5-2023

Digital Simulations of Memristors Towards Integration with Reconfigurable Computing

Ivris Raymond
University of Arkansas, Fayetteville

Digital Simulations of Memristors Towards Integration with Reconfigurable Computing

Digital Simulations of Memristors Towards Integration with Reconfigurable Computing

An Undergraduate Honors College Thesis

in the

Department of Computer Science and Computer Engineering
College of Engineering
University of Arkansas
Fayetteville, AR
May, 2023

by

Ivris Raymond

Abstract

The end of Moore's Law has been predicted for decades. Demand for increased parallel computational performance has been increased by improvements in machine learning. This past decade has demonstrated the ever-increasing creativity and effort necessary to extract scaling improvements in CMOS fabrication processes. However, CMOS scaling is nearing its fundamental physical limits. A viable path for increasing performance is to break the von Neumann bottleneck. In-memory computing using emerging memory technologies (e.g. ReRam, STT, MRAM) offers a potential path beyond the end of Moore's Law. However, there is currently very little support from industry tools for designers wishing to incorporate these devices and novel architectures. The primary issue for those using these tools is the lack of support for mixed-signal design, as HDLs such as Verilog were designed to work only with digital components. This work aims to improve the ability for designers to rapidly prototype their designs using these emerging memory devices, specifically memristors, by extending Verilog to support functional simulation of memristors with the Verilog Procedural Interface (VPI). In this work, demonstrations of the ability for the VPI to simulate memristors with the nonlinear ion-drift model and the behavior of a memristive crossbar array are presented.

ACKNOWLEDGEMENTS

I'd like to first thank the University of Arkansas Honors College for supporting this research through the Honors College Research Grant.

I would like to thank my advisors Dr. Alexander Nelson and Dr. David Andrews for mentoring and guiding me through this work. I would also like to thank my committee for giving me helpful insight and ideas during this project. I would like to thank my lab mates, Arafat and Nathan, for their help coming up with the idea for this project and offering advice along the way. Finally, I'd like to thank my stepfather, Tim, for first introducing me to the world of electronics.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 Introduction

The von Neumann architecture, for all its successes, introduced a performance barrier related to memory. Each instruction and its data must first be loaded from memory into registers before it may be computed. This is defined as the von Neumann bottleneck and has long plagued computer architects in their ceaseless pursuit of high-performant computing machines. The architecture community has utilized a number of workarounds to enable ever increasing performance despite this bottleneck, but the rapidly approaching end to Moore's Law has all but eliminated future prospects for these workarounds [1]. Out-of-order execution, branch prediction, and multi-tier memory caching were all improvements that were greatly aided by the exponentially increasing transistor density of the past.

However, even as Moore's Law is ending, the demands for increasing computational performance continue to grow. Currently, AI and machine learning algorithms dominate the use-case for highly parallel systems. Many of these algorithms are limited by memory read and write operations [2]. Solutions to this problem have been explored in recent years with processing-in-memory designs in both reconfigurable and Application Specific Integrated Circuit (ASIC) implementations [3], [4]. These designs seek to circumvent the limitations of the von Neumann architecture by processing data directly in the memory rather than performing the usual load, process, store flow.

Many of these solutions also make use of novel components, some of which are analog in nature. For those that are still digital in nature, current tools for simulation, emulation, and layout of these devices are adequate. However, for those designs that utilize analog components for essential functionality, many tools provide very little support throughout the early stages of the design flow.

This thesis presents a method for functional simulation of these non-CMOS devices. This enables the designer to rapidly prototype according to simulation

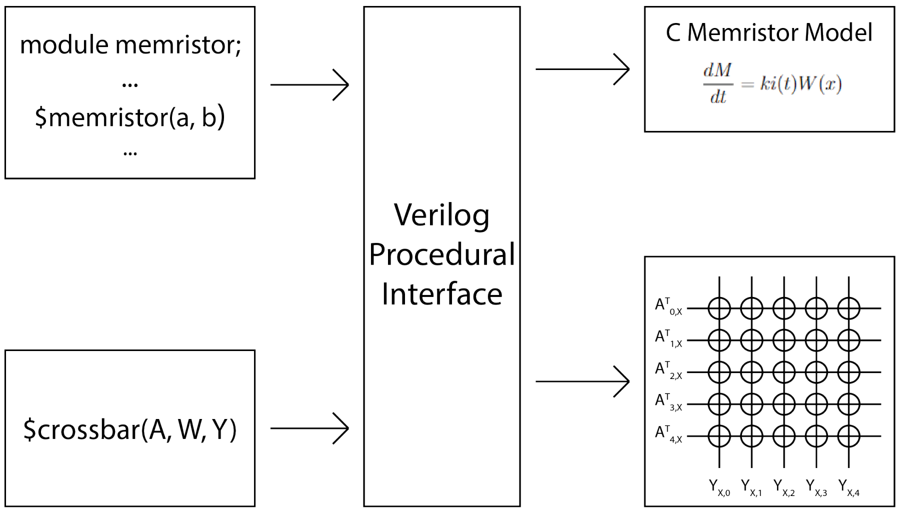

Figure 1.1: Outline of the process through which this work is able to support Verilog simulation of memristors. The designer makes calls to the Verilog Procedural Interface, which then refers to C code injected at runtime to simulate the behavior of memristors.

results while still supporting industry-standard and academic tools. This work will also discuss the ability for this flow to provide functional simulation in the FABulous tools, a recently developed suite of tools for developing embedded FPGAs with custom fabrics of both traditional CMOS and Post-Moore non-CMOS components [5].

2 Related Works

This work builds on prior art in several fields. This section categorizes and summarizes related work necessary to properly place the work performed in this thesis.

2.1 Post-CMOS

The architecture community has seen transistor scaling slow in the face of the end of Dennard Scaling, a statement that originally postulated transistor power density remains constant as device area is reduced [6], [7]. Post-Dennard Scaling, the architecture and fabrication communities continued to offer significant performance improvements through multicore architectures. The continued reduction of transistor size allowed for packing more and more cores onto a die without needing to develop more sophisticated interconnects or latency tolerant architectures.

However, the failure of Moore's Law to produce sufficiently smaller devices has already begun to change the way hardware designers develop multicore architectures. Recent industry developments in chiplet architectures and interconnects for these devices, such as AMD's Infinity Fabric, demonstrate the need to reevaluate potential improvements in interconnections between memories and CPUs. [8]

While the end of Moore's Law has long been touted as a disaster for the architecture community, it also presents an excellent opportunity for revisiting architectures and devices that fell to the wayside in the past due to their lack of necessity compared to the scaling enabled by Moore's Law.

2.1.1 Memristors

In 1971, Chua published work presenting a fourth passive circuit element yet undiscovered by the scientific community [9]. The memristor, as he described it, related charge and flux-linkage. This device completed the set of 4 devices expected to exist based on how electrical properties passive circuit elements were expected to relate. The memristor is a two-terminal device that behaves similarly to a resistor, but has a variable resistance related to the amount of charge that passes through it. Furthermore, this memristance ”memory resistance” does not require constant power due to the passive nature of the device, meaning that the memristor is non-volatile, unlike traditional computational memories like DRAM and SRAM.

While Chua published work presenting what should be the final passive circuit element, it wouldn't be until nearly 40 years later that a device that behaved in a manner similar to the theoretical memristor would exist. In 2008, HP created what would arguably be the first memristor utilizing a two-terminal titanium-oxide design [10]. However, HP's implementation of the memristor being the first also suffered from issues with reprogramming and limited reads [11].

Since HP's first successes at fabricating a memristive circuit element, many alternative designs and processes have been developed that improve upon many aspects of the device. Alternative circuit layouts for memristive systems have also been demonstrated to improve the resilience of memristive systems. One such example is [12], which utilizes a series resistor to improve the resistance to drift in the memristor upon reading. Furthermore, problems with resistance drift have been exacerbated by poor heat dissipation in memristive systems. However, work has already been done to demonstrate techniques for improving heat dissipation to attain a substantial reduction in resistance drift as values are read from and written to memristors [13]. Process yield has also greatly improved for nanoscale memristors recently, as in 2021 the work done in [14] presented a 90% yield rate for a 100nm process.

However, practical issues with adopting memristors for use in computational devices is not just limited to manufacturing. One of the biggest challenges a designer faces when making a memristive circuit is sneak paths, as they can greatly reduce the accuracy of a network whether it be used for memory, neuromorphic computing, etc. There has already been a lot of work discussing how this issue arises, and examples of how it can be solved in [15], [16]. This particular issue also affects logical conventions in memristors like imply logic [17]. While a wider adoption in the industry would likely result in more issues being found, the current success researchers have had in mitigating these issues sets a promising trend for the practicality of memristors.

Currently, there are several offerings of process nodes that include CMOS and non-volatile RAMs on the same substrate. One such example is the SkyWater 90nm and 130nm nodes which offer memristors for fabrication of chips that implement heterogeneous architectures [18]. The manufacturing of these devices is not limited to the SkyWater ChipIgnite programs either though, as other fabs such as TSMC have also manufactured devices used in published work [19]. Furthermore, TSMC specifically is one fab that has manufactured many chips utilizing the alternative devices discussed later in this work in published works [20], [21]. Companies are already manufacturing these devices, and while the scales may not be as high as CMOS processing nodes, there is a good reason to rapidly develop tools that support designers utilizing these technologies.

One method for overcoming some of the challenges of supporting memristive circuits for data processing and storage has been to add a series transistor on the bit-line. This 1 memristor 1 transistor (1M1T) layout can be seen in Figure 2.1 (a). This prevents issues such as sneak paths when several memristors are in parallel in a large array. The disadvantage of this is similar to those of CMOS Domino Logic (Fig. 2.1 (b)), as the benefits of using a non-CMOS technology are reduced with the introduction of additional correcting logic [22]. While a single transistor or pass-gate may not be CMOS, any transistor must be fabricated in silicon. Memristors are fabricated in the metal layers. This reduces the potential for area reduction, one of the primary advantages of memristors in heterogeneous systems.

One of the most promising use-cases for memristors is neuromorphic computing due to their analog nature providing outstanding support for multiply-

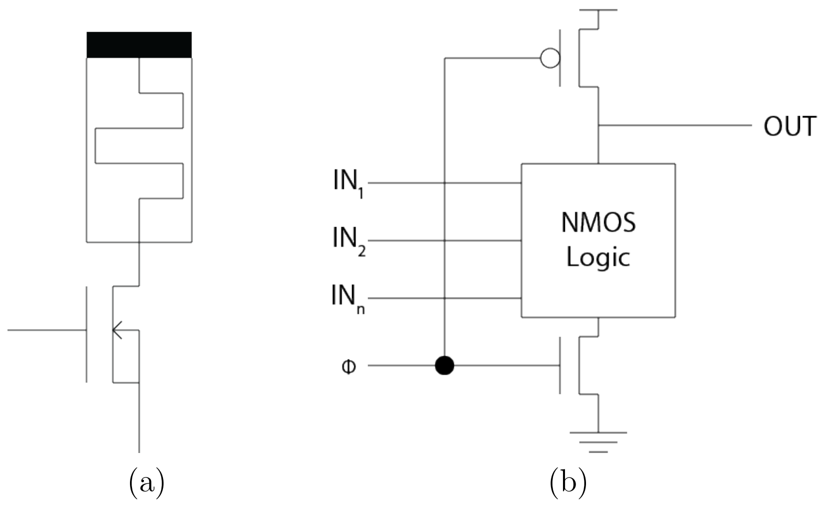

Figure 2.1: (a) 1m1t Cell. A memristor (top) in series with a transistor. (b) Domino logic unit

accumulate operations. The most common way to implement this behavior on a memristive crossbar array is by flashing the weights matrix to the memristors, then putting each vector on the rows of the array. It is necessary to transpose the input matrix using this method. This setup is shown in figure 2.2. This allows us to calculate an entire row of the output at once by measuring the value on the lines making up the columns of the array. This particular setup is very vulnerable to the sneak-paths issue discussed earlier in this section.

There are other logic families designed for memristors that can provide support for conventional Boolean logic traditionally done on CMOS components. Good examples of these are MAGIC and IMPLY, which vary highly in their techniques for implementing logical operations [23], [17].

2.1.2 Alternative non-CMOS Technologies

Other nonvolatile memory technologies are also being explored for their potential to accelerate neuromorphic computation. One of these options is Magnetic RAM, specifically Spin Torque Transfer (STT) RAM [24]. STT-RAM does offer some advantages compared to Resistive RAM (ReRAM). One of the most impor-

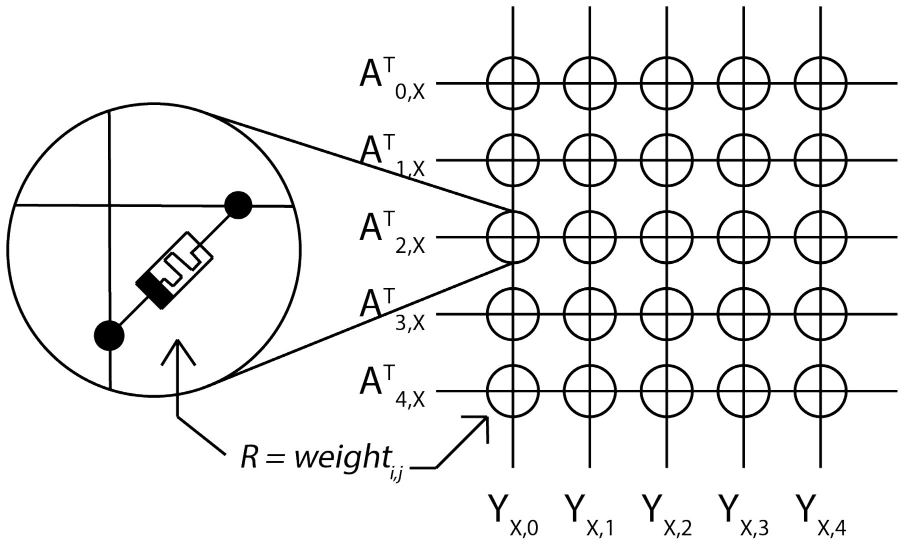

Figure 2.2: Memristor Crossbar Array for Multiply-Accumulate Operations. The input matrix (A) is transposed, then the matrix's column vectors are written to the horizontal lines. The vertical lines can then be read for the accumulated result, with each input vector resulting in a vector of the output matrix (Y).

tant differences is in the read and write speeds of STT-RAM, which tends to be much closer to SRAM than most of the other post-CMOS alternative technologies. With the development of STT-RAM, the scalability of STT-RAM was also greatly improved over the first variations of MRAM, which were largely Magnetic Tunnel Junction RAM due to the reduction in area [25].

Another alternative to ReRAM is Phase Change RAM (PC-RAM). PC-RAM does also benefit from non-volatility, so it does not require a refresh cycle. It also has a symmetric write and read cycle, both of which are non-destructive. This makes it a compelling option for replacing something like HDDs, as it doesn't suffer from the reliability issues of NAND flash. However, it does have a higher latency than DRAM. In many ways, PC-RAM is more likely to replace the role of NAND in a system rather than that of SRAM or a BRAM in an FPGA.

2.2 Simulation of Memristive Technologies

Researchers have already created many suites for simulating memristors at higher levels using a variety of models. Many of these are implemented in languages such as Python and C++, making them accessible to the general programmer. These simulation libraries are also often focused specifically on neuromorphic computing with memristors, making them ideal for testing the potential for memristors to implement a particular model [26], [27]. Furthermore, there has been extensive work in simulating memristors in very low-level scenarios, such as in SPICE simulators [28], [29]. In this way, there has been substantial exploration of simulating memristors at the circuit level, and application level. However, there has been little exploration of simulating memristors at the behavioral level.

2.3 Processing in Memory

Processing in memory is not a new idea. The general concept is to subvert the limitations of the von Neumann architecture by processing data directly in the memory instead of first having to move it out of memory, then back in after the computation [30]. However, CMOS has always been inherently limited for this architecture. In CMOS, designers don't usually have elements they can use as both memory and compute elements. This has resulted in a lot of the work on processing in memory done in CMOS being closer to processing *near* memory rather than *in* memory. Typically, you'll see this as something like processing elements, complicated or simple, added to bit lines at the output of the sense amps in something like a Block RAM unit in an FPGA [3], [31].

2.4 Mixed Signal VLSI

Many tools often used by hardware designers throughout the industry provide little to no support for mixed-signal circuit VLSI designs. This is because there was little need to use analog components for digital simulation early on in the process, as they often didn't provide additional functionality so much as doing things such as power filtering. Therefore, the addition of analog components was assumed to only take place at the layout stage of the design process. These tools could then export your design to a SPICE netlist for simulation, but it obviously isn't possible to go back up the chain from a netlist in a layout tool to an RTL Verilog description. Even with recently developed open-source tooling such as OpenRoad, there is no simulation flow for mixed-signal designs aside from compiling your gate-level netlist to a SPICE compatible netlist then running simulations in SPICE [32]. This problem is further compounded by the fact that many tools like FABulous are being developed to make development of heterogeneous architectures easier as long as your expected flow is verification in SPICE then tapeout.

For the first time, though, there is need for the ability to do functional simulation of mixed signal designs earlier in the process. This is because many emerging technologies, such as memristors, are analog in nature and require different simulation support than that which traditional Verilog event-driven simulators are able to provide. This has left many designers with very limited simulation options, often requiring them to go to layout then export that design to a SPICE netlist for simulation. This does work, but SPICE provides a lot more information than is required for functional verification, and at a hefty price in terms of simulation time even for simple designs [33].

2.5 Reconfigurable Computing

FPGAs are reprogrammable hardware useful for prototyping designs and implementing logic that may change often. There are two popular FPGA architectures in the industry currently: Island Style and Hierarchical [34]. More recently, the industry has grown interested in embedded-FPGAs (eFPGAs), which embed a chip (often an SoC) within FPGA fabric that extends its functionality in some helpful way for the designer's use-case. For instance, if a designer wanted to add vector instructions to an ARM SoC that didn't natively support them, they could do so with an eFPGA that extended the ISA of the chip such as in [35]. While the role FPGAs have played in computing outside of very specific use-cases has been relatively limited, the rapid increase in demand for machine learning acceleration has sparked new interest in their abilities. Recently, researchers have started exploring the best ways to utilize memristors and other emerging memory devices in FPGA fabric for improving performance, power, and area [36].

2.5.1 FABulous

FABulous is a recently developed suite of tools for customizing the fabric of eFPGA devices to better suit a user's application [5]. Most of the work FABulous does is at the RTL level and above, as it allows a user to define their fabric using CSV files to instantiate primitives. These primitives can be provided by the fab, or they can be custom primitives designed by the user.

FABulous provides most of its utility through abstraction. Primitives exist in tiles along with configuration blocks and switch matrices. Those tiles may then be arranged into supertiles, which can also separately contain more switch matrices and configuration blocks. This degree of abstraction makes fabric description for something like an Island Style architecture FPGA much faster. The user is able to write individual elements of the fabric, BRAMs, DSPs, etc... Those elements can then be instantiated in whatever pattern the designer wants to support using the CSV fabric descriptions rather than a Verilog Top Module. The output of FABulous is then an RTL description of your entire fabric, which can then be used to do functional simulation, place and route emulation, or coverted to a gate-level netlist and used for layout.

3 Implementation and Architecture

For this work, most of the implementation was done in C. The code used for the single memristor implementation and the crossbar array can be found in Appendix A. This work was written on a system running a version of Ubuntu with the packages required for GCC and IVerilog.

3.1 Memristors in VPI

My original intention for this project was to demonstrate how FABulous can be used to customize specific primitives in an FPGA fabric without needing to redesign the entire fabric. FABulous does indeed support this use-case as long as you intend to work in CMOS. However, the goal was to do so using memristive elements for a BRAM rather than standard SRAM. If this work were scoped to take the project to tapeout, that CAD flow would have been well supported. As previously mentioned, it would have been as simple as using FABulous to generate the RTL description of the fabric, then compiling to a gate-level netlist and doing layout with tools like OpenRoad [32], [5]. The issue was that this project would not be going to tapeout, and instead the intention was to do functional simulation to demonstrate the potential for the project as a proof of concept.

As previously mentioned, research in next-generation architectures utilizing novel non-CMOS components has been largely restricted by the tools available to researchers and designers. Currently, most of the industry uses tools meant for creating digital circuits, without much need for complicated mixed-signal designs or any analog components until the layout stage. When designing a chip utilizing an architecture that seeks to do computation with these novel components, though, it is often helpful to do functional simulation of the circuit without the computational overhead of a full SPICE simulation. Historically, this has been done with simple event-driven RTL-HDL simulators (eg. IVerilog).

ReRAM elements are analog in nature and cannot be simulated in HDL under the current IEEE standards of Verilog, SystemVerilog, or VHDL without an extension of the language. However, the Verilog standard (IEEE 1364) does include a method for interfacing with procedural languages such as C. Namely, the Verilog Procedural Interface [37]. Utilizing the VPI it is possible to write a task in Verilog that runs some C code during simulation of the Verilog netlist. Since we have a memristor model in C, the implementation utilized that along with the VPI to call the memristor code at runtime and provide arguments for the digital logic values going into the memristor. The C code makes use of the nonlinear ion-drift model presented in [38], [39], and was based heavily on work presented using C++ in [40]. Note that unlike the model presented in [40], this model only calculates the state variable for a single given time step in the below C code. The variables defined in this model also are not accurate to any particular process and would change to reflect those of the memristors produced by the process the user will be fabricating on. Below shows the model used in this work, as well as a selection of the code found in Appendix A.

$$\frac{dM}{dt} = ki(t)W(x)$$

Where k is a constant determined by the device properties, M is memristance, and *W(x)* is an implementation of the window function based on that presented in [39]. This was implemented as follows:

Two memristor models were implemented for this work, both of which were nonlinear. One included a window function for correcting sudden swings in memristance, while the other did not. The effects of this are discussed later in this thesis.

3.2 Larger Memristive Circuits in VPI

As previously mentioned, the original need for this work was in support of simulating memristive circuits in primitives included within FPGA fabric gener-

```
    double window = (1 - pow((2*(prev_state+0.00001) - 1), (2*P)));
    double change = K*(i*dt)*window;
    if (fabs(i) < THRESHOLD) {
        printf("NOT CHANGED\n");
        return new_state;
    }
    else if (i < 0) {
        M = M - change;
        if (M < R_ON) {
            M = R_ON;
        }
        else if (M > R_OFF) {
            M = R_OFF;
        }
        new_state = (M - R_ON) / (R_OFF - R_ON);
        return new_state;
    }
    else {
        M = M + change;
        if (M < R_ON) {
            M = R_ON;
        }
        else if (M > R_OFF) {
            M = R_OFF;
        }
        new_state = (M - R_ON) / (R_OFF - R_ON);
        return new_state;
    }
```

Figure 3.1: Code illustrating part of the implementation for the nonlinear ion-drift model of memristor behavior in C. Many of the physical constants are not shown here, but this information can be found in Appendix A.

ated by the FABulous tools. Since designers can provide a custom description of a primitive in FABulous, the user is able to replace something like a generic FPGA BRAM with one that implements processing in-memory with memristors. Previously, though, the designer was unable to simulate this until they put the fabric description into a layout tool and generated a SPICE netlist. The work presented in this thesis provides a CAD flow for simulation before layout, preventing the need to go up and down the stack just to verify functionality. Since the FABulous tools output a Verilog netlist description of the eFPGA fabric the user designed, the user could then compile that with the VPI in IVerilog before using something like next-pnr for simulated place and route.

A behavioral implementation of a memristive crossbar array in C for multiply-accumulate options was done in support of modifying an FPGA BRAM. In this case, the C function that is called is just a standard matrix multiplication function. The outer Verilog wrapper that calls the function performs mutations on the data based on the instruction it is given. In this case, it is possible to do matrix-matrix multiplication, matrix-vector multiplication, and vector-vector multiplication. In order to accomplish this, the outer Verilog wrapper aligns the data in different ways, prioritizing the left-most column for data that does not fill the array. Figure 2.2 also demonstrates how matrix multiplication is performed using a memristor crossbar array. The difference is that we multiply column-wise on both matrices, meaning that the transpose of the input matrix must first be taken. The weights matrix is already represented in the memristors by flashing the memristors with appropriate resistances ahead of time. For this work 4x4, 8x8, and 16x16 matrices were tested for matrix multiply validity. This work was chosen to match up with that discussed in [41], where the solution did not converge for 8x8 and 16x16 cases.

4 Results

Table 4.1: Matrix Multiply Successes in Various Simulators

Implementation	4x4	8x8	16x16	Digital vs Analog
VPI Crossbar	✓	✓	✓	Digital
SPICE Crossbar [41]	✓	✗	✗	Analog
PWL SPICE Crossbar [42]	✓	✓	✓	Analog†
Mem-torch Simulator [27]	✓	✓	✓	Digital*
IBM Analog Accel Kit [26]	✓	✓	✓	Digital*

The table above (Table 4.1) compares this work with alternatives for simulating memristive crossbars. Data from the papers cited was used for the information in this table. ✓ and ✗ indicate whether or not the simulation model can finish simulating a memristive crossbar array of the corresponding size. The high-level and low-level notes indicate whether the model is meant to prove the functionality of the design or to verify more detailed information (eg. power usage). * These simulations are run at a higher level than the traditional hardware stack (C++ and above). † These simulations sacrifice some accuracy compared to traditional SPICE simulations.

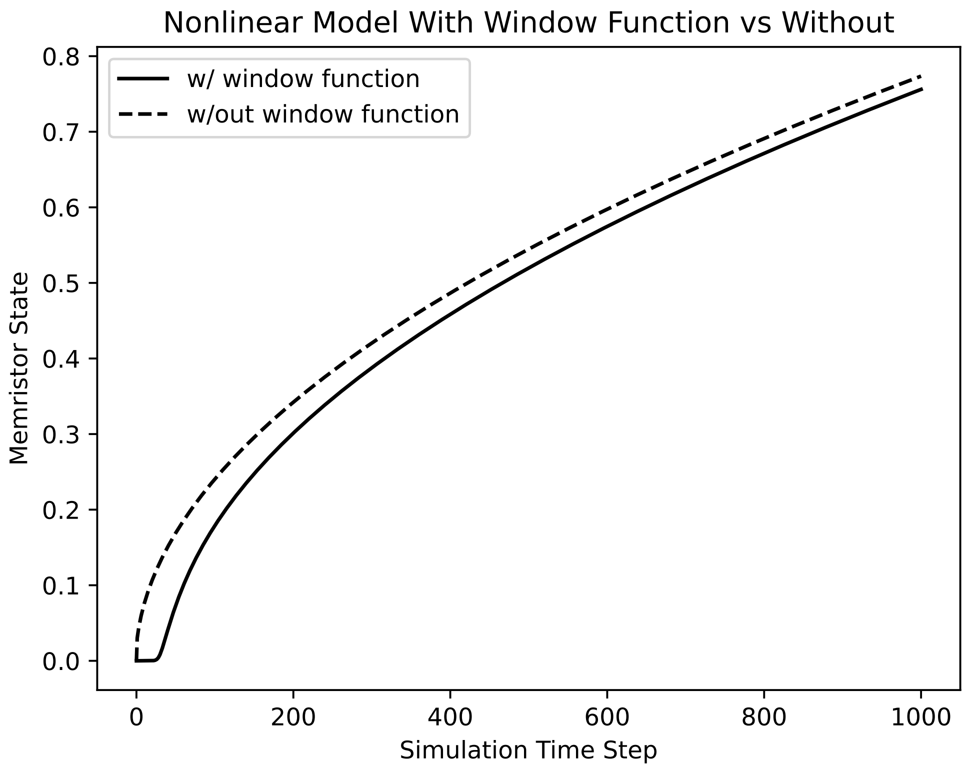

Figure 4.1: The graph of memristors transitioning from high resistance state to low resistance state as modeled in C for this work. The non-linear ion-drift model that omits the window function is the dashed line, while the nonlinear ion-drift model with the window function is the solid line. The time steps are an arbitrary unit for each iteration a current was applied for a fixed interval.

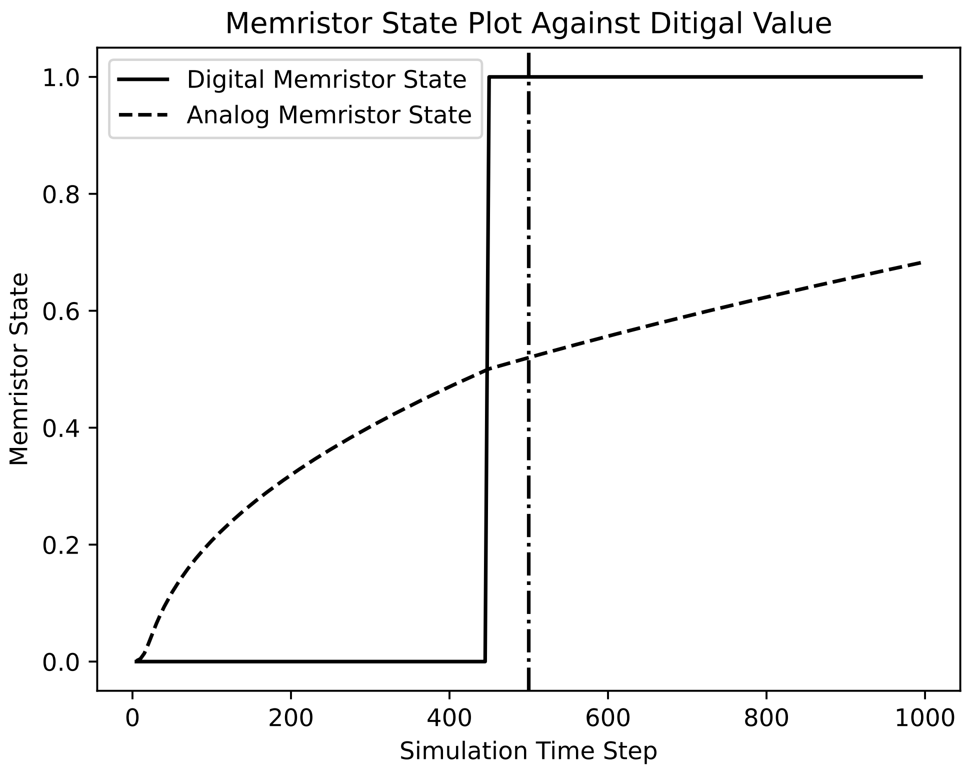

Figure 4.2: This plot illustrates the conversion of the state of the memristor vs the binary state it is considered to be in. The dotted-dashed line indicates the halfway point in the data collection time. Because of the non-linearity of this model, the halfway point does not line up with the transition from 0 to 1.

5 Discussion

The biggest issue with memristors from a designer's perspective is the lack of a streamlined workflow. Currently, hardware designers are cobbling together many tools with various contributions to get us through place and route. Once the architecture community gets there, it will still need to put together several pieces to accomplish mixed signal simulation in most cases. This work seeks to help streamline that process by making it unnecessary to move down the stack rapidly to layout in order to do functional verification. Currently all a designer needs to do is provide a C implementation of the component they want to simulate the functionality of, then inject that at runtime using VPI. This means a lot less code modification is necessary compared to writing several Verilog modules to simulate the behavior of an analog component in pure Verilog, but without the disadvantage of no longer being able to use the much faster Verilog simulators. Figure 5.1 depicts a typical CAD flow when designing devices. Highlighted in red is the point at which functional simulation takes place, notably earlier than the point at which other options such as SPICE simulation can take place.

Currently the process outlined in this paper does require some work on the part of the hardware designer in the event that a C implementation of the behavior they want hasn't already been made. Furthermore, there are some timing limitations when it comes to using the VPI. With the examples given, this work passes data back and forth from a task using the provided arguments to the task call in Verilog. The VPI offers less intuitive support for passing vectors as arguments, so in this work vectors were broken up into the number of components and passed as one argument per component. Consuming simulation time within a VPI task is also difficult, as by default no matter how long it takes your task to run, for simulation purposes it is finished in 0ns. This behavior is desirable for simulation, but the method for telling the simulator how much simulation time a task should consume

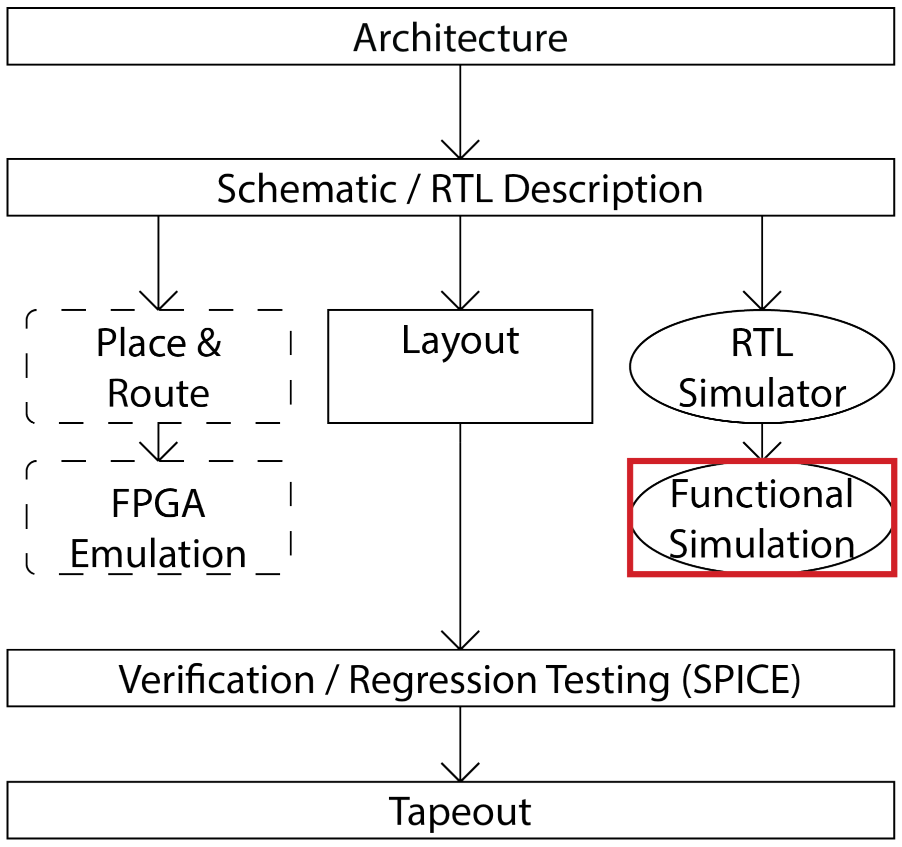

Figure 5.1: Typical CAD flow for designing devices.

is non-trivial. In general, the VPI can be difficult to work with for anyone not especially experienced with the more obtuse behaviors of the C language. There is potential for automating a lot of these processes though, and integrating the work presented here with other tools such as OpenRoad. There is further discussion of this in future works. Figure 4.1 makes a good case for differences in modeling being a worthwhile time investment for a designer, though, as the difference even for a small change in a function can have a pretty substantial effect for these devices.

A good demonstration of the ability for this work to help designers make decisions about their implementations can be seen in Figure 4.2. In the figure, it is obvious that the binary threshold between 0 and 1 does not occur midway through the simulation. Instead, it occurs a little sooner than would be expected due to the nonlinear nature of the memristive devices. This behavior may be desirable for some scenarios, such as those in which a CMOS skewed gate would have been appropriate, but in most cases a symmetrical gate is desired. The designer could make adjustments to their analog-to-digital conversion hardware in order to account for this, so instead of shifting the binary state when the state variable is halfway between the minimum and maximum value, it could be done earlier or later depending on the device behavior.

A valid concern about doing functional simulation in this manner is whether or not it will accurately reflect the behavior of the hardware. With the work presented, the simulation is only as good as the model the user is willing to write in C. However, whenever it comes to something like the crossbar example, it allows the user to validate the other portions of the circuit assuming they have a working crossbar. This also enables a divide-and-conquer approach to hardware development where one engineer may develop the analog accelerator portion separately from the team developing the digital logic. Whether or not it is necessary to write in such a manner to directly simulate the behavior of the hardware mathematically largely depends on the desired results from the simulation. In order to do something like test for sneak-paths, it may be necessary to do so, but if someone just needs the behavior to interface with there is no need to write the nonlinear ion-drift model in C.

It is also important to note that this work does not replace something like a SPICE simulator or doing emulation on an FPGA before pushing a design to tapeout. The role of this work is in the early phases of design, when an engineer wants to get a working version in the simulator and isn't yet concerned with power characteristics, etc. As mentioned previously, this enables rapid prototyping early on in the development process. The purpose of doing SPICE simulations during the hardware design process is often to prove the design will work on real hardware that tends to behave non-linearly rather than in a Verilog simulator at gate level. This is an important step in verification of a design, and the work presented in this paper should not be used to replace any portion of this step. The contribution of this work when compared to other works contributing higher level simulations of emerging memory devices is in the integration with the early stages of the design process. Where those other implementations would require a behavioral description of your digital architecture in higher level languages to communicate with the simulation, the method presented in this work interfaces directly with the RTL description of the digital logic.

5.1 Applying this work to Alternative non-CMOS Technologies

As discussed previously in this work, there are other emerging technologies that may provide similar benefits to specific application to memristors. These technologies such as MRAM and PCM also benefit from non-volatility, and some are also capable of representing more than one bit of information per cell. These technologies could also benefit from work similar to that discussed in this paper, as being able to represent that much information per cell also makes them unable to be represented in digital logic without a lot of overhead in both performance and in writing the behavioral HDL.

Historically, the software development community has benefited from the relatively homogeneous hardware stack that Moore's Law and CMOS scaling enabled. However, as this work has discussed, the goal of memristors and other emerging technologies like them is to aid CMOS with certain common operations rather than entirely replace it. This means a user won't have just one type of hardware in their system, but instead will have a variety of accelerators at their disposal. Current compilers are unable to map algorithms implemented by any given programmer to these accelerators. The general trend in research so far has been that most of the people interested in working with memristors also happen to be capable of mapping the algorithm they want to use to the appropriate accelerator in an appropriate manner. The architecture community obviously can't expect this of programmers that work largely in higher level languages such as C++ and Python, and certainly can't expect this from users. Thus far, efforts have been relatively limited on this problem due to the lack of need, but some work has been done that attempts to address these issues ahead of time. One example of using machine learning to map algorithms to the appropriate accelerator in an accelerator-rich system is [43].

5.1.1 Potential Applications

The most obvious application of this work is the goal that originally motivated this work: to design an eFPGA fabric that utilizes memristors in the BRAM to improve the performance of array processor implementations mapped to the fabric. However, works such as [36] also demonstrate the ability for memristors to substantially improve the communication circuitry in FPGAs. Emerging academic tools such as FABulous offer a promising method for future researchers to continue exploring this research. As mentioned previously, the improvement in support for the research use-case was the primary motivation for this work, as tool suite originally intended to use for this work. FABulous, didn't offer the simulation support for memristors that was needed for this timeline. This work was then pivoted to researching a way to provide that support. This work should provide an avenue for simulation that didn't exist previously with tools like FABulous for the designer that wants to make use of non-CMOS components.

Memristors are already being explored for their potential applications in IoT security for reducing power consumption and increasing performance compared to traditional AES Encryption cores [44]. Currently, some of the most well-researched alternatives for replacing traditional key establishment mechanisms in a post-quantum world involve solving complicated problems on lattice matrices with error components. Traditionally, this error is generated using dedicated hardware, such as random number generators. However, with the current state of memristor manufacturing, it may be possible to use resistance drift to our advantage for computational efficiency. Work has already been done demonstrating the ability for memristors to be used for securing communications with chaotic systems [45]. The advantage of using memristors demonstrated in [45] is that we can manufacture secure, trusted systems with untrusted foundries.

This work was also in support of exploring the potential memristors have for accelerating IoT applications to speeds and efficiency that nears ASIC. The primary advantage of memristors is in area and reconfigurability. Unlike ASIC, it is possible for the device to be reconfigured to some degree. Memristive processing in-memory elements offer many of the advantages of FPGA's reconfigurable fabric while substantially reducing the performance and power overhead of a traditional CMOS FPGA. Furthermore, IoT devices will also need new Key Establishment Mechanisms (KEM) as we approach a post-quantum computing world. Due to the computational demands of many of the algorithms presented in the NIST PQC competition, IoT devices often require substantial amounts of time and memory (relative to their comparatively limited DRAM pool) to perform the computation [46]. Efforts have already begun on exploring the potential for memristors to accelerate other security algorithms on IoT devices [47]. There is an immediate need for the acceleration of the PQC KEM algorithms though, and this use-case has yet to be addressed. This work would be beneficial in the process of designing an IoT KEM accelerator.

A final security implication of including memristors on IoT devices is an improvement in data privacy in devices using machine learning. Currently, many systems that perform inference on some user input must first transfer that data to a cloud server with more substantial processing capabilities. While this data is often encrypted to protect from man-in-the-middle attacks, it is not safe from bad actors at the destination engaging in maleficence with personally identifiable information (PII). Memristors could provide smaller IoT devices with the ability to perform inference on-device, thus eliminating the risks involved with sending potentially sensitive data to an off-site server for processing. Work has already been done demonstrating the efficacy of several emerging memory devices for this purpose, including ReRAM [48].

6 Conclusion and Future Work

This work demonstrated the potential for non-CMOS compontents to be simulated in the early phases of hardware design by expanding the Verilog standard (IEEE 1364) using the Verilog Procedural Interface. This work demonstrates the ability to do so in a manner that very closely simulates the actual functionality of a memristor using the nonlinear ion-drift model [38]. This thesis also demonstrated the ability for a designer to instead opt for a more behavioral-focused simulation that operates at a higher level to simulate a complete memristive system, such as a crossbar array used for multiply-accumulate operations [4], [49].

This work enables the designer to rapidly prototype and test their design using Verilog simulations rather than lower-level simulations such as SPICE or emulation methods. A designer with less intricate knowledge of SPICE simulations or less time to move through the stack before tapeout could utilize this technique to expedite their functional testing. Furthermore, this allows a designer to more easily integrate their design with tools such as FABulous [5], that can allow for abstraction that further expedites hardware design and modification.

This work is timely, because while memristors may currently struggle with reliability and resilience when it comes to reading and rewriting their current values, the materials community is rapidly improving processing techniques to solve these problems. As yield and resiliency of these devices improves, designers will be interested in the potential for these devices to accelerate their target use-cases. This work enables designers to more easily ensure functionality of their designs at a high level.

6.1 Future Work

There is still a lot that can be done using the techniques outlined in this thesis. This work primarily focuses on proving the efficacy of this CAD flow and discussing the use-cases in which this work is most valuable. Very few designs are given in this work which designers may want to actually use, due to the limited timeframe in which this research was conducted. A great place to start for the short term is to develop more designs utilizing the VPI for a variety of memristor use-cases.

In the short to mid-term, another great expansion of this work would be to provide designers with the ability to convert a SPICE netlist directly into VPI compatible C. Another worthwhile idea to explore would be providing the designer with semantics they can use in Verilog for tagging a particular module as being simulated with the VPI. Then, when compiling to RTL and gate-level netlists, it is then automatically swapped out with the appropriate blackbox module instantiation. This would further improve the ability of the user to rapidly prototype their mixed-signal design, which was the primary objective of this work. Providing an automated method for re-implementing certain portions of the designer's HDL would further reduce the amount of extra work that needs to be done on the part of the designer between functional simulation and emulation/RTL compilation.

Another long-term goal for this work could be to support existing higher level simulation libraries for memristors, such as those mentioned earlier in this work [27], [26]. For instance, this work could be extended to provide a Python library capable of wrapping these libraries for C function calls. The library would then implement the VPI code automatically around the simulation functions, substantially lowering the required expertise on part of the designer. Rather than needing to provide an implementation of their desired behavior in C themselves, the designer would be able to reuse work already done that behaves in the manner they desire without needing to write any C code themselves. The difficulty of creating the wrappers around these libraries would vary, but many of them provide implementations in C++ already. It may be relatively trivial to write C that wraps around the C++. For those that do not implement their behavior in C++, the work may be less trivial, but still a worthwhile effort. This would allow for compartmentalization of development, as a programmer not familiar with HDLs or even C could provide a simulation framework to hardware designers to suit their desired use-case.

Bibliography

[1] T. M. Conte, E. P. DeBenedictis, P. A. Gargini, and E. Track, “Rebooting computing: The road ahead,” *Computer*, vol. 50, no. 01, pp. 20–29, jan 2017.

[2] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. luc Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, and J. Ross, “In-datacenter performance analysis of a tensor processing unit,” in *International Symposium on Computer Architecture*, 2017. [Online]. Available: https://arxiv.org/pdf/1704.04760.pdf

[3] A. Arora, T. Anand, A. Borda, R. Sehgal, B. Hanindhito, J. Kulkarni, and L. K. John, “Comefa: Compute-in-memory blocks for fpgas,” 2022.

[4] M. Hu, C. E. Graves, C. Li, Y. Li, N. Ge, E. Montgomery, N. Davila, H. Jiang, R. S. Williams, J. J. Yang, Q. Xia, and J. P. Strachan, “Memristor-based analog computation and neural network classification with a dot product engine,” *Advanced Materials*, vol. 30, no. 9, p. 1705914, 2018. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/adma.201705914

[5] D. Koch, N. Dao, B. Healy, J. Yu, and A. Attwood, “Fabulous: An embedded fpga framework,” in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 45–56. [Online]. Available: https://doi.org/10.1145/3431920.3439302

[6] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, “Design of ion-implanted mosfet’s with very small physical dimensions,” *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct 1974.

[7] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” *SIGARCH Comput.*

Archit. News, vol. 39, no. 3, p. 365–376, jun 2011. [Online]. Available: https://doi.org/10.1145/2024723.2000108

[8] AMD, "Amd infinity architecture," 2019. [Online]. Available: https://www.amd.com/system/files/documents/LE-70001-SB-InfinityArchitecture.pdf

[9] L. Chua, "Memristor-the missing circuit element," *IEEE Transactions on Circuit Theory*, vol. 18, no. 5, pp. 507–519, 1971.

[10] D. Strukov, G. Snider, D. Stewart, and S. Williams, "The missing memristor found," *Nature*, vol. 453, pp. 80–3, 06 2008.

[11] A. V. Fadeev and K. V. Rudenko, "To the issue of the memristor's hrs and lrs states degradation and data retention time," *Russian Microelectronics*, vol. 50, pp. 311–325, 2021.

[12] K. M. Kim, J. J. Yang, J. W. Strachan, E. Grafals, N. Ge, N. Dávila, Z. Li, and S. Williams, "Voltage divider effect for the improvement of variability and endurance of taox memristor," *Scientific Reports*, vol. 6, p. 20085, 02 2016.

[13] H. An, M. S. Al-Mamun, M. K. Orlowski, L. Liu, and Y. Yi, "Robust deep reservoir computing through reliable memristor with improved heat dissipation capability," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 3, pp. 574–583, 2021.

[14] B. Zhang, W. Chen, J. Zeng, F. Fan, J. Gu, X. Chen, L. Yan, G.-J. Xie, S. Liu, Q. Yan, S. J. Baik, Z.-G. Zhang, W. Chen, J. Hou, M. El-Khouly, Z. Zhang, G. Liu, and Y. Chen, "90% yield production of polymer nano-memristor for in-memory computing," *Nature Communications*, vol. 12, p. 1984, 03 2021.

[15] M. A. Zidan, H. A. H. Fahmy, M. M. Hussain, and K. N. Salama, "Memristor-based memory: The sneak paths problem and solutions," *Microelectronics Journal*, vol. 44, no. 2, pp. 176–183, 2013. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0026269212002108

[16] Y. Cassuto, S. Kvatinsky, and E. Yaakobi, "Sneak-path constraints in memristor crossbar arrays," in *2013 IEEE International Symposium on Information Theory*, 2013, pp. 156–160.

[17] S. Kvatinsky, G. Satat, N. Wald, E. Friedman, A. Kolodny, and U. Weiser, "Memristor-based material implication (imply) logic: Design principles and methodologies," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 22, pp. 2054–2066, 10 2014.

[18] S. Technologies, "Cmos: 90 nm and 130 nm cmos technologies with embedded non-volatile memory," 2023. [Online]. Available: https://www.skywatertechnology.com/cmos/

[19] M. Giordano, K. Prabhu, K. Koul, R. M. Radway, A. Gural, R. Doshi, Z. F. Khan, J. W. Kustin, T. Liu, G. B. Lopes, V. Turbiner, W.-S. Khwa, Y.-D. Chih, M.-F. Chang, G. Lallement, B. Murmann, S. Mitra, and P. Raina, "Chimera: A 0.92 tops, 2.2 tops/w edge ai accelerator with 2 mbyte on-chip foundry resistive ram for efficient training and inference," in *2021 Symposium on VLSI Circuits*, 2021, pp. 1–2.

[20] H. L. Chiang, J. F. Wang, T. C. Chen, T. W. Chiang, C. Bair, C. Y. Tan, L. J. Huang, H. W. Yang, J. H. Chuang, H. Y. Lee, K. Chiang, K. H. Shen, Y. J. Lee, R. Wang, C. W. Liu, T. Wang, X. Bao, E. Wang, J. Cai, C. T. Lin, H. Chuang, H. S. P. Wong, and M. F. Chang, "Cold mram as a density booster for embedded nvm in advanced technology," in *2021 Symposium on VLSI Technology*, June 2021, pp. 1–2.

[21] W. S. Khwa, K. Akarvardar, Y. S. Chen, Y. C. Chiu, J. C. Liu, J. J. Wu, H. Y. Lee, S. M. Yu, C. H. Lee, T. C. Chen, Y. C. Lin, C. F. Hsu, T. Y. Lee, T. K. Ku, C. H. Kuo, J. Y. Wu, X. Y. Bao, C. S. Chang, Y. D. Chih, H.-S. P. Wong, and M. F. Chang, "Mlc pcm techniques to improve nerual network inference retention time by 105x and reduce accuracy degradation by 10.8x," in *2021 Symposium on VLSI Technology*, June 2021, pp. 1–2.

[22] A. Rjoub and O. Koufopavlou, "Low-power domino logic multiplier using low-swing technique," in *1998 IEEE International Conference on Electronics, Circuits and Systems. Surfing the Waves of Science and Technology (Cat. No.98EX196)*, vol. 2, 1998, pp. 45–48 vol.2.

[23] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Magic—memristor-aided logic," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 11, pp. 895–899, 2014.

[24] P. Rzeszut, J. Checiński, I. Brzozowski, S. Zietek, W. Skowroński, and T. Stobiecki, "Multi-state mram cells for hardware neuromorphic computing," 2021.

[25] J. Park, "Neuromorphic computing using emerging synaptic devices: A retrospective summary and an outlook," *Electronics*, vol. 9, no. 9, 2020. [Online]. Available: https://www.mdpi.com/2079-9292/9/9/1414

[26] M. J. Rasch, D. Moreda, T. Gokmen, M. Le Gallo, F. Carta, C. Goldberg, K. El Maghraoui, A. Sebastian, and V. Narayanan, “A flexible and fast pytorch toolkit for simulating training and inference on analog crossbar arrays,” in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, June 2021, pp. 1–4.

[27] C. Lammie, W. Xiang, B. Linares-Barranco, and M. Rahimi Azghadi, “Memtorch: An open-source simulation framework for memristive deep learning systems,” *Neurocomputing*, vol. 485, pp. 124–133, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0925231222002053

[28] Z. Biolek, D. Biolek, and B. V, “Spice model of memristor with nonlinear dopant drift,” *Radioengineering*, vol. 18, 06 2009.

[29] X. Guan, S. Yu, and H.-S. P. Wong, “A spice compact model of metal oxide resistive switching memory with variations,” *IEEE Electron Device Letters*, vol. 33, no. 10, pp. 1405–1407, 2012.

[30] O. Mutlu, S. Ghose, J. Gómez-Luna, and R. Ausavarungnirun, “A modern primer on processing in memory,” 2022.

[31] X. Wang, V. Goyal, J. Yu, V. Bertacco, A. Boutros, E. Nurvitadhi, C. Augustine, R. Iyer, and R. Das, “Compute-capable block rams for efficient deep learning acceleration on fpgas,” in *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2021, pp. 88–96.

[32] D. B. T. Ajayi, “Openroad: Toward a self-driving, open-source digital layout implementation tool chain,” *Proceedings of Government Microcircuit Applications and Critical Technology Conference*, 2019. [Online]. Available: https://par.nsf.gov/biblio/10171024

[33] N. Kapre and A. DeHon, “Performance comparison of single-precision spice model-evaluation on fpga, gpu, cell, and multi-core processors,” in *2009 International Conference on Field Programmable Logic and Applications*, 2009, pp. 65–72.

[34] I. Kuon, R. Tessier, and J. Rose, *FPGA Architecture: Survey and Challenges.* Now Foundations and Trends, 2008.

[35] N. Dao, A. Attwood, B. Healy, and D. Koch, “Flexbex: A risc-v with a reconfigurable instruction extension,” in *2020 International Conference on Field-Programmable Technology (ICFPT)*, 2020, pp. 190–195.

[36] J. Cong and B. Xiao, "mrfpga: A novel fpga architecture with memristor-based reconfiguration," in *2011 IEEE/ACM International Symposium on Nanoscale Architectures*, 2011, pp. 1–8.

[37] C. Dawson, S. Pattanam, and D. Roberts, "The verilog procedural interface for the verilog hardware description language," in *Proceedings. IEEE International Verilog HDL Conference*, 1996, pp. 17–23.

[38] V. Mladenov and S. Kirilov, "A nonlinear drift memristor model with a modified biolek window function and activation threshold," *Electronics*, vol. 6, no. 4, 2017. [Online]. Available: https://www.mdpi.com/2079-9292/6/4/77

[39] S. Thomas, S. Prakash, and K. Priya, "Characterization of memristor based on non-linear ion drift model," in *2017 International Conference on Energy, Communication, Data Analytics and Soft Computing (ICECDS)*, 2017, pp. 2189–2192.

[40] A. Bala, A. Adeyemo, X. Yang, and A. Jabir, "High level abstraction of memristor model for neural network simulation," in *2016 Sixth International Symposium on Embedded Computing and System Design (ISED)*, 2016, pp. 318–322.

[41] C. Yakopcic, T. M. Taha, G. Subramanyam, and R. E. Pino, "Memristor spice model and crossbar simulation based on devices with nanosecond switching time," in *The 2013 International Joint Conference on Neural Networks (IJCNN)*, 2013, pp. 1–7.

[42] Y. Zhang, H. Xu, Z. Li, Y. Sun, H. Yu, and C. Chen, "An efficient pwl memristor model with mmse parameter fitting," *IEEE Transactions on Electron Devices*, vol. 69, no. 3, pp. 1545–1552, 2022.

[43] H. T. Kassa, T. Verma, T. Austin, and V. Bertacco, "Chipadvisor: A machine learning approach for mapping applications to heterogeneous systems," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, April 2021, pp. 292–299.

[44] H. Rady, H. Hossam, M. Saied, and H. Mostafa, "Memristor-based aes key generation for low power iot hardware security modules," in *2019 IEEE 62nd International Midwest Symposium on Circuits and Systems (MWSCAS)*, Aug 2019, pp. 231–234.

[45] R. Vishwakarma, R. Monani, A. Hedayatipour, and A. Rezaei, "Reliable and secure memristor-based chaotic communication against

eavesdroppers and untrusted foundries," 2022. [Online]. Available: https://doi.org/10.21203/rs.3.rs-2331476/v1

[46] D. Atkins, "Requirements for post-quantum cryptography on embedded devices in the iot," in *Third PQC Standardization Conference*, 2021. [Online]. Available: https://csrc.nist.gov/CSRC/media/Events/third-pqc-standardization-conference/documents/accepted-papers/atkins-requirements-pqc-iot-pqc2021.pdf

[47] M. Uddin, A. S. Shanta, M. Badruddoja Majumder, M. S. Hasan, and G. S. Rose, "Memristor crossbar puf based lightweight hardware security for iot," in *2019 IEEE International Conference on Consumer Electronics (ICCE)*, Jan 2019, pp. 1–4.

[48] A. Singh, S. Diware, A. Gebregiorgis, R. Bishnoi, F. Catthoor, R. V. Joshi, and S. Hamdioui, "Low-power memristor-based computing for edge-ai applications," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2021, pp. 1–5.

[49] J. Chen, J. Li, Y. Li, X. Miao, J. Chen, J. Li, Y. Li, and X. S. Miao, "Multiply accumulate operations in memristor crossbar arrays for analog computing," *Journal of Semiconductors*, vol. 42, 09 2020.

7 Appendix

7.1 Appendix A

All of the code and data for this project can be found at the following GitHub Repo: https://github.com/Arenile/memristor-vpi

7.1.1 VPI Implementation of nonlinear ion-drift memristor

```
    double vIn = vInc - vDec;
    double M = ((R_OFF * prev_state) + (R_ON * (1 - prev_state)));
    double i = vIn / M;
    double new_state = prev_state;
    // Need window functin for nonlinear behavior
    double window = (1 - pow((2*(prev_state+0.0001) - 1), (2*P)));
    double change = K*(i*dt)*window;
    if (fabs(i) < THRESHOLD) {
        printf("NO CHANGE\n");
    }
    else if (i < 0) {
        M = M - change;
        if (M < R_ON) {
            M = R_ON;
        }
        else if (M > R_OFF) {
            M = R_OFF;
        }
        printf("IN i < 0\n");
        new_state = (M - R_ON) / (R_OFF - R_ON);
    }
    else {
        M = M + change;
        if (M < R_ON) {
            M = R_ON;
        }
        else if (M > R_OFF) {
            M = R_OFF;
        }
        new_state = (double)(M - R_ON) / (double)(R_OFF - R_ON);
    }
    return new_state;
}
```

7.1.2 Standard Matrix Multiply Function used in this work

```
// Matrix Multiply
void reconfig_crossbar(size_t n, char* input, char* weights,
                       char* output) {
    // Note that this assumes an nxn matrix
    char sum;
    for (int row = 0; row < n; row++) {
        for (int col = 0; col < n; col++) {
            sum = 0;
            for (int i = 0; i < n; i++) {
                sum += input[row*n+i] * weights[i*n+col];
            }
            output[row*n+col] = sum;
        }
    }
}
```

7.2 Appendix B: Outputs from Simulations

7.2.1 4x4 Matrix Multiply

```
INPUT
---------------------------
 9 2 2 5
 3 6 6 8
 2 1 9 2
 8 5 6 5
---------------------------
WEIGHTS
---------------------------
 5 8 2 1
 3 6 1 1
 7 6 1 5
 4 4 3 2
---------------------------
OUTPUT
---------------------------
 85 116 37 31
 107 128 42 55
 84 84 20 52
 117 150 42 53
---------------------------
```

7.2.2 8x8 Matrix Multiply

```
INPUT
---------------------------
 9 2 2 5 3 6 6 8
 2 1 9 2 8 5 6 5
 9 2 2 5 3 6 6 8
 2 1 9 2 8 5 6 5
 9 2 2 5 3 6 6 8
 2 1 9 2 8 5 6 5
 9 2 2 5 3 6 6 8
 2 1 9 2 8 5 6 5
---------------------------
WEIGHTS
---------------------------
 6 3 2 3 8 3 5 1
 2 1 5 2 8 5 6 5
 5 3 2 5 3 6 4 8
 7 2 8 7 8 5 6 2
 8 4 2 1 3 3 3 5
 1 1 9 6 8 5 4 4
 1 2 3 3 3 5 8 8
 2 1 3 4 8 5 9 5
---------------------------
OUTPUT
---------------------------
 155 83 174 165 273 183 248 172
 158 92 137 143 189 179 201 216
 155 83 174 165 273 183 248 172
 158 92 137 143 189 179 201 216
 155 83 174 165 273 183 248 172
 158 92 137 143 189 179 201 216
 155 83 174 165 273 183 248 172
 158 92 137 143 189 179 201 216
---------------------------
```

7.2.3 16x16 Matrix Multiply

```
INPUT
---------------------------
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
---------------------------
WEIGHTS
---------------------------
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7
 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
---------------------------
OUTPUT
---------------------------
 250 310 230 290 270 330 220 280 230 290 250 310 230 290 270 330
 330 406 262 338 254 330 356 432 318 394 330 406 262 338 254 330
 250 322 294 366 298 370 312 384 286 358 250 322 294 366 298 370
 250 318 306 374 222 290 248 316 334 402 250 318 306 374 222 290
 290 374 298 382 366 450 344 428 382 466 290 374 298 382 366 450
 250 310 230 290 270 330 220 280 230 290 250 310 230 290 270 330
 330 406 262 338 254 330 356 432 318 394 330 406 262 338 254 330
 250 322 294 366 298 370 312 384 286 358 250 322 294 366 298 370
 250 318 306 374 222 290 248 316 334 402 250 318 306 374 222 290
 290 374 298 382 366 450 344 428 382 466 290 374 298 382 366 450
 250 310 230 290 270 330 220 280 230 290 250 310 230 290 270 330
 330 406 262 338 254 330 356 432 318 394 330 406 262 338 254 330
 250 322 294 366 298 370 312 384 286 358 250 322 294 366 298 370
 250 318 306 374 222 290 248 316 334 402 250 318 306 374 222 290
 290 374 298 382 366 450 344 428 382 466 290 374 298 382 366 450
 250 310 230 290 270 330 220 280 230 290 250 310 230 290 270 330
---------------------------
```