Verification and Automated Synthesis of Memristor Crossbars

2016

Arya Pourtabatabaie
University of Central Florida

VERIFICATION AND AUTOMATED SYNTHESIS OF MEMRISTOR CROSSBARS

by

ARYA POURTABATABAIE
B.S. University of Tehran, 2014

A thesis submitted in partial fulfilment of the requirements
for the degree of Master of Science in Computer Science
in the Department of Electrical Engineering and Computer Science
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2016

Major Professor: Sumit Kumar Jha

ABSTRACT

The Memristor is a newly synthesized circuit element correlating differences in electrical charge and magnetic flux, which effectively acts as a nonlinear resistor with memory. The small size of this element and its potential for passive state preservation has opened great opportunities for data-level parallel computation, since the functions of memory and processing can be realized on the same physical device.

In this research we present an in-depth study of memristor crossbars for combinational and sequential logic. We outline the structure of formulas which they are able to produce and henceforth the inherent powers and limitations of Memristive Crossbar Computing.

As an improvement on previous methods of automated crossbar synthesis, a method for symbolically verifying crossbars is proposed, proven and analysed.

TABLE OF CONTENTS

LIST OF TABLES

CHAPTER 1: BACKGROUND

1.1 The Memristor

A memristor is a passive circuit element which changes its resistance in response to how much charge has passed through it. Imposing a current in one direction causes its (symmetrical) resistance to drop and imposing current in the other direction would cause it to increase.

The memristor was first theorized in 1971 by Leon Chua[3]. However, they were only fabricated in 2008. Their small size and passive memory, which would allow for low-power electronics, is promising for making efficient in-memory computation possible and thus removing the memory-processor communication bottleneck.

1.2 Memristive Crossbar Computing of Boolean Formulas

1.2.1 Memristor Crossbars

A crossbar is essentially a mesh of horizontal and vertical wires, with certain circuit elements at their junction. A memristor crossbar is one where there is a memristor at each junction. In order to tailor this setting to Boolean Combinational computing, we have to attach logical semantics to physical phenomena. For wires, the traditional mapping of supply voltage to 1 and ground voltage to 0 is acceptable. For memristors, it seems natural to have the resistance extrema represent boolean values: High conductance corresponds to logical 1 and high resistance corresponds to 0. Just as medium voltage levels for wires are considered "illegal" values, so are the memristors presumed to be either in a state of perfect conductance or in a state of perfect resistance.

We index memristors in accordance to the rows and columns they connect, respectively. Each memristor is either in a state of perfect conductance (corresponding to logical 1, by convention) or that of perfect resistance (corresponding to logical 0).

With all components in such a circuit being passive, every wire is at the 0 logical state if the circuit

is not somehow powered externally. In this thesis, unless otherwise stated, the circuit is powered by feeding the supply voltage to the bottom row, which we call r_0. This means that the logical state of any wire in the circuit is determined by whether or not it is connected to r_0 through a path of low resistance (namely, through wires and conducting memristors), such paths are called *Sneak Paths*.

Example 1.2.1. Consider the circuit below.

		c_0	c_1
	r_2	1	0
	r_1	0	0
$\rightarrow$	r_0	1	0

r_0 is at supply voltage and holds the logical value 1 by definition. With m_{00} being in the conducting state, we infer that c_0 is also conducting. Additionally, r_2 can be powered from c_0 through m_{20} and therefore also holds the value 1. r_1 and c_1 are at a 0-state, because there is no conducting path through which they can be powered.

The crossbar considered above is static and is incapable of performing interesting computations. It yields the same result every time it is powered. In order to be able to implement a Boolean function, we have to introduce some change in the crossbar with respect to the input vector in consideration.

1.2.2 Crossbar Designs

We can implement Boolean function with conditional pre-programming of memristors with respect to each input vector. In its simplest form, which is the focus of this thesis, each junction corresponds to a literal, that is, an atom in the input vector or its negation. For example, a memristor set to $\neg A$ will be set to high impedance whenever $A = 1$, and to high conductance wheneber $A = 0$.

Example 1.2.2. In fact, the static crossbar shown in 1.2.1 was an instance of a half-adder, in a case where $A = 0, B = 1$. The full design is as follows:

$$\begin{array}{cccc} & \overline{A} & A & \rightarrow (A+B)[0] \\ & A & 0 & \\ 1 \rightarrow & B & \overline{B} & \end{array}$$

As evident, r_0 is constantly powered and the result of the operation appears on r_2.

1.3 Examples of Simple Circuits

The following results have been obtained through synthesis methods based on that proposed by Velasquez and Jha[1]. The z3 SMT solver was used on a single compute node on the Comet service from the San Diego Supercomputer Center. These designs can serve as examples for making crossbar computing more intuitively clear.

Example 1.3.1. *Half Adder*

$$\begin{array}{ccc} A & B & 0 \\ 0 & 0 & A \\ \overline{B} & \overline{A} & B \end{array}$$

Note the sparsity of the design. Such sparsity if often indicative of the existence of more compact designs, as is the case with this design and 1.2.2.

Example 1.3.2. *3-Bit Parity*

The solution was obtained from Comet in a matter of a few seconds. The design is as follows:

$$\begin{array}{ccc} B & A & \overline{B} \\ \overline{C} & 1 & C \\ \overline{B} & \overline{A} & B \end{array}$$

Example 1.3.3. *4-Bit Parity*

$\overline{D}$ $\overline{C}$ C D

$\overline{A}$ $\overline{B}$ B A

D C $\overline{C}$ $\overline{D}$

This took about half an hour on Comet.

Example 1.3.4. *Unsigned 2-Bit Comparator*

Note that this component can essentially serve as providing the sign bit of a 2's complement representation of the subtraction of the two input numbers A and B. Here, the index under a letter denotes the index of significance of the bit inside the number in question, with index 0 representing the least significant bit.

A_1 $\overline{A_1}$

B_0 $\overline{A_0}$

$\overline{A_1}$ B_1

Example 1.3.5. *Full Adder*

This design is interesting in that it provides two outputs rather than one. r_3 holds the sum and r_2 holds the carry-out. There are two main inputs a and b, and a carry-in input c.

A $\overline{B}$ $\overline{B}$ 0 B $\rightarrow$ Sum

C 1 A A $\overline{A}$ $\rightarrow$ Carry-Out

B 0 0 C 0

0 $\overline{C}$ B $\overline{C}$ $\overline{B}$

This was designed in a matter of seconds.

Example 1.3.6. *2-Bit Subtractor with No Carry-In or Carry-Out*

- 5×5:

$\overline{A_1}$	B_0	A_1	0	0	$\rightarrow$ Result[1]
0	0	0	1	$\overline{A_0}$	$\rightarrow$ Result[0]
B_1	$\overline{A_0}$	$\overline{B_1}$	0	A_0	
0	A_0	0	$\overline{B_0}$	$\overline{B_0}$	
A_1	$\overline{B_0}$	$\overline{A_1}$	0	B_0	

  Sparsity in the resulting crossbar hints at the probability of there existing smaller designs.

- 5×4:

r_4	0	0	$\overline{A_1}$	A_1	$\rightarrow$ Result[1]
r_3	A_0	$\overline{A_0}$	0	0	$\rightarrow$ Result[0]
r_2	B_0	$\overline{A_0}$	$\overline{B_1}$	B_1	
r_1	$\overline{B_0}$	A_0	B_1	$\overline{B_1}$	
r_0	$\overline{B_0}$	B_0	0	0	
$*$	c_0	c_1	c_2	c_3	

  This design is particularly interesting since it shows an order in design which is composed of distinct object, is provably correct and is intuitive to a human observer therefore is a great candidate to be used as a heuristic in automated large-scale synthesis.

  The rows r_4 and r_3 can never create a new sneak path, since neither of them at any moment can have more than single memristor on. Therefore removing them from the design does not affect the logical function of other wires (The logical function of a wire being $f : \mathbb{B}_{\not\leftarrow}{}^n \rightarrow \mathbb{B}_{\not\leftarrow}$ that describes the conducting and non-conducting state of the wire in terms of input in a particular design).

- 4×5:

  0 $\overline{A_1}$ 0 A_1 B_0 $\rightarrow$ Result[1]

  B_0 0 $\overline{B_0}$ 0 0 $\rightarrow$ Result[0]

  $\overline{B_0}$ B_1 A_0 $\overline{B_1}$ $\overline{A_0}$

  $\overline{A_0}$ A_1 A_0 $\overline{A_1}$ $\overline{B_0}$

- There exists no 4×4 design for a 2-bit subtractor.

It is fairly obvious that a negative result (i.e. the nonexistence of any implementation of a function) on the $R \times C$ crossbar implies the same on any smaller crossbar. Namely, if no $R \times C$ design exists, neither does an $R' \times C'$ design where $R' \leq R$ and $C' \leq C$. This will be formally proven in the following chapter.

Example 1.3.7. *2-Bit Subtractor: No Carry-In, Providing Carry-Out*

- No 5×5 solution exists.

- 5×6:

  B_1 0 $\overline{B_1}$ 0 0 $\overline{A_1}$ $\rightarrow$ Result[2](Carry-Out)

  A_1 0 $\overline{A_1}$ 0 $\overline{A_0}$ $\overline{B_1}$ $\rightarrow$ Result[1]

  0 A_0 0 B_0 0 0 $\rightarrow$ Result[0]

  $\overline{B_1}$ $\overline{B_0}$ B_1 $\overline{B_0}$ B_0 A_1

  $\overline{A_1}$ $\overline{B_0}$ 0 $\overline{A_0}$ A_0 B_1

  Note: In general, it appears that for each given area set for a crossbar design, those with more width and less height are more favorable, since there is a higher number of wires to provide logical "intermediate results". This is in spite of the fact that square-shaped crossbars are simpler to fabricate.

  Observation: Addition (Subtraction) is a computation with "limited parallelization", meaning that even though there is some dependency between the results of each output line, those dependencies can be propagated through the vertical lines, and therefore, we see that in the

uppermost wire, all memristors only depend on the most significant bits, on the second, there is a mixture and on the lowest wire, all memristors depend only on the least significant bits (that is of course obvious, because the least significant bit of the subtraction depends only on the least significant bits of the input.)

- The 6×5 design:

  $0 \quad \overline{B_1} \quad B_1 \quad \overline{B_1} \quad 0 \quad \rightarrow$ Result[2] (Carry)

  $0 \quad 0 \quad A_1 \quad \overline{A_1} \quad 0 \quad \rightarrow$ Result[1]

  $\overline{A_0} \quad B_1 \quad 0 \quad 0 \quad A_0 \quad \rightarrow$ Result[0]

  $0 \quad B_0 \quad \overline{A_0} \quad 0 \quad \overline{B_1}$

  $A_0 \quad 0 \quad \overline{B_1} \quad B_1 \quad \overline{B_0}$

  $B_0 \quad 0 \quad \overline{A_1} \quad 0 \quad \overline{B_0}$

  Observation: Very rarely is a memristor seen to be unconditionally set to 1. Therefore this was enforced as a heuristic to shrink the design search space.

  Observation: One may also think of the heuristic that the row providing the least significant bit of the result should only have its memristors depend on the least significant bit of the input.

  Enforcing the heuristics, the following design was obtained:

  $A_1 \quad 0 \quad \overline{A_1} \quad 0 \quad 0 \quad \rightarrow$ Result[2] (Carry)

  $B_1 \quad 0 \quad \overline{B_1} \quad 0 \quad A_1 \quad \rightarrow$ Result[1]

  $0 \quad B_0 \quad 0 \quad \overline{B_0} \quad 0 \quad \rightarrow$ Result[0]

  $\overline{A_1} \quad \overline{B_0} \quad A_1 \quad A_0 \quad \overline{B_1}$

  $0 \quad 0 \quad B_0 \quad 0 \quad \overline{A_0}$

  $0 \quad \overline{A_0} \quad B_1 \quad A_0 \quad \overline{A_1}$

  However, no considerable speedup was observed.

- The 6×6 design:

A_0	0	$\overline{B_1}$	$\overline{B_0}$	B_1	0	$\rightarrow$ Result[2] (Carry)
$\overline{A_0}$	0	B_1	0	$\overline{B_1}$	0	$\rightarrow$ Result[1]
0	$\overline{B_0}$	A_1	0	$\overline{A_1}$	$\overline{B_0}$	$\rightarrow$ Result[0]
0	0	0	A_0	0	B_0	
$\overline{A_1}$	$\overline{A_0}$	B_1	0	0	$\overline{A_0}$	
A_0	B_0	$\overline{A_1}$	$\overline{B_0}$	0	0	

Example 1.3.8. *2-Bit Subtractor with Carry-In*

It is desirable to be able to cascade crossbar designs in the same way as transistor logic circuits, in order to build logic capable of large-scale computation. In order to be able to cascade the modules built to make larger ones, the synthesis of 2-Bit Adders with a Carry-In input was attempted. (The result then would be $Sum = A - B - c_i$)

- 5×6, 6×5 and 5×7 designs do not exist.
- We could not determine the existence of a 6×6 design in 48 hours. In order to facilitate achieving a positive result given that one exists, the following constraints were imposed:

  1. Literals only depend on the most significant bits of the input on the row producing the carry-out.
  2. Literals only depend on the least significant bits of the input and the carry-in on the row producing the least significant bit of the result.
  3. No 'always true' literals.

  The search was finished in less than a day with a negative result. However, negative results for a restricted design space are not conclusive.

1.4 Scaling Up

The last few examples demonstrate the difficulty of designing larger crossbars implementing more complex functions. In order to be able to achieve complex computations, the following ways come to mind:

1. Designing sparse crossbars constructively
2. Modular Design, cascading multiple smaller crossbars, for each the design problem is manageable
3. Designing more efficient verification techniques
4. Approximate implementations through non-exhaustive search methods

The first way has been the subject of a fair amount of research (citations...).
The remainder of this chapter will be concerned with the second way. Efficiency of verification techniques and approximations will be the subjects of later chapters 3 and 4, respectively.

The following designs were obtained in an attempt to develop BCD subtractors for the purpose of designing specialized hardware for pairs trading, a short selling method in financial markets.

Example 1.4.1. *BDC Comparator (Less Than)*
This comparator is useful for providing the sign bit of a BCD subtractor's output.

- 4×5:

  $\overline{A_2}$ $\overline{A_1}$ $\overline{A_3}$ B_1 0 $\rightarrow$ Result[0]

  0 B_0 0 $\overline{A_0}$ 0

  B_2 B_1 B_3 $\overline{A_1}$ $\overline{A_2}$

  A_2 0 B_3 0 $\overline{A_3}$

  was obtained in 6 hours.

- 5×4:

  B_2 $\overline{A_2}$ 0 B_3 $\rightarrow$ Result[0]

  A_0 $\overline{A_0}$ B_0 0

  $\overline{B_3}$ 0 B_1 $\overline{A_2}$

  $\overline{A_1}$ B_1 $\overline{B_1}$ 0

  B_3 A_2 0 $\overline{A_3}$

  was obtained in 9 hours.

  There are smaller designs feasible for this logic compared to a 4-Bit Comparator. This is because of the presence of "Don't Care" literals. However, for large crossbar sizes when the search is indecisive, in is better to revert to a more restricted instance of the function.

Example 1.4.2. *2-Bit Subtractor*

Example 1.4.3. *2-Bit Subtractor with Carry-Out*

- 5×6:

  B_1 0 $\overline{B_1}$ 0 0 $\overline{A_1}$ $\rightarrow$ Result[2](Carry-Out)

  A_1 0 $\overline{A_1}$ 0 $\overline{A_0}$ $\overline{B_1}$ $\rightarrow$ Result[1]

  0 A_0 0 B_0 0 0 $\rightarrow$ Result[0]

  $\overline{B_1}$ $\overline{B_0}$ B_1 $\overline{B_0}$ B_0 A_1

  $\overline{A_1}$ $\overline{B_0}$ 0 $\overline{A_0}$ A_0 B_1

- 6×5:

  0 $\overline{B_1}$ B_1 $\overline{B_1}$ 0 $\rightarrow$ Result[2] (Carry)

  0 0 A_1 $\overline{A_1}$ 0 $\rightarrow$ Result[1]

  $\overline{A_0}$ B_1 0 0 A_0 $\rightarrow$ Result[0]

  0 B_0 $\overline{A_0}$ 0 $\overline{B_1}$

  A_0 0 $\overline{B_1}$ B_1 $\overline{B_0}$

  B_0 0 $\overline{A_1}$ 0 $\overline{B_0}$

Enforcing a heuristic: The row providing the least significant bit of the result should only have its memristors depend on the least significant bit of the input. With that heuristic enforced we have the following design:

A_1	0	$\overline{A_1}$	0	0	$\rightarrow$ Result[2] (Carry)
B_1	0	$\overline{B_1}$	0	A_1	$\rightarrow$ Result[1]
0	B_0	0	$\overline{B_0}$	0	$\rightarrow$ Result[0]
$\overline{A_1}$	$\overline{B_0}$	A_1	A_0	$\overline{B_1}$	
0	0	B_0	0	$\overline{A_0}$	
0	$\overline{A_0}$	B_1	A_0	$\overline{A_1}$	

CHAPTER 2: FORMAL ANALYSIS

First we study the conductance properties of a crossbar in response to external voltages enforced on the wires and the resistances recorded on the memristors. That behavior can be completely determined by knowing only the size of the crossbar (Number of rows and number of columns).

Then we proceed to consider the behavior of crossbars as implementations of Boolean functions. In this context, a "design" would be a description of memristor resistances and initial wire conductances as functions of some Boolean input, which would imply the voltage of all wires will be functions of the same input.

2.1 Static Crossbars

Definition 2.1.1. A Static Crossbar is a set $\{W, M, W_0\}$ of wires $W : \mathbb{N} \to \mathbb{B}$, conjunctions $M : \mathbb{N} \to \mathbb{B}$ and external voltages $W_0 : \mathbb{N} \to \mathbb{B}$, where the law of conductance (as formalized in 2.1.18) applies.

Remark 2.1.2. For notational convenience, junctions will be referenced bidimensionally (m_{ij}), and wires will be referenced as either rows (r_i) or columns (c_j).

For any wire, the predicate $Row(w)$ asserts that it is a row and $Col(w)$ asserts that it is a column.

Remark 2.1.3. In our notation, indexes start at 0.

Definition 2.1.4. An $R \times C$ crossbar is one for which we have,

$$\forall i \geq R : r_i = 0$$

and

$$\forall j \geq C : c_i = 0$$

Remark 2.1.5. Naturally, every real-world crossbar is bounded. However, this assumption can sometimes complicate our analysis process. Therefore, whenever possible, the definitions and theorems are expressed abstractly, applicable to bounded and infinite crossbars alike.

Remark 2.1.6. Inclusion of the enforced voltage profile was necessary to fully capture the physical behavior of the crossbar. To have a wire's enforced voltage to be 1, V_d is forced on it. This means that no matter how the sneak paths inside the crossbar play out, that wire would have an "on" voltage. However, the meaning of the enforced voltage to be 0 is merely having the wire's power connection to be of high impedance, it does not prevent the wire from *having* a high voltage through memristor conductance.

With all that said, a disconnected wire would have a ground voltage in practice due to electric leakage, and we will capture that with the next set of definitions.

Definition 2.1.7. A Crossbar Path is a finite sequence of wires $p = \langle w_0, ..., w_K \rangle$ where the wires are of alternating types.

Definition 2.1.8. An Acyclic Crossbar Path is a crossbar path $p = \langle w_0, ..., w_K \rangle$ where no two wires are identical.

Theorem 2.1.9. *For every path, there exists an acyclic subpath.*

Proof. Considering the crossbar as a complete bipartite graph with rows on one side and columns on the other, the result follows from elementary graph theory. □

Remark 2.1.10. The last two definitions are only concerned with the positions of the wires, not their logical values. Therefore they are invariant with respect to different crossbars.

Definition 2.1.11. A Sneak Path in a static crossbar is a path wherein for every consecutive wires w'_i and w'_{i+1} their junction m'_i has value 1.

Definition 2.1.12. An Acyclic Sneak Path in a static crossbar is a Sneak Path for which the underlying path is acyclic.

Theorem 2.1.13. *For every Sneak Path p, there exists an acyclic Sneak Subpath.*

Proof. The edges of the acyclic subpath of p's underlying path constructed by 2.1.9 is a subsequence of p's underlying path's edges. Since edges correspond to junctions, all junctions in the acyclic subpath are also conductive and the acyclic subpath is therefore a Sneak Path. □

Definition 2.1.14. We call a pair of wires w_i and w_o *Connected* iff there exists a Sneak Path between them. We write this relation as $Connected(w_i, w_o, M)$

Remark 2.1.15. In other words, w_i and w_o are connected if and only if an Acyclic Sneak Path exists between them.

Theorem 2.1.16. *Connectedness is an equivalence relation. Namely,*

Reflexivity $Connected(w, w)$

Symmetry $Connected(w, w') \Rightarrow Connected(w', w)$

Transitivity $Connected(w, w'), Connected(w', w'') \Rightarrow Connected(w, w'')$

Proof. **Reflexitivity** The path $\langle w \rangle$ connects any wire w is to itself, regardless of the state of any junction.

Symmetry For any path $\langle w, w_0, ..., w_k, w' \rangle$ that connects w to w', the path $\langle w', w_k, ..., w_0, w \rangle$ connects w' to w.

Transitivity For any path $\langle w, w_0, ..., w_k, w' \rangle$ connecting w to w' and any path $\langle w', w'_0, ..., w'_{k'}, w'' \rangle$ connecting w' to w'', the path $\langle w, w_0, ..., w_k, w', w'_0, ..., w'_{k'}, w'' \rangle$ connects w to w''. This path may be cyclic but it certainly contains an acyclic subpath that connects w to w''.

□

Definition 2.1.17. The *Transfer Function* from wire w_i to w_o (and back) as a predicate denoting whether they are connected, as a function of a conductance profile.

Definition 2.1.18. *The law of conductance*:

In a static crossbar, each wire w we have:

$$w = 1 \Leftrightarrow \exists w' : \Big[Connected\{w, w'\} \wedge w'_0 = 1\Big]$$

Remark 2.1.19. By this definition, for any initial voltage enforcement and a conductance profile for the junctions, the value of all wires is uniquely determined. As such, the value of wires can be thought of as a *function* of the former two.

Theorem 2.1.20. *Given two wires w and w' of the same type and a wire w'' (no two of the three being necessarily distinct), for any acyclic path between w'' and w, there is an acyclic path between w'' and w' in which every occurrence of w' has been replaced by w and vice versa.*

Proof. Any case where any of the two wires are identical is trivial. Consider the sequence $< w'', w_0, ..., w_k, w >$. We consider the two following cases:

w' does not occur in the original sequence In this case $< w'', w_0, ..., w_k, w' >$ is a legitimate acyclic path between w'' and w'. w_k and w' are of alternating types and can be collocated, and w' only appears once in the whole sequence.

w' appears as a w_i In this case the original sequence could be written as $< w'', w_0, ..., w_{i-1}, w_i = w', w_{i+1}, ..., w_k, w >$. Note that w_i is the only place where w' occurs because it cannot appear more than once. The sequence $< w'', w_0, ..., w_{i-1}, w, w_{i+1}, ..., w_k, w' >$ is a legitimate acyclic path between w'' and w. w and w' still appear only once, and they collocate properly with the rest of the sequence.

□

2.2 Crossbars as Boolean Functions

So far we have considered the way in which the truth value of the wires is determined by those of the junctions and the initially enforced truths. Now we can extend our study to a case where each of those values are a function of some Boolean vector.

Definition 2.2.1. A Crossbar Function is a function that maps the set of natural numbers to the set of static crossbars. $C : \mathbb{N} \to CR$

Again, as in the case of the concept of crossbar size, the definition is quite abstract and further restrictions are necessary to make it applicable to actualities.

Definition 2.2.2. An n-ary Crossbar Function is one where

$$\forall x \geq 2^n : C(x) = 0$$

Here, 0 denotes the null static crossbar.

Apart from this, there are further non-obvious restrictions on how crossbars are fabricated. One would expect the "forced" values on the crossbar (i.e. the forced voltages and the junction values) to be simple in terms of the input. Common sense supports this to some extent since the availability of complex computations would defeat the purpose of trying to design crossbars altogether.

Definition 2.2.3. A simple n-ary Crossbar Function is one where for each function $F \in W_0 \cup M$ one of the following holds:

1. $F(x) = 0$ for all x
2. $F(x) = 1$ for all x
3. $F(x) = x_i$ for all x and some $i \in \mathbb{Z}_{0..n-1}$

4. $F(x) = \neg x_i$ for all x and some $i \in \mathbb{Z}_{0..n-1}$

where $x = \sum_{i=0}^{n-1} 2^i x_i$

2.3 Number of Acyclic Paths

This section studies the number of acyclic paths in a crossbar, partly demonstrating their potential for implementing a great number of functions.

2.3.1 Paths Between Two Rows

The path has the form $r_0, c_0, r_1, c_1, ..., c_{k-1}, r_k$ with r_0 and r_k being given. Counting the paths, we have to choose $k-1$ rows and k columns from the wires in the crossbar and place them in distinct positions. Therefore between any two given rows, there are $P(R-2, k-1) \times P(C, k)$. That is the number of Acyclic Paths between with k intermediary columns two Rows.
The minimum acceptable number for k is 1. The maximum is determined by the equations: $k-1 \leq R-2, k \leq C$. Therefore, $k \leq min\{C, R-1\}$.
Knowing that, we can say that the number of acyclic paths between two rows is:

$$Paths(r_i, r_o) = \sum_{k=1}^{min\{C,R-1\}} P(C,k)P(R-2,k-1)$$

Which means that there are $\binom{R}{2}$ pairs of rows as the ends of the path, there are

$$\binom{R}{2} \sum_{k=1}^{min\{C,R-1\}} P(C,k)P(R-2,k-1)$$

paths between rows in a crossbar.

Order of Magnitude:

$R-1=C$ **case** It is natural to take our crossbars to be more or less square-shaped.

$$f(k) := P(C,k)P(R-2,k-1)$$

$$Paths(r_i, r_o) = \sum_{k=1}^{C} f(k)$$

$$f(k) = C \times (P(C-1,k-1))^2$$

$$f(C) = C \times (P(C-1,C-1))^2 = C \times ((C-1)!)^2 = \frac{(C!)^2}{C} = \theta(\frac{CC^{2C}}{Ce^{2C}}) = \theta((\frac{C}{e})^{2C})$$

Lemma 2.3.1. *For an increasing non-negative series a_n, if there are constants $\delta_0, C > 0$ and $0 \leq \alpha < 1$ where for every natural number $\delta \geq \delta_0$ we have $a_{n-\delta} \leq C\alpha^{\delta} a_n$, we have*

$$\sum_{-\infty}^{N} a_n = \theta(a_N)$$

Proof. The sum is obviously $\Omega(a_N)$.

Additionally,

$$\sum_{-\infty}^{N} a_n = \sum_{-\infty}^{N-\delta_0} a_n + \sum_{N-\delta_0+1}^{N} a_n \leq \sum_{\delta_0}^{\infty} a_{N-\delta} + \sum_{N-\delta_0+1}^{N} a_N \leq \sum_{\delta_0}^{\infty} C\alpha^{\delta} a_N + \delta_0 a_N$$

$$= (\frac{C\alpha^{\delta_0}}{1-\alpha}) a_N = O(a_N)$$

□

Going back to considering the number of Sneak Paths, we have

$$\frac{f(C)}{f(C-\delta)} = \frac{C((C-1)!)^2}{C(\frac{(C-1)!}{\delta!})^2} = (\delta!)^2 \geq (C-K+1)^{\delta}$$

Taking $\delta_0 = 2$, we have $\frac{f(C)}{f(C-\delta)} \geq 4^{\delta-1} = \frac{1}{4}4^{\delta}$ and can say $Paths(r_i, r_o) = \theta(f(C)) = \theta((\frac{C}{e})^{2C})$

$R - 1 \leq C$ **case**

$$Paths(r_0, r, 1) \geq P(C, R-1)P(R-2, R-2) = (R-2)!\frac{C!}{(C-R+1)!}$$

In this case we will have

$$Paths(r_i, r_o) = \sum_{k=1}^{R-1} P(C,k)P(R-2, k-1)$$

$$f(k) = P(C,k)P(R-2,k-1) = \frac{C!(R-2)!}{(C-k)!(R-k-1)!}$$

$$\frac{f(R-1)}{f(R-1-\delta)} = \frac{(C-R+1+\delta)!\delta!}{(C-R+1)!} \geq \frac{1}{2(C-R+3)}(2(C-R+3))^{\delta}$$

Meaning that

$$Paths(r_i, r_o) = \sum_{k=1}^{R-1} f(k) = \theta(f(R-1)) = \theta(P(C, R-1)P(R-2, R-2)) = \theta(\frac{(R-2)!C!}{(C-R+1)!})$$

Applying Stirling's approximation to $C!$ we have:

$$Paths(r_i, r_o) = \theta(\frac{\sqrt{C}(R-2)!C^C}{(C-R+1)!e^C})$$

For $R = o(C)$, we also have $R = o(C-R)$ and $\sqrt{C-R+1} = \theta(\sqrt{C})$. Consequently:

$$Paths(r_i, r_o) = \theta(\frac{(R-2)!C^C e^{C-R+1}}{(C-R+1)^{C-R+1}e^C}) = \theta(\frac{(R-2)!C^C}{(C-R+1)^{C-R+1}e^{R-1}})$$

Now if $R = O(1)$, making the crossbar a thin tape, we would have:

$$Paths(r_i, r_o) = \theta(\frac{C^C}{(C-R+1)^{C-R+1}}) = \theta(\frac{C^{C+R-1}}{(C-R+1)^C})$$

$$= \theta((\frac{C-R+1+R-1}{C-R+1})^{C-R+1} \times C^{2R-2}) = \theta((1+\frac{R-1}{C-R+1})^{C-R+1} \times C^{2R-2})$$

$$= \theta(e^{R-1} \times C^{2R-2}) = \theta(C^{2R-2})$$

$R - 1 \geq C$ **case**

$$Paths(r_i, r_o) = \sum_{k=1}^{C} P(C,k)P(R-2,k-1)$$

$$f(k) = P(C,k)P(R-2,k-1) = \frac{C!(R-2)!}{(C-k)!(R-k-1)!}$$

$$\frac{f(C)}{f(C-\delta)} = \delta! \frac{(R-1-C+\delta)}{(R-1-C)!}$$

For $\delta_0 = 2$, we have $\frac{f(C)}{f(C-\delta)} \geq \frac{1}{2}2^\delta$, therefore:

$$Paths(r_i, r_o) = \theta(f(C)) = \theta(\frac{C!(R-2)!}{(R-C-1)!})$$

The Stirling approximation on $(R-2)!$ would yield:

$$Paths(r_i, r_o) = \theta(\frac{C!(R-2)^{(R-2)}\sqrt{R-2}}{(R-C-1)!e^{(R-2)}}) = \theta(\frac{C!(R-2)^{R-2}\sqrt{R}}{(R-C-1)!e^R})$$

Going further, if $C = o(R)$, we have $C = o(R-C-1)$ and $\sqrt{R-C-1} = \theta(\sqrt{R})$:

$$Paths(r_i, r_o) = \theta(\frac{C!(R-2)^{R-2}e^{R-C-1}}{(R-C-1)^{R-C-1}e^R}) = \theta(\frac{C!(R-2)^{R-2}}{(R-C-1)^{R-C-1}e^C}) =$$

$$\theta(\frac{C!(R-2)^{R+C-1}}{(R-C-1)^R e^C})$$

And if $C = O(1)$,

$$Paths(r_i, r_o) = \theta(\frac{(R-2)^{R-2}}{(R-C-1)^{R-C-1}}) = \theta((\frac{R-C-1+C-1}{R-C-1})^{R-C-1} \times (R-2)^{C-1})$$

$$= \theta((1 + \frac{C-1}{R-C-1})^{R-C-1} \times (R-2)^{C-1}) = \theta(e^{C-1} \times (R-2)^{C-1}) = \theta((R-2)^{C-1})$$

2.3.2 Paths Between Two Columns

A similar analysis would yield $P(C-2, k-1) \times P(R, k)$ paths with k intermediary rows between any two given columns,

$$\sum_{k=1}^{min\{R,C-1\}} P(R,k)P(C-2,k-1)$$

paths of all possible lengths between two given columns and

$$\binom{C}{2} \sum_{k=1}^{min\{R,C-1\}} P(R,k)P(C-2,k-1)$$

inter-columnar paths.

Similarly to the case of paths between two rows, we have the following cases:

$C - 1 = R$ **case** And in a somewhat square crossbar with $C - 1 = R$, we can say the order of magnitude of the number of paths between any two given columns are,

$$Paths(c_i, c_o) = \theta((\frac{R}{e})^{2R})$$

$C - 1 \leq R$ **case** Generally,

$$Paths(c_i, c_o) = \theta(\frac{\sqrt{R}(C-2)!R^R}{(R-C+1)!e^R})$$

If $C = o(R)$,

$$Paths(c_i, c_o) = \frac{(C-2)!R^R}{(R-C+1)^{R-C+1}e^{C-1}}$$

If $C = O(1)$,

$$Paths(c_i, c_o) = \theta(R^{2C-2})$$

$C - 1 \geq R$ **case** Generally,

$$Paths(c_i, c_o) = \theta(\frac{R!(C-2)^{C-2}\sqrt{C}}{(C-R-1)!e^C})$$

If $R = o(C)$,

$$Paths(c_i, c_o) = \theta(\frac{R!(C-2)^{R+C-1}}{(C-R-1)^C e^R})$$

And if $R = \theta(1)$,

$$Paths(c_i, c_o) = \theta((C-2)^{R-1}) = \theta(C^{R-1})$$

2.3.3 Paths Between a Row and a Column

Without loss of generality, we assume w_0 to be a row and w_{-1} to be a column. The path would have the form $r, c_0, r_0, ..., c_{k-1}, r_{k-1}, c$. Therefore, assuming k intermediary column-row pairs, there are $P(R-1,k)P(C-1,k)$ paths between the wires. Therefore, there are $P(R, k+1)P(C, k+1)$ paths with k intermediary column-row pairs in the entire crossbar. There is

$$\sum_{k=0}^{min\{R-1,C-1\}} P(R, k+1)P(C, k+1)$$

paths between rows and columns in the entire crossbar.

Order of magnitude: Let us again concern ourselves with the number of paths between a given row and a given column.

$R = C$ **case**

$$Paths(r_i, c_o) = \sum_{k=0}^{R-1} P(R-1,k)P(C-1,k) = \sum_{k=0}^{R-1} f(k),$$

where $f(k) = P(R-1,k)^2$.

$$\frac{f(R-1)}{f(R-1-\delta)} = (\frac{P(R-1,R-1)}{P(R-1,R-1-\delta)})^2 = (\delta!)^2$$

Fast growing rate shows that

$$Paths(r_i, c_o) = \theta(f(R-1)) = \theta(P(R-1,R-1)^2) = \theta(((R-1)!)^2)$$

$$= \theta(\sqrt{R}(R-1)^{R-1}e^{-(R-1)})$$

$R \leq C$ **case**

$$Paths(r_i, c_o) = \sum_{k=0}^{R-1} P(R-1,k)P(C-1,k) = \sum_{k=0}^{R-1} f(k),$$

where $f(k) = P(R-1,k)P(C-1,k)$.

$$\frac{f(R-1)}{f(R-1-\delta)} = \frac{\delta!(C-R+\delta)!}{(C-R)!}$$

It is a fast growing function, therefore

$$Paths(r_i, c_o) = \theta(f(R-1)) = \theta(\frac{(R-1)!(C-1)!}{(C-R)!}) = \theta(\frac{\sqrt{C}(R-1)!(C-1)^{C-1}}{(C-R)!e^C})$$

$$= \theta(\frac{\sqrt{C}(R-1)!(C-1)^C}{(C-1)(C-R)!e^C}) = \theta(\frac{(R-1)!(C-1)^C}{\sqrt{C}(C-R)!e^C}) = \theta(\frac{(R-1)!C^C(1-\frac{1}{C})^C}{\sqrt{C}(C-R)!e^C})$$

$$= \theta(\frac{(R-1)!C^C}{\sqrt{C}(C-R)!e^C}) \times \theta((1-\frac{1}{C})^C) = \theta(\frac{(R-1)!C^{C-\frac{1}{2}}}{(C-R)!e^{2C}})$$

Further, if $R = o(C)$,

$$Paths(r_i, c_o) = \theta(\frac{(R-1)!C^{C-\frac{1}{2}}}{(C-R)!e^{2C}}) = \theta(\frac{e^{C-R}(R-1)!C^{C-\frac{1}{2}}}{\sqrt{C}(C-R)^{C-R}e^{2C}}) = \theta(\frac{e^{C-R}(R-1)!C^{C-1}}{(C-R)^{C-R}e^{2C}})$$

$$= \theta(\frac{(R-1)!C^{C-1}}{(C-R)^{C-R}e^{C+R}})$$

And if $R = \theta(1)$,

$$Paths(r_i, c_o) = \theta(\frac{C^{C-1}}{(C-R)^{C-R}e^{C}}) = \theta(C^{-1}(C-R)^{R}e^{-C}) \times \theta((\frac{C-R}{C})^{-C})$$

$$= \theta(C^{-1}(C-R)^{R}e^{-C}) \times \theta((1-\frac{R}{C})^{-C}) = \theta(C^{-1}(C-R)^{R}e^{-C}) \times \theta(e^{R})$$

$$= \theta(C^{-1}(C-R)^{R}e^{-C}) = \theta(C^{R-1}e^{-C})$$

$R \geq C$ **case** Symmetrically,

$$Paths(r_i, c_o) = \theta(\frac{(C-1)!R^{R-\frac{1}{2}}}{(R-C)!e^{2R}})$$

In cases where $C = o(R)$,

$$Paths(r_i, c_o) = \theta(\frac{(C-1)!R^{R-1}}{(R-C)^{R-C}e^{C+R}})$$

And if $C = \theta(1)$,

$$Paths(r_i, c_o) = \theta(R^{C-1}e^{-R})$$

2.4 Lower Bounds on Crossbar Size

In this section we will go through several theorems indicating the minimum size of a crossbar for implementing a function, or a class of functions.

Definition 2.4.1. A *void path* is an acyclic path along which a pair of incompatible literals or a 0

exists. By incompatible literals we mean x and $\neg x$ for some binary variable x. In other words, a non-void path is one whose literals make up a well-formed conjunction.

Theorem 2.4.2. *In a crossbar implementing a Boolean formula, the junction labels along any non-void path from the supply wire to the target wire should be a superset of the literals in some prime implicant of the target function.*

Proof. If a non-void path exists over the set of literals C, it means that the implemented Boolean function evaluates to 1 whenever all literals in L hold. The conjunctive clause $\wedge_{l \in C} l$ corresponding to this path is therefore an implicant of f, and is therefore a subset of some prime implicant of f. □

Theorem 2.4.3. *For implementing a function having a d-ary prime implicant, a crossbar of size $(d+1) \times d$ or more is required.*

Proof. It has been established in 2.3.1 that the longest row-to-row Sneak Path in an $R \times C$ crossbar has $\min\{R-1, C\}$ junctions. Since a path of length at least d is required, we will have:

$$\min\{R-1, C\} \geq d \Rightarrow R \geq d+1 \text{ and } C \geq d$$

□

Theorem 2.4.4. *A Lower Bound on the Number of Functions Represented by a Crossbar*

Proof. Any $R \times C$ crossbar belongs to a class of $(R-2)! \times C!$ crossbars all implementing the same function. With there being $(2|I|+2)^{RC}$ different crossbar designs, there is at most $\frac{(2|I|+2)^{RC}}{(R-2)!C!}$ distinct functions implemented by such crossbars. In order for a crossbar space to be able to implement every Boolean function of $|I|$ inputs, we should have:

$$\frac{(2|I|+2)^{RC}}{(R-2)!C!} \geq 2^{2^{|I|}} \Leftrightarrow RC + RC\log(|I|+1) \geq \log\left[(R-2)!\right] + \log(C!) + 2^{|I|}$$

Asymptotically,

$$RC + RC\log(|I|+1) = O\Big(\log\big[(R-2)!\big] + \log(C!) + 2^{|I|}\Big)$$

$$\Rightarrow RC + RC\log(|I+1|) = O\big(\log(R!) - 2\log R + \log(C!) + 2^{|I|}\big) = O(R\log R + C\log C + 2^{|I|})$$

□

Using the result from 2.4, we can compute a lower bound for the size of a crossbar that is capable of implementing *all* Boolean functions of size $|I|$. It is important to take into account the limitations of . Firstly, the fact that not all functions of, say, 9 input variables can be implemented on a 10×10 crossbar does not mean that none of them can. The function $x_1x_2...x_9$ which non-trivially depends on all input variables can be implemented on a 10×10 crossbar. Furthermore, exceeding the lower bounds implied by 2.4 does not guarantee the possibility of the crossbar size accommodating all Boolean functions. For example, for $|I| = 3$ it is implied by 2.4.3 that a 4×4 crossbar is required to implement $x_1x_2x_3$.

Table 2.1: Lower Bounds on Square Crossbar Size Supporting All Boolean Functions

Size of Input Vector	Minumum Crossbar Width Required
2	2
3	2
4	3
5	4
6	5
7	6
8	9
9	12
10	16
11	22
12	31
13	43
14	59
15	82
16	115

CHAPTER 3: VERIFICATION OF CROSSBARS

In this chapter we discuss various methods for verifying a crossbar against a target function, possibly but not necessarily constructing a representation of the functions implemented by the crossbar itself.

3.1 The Naive Method

This temporal-logic based approach describes a set of transition rules corresponding to inductive inference. It describes the value of a wire given an input vector through an eventuality proposition. The automaton can be described as follows:

- The state vector: $\langle r_0, ..., r_{R-1}, c_0, ..., c_{C-1} \rangle \in \mathbb{B}^{R+C}$ with each term representing the conductance state of its corresponding wire.
- The junction conductance matrix $[M_{ij}]$, with the value of the junctions being a function of the input vector, but fixed throughout the evaluation against that vector.
- Initial state: $s_0 = \langle 1, 0, ..., 0 \rangle$
- The transition rules:

$$Xr_i \Leftrightarrow r_i \vee \left(\exists j : m_{ij} \wedge c_j\right)$$

$$Xc_j \Leftrightarrow c_j \vee \left(\exists i :_{ij} \wedge r_i\right)$$

- Interpretation: $\hat{f}(I) = 1 \Leftrightarrow Fr_{R-1}$. More generally, for each wire w, $\hat{f}_w(I) = 1 \Leftrightarrow Fw$

One may note the structural similarities with an inductive argument.

A naive implementation of this method would require $O(RC(R+C)2^{|I|})$ running time. However, this mode of analysis provides a rigorous and simple method of formal analysis, much

simpler than what we developed in chapter 3.

Also, because of its simplicity and being fully predicated, this formulation of the problem can be fed into a SAT solver, essentially fusing synthesis and verification into a single process. This allows the SAT solver to intelligently apply optimizations on the process.

This approach was pioneered by Velasquez and Jha[1], and most designs presented in chapter 2 were obtained from various implementations of the same method.

3.2 Iterated Matrix Multiplication

The previous approach can be formulated in terms of Boolean Linear Algebra. The existential quantifiers in the transition rules have a fixed and finite domain, therefore they can be statically expanded as:

$$r_i(t+1) = r_i(t) + \sum_{j=0}^{C-1} m_{ij} c_j(t) = \sum_{j=0}^{C-1} m_{ij} c_j(t)$$

$$c_j(t+1) = c_j(t) + \sum_{i=0}^{R-1} m_{ij} r_i(t) = \sum_{i=0}^{R-1} m_{ij} r_i(t)$$

This can be reiterated in the matrix form as:

$$\begin{bmatrix} r_0 \\ \vdots \\ r_{R-1} \\ c_0 \\ \vdots \\ c_{C-1} \end{bmatrix} (t+1) = \left[\begin{array}{ccc|ccc} 0 & \cdots & 0 & m_{00} & \cdots & m_{0(C-1)} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ 0 & \cdots & 0 & m_{(R-1)0} & \cdots & m_{(R-1)(C-1)} \\ \hline m_{00} & \cdots & m_{(R-1)0} & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ m_{0(C-1)} & \cdots & m_{(R-1)(C-1)} & 0 & \cdots & 0 \end{array}\right] \begin{bmatrix} r_0 \\ \vdots \\ r_{R-1} \\ c_0 \\ \vdots \\ c_{C-1} \end{bmatrix} (t) =$$

$$\left[\begin{array}{c|c} 0 & M \\ \hline M^T & 0 \end{array}\right] X(t)$$

Rewritten through renaming as

$$X(t+1) = M_T X(t) = (M_T)^{t+1} X(0)$$

, where $X(0) = \begin{bmatrix} 1 & 0 & \cdots & 0 \end{bmatrix}^T$, and the multiplication and addition operators correspond to Boolean conjunction and disjunction, respectively.

The sequence $X(.)$ is monotonically increasing, and it is bounded by $\begin{bmatrix} 1 & \cdots & 1 \end{bmatrix}^T$. Therefore it is convergent, and the result can be expressed as $\hat{f}(I) = \lim_{t\to\infty}(M_T)^t X(0)$. The sequence converges in fewer steps than $m+n$, and each multiplication operator takes $O\big((R+C)^2\big)$ time. Therefore, computing the output to any single input vector should take $O\big((R+C)^3\big)$ time, the same level of complexity as the previous method.

Remark 3.2.1. One might think that implementing this formulation should necessarily perform worse than a naive implementation of the original temporal logic formulation. That is not the case. Smaller matrices could be used to transform the state transition relation into a more compact form:

$$\lim_{t\to\infty} X_{0..(R-1)}(t) = \lim_{t\to\infty} (MM^T)^t X_{0..(R-1)}(0)$$

which would take $O\big(R^2(R+C)\big)$.

In fact, it can also be rewritten as

$$\lim_{t\to\infty} M(M^T M)^t M^T X_{0..(R-1)}(0)$$

which takes $O(C^2(R+C))$ time. Therefore, it can performs no worse and in fact outperforms the trivial method in cases where R and C are not of the same order.

The equivalent formula $\hat{f}(I) = \lim_{t\to\infty}(M_T)^{2^t} X(0)$ converges in $O(\log(R+C))$. Unfortunately, because the addition operator in the Boolena field is irreversible, the Strassen optimization

cannot be applied and each exponentiation should take $O((R+C)^3)$ time.

Remark 3.2.2. Applying the same type of optimization as discussed in 3.2.1, we would obtain a $O(\min\{R,C\}^3 \log(R+C)$ runtime, that would outperform the method in 3.1 if and only if the crossbar is narrow (meaning one of the dimensions is considerably larger than the other).

Remark 3.2.3. All runtimes addressed in this subsection apply to the evaluation of the crossbar with respect to a single input vector. Therefore, they all should be multiplied by a factor of 2^i in order to specify the runtime of a full crossbar verification.

3.3 Graph-based Verification

Another formulation of evaluation is as a graph reachability problem. The crossbar can be modeled as a bipartite graph, with nodes corresponding to wires and edges corresponding to junctions. The lack of direct connection among rows and among columns justifies the bipartite assumption. Rows can be placed on one side and columns on the other. A wire evaluates to 1 if and only if its corresponding node is reachable from r_0's node.
This approach yields better results from previously considered algorithms, having $O(|E|) = O(RC)$ worst-case time complexity and $O(|V|) = O(R+C)$ worst-case space complexity, if trivial Breadth-First or Depth-First algorithms are used.

3.4 BDD-based Symbolic Verification

The algorithms hitherto presented all view crossbars and target functions as functions. The following algorithm views those as data, represented by Reduced Ordered Binary Decision Diagrams.

3.4.1 Preliminaries

3.4.1.1 Probably Inevitable Exponential Time

Theorem 3.4.1. *Any representation scheme of Boolean functions with $|I|$ inputs is of $\Omega(2^{|I|})$ on the worst case.*

Proof. There are $2^{2^{|I|}}$ distinct functions mapping $\{0,1\}^{|I|}$ to $\{0,1\}$. The depth of any DAG with $2^{2^{|I|}}$ distinct leaves is at least $\log_2 2^{2^{|I|}} = 2^{|I|}$ □

This loosely suggests that verifying crossbars is inherently difficult. Assuming (unjustifiably) that a crossbar of a certain size is able to implement all binary functions over input vectors of size i, its verification should be $\Omega(2^i)$ worst-case, simply because certain outputs of the algorithm have to be of that size.

3.4.1.2 The Gist of a Junction

We base our algorithm on a simple observation, that the entire role of a simple (referring to a literal) junction in the composition of the crossbar function is that it equates certain a cofactor (restrictions) of the two wires it connects. Therefore, a junction can have either of the three effects based on the function through which it is set:

1. If $m_{mn} = 1$ for all input vectors, r_m = c_n for all input vectors.

2. If $m_{mn} = x_i$ (or $\neg x_i$) for some input atom x_i, r_m and c_n are equal over a half of the input values, namely, the cases where x_i ($\neg x_i$) holds. Another way to express this is to say that the cofactors (restrictions) of r_i and c_j under $x_i = 1$ or 0) are equal. Formally,

$$(r_m)|_{x_i=1} = (c_n)|_{x_i=1}$$

, likewise for $x_i = 0$.

3. If $m_{mn} = 0$ for all inputs, this junction bears no information. This also does not mean that they are necessarily *NOT* equal. There can be a case when two totally disconnected rows can have the exact same functions.

   Example 3.4.2. In the crossbar below,

   $$\begin{array}{cc} 1 & 0 \\ A & A \end{array} \quad \begin{array}{l} \\ r_0 \rightarrow \end{array}$$

   r_1 and c_1 implement the same function (A) in spite of the fact that they are in no direct contact with each other. That fact is caused by other junctions, not the one connecting the two.

Remark 3.4.3. It is not always straightforward to compute a restriction of a Ordered Binary Decision Diagram. It is only so when the restricting variable is on the top level.

3.4.2 The Symbolic Verification Algorithm

This algorithm iteratively applies a slightly modified version of the Restrict operation as originally suggested by Bryant[2], and a new operation named Equate. The Equate function is run on every junction, equating in each iteration the relevant restrictions of the wires it connects, based on its label.

Our algorithm will proceed working on "incomplete" ROBDDs, therefore a few notions have to be introduced:

- A "white" node is an incomplete node without an associated decision variable or edges.
- A node implements an "incomplete" function if and only if there is a white node descending from it. Such a node can itself be called incomplete as well.
- The "depth" of a node is defined as the variable following its parent's decision variable if it is a white node and its own decision variable otherwise.

3.4.2.1 Equate and Modified Reduce Algorithms

The reduce algorithm will be modified by simply adding the following assumption:

Any white node is considered distinct from any other node.

This means that in presence of uncertainty, the reduction procedure should act conservatively.

The Equate algorithm is based on the original Restrict algorithm, and it operates as follows:

function EQUATE(n_0, n_1, x, v)

Ensure: $n_0|_{x=v} = n_1|_{x=v}$

if depth(n_0) $>$ depth(n_1) **then**

SWAP(n_0, n_1) ▷ Acts only on the references, not the actual nodes

end if

if $n_0 = n_1$ **then return** success ▷ Any two equal functions share $x = v$ cofactors

else if depth(n_0) $\leq x$ **then**

if n_0 is white **then**

LABELANDCREATECHILDREN(n_0)

end if

if depth(n_1) =depth(n_0) **then**

if n_1 is white **then**

LABELANDCREATECHILDREN(n_1)

end if

$(n_{11}, n_{10}) \leftarrow (\text{high}(n_1), \text{low}(n_1))$

else

$(n_{11}, n_{10}) \leftarrow (n_1, n_1)$

end if

if depth(n_0) $< x$ **or** $v = 1$ **then**

EQUATE(high(n_0), high(n_1), x, v)

end if

if depth(n_0) $< x$ **or** $v = 0$ **then**

EQUATE(low(n_0), low(n_1), x, v)

end if

return success **if** all calls returned success, failure otherwise

else if depth(n_0) $> x$ and n_0 is a white node **then**

$n_0 \leftarrow n_1$

REDUCE(n_0)

return success

else if depth(n_0) $> x$ and n_0 is not a white node **then return** failure

else if n_0 and n_1 are not white **then return** failure

end if

end function

3.4.2.2 Correctness

The correctness of the algorithm can be established by proving two propositions:

1. The Equate algorithm presented in 3.4.2.1 is correct.

2. $R \times C$ equal cofactor statements are sufficient description for the crossbar.

Theorem 3.4.4. *In an ROBDD derived from the iterative cofactor equating process, the* 1 *terminal node is reached from a wire root on an input vector* I *if and only if* r_0 *and wire* w *in the corresponding crossbar are connected under* I*.*

Proof. Assuming the correctness of the modified Reduce algorithm, the forward direction is straightforward:

If r_0 and w are connected through a sequence of conducting junctions $(m_{i_n j_n})$, there will be a

sequence of corresponding cofactor propositions $w^n_{|x_n=a_n} = w^{n+1}_{|x_n=a_n}$, where $w^0 = r_0$ and $w^{n_0} = w$ for some n_0, and every $x_n = a_n$ actually holds under I. Therefore, for the function values of the wires under I, we have, $r_0(I) = w^0(I) = ... = w^{n_0}(I) = w(I)$.

For the converse, assuming the correctness of the modified Reduce algorithm, anything resulting from reasoning based on the ROBDD can also be obtained from the same OBDD had no reduction occurred. In that scenario, any path from a root node w to the terminal 1, goes through decision nodes equating the cofactors of two adjacent wires. □

3.4.2.3 Performance Analysis

The algorithm iterates over all junctions, performing one Equate operation each time. This means that it is $O(mn2^{|I|})$ in the worst case. This is not better than the graph-based verification algorithm presented in 3.3. However, the actual amount of time needed to perform a Equate operation is bounded depends on the size of the OBDD representations of the two wire functions being equated and is not necessarily equal to $2^{|I|}$.

CHAPTER 4: AUTOMATED SYNTHESIS OF DENSE CROSSBARS

In this chapter we consider how all the analytic and algorithmic tools developed hitherto can contribute to automatically designing memristor crossbars implementing target Boolean functions.

Fundamentally, all the methods for constructing "dense" circuits (i.e. those with more than an $O(\frac{1}{\sqrt{RC}})$ of its memristors being non-zero) are based on some form of directed search.

A Brute Force exhaustion of the design space is not possible, because of its colossal size. A simple calculation demonstrates how quickly the design space can grow:

There are $R \times C$ junctions in a crossbar. Each of those crossbars can take one of $2|I| + 2$ values. This would mean that the number of possible designs sums up to $(2|I| + 2)^{RC}$. This necessitates one of the following approaches:

1. Trying to apply heuristics to guide and restrict the search, as discussed in chapter 1
2. Abandoning the idea of dense crossbars entirely, and systematically designing sparse crossbars. This method is not covered in this thesis.
3. Settling for an approximation of the target function, one that agrees with it on a majority of inputs. Such functions are much more numerous and therefore more likely to be found even in a purely randomized search.

Every synthesis algorithm is coupled with a verification algorithm that gives it feedback on a measure of agreement between the crossbar and the target function. The feedback can be of a binary equal / not equal, or perhaps more helpfully, a ratio describing how many of the crossbar's outputs match that of the target function.

4.1 Exact Methods - SAT Solver Based

As previously mentioned, this method combines the verification and synthesis processes in a single description. The SAT solver is told the rules through which it can infer whether or not a design matches a target function, and is asked to reverse-engineer that inference process, finding a design that satisfies the condition (that of matching the output function completely), or report that no such design exists.

The advantage of this method is that it can make use of all the pruning power of an established general purpose SAT solver. SAT solvers have been under development for a long time, and they have yielded impressive results solving SAT problems with hundreds of variables.

The same issue, namely the use of a general purpose tool may also be this method's downfall. There is reasonable speculation that because of the narrow scope of the crossbar design problem, less overall sophisticated but more specialized tools would outperform the SAT solver based methods. We believe our work provides evidence to support this claim.

4.2 Approximate Methods

This approach views crossbar design as an optimization problem. The objective is to maximize through iteration, the fit of the crossbar with respect to the target function. The fitness measure obtained from the verifier at each iteration is used to direct the search for an optimum with making a small change to the crossbar. The legitimacy of this method is not indisputable, because there is no conclusive evidence that fitness functions act "continuously", meaning that there might be no strong connection between being close to an optimal crossbar design and having a high fitness measure. Despite this, empirical evidence leans towards affirming the usefulness of this approach.

4.2.1 Simulated Annealing

We experimented with randomized search using Simulated Annealing, and the following results were obtained for 15×15 crossbars implementing functions with 8 inputs, something not realized by previous methods.

Example 4.2.1. 3-Bit Adder

$$\begin{array}{cccccccccc}
\overline{A_0} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \overline{A_0} & \overline{B_0} & \overline{B_2} & A_0 & 0 \\
A_1 & B_0 & B_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \overline{A_2} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & B_0 \\
A_2 & 0 & B_2 & 0 & 0 & 0 & 0 & B_2 & 0 & 0 \\
0 & 0 & 0 & 0 & B_2 & 0 & 0 & 0 & B_2 & 0 \\
0 & B_0 & 0 & 0 & B_2 & 0 & 0 & 0 & 0 & 0 \\
B_1 & A_0 & A_1 & 0 & 0 & 0 & B_2 & 0 & 0 & 0 \\
0 & 0 & A_2 & 0 & 0 & 0 & A_1 & \overline{A_1} & 0 & \overline{A_0}
\end{array}$$

This result was obtained in less than 2 minutes.

Example 4.2.2. 4-Bit Carry-Out

This circuit computes whether two 4-bit numbers sum up to 16 or more. It can serve as a component of a 4-bit adder.

0	0	0	0	0	0	0	0	0	B_1	0	1	0	0	$\overline{B_3}$
0	$\overline{B_0}$	0	0	0	0	0	B_2	0	0	0	A_2	0	0	0
B_2	1	1	0	0	0	0	0	A_2	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	A_3	0	0	0	0	0
B_1	0	B_2	0	0	0	0	0	B_0	0	$\overline{A_2}$	0	0	0	0
0	0	B_2	0	0	0	0	0	0	0	0	A_2	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
B_1	0	0	$\overline{B_3}$	0	0	0	A_2	0	0	0	0	B_2	0	$\overline{A_3}$
0	0	0	A_3	$\overline{A_0}$	0	0	0	0	0	0	B_3	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	B_0	0	0	0	0	0	0	0	0	A_0	0	0	0
0	0	0	0	0	0	0	A_1	0	B_1	0	0	$\overline{B_1}$	0	0
0	0	$\overline{A_3}$	0	0	0	0	0	0	0	0	0	0	0	0
0	0	B_0	0	0	0	0	A_1	0	0	0	0	0	$\overline{B_2}$	0
0	0	0	A_3	0	0	0	0	0	0	0	0	0	0	B_3

A design with such large dimensions and input vector size was obtained in only one hour. However, a smaller design was also found:

0	0	A_3	0	B_3
0	B_1	B_2	A_1	A_2
$\overline{A_1}$	B_0	0	A_0	0
A_0	A_1	A_2	0	B_2
0	0	B_3	0	$\overline{B_3}$

LIST OF REFERENCES

[1] A. Velasquez, S. K. Jha. *Automated Synthesis of Computing Nanoscale Crossbars using Formal Methods* . IEEE/ACM International Symposium on Nanoscale Architectures, 2015.

[2] R. E. Bryant. *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers, 1986.

[3] L. Chua. *Memristor - The Missing Circuit Element*. IEEE Transactions on Circuit Theory, 1971.

[4] Stirling's Approximation,
`http://mathworld.wolfram.com/StirlingsApproximation.html`