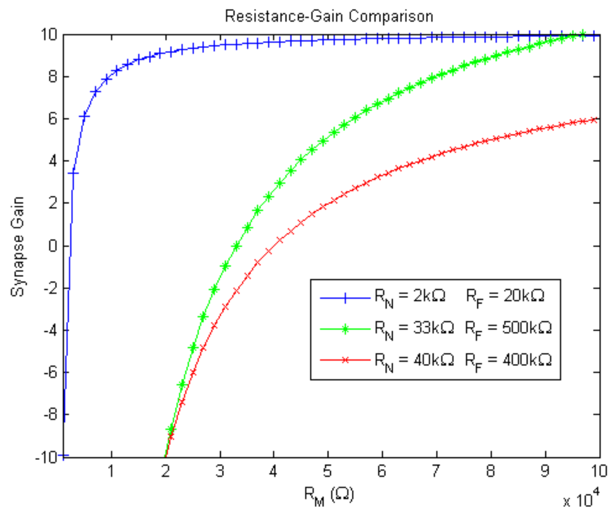


Figure 4.7: Synapse gain curve comparison

4.5 Alternative Synapse Circuits

The synapse introduced in Section 4.3 is not the only solution to negative weights. Adhikari et al. recently published their work on a memristor bridge synapse [1]. The synapse can be seen in Figure 4.8. At first glance, it seems that this circuit is more elegant than what has been shown so far. However, while this circuit implements both positive and negative weights, it is not able to provide amplification, meaning that the synapse's gain is never greater than or equal to 1. The weights of an ANN can essentially all be normalized to lie within a range of [0,1] or even [-1,1]. This is not a big issue for computer simulations where we have double-precision floating point accuracy available. Even though analog signals are continuous, they are only in theory more accurate than their digital representation. As was discussed in Chapter 3, small feed-forward voltages can cause the attenuated signal to be in the noise floor. Weight scaling or normalization is therefore not always a good idea and it is of benefit to have a synapse supporting gains greater than 1 and less than -1.

Another issue the circuit in Figure 4.8 poses is that there are multiple memristors. Memristor models available today are idealized and do not exhibit the true memristor behavior. To the best of the author's knowledge, literature has yet to present a way to accurately program memristors. The current issue is that each memristor has a unique intrinsic characteristic, which can change dependent on the pulses applied to it. Therefore, it becomes exponentially difficult to program a synapse with four memristors as opposed to a single memristor. The synapse proposed by Adhikari et al. was only simulated and might seem like a good approach to a more linear adaptive synapse, but it is very hard to control in hardware.

An alternative is to replace three of the four memristors with regular fixed resistors, which will turn the bridge synapse into a design similar to the one described in this chapter. One output of the bridge is fixed and provides an offset or reference voltage, while the other output can swing above and below the reference such that positive and negative weights can be achieved. The resistance-weight curve of this circuit can be adjusted by changing the values of the fixed resistors. The resistance-weight curve of this circuit is nonlinear as seen in Figure 4.9, therefore posing similar programming difficulties as the synapse circuit introduced in Section 4.3. The synapse gain curve in Figure 4.9 was achieved by replacing R_{M1}, R_{M2}, and R_{M3} with fixed

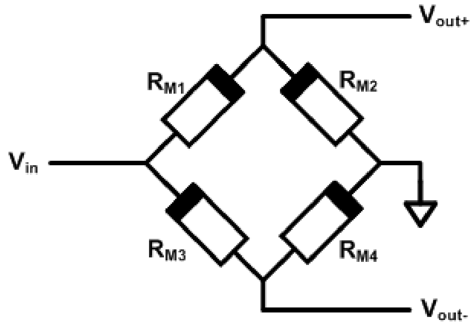


Figure 4.8: Memristor bridge synapse

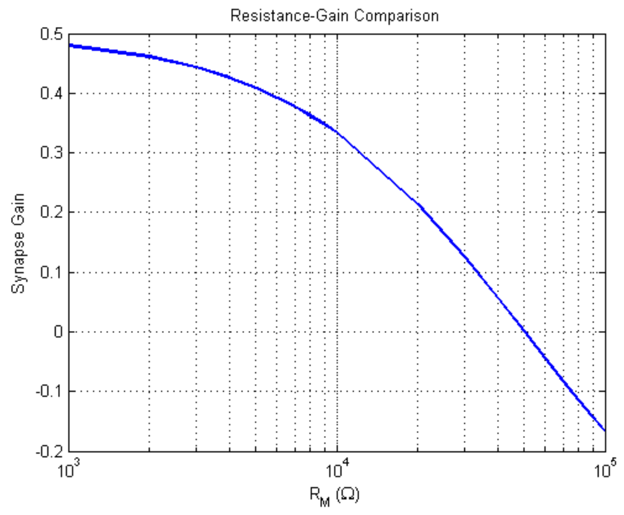


Figure 4.9: Gain curve of the memristor bridge synapse

resistors of $50k\Omega$ and R_{M4} assuming a resistance range of $1k\Omega$ - $100k\Omega$.

4.6 Putting Everything Together

In this section, the programmer and network is put together to form the final programmable network. Chapter 3 describes the programing circuitry while this chapter elaborates on the network structure itself. Referring to the objectives in Section 1.4, we see that Chapters 3 and 4 have already covered the development of the programming circuit and the feed-forward network with synapses and neurons. In order to build a fully hardware implemented ANN, the programming circuit needs to be connected with the network itself such that each synapse consisting of a memristor can be programmed. Figure 4.10 shows a conceptual depiction of the programming circuit connected to the network. It is a single TLG with two inputs (V_{in1} and V_{in2}) and one bias node (V_{in0}). Note that the three synapses are depicted in a simplified manner. The negative offset circuit is the bottom half of the circuit seen in Figure 4.3,

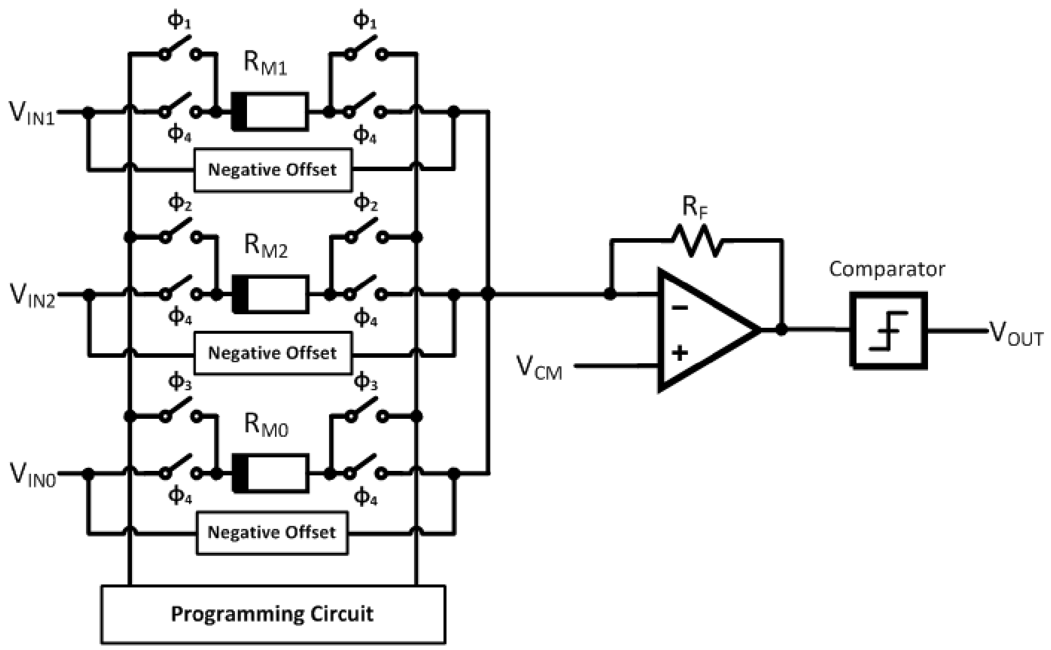


Figure 4.10: Network with Programming Circuit

producing the current I_2.

For programming, a memristor has to be fully disconnected from the feed-forward network and connected to the programming circuit. Once programming is done, the memristor is put back into the network. Figure 4.10 shows switches that allow each memristor to be connected to either the programmer (see Figure 3.17) or the network. Only one memristor can be programmed at a time. Switches ϕ_1 connect memristor R_{M1} to the programmer, ϕ_2 selects R_{M2}, etc. while ϕ_4 connects all memristors to the network for feed-forward operation. The switches are controlled by the switch control module (see Section 3.3.4), which allows proper addressing of the memristor array, such that the desired memristor can be programmed. The switch control module implements necessary lock-out conditions, preventing unsafe operation of the circuit, which could potentially harm the memristors.

The comparator is used as a hard-thresholding activation function and compares the neuron's sum to V_{CM}. If the sum is greater than V_{CM}, then the comparator's output is 1 (5V), otherwise it is 0 (0V). Note that the TLG's output (V_{OUT}) is only valid during feed-forward operation when all ϕ_4 switches are closed. For this work V_{CM} was set to zero.

Another important part is the signal conditioning of the inputs to the network. As stated in Section 2.2, the voltage across a memristor should be as low as possible if we wish to operate in the linear region, which is required for feed-forward operation. Therefore, the input signals from the FPGA are scaled down and shifted to +/-20mV.

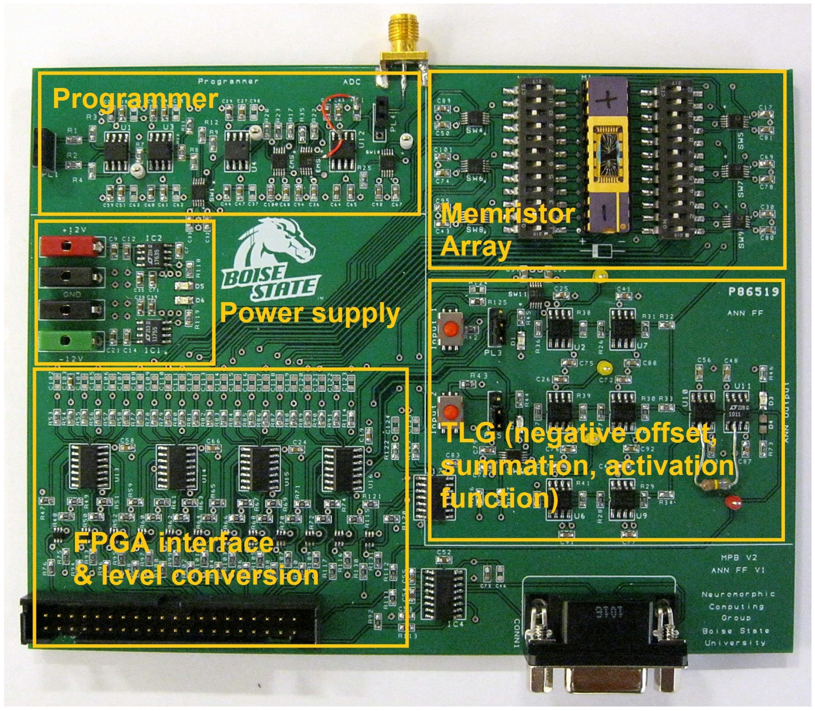

Programmer
Memristor Array
Power supply
TLG (negative offset, summation, activation function)
FPGA interface & level conversion

Figure 4.11: PCB built for this project consisting of programming circuit, memristor array, and ANN

Figure 4.11 shows the final four layer PCB consisting of programming circuit, memristor array, and TLG. The PCB also consists of a power supply and level conversion circuit, which translates the FPGA's 2.5V digital levels to the PCB's

5V level. All necessary control signals are routed through the FPGA interface, while programming pulses are brought in from the DAC through the connector on the top left. The connector seen in the top middle is used to bring the response of a reading pulse to the ADC.

4.7 Scalability and Expansion

When designing an ANN in software, the architecture is often not a big problem, as only the feed-forward network has to be implemented. When "programming" the weights, the software simply writes the updated weight value in the corresponding memory location. However, when building an ANN in hardware, we can't simply update a memory location, but rather have to physically connect the synapse (in our case a memristor) to the programming circuit. This requires a special architecture, which adds some overhead. As Figure 4.10 shows, this architecture consists of a lot of switches, allowing the programmer to individually address each memristor. While this approach is acceptable for smaller networks, the question arises as to whether this architecture is scalable with increasing network complexity.

The network seen in Figure 4.10 requires 4 switches for each memristor, for a total of 12 switches. An XOR network requires a second layer with a 2-1 topology (2 neurons in the first layer, 1 neuron in the second layer). Using bias weights, the XOR network requires 9 memristors, meaning that the total number of switches is 36. Clearly, this is not scalable, as the overhead architecture gets too complex and too resource intensive with increasing network size. Especially for this work, which was built on a PCB for proof-of-concept purposes, the rapidly increasing number of switches would blow-up the area of the PCB and the needed control lines from the

FPGA. In addition to that, the ADG1221 switches used for this work are relatively expensive. When integrating everything on-chip, the architecture can be compacted, but is still not ideal for larger networks. While the architecture used here is a viable solution for small networks, a simpler solution has to be found for larger ANN implementations.

If the network were expanded to implement a multilayer TLG as seen in Figure 4.12, there are some minor modifications that need to be done. The output of each neuron is a signal in the range of 0-5V. If this signal is passed to other synapses in a second stage, the same signal conditioning as for the network input has to be performed (i.e., scaling and shifting the 0-5V signal to +/-20mV). With that, the single layer developed in this chapter can simply be combined to form a multilayer network. This modification is sufficient as long as no learning algorithm is used. If a learning algorithm like MRII were to be added on to a multilayer network,

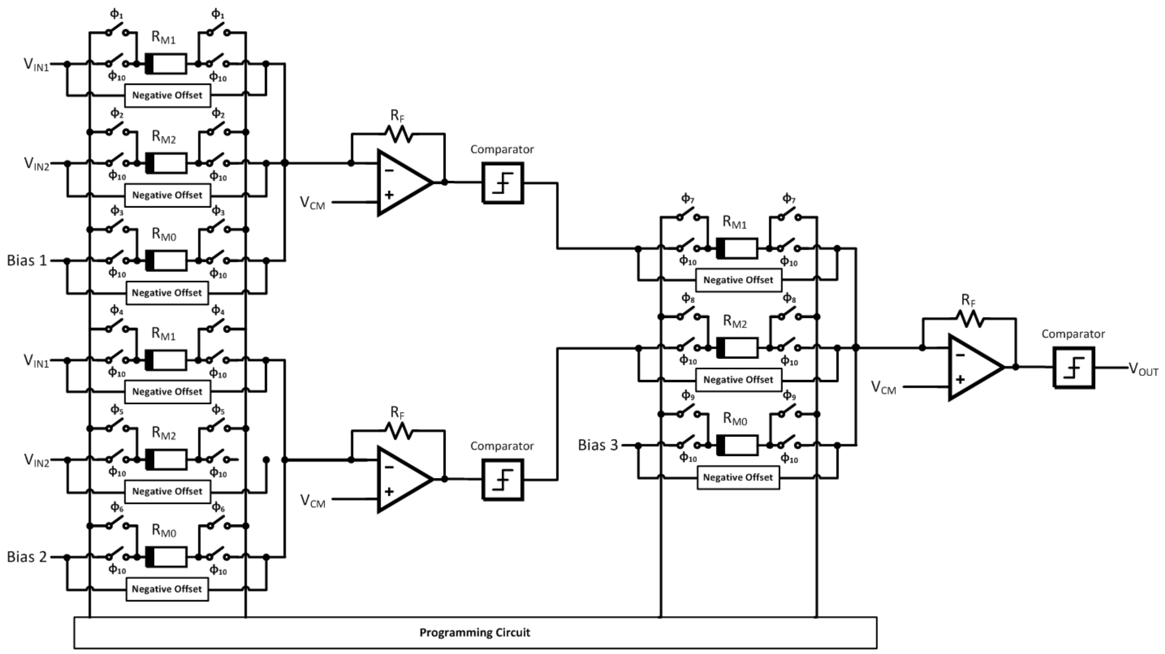


Figure 4.12: Network from Figure 4.10 expanded to a multilayer network

the algorithm would need access to the signal between the summing stage and the comparator of each neuron. This is needed for the algorithm to establish the level of confidence of each neuron, which is used to decide which synapses need to be disturbed (see Section 2.1.6). For a single layer, single neuron network like the TLG presented here, the tapping into the neuron is not necessary as the algorithm would always disturb the synapses of the single neuron and would not have to distinguish between others.

CHAPTER 5

RESULTS

In Chapter 4, a fully reconfigurable TLG was built using the memristor devices from Chapter 2 and the programming circuit developed in Chapter 3, meeting objectives 1, 2, and 3 (see Section 1.4). In order to meet the remaining objectives, memristors need to be characterized, TLG performance determined, and training algorithms evaluated. This is necessary to determine whether it is feasible to use memristors as synapses in TLGs or ANNs in general. This chapter presents the results and findings of the programming circuit, memristor characterization, TLG performance, and training algorithm evaluation.

5.1 Circuit Characterization

It is best to first characterize the circuitry described in Chapters 3 and 4 to make sure that everything is working the way it is supposed to. Furthermore, it is important that the programming circuit is calibrated such that when a 1.5V pulse is needed, a 1.5V pulse is indeed produced by the pulse generator. Figure 5.1 shows the calibration curve for the programmer's output. The DAC is controlled by an incremental value between 0 and 16383 (14-bit) where 8191 is absolute zero. Incremental values above 8191 will generate a positive voltage and values below 8191 generate a negative voltage. As Figure 5.1 shows, the amplitude peaks at roughly 3.2V for both the negative and positive swing. This is a limitation of the AD8055 preamplifier opamp in the programmer's amplification stage. Using a least-squares approximation of the measured output voltage values, a voltage/increment calibration value can be computed, which is then used in the software to calculate the correct incremental value for the DAC.

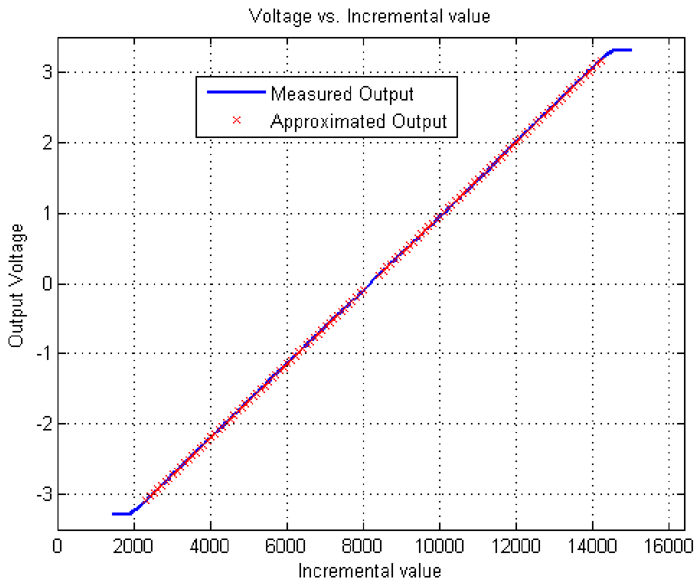


Figure 5.1: Programmer calibration plot

Another important parameter is the pulse length. Besides the incremental value for the DAC, the pulse generator also requires an integer value for the pulse length, so that the appropriate pulse can be formed and sent to the DAC. The pulse length is also an incremental number, indicating how many clock cycles the pulse should last. At a frequency of 50MHz, the period of a clock cycle is 20ns, which is the minimal pulse width. Table 5.1 summarizes the calibration values.

Section 3.2.2 mentioned glitches and opamp oscillations propagating through the circuit. While these glitches are expected from the switches and are not a big issue

Table 5.1: Calibration values

Parameter	**Value**
Voltage/Increment	0.00052489V/Inc
Pulse length/Increment	20ns/Inc

for the circuit itself, it poses a problem for the memristors as a high enough glitch could cause a memristor's state to change. As described in Section 3.2.3, the existing switches were replaced with the AD1221 analog switch, which has very low charge injection. Using the AD1221 and applying a better switching scheme, which makes sure that the memristors are isolated from the programmer whenever the PGA is reconfigured, the glitches were found to be lower than the reading pulse amplitude.

Cross-talk is another concern with respect to signal and memristor device integrity. If a programming pulse propagates to another trace because of cross-talk, a memristor could potentially be programmed even if it was not selected for programming. Figure 5.2 shows the cross-talk between the three memristors's signal lines. Note that a programming pulse of maximal amplitude is applied to memristor 2 (signal 2 @ 2V/div). Both memristor 1 and 3 (signals 1 and 3 @ 20mV/div) show the same signal due to cross-talk, but have a very small peak amplitude of 40mV. Since the cross-talk does not exceed the reading voltage in the worst-case scenario, it is within the allowed boundaries and therefore of no concern.

5.2 Memristor Characterization

This section elaborates on the memristor characterization results that will help to improve or adjust existing programming techniques such that the ANN can quickly and accurately be reprogrammed. At this point the reader should refer back to Sec-

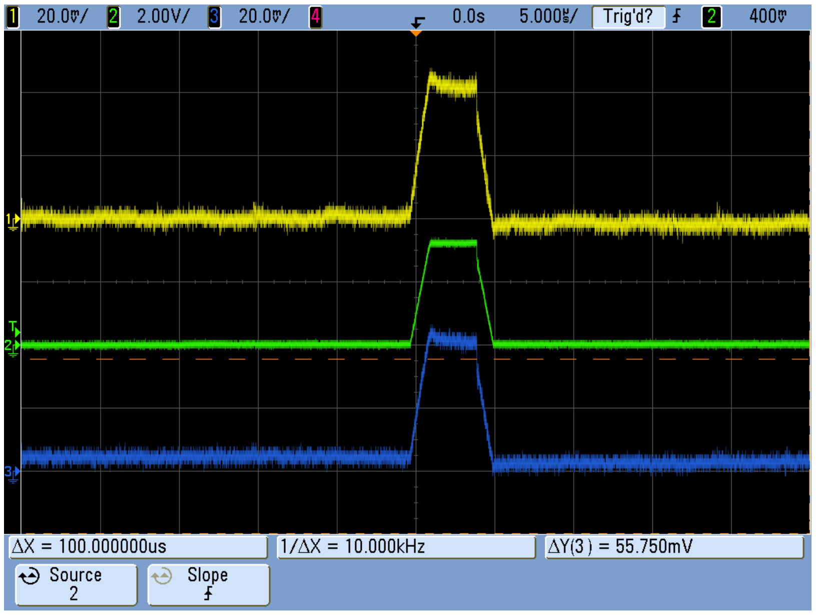


Figure 5.2: Cross-talk

tion 2.2.4, which lists some general rules that need to be observed when programming memristors. Experiments were conducted on two different device types: One using silver, the other copper, as the conduction material. Unless otherwise stated, all characterization experiments were conducted using the programming circuit developed in Chapter 3.

5.2.1 Threshold

As stated in Section 2.2.3, a memristor's state can only be changed if the applied voltage reaches a threshold. While this is true for a change in the device's conductance, the conduction material (silver or copper) can still be moved into the amorphous medium (see Figure 2.8 and Section 2.2.4) at voltages below the threshold. It is therefore important that no unnecessary pulses below the threshold voltage are applied to the memristor. However, the threshold of a device can change during its

Table 5.2: DC Threshold levels

	Silver	**Copper**
Writing	0.23V	1.2V
Erasing	-0.58V	-1.5V

lifetime and was found to exhibit significant fluctuations between devices. In general, the writing and erasing thresholds for silver devices were found to be much lower than for the copper devices. Table 5.2 shows the threshold voltages that were found to work best for the devices under study. Note that these values are only valid for DC operation (slow pulses) and are not representative of the AC behavior of these devices. Figure 5.3 shows erasing and writing pulses applied to a silver-based device. The device is initially in low resistance state. Applying a low erasing pulse does not change its state, unless the erasing threshold is reached. At 15μs, the device switches to a higher resistance because the erasing threshold listed in Table 5.2 is exceeded. The same is true for programming the device back to a lower resistance at 25μs. The three programming pulses between 30 and 40μs are bad examples as they violate one of the programming rules. If a larger ΔR is observed when writing, no more subsequent writing pulses should be applied as it could potentially harm the device.

Note that the resistance in Figure 5.3 was only slightly reduced by the first pulse at 25μs, indicating that the device has been sufficiently written. At 50μs the amplitude needed to erase the device is significantly higher than previously reported. In fact, the erasing pulse at 45μs did not alter the device's state at all. This indicates that the threshold levels can change while programming the device. Programming pulses applied between 55 and 80μs show that the threshold levels for both erasing and writing decreased, causing irregular changes in device state.

The erratic behavior of the device seen in Figure 5.3 could have been due to

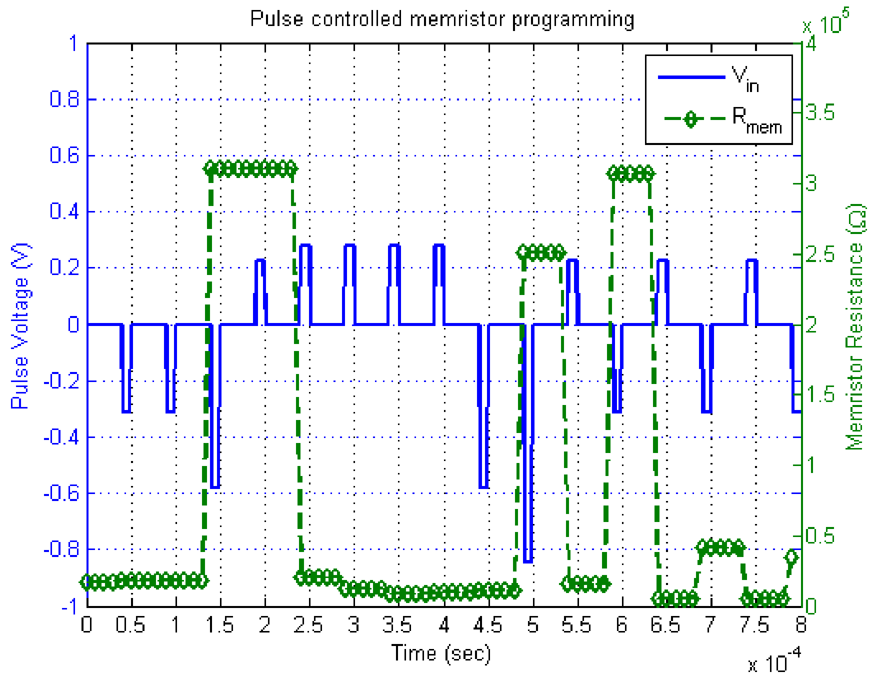


Figure 5.3: Moving threshold

damage caused by applying subsequent writing pulses. However, the device used for this experiment performed well for many more cycles, indicating that its integrity was not decreased. In fact, the "moving threshold" phenomenon was observed in all devices under study (both silver and copper). The threshold levels listed in Table 5.2 were found to generally hold true, but exhibited a deviation of up to 50% over a memristor's lifetime.

5.2.2 Varying the Device State

In her work, Tran has shown that a pulse train of writing pulses continues to decrease a memristor's resistance, while an erasing pulse train continually increased its resistance [32]. Her work was based on an ideal simulated model, which does not have to follow the memristor programming rules as outlined in Section 2.2.4. When dealing with physical devices, the reality often looks significantly different compared to a simulated environment. This is certainly the case here as well. Figure 5.3 indicates that the change of resistance is not simply a function of the applied pulse. Experiments on both silver and copper devices confirmed this indication and it was found that continuously increasing or decreasing the resistance is extremely difficult to achieve. This is especially true for continuous writing, as the programming rules prohibit us from continually sending writing pulses. It is therefore better to write the device and continually erase it. The resistance change was not found to be related to the previous state and the applied pulse. In most cases the devices would perform hard-switching (i.e., jump from a low resistance to a high resistance and vice-versa). While this is not ideal for ANNs, we shall briefly investigate the hard-switching behavior.

Figure 5.4 shows the result of applying a pulse train of writing and erasing pulses with pulse amplitudes that are only slightly above the silver threshold levels (see Table 5.2) to a silver device. As can be seen, the resistance jumps up and down by at least two orders of magnitude. Note that the deviation from the average high and low values is pretty significant. The standard deviation was found to be around 45% for both high and low states. While this is a pretty significant error, the memristors can still be used as a digital memory when toggling between the two states (high and low). The average high state resistance was found to be at 392kΩ and the average low state resistance at 7.3kΩ.

Due to their intrinsic device structure and materials characteristics, silver-based devices are expected to exhibit a certain hard-switching behavior, while copper-based devices should work better with intermediate resistance values [5]. However, a similar hard-switching behavior was found when investigating copper-based devices. The reason for this discrepancy could be due to the fact that the programmer used for

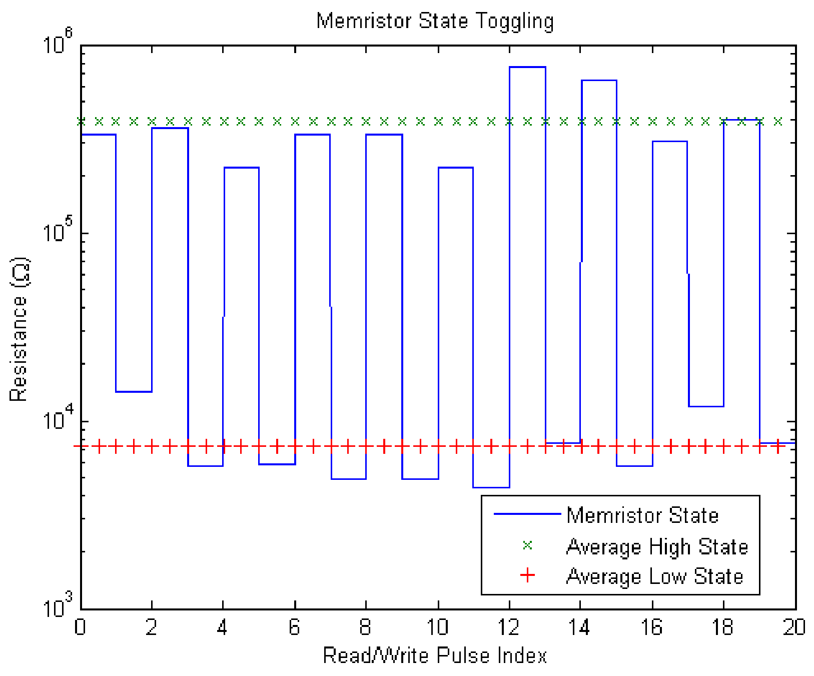


Figure 5.4: Toggling between memristor states

this work is purely voltage controlled. While the threshold voltage suggests that the programmer should be voltage controlled, it is possible that the device should be programmed in current mode once the threshold voltage is reached. In fact, literature suggests that memristors are both a voltage and current controlled device [26, 30], but no solutions are presented as to how a memristor is best controlled in a practical application such as an ANN programming circuit.

When looking at Figure 5.4, it seems as though the deviation of the high and low states is random and does not suggest that a relationship between previous state (resistance) and the change in resistance exists. However, Figure 5.4 is deceiving and the change in resistance was analyzed numerically. It was found that a relationship between the change in resistance and the previous resistance does exist, but only when writing. Figure 5.5 shows data-points from selected datasets that exhibited nice hard-switching behavior of both silver- and copper-based devices. When writing the devices to a lower resistance, the change in resistance is dependent on the previous state, but is independent of the pulse amplitude (the pulse amplitudes used here were all the same). The resistance change can be said to be proportional to $-1/R^2$, where R is the resistance of the current state (before applying the pulse). This is consistent with literature investigating spike-timing-dependent-plasticity (STDP), which governs how synapses learn in biology [37]. What is interesting about the relationship between resistance change and previous state is that it only holds for programming (i.e., reducing the resistance). When erasing, there seems to be no apparent pattern between the change and previous state. Given that the relationship is proportional to $-1/R^2$, it makes sense that this only applies when writing a memristor to a lower state, as the change in resistance will always be a negative number, indicating a reduction in resistance. The copper-based devices seem to be a little more scattered than the

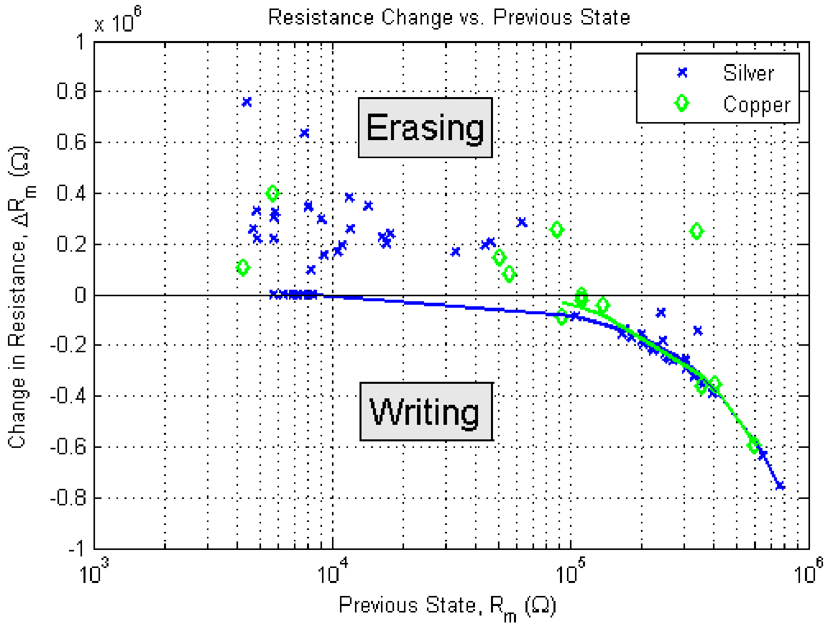


Figure 5.5: Relationship between resistance change and previous state

silver-based devices, whose data-points are much closer to each other, but overall the two device types agree very well with each other.

Another observation was made that could explain why it is so hard to accurately program memristors. Figure 5.6 shows the output of the programming opamp during programming (top pulse, yellow). The figure depicts that the $10\mu s$ programming pulse starts writing the device as soon as the threshold level is reached. The writing process lasts for about $7\mu s$ before the memristor settles in at the new resistance. What is interesting about the writing process is the stepwise change in voltage and therefore the current through the device. Jo et al. observed the same behavior and linked it to the formation of individual filaments in the amorphous medium [19]. The filaments actually already exist, but they are filled with silver during the writing process. Characterizing the average wait time until the first or subsequent transitions in voltage steps occur leads to a much better understanding of the device's switching

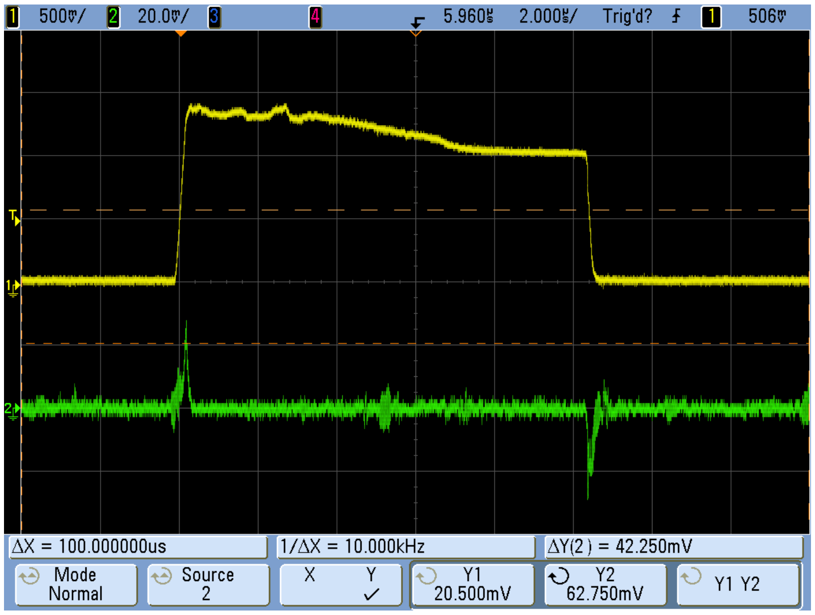


Figure 5.6: Programming pulse and its effect on the memristor

behavior and its relationship to the threshold voltage [19]. Since the memristor devices used for this work exhibited the same behavior as in the study by Jo et al., it is worth investigating this matter further.

5.2.3 State Drift

Since memristors are ion-conduction devices, it is expected that the resistance of a particular device could drift off over time due to unintentional ion migration. This is especially true after very short programming pulses, as the ion migration does not completely finish and some of the metal ions move back to the top electrode [5]. As Section 5.2.4 shows, very short erasing pulses can actually damage the device. The applied programming pulse therefore has to be long enough such that the metal ions are successfully moved into (or removed from) the amorphous medium. For silver-based devices, the ideal programming pulse width is about 5μs and about 10 times higher or 50μs for copper-based devices [5]. Theory suggests that the longer the pulse, the better the state will persist and not drift off [5].

Both silver- and copper-based devices were investigated for state drift. The silver-based devices did not show any drift in resistance at all. Even when the pulse length was reduced from 5μs to 1μs, the resistance of the device did not drift off. However, copper-based devices showed significant drift. Figure 5.7 shows that after applying a writing or erasing pulse the state drifted up or down respectively. Note that the experiment leading to the plot in Figure 5.7 was conducted by applying either a writing or erasing pulse and then reading the resistance approximately every 10 seconds until the resistance stayed constant. The index on the plot's x-axis therefore shows a multiple of 10 second intervals. The experiment performed on this particular device showed that the resistance tends to drift back to its previous state (i.e., when

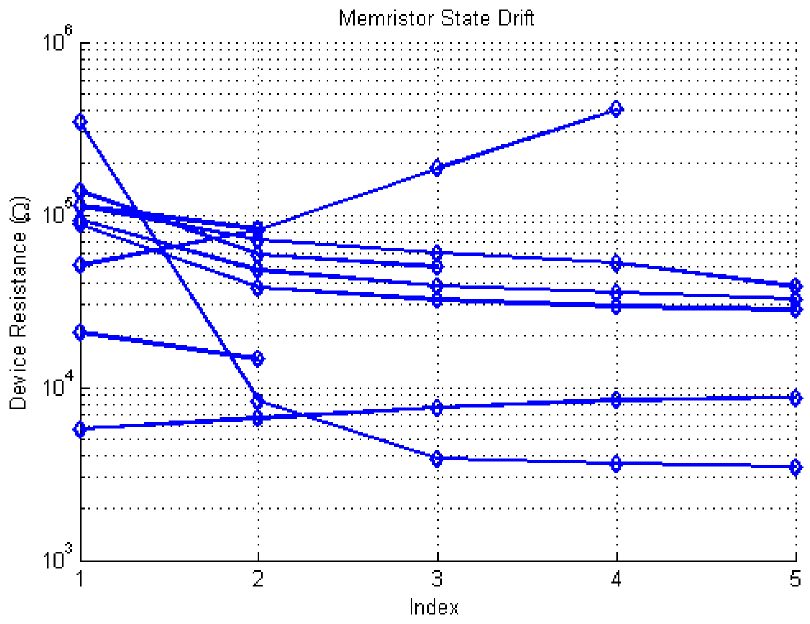


Figure 5.7: Copper-based memristor state drift

an erasing pulse is applied, the resistance is increased initially, but then decreases back towards its previous value).

5.2.4 Negative Differential Resistance

From physics we know that resistance is always a positive value and that physical devices cannot exhibit negative resistance. However, this is not completely true. Tunnel diodes often exhibit the so called “negative differential resistance” (NDR) phenomenon. When looking at the I-V curve, at some point the current starts to decrease even though the voltage continues to increase. Continuing on, the current will eventually increase and follow the expected behavior. Figure 5.8 shows a typical NDR curve.

It turns out that memristors can exhibit NDR behavior as well. This is only the case if the device is damaged by applying an extremely short, but high erasing

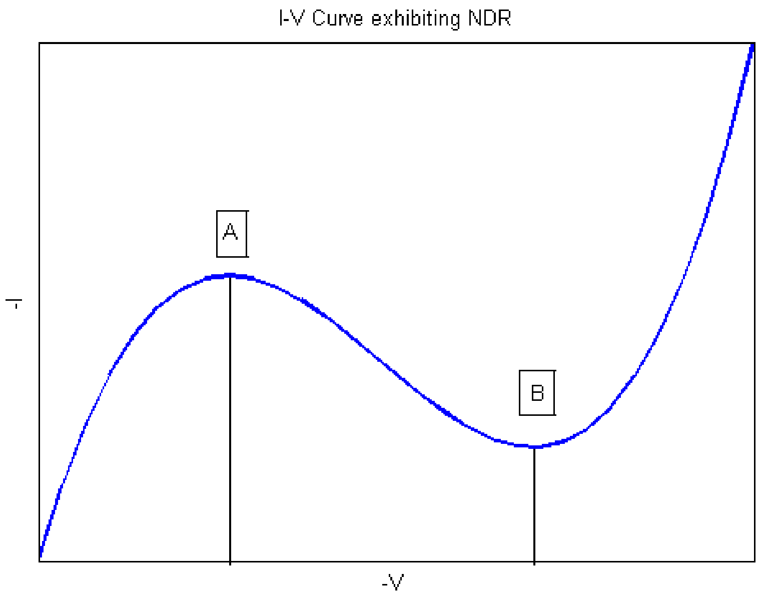


Figure 5.8: Typical NDR curve

pulse [5]. Figure 5.8 shows the two vertices of the curve (A and B). They govern the area where the device exhibits NDR behavior. If the voltage applied is between 0V and A, then no change to the device's resistance is made. In other words, A can be thought of as the erasing threshold value. The region between A and B is where the device exhibits NDR. When an erasing pulse is applied, which falls between A and B, the device's resistance is decreased instead of increased (writing). Once the pulse amplitude passes B, the device erases normally [5]. Figure 5.9 shows the NDR behavior of a silver-based device when pulses with different amplitudes are applied. At 15μs, the erasing pulse causes the device to reduce its resistance just like the previous writing pulse. The device does not erase until the pulse amplitude exceeds point B as seen at 40μs. The same behavior can be observed between 45μs and 55μs.

NDR could be useful mechanism to program memristors, as it seems that the

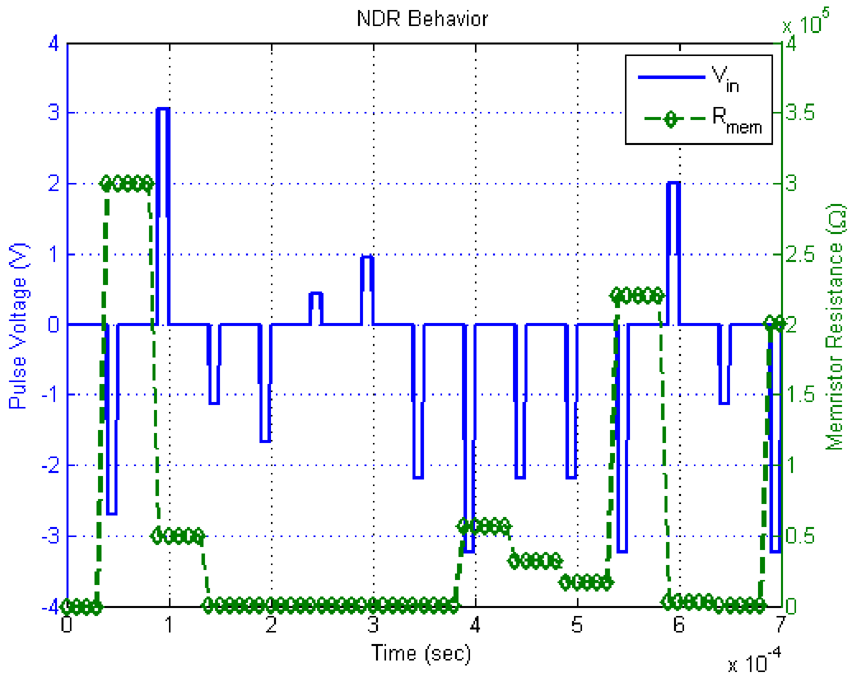


Figure 5.9: Memristor NDR behavior

device's resistance can be stepped down much better than when applying a single writing pulse. It is therefore worth investigating whether NDR is a viable mechanism for accurate memristor programming. However, as mentioned before, NDR is an artifact of device damage and needs to be studied and understood better, before it is used as a programming mechanism. The other problem is that points A and B move up and down depending on the device's state and the applied pulses [5]. This means that the threshold voltages change, requiring the programming algorithm to be re-tuned to each device on a regular basis. The phenomenon of moving thresholds is consistent with the observations made in Section 5.2.1.

5.3 Evaluation of Programming Algorithms

Section 2.1.6 already alluded that the widely used backpropagation algorithm is not useful here because a hard-thresholding activation function is used for this work. Instead, the MRII training algorithm is a viable alternative and was shown to perform very well in simulation [32]. The majority of training algorithms are designed for computer simulations where a weight is simply a memory location that can be updated by the training algorithm. When dealing with programmable weights such as memristors, the memory updating process has to be replaced by a programming algorithm, making sure that the synapse or weight is set to the correct value. Tran used a simple P-type programming algorithm, which applies a programming pulse proportional to the error ($V_{prog} = K_p$*($R_{m,desired}$ - $R_{m,current}$)). This works for an idealized model that does not change its characteristics over its lifetime. However, the content of Section 5.2 indicated that the physical memristor devices do not follow the simulation model and do, in fact, change their characteristics. This is for the most part because not enough information about the characteristics of these devices is known and no ideal programming mechanism has been discovered yet. Since no consistent behavior was found thus far, it is not yet possible to develop a reliable programming algorithm. This is mostly due to the fact that after each programming pulse the devices changed their characteristic in an unpredictable manner. More characterization research on both device level as well as programming circuitry has to be done to proceed with the development of a programming algorithm suitable for use with physical devices.

When looking at Figure 5.4, we see that silver-based devices work reasonably well when they are toggled between high and low states. Even with an error of up to

50%, a repeated writing/erasing pulse train will eventually program the device to the desired high or low state. Therefore, a simple algorithm applying a repeated writing/erasing pulse train can be used for look-up table programming and even simple binary learning.

5.4 TLG Results

The main objective of this work was to build a memristor-based neuromorphic computing application. More specifically, a memristor-based TLG was developed in Chapter 4. An FPGA controlled programming circuit, developed in Chapter 3, was then used to program the memristors such that the TLG would exhibit different logic functions (AND, OR, NAND, NOR). Its results are presented in this section.

Section 5.2 mentions that the memristor devices essentially work, but there are a lot of things that are not yet well understood and therefore compromise full physical ANN realization. The issues with moving threshold, erratic behavior when erasing, and low device lifetime (due to the programming circuit's inability to recover devices in very high resistance state) did not allow a conclusive study on the TLG to be conducted.

The performance of the TLG was measured by the four basic logic operations (AND, OR, NAND, NOR), but there are a total of 16 2-input logic operations possible. The invalid cases mentioned in this section are not necessarily invalid logic operations as they might be part of the rest of the 16 functions. In Figure 5.10 for example, the TLG outputs show TRUE, NOT(Input1), and NOT(Input2) functionality among others. This clearly shows that the TLG can be reconfigured to many more possible functions. For simplicity's sake, only AND, OR, NAND, and NOR were used for this work. Table 5.3 shows all 16 possible logic operations.

Table 5.3: 16 possible binary logic operations

INPUTS	A (Input 1)	0	1	0	1
	B (Input 2)	0	0	1	1
OUTPUTS	FALSE	0	0	0	0
	A AND B	0	0	0	1
	B DOESN'T IMPLY A	0	0	1	0
	TRUE B	0	0	1	1
	B IS NOT IMPLIED BY A	0	1	0	0
	TRUE A	0	1	0	1
	A XOR B	0	1	1	0
	A OR B	0	1	1	1
	A NOR B	1	0	0	0
	A XNOR B	1	0	0	1
	NOT A	1	0	1	0
	B IS IMPLIED BY A	1	0	1	1
	NOT B	1	1	0	0
	B IMPLIES A	1	1	0	1
	A NAND B	1	1	1	0
	TRUE	1	1	1	1

The results of the memristor characterization showed that it was not possible to alter the device resistance in fine steps like Tran showed with her simulated TLG [32]. Instead, Figure 5.4 showed that the silver-based devices can be used to switch back and forth between a “High” and “Low” state. This hard-switching behavior is used to reconfigure the TLG to either an AND, OR, NAND, or NOR gate. As reported in Section 5.2.2, the average high state resistance was around 392kΩ and the average low state resistance at 7.3kΩ. Before any feed-forward operation can be done, the synapses have to be configured for this resistance range such that the appropriate gains can be realized. Referring back to Section 4.4 and Equations 4.5 and 4.6, new values for R_n and R_f can be computed. However, since the average high and low state resistances are likely to change from memristor to memristor, the general synapse based on the second curve of Figure 4.7 was implemented ($R_n = 33.3k\Omega$, $R_f = 500k\Omega$).

Table 5.4 shows the "logic" memristor resistance states needed to produce a particular pattern. This is based on the high and low values seen in Figure 5.4 applied to the weight cube in Figure 4.6. In short, using high and low states, the memristors are programmed such that the network operates only in the four corners of the weight cube where a valid pattern is located. Due to the high error of the high and low states (45%, see Section 5.2.2), the programmed resistance may cause the TLG's operating point to lie outside of the targeted tetrahedron.

Table 5.4: TLG logic configuration states

Configuration	**Memristor 1**	**Memristor 2**	**Memristor 3**
AND	High	High	Low
OR	High	High	High
NAND	Low	Low	High
NOR	Low	Low	Low

Based on Table 5.4, the operator or control algorithm knows which memristor to write or erase to achieve the desire logic operation. Once the programming (writing or erasing) is done and feed-forward does not produce satisfactory results, each memristor is simply toggled to attain a new resistance value. This process is repeated until a correct output is obtained. Essentially, the devices are perturbed using two fixed pulse amplitudes for writing and erasing. This is reminiscent of the MRII algorithm, which also perturbs the devices in a random way. However, here we restrict the perturbation to only two pulses, simplifying the programming process to a simple toggling action.

Due to the irregular device behavior and the low device lifetime, the programming of the three memristor devices used for the TLG was done manually through the

Nios II processor's RS232 data link to a host computer. Figure 5.10 shows the result of reconfiguring the TLG. The first two plots show the two input signals. Note that their signal levels are not logic 1 or 0, but -1 and 1 (-20mV and 20mV) instead (see Section 2.1.4). The third, fourth, and fifth plot shows the output of the TLG. The two input signals were continuously applied to the TLG to obtain the three output plots. During the programming phase, the input signals are kept at 0V. Figure 5.10 clearly shows that the TLG was reconfigured to exhibit AND, OR, NAND, and NOR functionality.

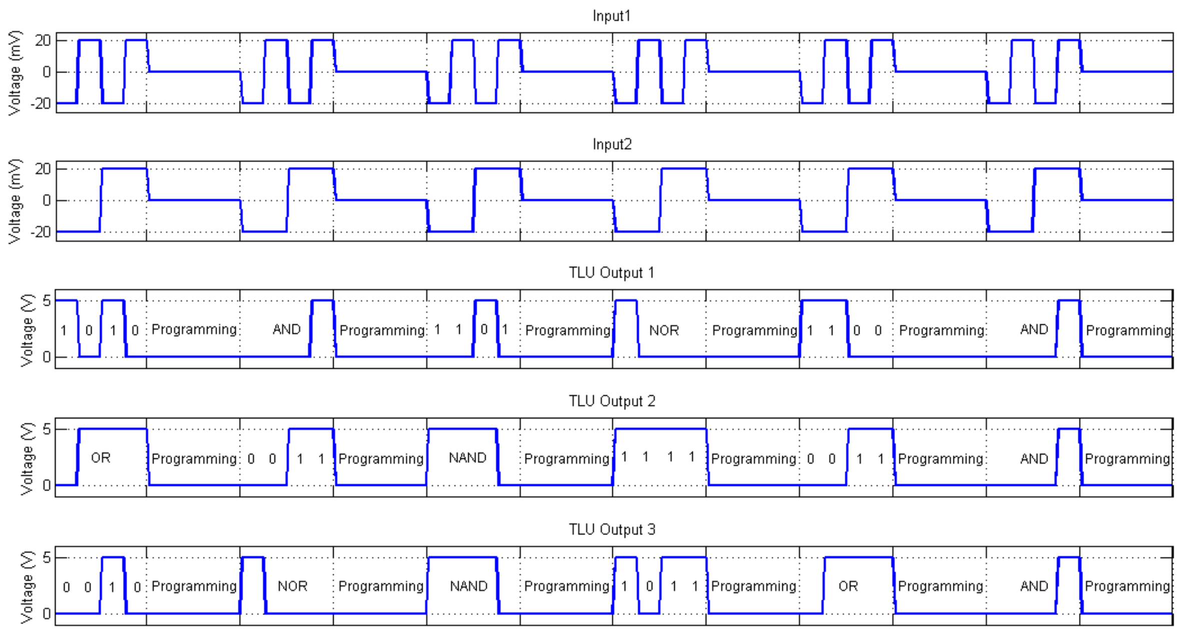


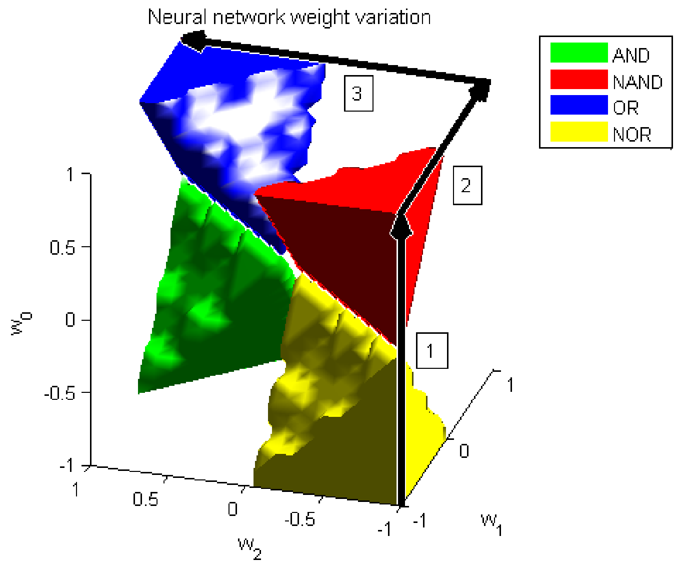
Figure 5.10: TLG inputs and output

The programming sections in Figure 5.10 indicate where a single memristor was programmed according to Table 5.4. As Table 5.4 shows, going from an AND to a NOR configuration requires two memristors to change (memristor 1 and 2 need to change from high to low; memristor 3 stays). Therefore, it required two programming cycles - one for each memristor - to reconfigure the TLG. In order to reconfigure the TLG from an AND to an OR or a NAND to a NOR configuration, only one programming cycle is required, as the bias weight (memristor 3) is the only one that needs to be changed.

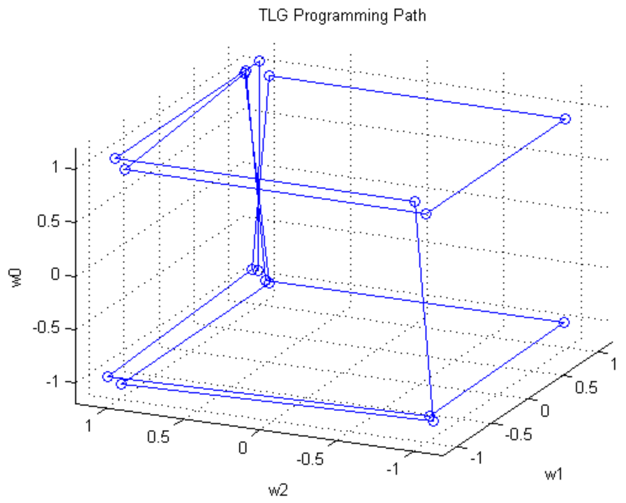
Figure 5.11a shows the weight space with a possible programming path for reconfiguring the TLG from NOR to OR. In this case, it will take a total of three programming cycles as all three memristors need to be changed. Note that changing only the bias weight (memristor 3) changes the network from NOR to NAND, but two memristors (memristor 1 and 2) need to be changed to go from NAND to OR. As Figure 5.11a shows, the path from NAND to OR leads through a corner, which is not one of the four functions used for this work. Figure 5.11b shows the programming paths required to obtain the results in Figure 5.10. As can be seen, the TLG is reconfigured to operate only in the corner regions. Four of the eight corners are valid configurations (AND, OR, NAND, NOR), while the rest are unused configurations and are simply used as way-points to reconfigure the TLG from one function to another. Table 5.5 shows a set of memristor resistances for each gate configuration. Note that these resistances agree with the high/low notation of Table 5.4 considering that a resistance near $10^3\Omega$ is interpreted as logic low and a resistance near $10^5\Omega$ as logic high.

Table 5.5: TLG memristor resistances

Configuration	**Memristor 1 (kΩ)**	**Memristor 2 (kΩ)**	**Memristor 3 (kΩ)**
AND	80	118	17
OR	73	62	133
NAND	8.2	9.8	114
NOR	5.8	5.5	19



(a)



(b)

Figure 5.11: (a) Weight cube indicating the programming path required to go from NOR to OR and (b) all programming paths required to obtain the results seen in Figure 5.10.

It is worth mentioning that it took an average of 2 programming pulses per programming cycle (see Figure 5.10) to program a device to the desired state. The standard deviation was found to be 1 programming pulse and the highest and lowest number of programming pulses needed was 4 and 1, respectively. This is just slightly higher than expected when looking at the results in Figure 5.4, which were obtained by using single pulses to switch the device from one state to the other. As Figure 5.11b shows, the memristors are programmed such that the network always operates in the corner regions. If a memristor's resistance is adjusted such that the network is not operating in one of the four valid corners (AND, OR, NAND, NOR) and most importantly is not within the tetrahedra depicted in Figure 5.11a, the network's configuration is invalid. Therefore, additional programming pulses are needed to move the network's operation point into the desired tetrahedron. On the device level, this issue arises because each device has a slightly different behavior and might not react the same way to the programming pulses. Therefore, it is expected that some devices need more pulses or even need characterization during programming to determine where the new threshold values are. A better understanding of the intrinsic memristor programming mechanisms and more research on hardware friendly algorithms are needed to automate network reconfiguration. The results shown in Figures 5.10, 5.11b, and Table 5.5 were obtained using only copper-based memristors and all experiments were run on a series of devices and proved to be consistent and reproducible.

Figure 5.12 agrees with the previously mentioned average of 2 programming pulses needed per programming cycle. However, when looking at the normalized histogram, it seems like the state of the memristor is not that hard to control as was previously mentioned. The data presented here suggests that about 33% of programming attempts succeed with only a single programming pulse. The curve seems to fall of exponentially with only 2% of all programming attempts needing 7 programming pulses. This data is somewhat missleading and requires more explanation. First

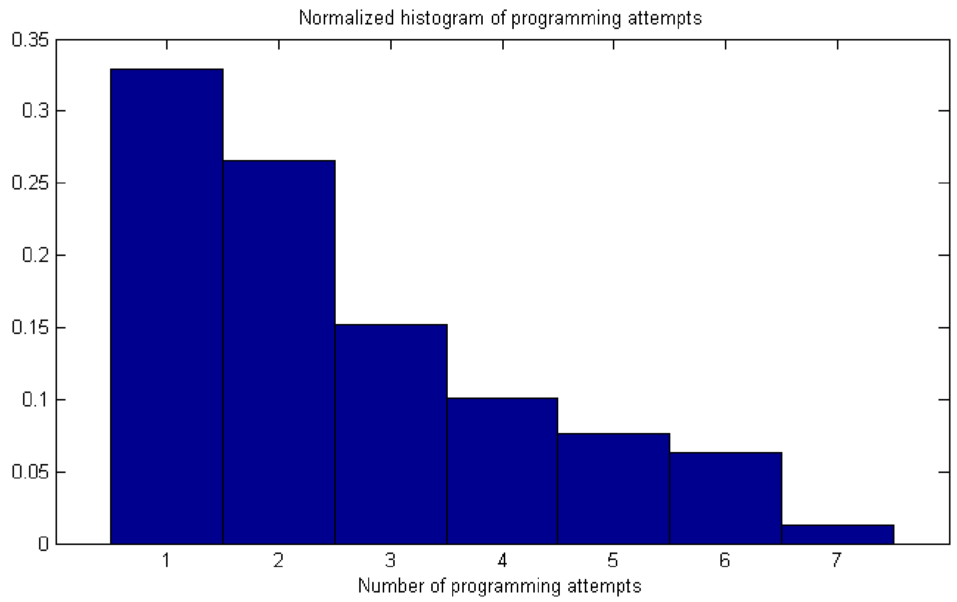


Figure 5.12: Histogram of memristor programming attempts needed

of all, this work is based on binary states, meaning that the devices can easily be toggled by using pulses big enough to cause the device's state to move well over the binary threshold. This cannot be compared to Tran's work, as her simulation was focused on varying the state in very small increments. Furthermore, all that needs to be accomplished during programming is to cross the binary threshold. It does not matter what the exact resistance of the memristor is. Lastly, a complete programming cycle as described here, does not imply a successful reconfiguration of the TLG. A complete programming cycle is simply the completion of setting a particular device to a "High" or "Low" state. If the feed-forward operation does not indicate a successful reconfiguration of the TLG, the memristor is simply perturbed (toggled) as explained earlier.

Since the result of the TLG is a function of the resistances of all three memristors, it is possible that their values cause one synapse to offset another, therefore producing

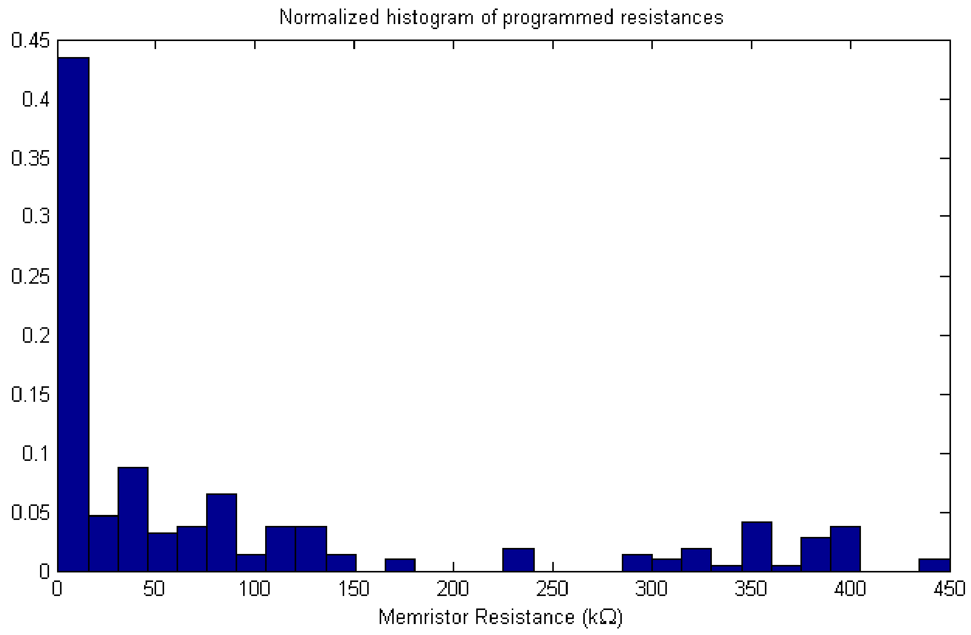


Figure 5.13: Histogram of attained memristor resistances

an incorrect outcome, even though all memristors are in their targeted logic state. It would therefore be desirable that the memristors would have a much narrower window of target resistance pertaining to each logic state, to decrease the number of incorrect programming cycles. Unfortunately, this is not possible due to the fact that memristors are hard to accurately control. Figure 5.13 shows the normalized histogram of all resistances attained during programming. It can be seen that the histogram is widely spread, indicating that the windows of the two “High” and “Low” target resistances are quite wide.

What is interesting about the histogram in Figure 5.13 is that almost 50% of the attained resistances were below 20kΩ. In fact, the majority of the attained resistances are in the lower resistance range. This is because the memristors used here are easier to write than to erase. In addition to that, when a writing pulse was applied, causing the device's resistance to decrease, the momentarily decreased resistance caused an increased current flow, which in turn caused the resistance to decrease even more. This avalanche efect is the reason that almost 50% of attained resistances are below 20kΩ. Dividing the histogram in Figure 5.13 into "High" and "Low" regions at around 200kΩ, it seems like the lower resistance region has more discernible distribution pattern than the higher resistance region. In fact, this corresponds to the relationship between resistance change and previous state as seen in Figure 5.5.

CHAPTER 6

CONCLUSION

This thesis revolved around building a memristor-based neuromorphic application in order to show the viability of using memristors as synapses in ANNs. This chapter summarizes this work and concludes with a list of recommendations for further improvement.

6.1 Summary of Work

In Chapter 4, a fully hardware-based ANN in the form of a TLG was developed. Even though that this has already been done by Rosenblatt, Widrow, and others as early as 1960 [24, 33], there are many advances that set this work apart from what has previously been done. Both Rosenblatt and Widrow built a fully reconfigurable hardware ANN, but their setup was very large in size and slow to adapt. Their synapses were based on a three terminal device; Rosenblatt used a variable resistor powered by a stepper motor [24] and Widrow used a resistive metal strip whose resistance could be altered by electro-deposition [33]. Clearly their three terminal synapses were not only large in size, but required a lot of overhead resources. The work presented here used two terminal memristive devices as synapses, which not only simplifies the setup but most importantly significantly reduces its size. It needs to be pointed out that the prototype built for this work was based on discrete circuit components, making the final network several thousand times bigger than if it were built on-chip. The findings presented in Chapter 5 prompt further improvements of the circuit architecture so that it can later be implemented in CMOS technology.

The other novelty about this work is the use of memristors in an application-based environment. Other research groups have reported on the characterization and use of memristors, but their work was mostly done in a protective lab environment using elaborate lab equipment such as SPAs [17, 18, 19, 26, 30]. The work presented here is application based and characterized, programmed, and reprogrammed memristors while they were physically part of the ANN. A simple pulse programming circuit was developed in Chapter 3 that allows the application to control the memristors. While Chapter 5 mentions some issues and shortcomings of both the application (ANN and programmer) as well as the memristors, it clearly shows the viability of using memristors in neuromorphic applications and indicates where further improvements need to be made in order to advance this technology.

6.2 Conclusion and Future Work

The objectives of this work are stated in Section 1.4 and are repeated here for easier referencing in this section.

1. Develop memristor programming and reading circuit.

2. Develop synapse and neuron circuit.

3. Build a fully hardware implemented TLG.

4. Evaluate and characterize memristors for use in a synapse.

5. Find the effect of the training environment on training algorithms.

Objectives 1, 2, and 3 were the focus of Chapters 3 and 4 and were implemented and demonstrated to work in Section 5.4. While these three objectives were successfully met, some improvements are necessary for further advance of this technology. The programming circuit needs to be able to cover a wider range of memristor resistance. While the resistance range of the memristors was restricted from 1kΩ to 100kΩ, it has been shown in Section 5.2.1 that the memristors are not only hard to control, but they also exceed this range when toggled between high and low states. Exceeding this range can be accounted for by adjusting R_n and R_f in the synapse-neuron circuit, but it will cause resistance measurement error as the PGA's range is exceeded. The programming opamp used for this work does not support high feedback resistances while supporting both high-speed and gains up to 10. In order to expand the PGAs range to accommodate the larger resistance variation of memristors, either a custom opamp is needed or a different programming circuit needs to be developed.

The synapse and neuron circuit developed in Chapter 4 proved to work as expected. However, Tran reported that the nonlinearity of the synapse circuit can lead to programming issues [32]. For "binary" state toggling, some memristors might operate in the steeper region of the synapse's gain curve, while others might operate in the flatter region. This is due to the characteristic variation among memristors. Being able to control memristors more accurately could alleviate this issue, but nonetheless the nonlinearity of the synapse is an undesired behavior. Several options such as the bridge-based synapse or the use of logarithmic opamps should be explored in order to linearize the synapse's gain curve. This is perhaps easier to do when building the network on-chip, as it allows for more flexibility with custom designed transistors and opamps.

The fully hardware implemented TLG as listed in objective 3 has been demonstrated to work as expected (see Figure 5.10). While this is only a prototype for a proof-of-concept neuromorphic application, more research in the area of network topologies and programming overhead architectures needs to be conducted. As mentioned in Section 4.7, the network built here does not scale for larger, more complex architectures.

The majority of Chapter 5 outlines the memristor characterization results, meeting objective 4. Even though the results indicate that the memristor devices used for this work were hard to control in terms of accurately adjusting the device resistance, it showed that they do exhibit some relationships that can be exploited for better device handling. Furthermore, the results indicate the areas where further research is needed. This is by no means limited to the device only and should in particular include other programming techniques such as voltage and current controlled programmers as well as spike-timing-dependent-plasticity (STDP).

Last, but not least, the MRII training algorithm was evaluated for use with a hardware ANN. Tran was able to successfully implement a slightly modified version of the MRII algorithm in her Matlab-Cadence co-simulation [32]. As mentioned before, her memristor model was based on idealized device assumptions. Therefore, the simulation was able to successfully employ the learning algorithm in a simulated hardware environment [32]. While the P-type programming algorithm worked well for her work, it would certainly not work with hardware devices that do not exhibit consistent behavior. With the improvement of memristor device behavior and acquisition of a better device understanding, reliable programming techniques can be developed, which would allow for software-based training algorithms to be employed. However, this is not an ideal situation as current learning algorithms are streamlined for software ANNs, meaning that the weight update process is simply done by updating a memory location. In hardware we have to take the weight's characteristic and more importantly its lifetime into account. This should not only be done in the programming algorithm, but also in the learning algorithm itself. For example, MRII falls back to the previous weights if the new weight change does not reduce the error [32, 36]. This is not very hardware friendly as it not only wears the memristors out, but adds unnecessary programming time for the process of going back to the previous weight. Further modification of the MRII algorithm or development of another hardware-friendly learning algorithm is therefore required. Even though the TLG was not equipped with a learning algorithm due to the issues outlined in this chapter, a more hardware-friendly learning algorithm can easily be coupled with the TLG in the future to exhibit learning.

REFERENCES

[1] S.P. Adhikari, Changju Yang, Hyongsuk Kim, and L.O. Chua. Memristor bridge synapse-based neural network and its learning. *Neural Networks and Learning Systems, IEEE Transactions on*, 23(9):1426 –1435, Sept. 2012.

[2] R. Jacob Baker. *CMOS Circuit design, layout, and simulation*. IEEE Press, 3rd edition, 2010.

[3] K. Basterretxea, J.M. Tarela, and I. del Campo. Approximation of sigmoid function and the derivative for hardware implementation of artificial neurons. *Circuits, Devices and Systems, IEEE Proceedings* -, 151(1):18 – 24, Feb. 2004.

[4] Nick Bostrom. Ethical issues in advanced artificial intelligence. In *Science Fiction and Philosophy: From Time Travel to Superintelligence*, pages 277–286. Wiley-Blackwell, 2009.

[5] Kristy A. Campbell. Personal conversation with Dr. Campbell, Sept 2012.

[6] Kristy A. Campbell and Beth R. Cook. Unpublished Data.

[7] Brian Casper. Energy efficient Multi-Gb/s I/O: Circuit and system design techniques. *IEEE Workshop on Microelectronics and Electron Devices*, 2011.

[8] L.O. Chua. Memristor-the missing circuit element. *Circuit Theory, IEEE Transactions on*, 18(5):507 – 519, Sept. 1971.

[9] L.O. Chua. The fourth element. *Proceedings of the IEEE*, 100(6):1920 –1927, June 2012.

[10] L.O. Chua and Sung Mo Kang. Memristive devices and systems. *Proceedings of the IEEE*, 64(2):209 – 223, Feb. 1976.

[11] Kolton Drake and Kristy A. Campbell. Chalcogenide-based memristive device control of a LEGO Mindstorms NXT servo motor. *AIAA Infotech@Aerospace Conference and Exhibit*, March 2011.

[12] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification*. John Wiley & Sons, 2006.

[13] Rafael Gadea, Joaquín Cerdá, Franciso Ballester, and Antonio Mocholí. Artificial neural network implementation on a single FPGA of a pipelined on-line backpropagation. In *Proceedings of the 13th International Symposium on System Synthesis*, pages 225–230, 2000.

[14] Rafael Gadea Girons, Ricardo Colom Palero, Joaqun Cerd Boluda, and Angel Sebastian Corts. FPGA implementation of a pipelined on-line backpropagation. *The Journal of VLSI Signal Processing*, 40:189–213, 2005.

[15] D.O. Hebb. *The Organization of Behavior: A Neuropsychological Theory*. Taylor & Francis, 2002.

[16] J.L. Holt and T.E. Baker. Back propagation simulations using limited precision calculations. In *Neural Networks, 1991, IJCNN-91-Seattle International Joint Conference on*, volume II, pages 121 –126, July 1991.

[17] Sung Hyun Jo, Ting Chang, Idongesit Ebong, Bhavitavya B. Bhadviya, Pinaki Mazunder, and Wei Lu. Nanoscale memristor device as synapse in neuromorphic systems. *Nano Letters*, 10(4):1297 – 1301, 2010.

[18] Sung Hyun Jo, Kuk-Hwan Kim, Ting Chang, S. Gaba, and Wei Lu. Si memristive devices applied to memory and neuromorphic circuits. In *Circuits and Systems (ISCAS), Proceedings of 2010 IEEE International Symposium on*, pages 13 –16, June 2010.

[19] Sung Hyun Jo, Kuk-Hwan Kim, and Wei Lu. Programmable resistance switching in nanoscale two-terminal devices. *Nano Letters*, 9(1):496–500, 2009.

[20] J.H. Kim, Wenle Zhang, Seung-Ki Ryu, and Yoon-Seuk Oh. An ADALINE neural network with truncated momentum for system identification of linear time varying systems. In *Industrial Technology (ICIT), 2012 IEEE International Conference on*, pages 292 –297, March 2012.

[21] Jihong Liu and Deqin Liang. A survey of FPGA-based hardware implementation of ANNs. In *Neural Networks and Brain, 2005. ICNN B '05. International Conference on*, volume 2, pages 915 –918, Oct. 2005.

[22] J.B. Lont and W. Guggenbühl. Analog CMOS implementation of a multilayer perceptron with nonlinear synapses. *Neural Networks, IEEE Transactions on*, 3(3):457 –465, May 1992.

[23] W.S. McCulloch and W.H. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115 – 133, 1943.

[24] G. Nagy. Neural networks-then and now. *Neural Networks, IEEE Transactions on*, 2(2):316–318, March 1991.

[25] Kristian R. Nichols, Medhat A. Moussa, and Shawki M. Areibi. Feasibility of floating-point arithmetic in FPGA based artificial neural networks. In *CAINE*, pages 8 – 13, 2002.

[26] A.S. Oblea, A. Timilsina, D. Moore, and K.A. Campbell. Silver chalcogenide based memristor devices. In *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pages 1 –3, July 2010.

[27] National Society of Professional Engineers. NSPE code of ethics for engineers, July 2012.

[28] Ernesto Ordoñez-Cardenas and Rene de J. Romero-Troncoso. MLP neural network and on-line backpropagation learning implementation in a low-cost FPGA. In *Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 333–338, 2008.

[29] Kevin T. Patton and Gary A. Thibodeau. *Anatomy & Physiology*. Elsevier, 2010.

[30] Dmitri B. Strukov, Gregory S. Snider, Duncan R. Stewart, and R. Stanley Williams. The missing memristor found. *Nature*, 453:80 – 83, May 2008.

[31] Linear Technology. LT1175 500mA low dropout micropower regulator.

[32] Thanh Tran. Simulations of artificial neural network with memristive devices. Master's thesis, Boise State University, Boise, Idaho, 2012.

[33] B. Widrow. An adaptive ADALINE neuron using chemical memistors. Technical Report No. 1553-2, Stanford Electronics Laboratory, Stanford, California, October 1960.

[34] Wikipedia. Harvard architecture, July 2012.

[35] Wikipedia. Von Neumann architecture, July 2012.

[36] Rodney Winter and Bernard Widrow. MADALINE Rule II: a training algorithm for neural networks. In *Neural Networks, IEEE International Conference on*, volume 1, pages 401 – 408, July 1988.

[37] Carlos Zamarreño-Ramos, Luis A. Camuñas-Mesa, Jose A. Pérez-Carrasco, Timothée Masquelier, Teresa Serrano-Gotarredona, and Bernabé Linares-Barranco. On spike-timing-dependent-plasticity, memrisitve devices, and building a self-learning visual cortex. *Frontiers in Neuroscience*, March 2011.