Equivalence Checking of Majority-based Function Mapping on ReRAM Crossbars

*Arighna Deb, **† Kamalika Datta,**† Rolf Drechsler
*School of Electronics Engineering, KIIT DU, Bhubaneswar, India
**Institute of Computer Science, University of Bremen, Bremen, Germany
† German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany
airghna.debfet@kiit.ac.in,{kdatta,drechsler}@uni-bremen.de

Abstract

Recent developments in *Resistive Random Access Memory* (ReRAM) technology have led to the fabrication of large-scale crossbar structures. The processor-memory speed gap in conventional computer architectures can be bridged using in-memory computing on ReRAM crossbars, with suitable mitigation of unwanted sneak path currents. Synthesis of Boolean functions and mapping them to such crossbars have been investigated by researchers. However, very little effort has been put in towards verification of such mapping approaches. For smaller designs, the verification of mapping is typically carried out through manual inspection and simulation. This is an important problem to address as real world designs are complex and require proper design verification. As such manual inspection and simulation based methods for larger designs are not practical. In this summary paper, we report an automated equivalence checking approach for majority-based in-memory designs on ReRAM crossbars, which was published recently. First, we introduce an intermediate data structure called *ReRAM Sequence Graph* (ReSG) to represent the logic-in-memory operations, which in turn is translated into *Boolean Satisfiability* (SAT) formulas. These formulas are verified against the original function specification using Z3 *Satisfiability* solver. The proposed approach has been validated by running on widely available benchmarks.

1 Introduction

Resistive Random Access Memory (ReRAM) or memristor [1] is an emerging technology that has opened up new possibilities in circuit design. In-memory computing on ReRAM crossbars (in which several ReRAM devices are arranged in a two-dimensional array structure) can help to bridge the processor-memory speed gap of conventional computing. Typically, the mapping approaches translate a given functional specification into a sequence of low-level micro-operations that can be directly executed on the crossbar [2, 3, 4]. The metrics commonly used to compare the approaches are: (a) number of cycles, (b) size of the crossbar, and (c) energy consumption to realize a function. Needless to say the complexity increases with increase in the size of the function. Most of these methods do not consider sneak path issues during crossbar operations, which is an important problem to address.

The existing mapping methods are based on an approach where correctness of the process is ensured for smaller circuit modules, and larger circuits comprising of the smaller modules are assumed to be correctly mapped. However, this process is not complete and does not cover all possible scenarios. This emphasizes the need for equivalence checking to prove that the crossbar micro-operations generated from the mapping tools are equivalent to the original functional specification. Thus far, few verification approaches exist [5, 6] that can check whether a mapping algorithm correctly realizes the desired function on ReRAM crossbars, leaving the verification of ReRAM-based designs largely unexplored.

In this paper, we present an automated equivalence checking methodology for majority-based in-memory designs on ReRAM crossbars[1]. In particular, we systematically verify the micro-operations performed on the crossbar against the golden functional specification as *Boolean Satisfiability* (SAT) formulas. For this purpose, we derive a *ReRAM sequence graph* (ReSG) from the logic-in-memory designs represented as crossbar micro-operations and then translate the ReSG into SAT formulas. These SAT formulas are verified against the original functional specification using Z3 solver. We validate our proposed method on several benchmark functions.

The rest of the paper is organized as follows. In Section 2, we present a brief background on ReRAM crossbars and *Boolean Satisfiability* (SAT). In Section 3, we present the design and verification methodology. Section 4 summarizes the experimental results. Finally, we conclude the paper in Section 5.

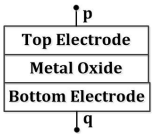
2 Background

In this section we briefly discuss about the ReRAM device and crossbar, and logic operations that can be performed on the crossbar. We also briefly discuss about the SAT.

[1] A more detailed version of this work is available in [6]

2.1 Resistive Random Access Memory (ReRAM)

A ReRAM is a resistive memory device that consists of an oxide layer sandwiched between two metal electrodes (p,q) in a Metal-Insulator-Metal structure as shown in Fig. 1(a). Such a device can be set to a low resistance state (LRS or logic 1) or a high resistance state (HRS or logic 0) by applying a suitable voltage across the device terminals. The behavior of the ReRAM device is shown in Fig. 1(b), where the values 0 and 1 at terminals p and q represent the voltages required to switch the internal state r. Essentially, the ReRAM device realizes the function $r_n = f(p,q,r) = p\overline{q} + pr + \overline{q}r$, where r_n denotes the next state of the ReRAM device [7]. Fig. 1(c) shows the symbolic representation of a ReRAM device. Such devices are typically laid out in a compact fashion as a crossbar structure as shown in Fig. 1(d), where p and q terminals of ReRAM devices are connected to the vertical and horizontal wires, respectively, of the crossbar. A vertical wire (or p terminal) is called a *bitline* or *column-line* and a horizontal wire (or q terminal) is called a *wordline* or *row-line*.



(a) ReRAM structure

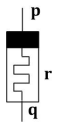
p	q	r	r_n
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0

$r_n = p \wedge \overline{q}$

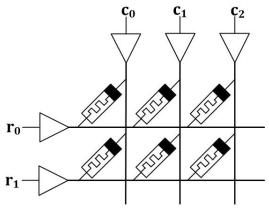
p	q	r	r_n
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

$r_n = p \vee \overline{q}$

(b) ReRAM operation [7]



(c) Circuit symbol

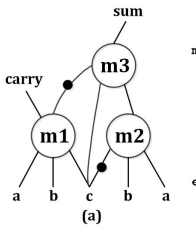


(d) ReRAM crossbar

Figure 1 ReRAM device and crossbar structure

To realize a Boolean functions on the crossbar, we sequentially execute several *micro-operations* in the crossbar depending on the given function representation, e.g. *Majority-Inverter Graph* (MIG) [8]. The micro-operations are performed by traversing each node in the corresponding MIG. Each node in a MIG (viz. MAJ3) realizes a 3-input majority function $f(a,b,c) = ab + ac + bc$.

Example 1 *Consider a full adder that takes three binary inputs a, b and c, and generates two outputs sum* $= a \oplus b \oplus c$ *and carry* $= ab + bc + ca$*. We express the sum and carry functions as a MIG as shown in Fig. 2(a), which essentially depicts a netlist comprising of three MAJ3 nodes (denoted as circles) and two inverters (denoted as solid dots). Fig. 2(b) depicts the equivalent Verilog description of the MIG structure. To map the given MIG to a crossbar circuit (of Fig. 1(d)), we traverse the MIG in a breadth-first manner and realize node* $m1$ *as a sequence of operations that include realization of* $\overline{b}$ *in crossbar located at row 1 (r1) and column 0 (c0) (denoted as 1x0) by applying input b and logic 1 (TRUE) at wordline 1 (row 1) and bitline 0 (column 0) respectively, followed by the realization of node* $m1$ *in the crossbar located at 1x2 (i.e. row 1 (r1) and column 2 (c2)) by applying input a and* $\overline{b}$ *to wordline 1 and bitline 2 respectively, provided the device at 1x2 is already initialized to input c (by applying logic 0 (FALSE) and input c at wordline 1 and bitline 2 respectively). In a similar manner, remaining nodes of the MIG are realized as a set of operations. The complete crossbar micro-operations realizing full adder are shown in Fig. 3(a). The micro-operations in lines 5-10 in the format (*`w x y z`*) indicate that the signal value x is applied to row w, and signal value z is applied to column y, of the crossbar. When we need to apply the state of a cell to a row or column, the crossbar controller reads the cell value and applies the corresponding voltage to the row or column. Further, the execution of each micro-operation leads to the realization of sub-functions in the crossbar as depicted in Fig. 3(b).*



(a)

```
module top(a, b, c, carry, sum);
  input a, b, c;
  output carry, sum;
  wire m1, m2;
  assign m1 = (a & b) | (b & c) | (c & a);
  assign m2 = (a & b) | (b & ~c) | (~c & a);
  assign carry = m1;
  assign sum = (c & ~m1) | (~m1 & m2) | (m2 & c);
endmodule
```

(b)

Figure 2 Full adder: (a) MIG, (b) Equivalent verilog code

```
1  .input a b c
2  .output carry sum
3  0 0 \a 1 \b 2 \c
4  1 TRUE 0 FALSE 1 FALSE 2 FALSE
5  1 0x1 0 TRUE
6  1 FALSE 2 0x2
7  1 1x0 2 0x0
8  1 FALSE 1 0x1
9  1 0x2 1 0x0
10 1 1x2 1 0x2
11 \sum 1x1
12 \carry 1x2
```

(a)

Steps	Rows	Columns 0	1	2
3	0	a	b	c
4	1	0	0	0
5	1	~b	0	0
6	1	~b	0	c
7	1	~b	0	m1
8	1	~b	b	m1
9	1	~b	m2	m1
10	1	~b	m3	m1

(b)

Figure 3 Micro-operations realizing full adder: (a) set of micro-operations, (b) resulting sub-functions in ReRAM devices after each micro-operation

2.2 Boolean Satisfiability (SAT)

The *Boolean Satisfiability* (SAT) is the problem of determining an assignment α to the variables of a Boolean function F such that F evaluates to TRUE (*Sat*). Otherwise, a proof is generated indicating that no such assignment exists (*Unsat*). Typically, F is expressed in *Conjunctive Normal Form* (CNF) consisting of conjunction of clauses. A clause is a disjunction of literals, where each literal is a normal variable or its negation.

3 Verification of in-memory Logic Design

In this section, we discuss the intermediate data structure to represent the operation sequence, and finally describe the overall verification methodology.

3.1 ReRAM Sequence Graph (ReSG)

As shown in Fig. 3(a), every line in the micro-operation sequence specifies the values applied to the word and bit lines of the crossbar, but does not specify the present state of the device. This makes the generation of functions from the micro-operations a difficult task. To overcome this, we use an intermediate data structure called *ReRAM Sequence Graph* (ReSG), which is a directed acyclic graph $H = (V, E)$ composed of four types of vertices, and represents the micro-operations in the crossbar. The first two types of vertices have no incoming edges and represent primary inputs with constant values of 0 and 1 respectively. The third type has no outgoing edges and represents primary output (or terminal) nodes. The fourth type has three incoming edges and an outgoing edge realizing the function $f(p,q,r) = p\overline{q} + pr + \overline{q}r$, with three kinds of incoming edges: two *regular* edges representing the inputs p and r, and a *complement* edge denoting the negation of the input q.

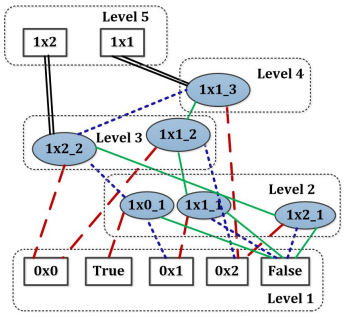


Figure 4 ReRAM sequence graph (ReSG)

Example 2 *Consider the micro-operations for a full-adder shown in Fig. 3 to be transformed into the equivalent ReSG. We traverse the sequence line-by-line from top to bottom, and from left to right. The complete ReSG is shown in Fig. 4. The regular incoming edges p and r of a functional node are highlighted in red and green respectively, while the incoming complement edge is denoted by a dashed blue line. For line 3, the primary input and two constant input nodes are inserted at* level 1 *of the ReSG. For line 4, we insert three nodes at* level 2 *with regular edge r connected to constant input node* FALSE. *Once these initial operations are translated into suitable nodes in ReSG, we consider the operations listed from lines (5 - 10) realizing the sub-functions. To realize operation at line 5, we apply 0x1 and* TRUE *respectively to the complement and regular edges of the functional node 1x0_1 at* level 2, *thereby realizing the negation of primary input b. For line 6, we apply* FALSE *and 0x2 to the complement and regular edge p respectively of node 1x2_1 at* level 2. *As a result, the primary input c is duplicated at node 1x2_1. For line 7, we add another node 1x2_2 at* level 3, *where we apply 0x0, 1x2_1 and 1x0_1 at the regular incoming edges p,r and a complement edge q respectively leading to the realization of primary output* carry. *In a similar fashion, lines 8 to 10 are translated into ReSG functional nodes at* level 3 *and* level 4 *as depicted in Fig. 4. Finally, we add two terminal nodes* sum *(1x1) and* carry *(1x2) at* level 5 *of the ReSG and connect them to the appropriate functional nodes.*

We label the functional nodes with the ReRAM crossbar locations and a number separated by an underscore (_). The number indicates the sequence number of the operations being executed sequentially on the same ReRAM device. For example, the node label 1x1_1 denotes that a sub-function is initially stored on the ReRAM device located at 1x1, the node label 1x1_2 indicates that the second sub-function is overwritten on the same ReRAM device, and so on.

3.2 Overall Verification Methodology

The verification methodology is depicted in Fig. 5, where we consider that a given function is represented as a *Majority-Inverter Graph* (MIG). The MIG and ReSG data structures are considered as the golden and reference representations respectively. A SAT-based equivalence checker then determines whether the two representations are equivalent or not.

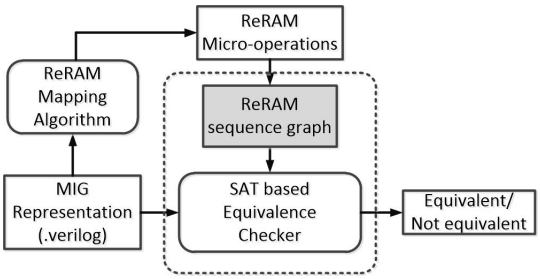


Figure 5 Proposed verification methodology

The general idea of SAT-based equivalence checking is to encode the problem as a Boolean Satisfiability instance, and use a SAT solver to solve it. In the present context, if the solver returns *unsatisfiable*, then the golden and the reference representations are equivalent. Otherwise, a counter-example is generated from the satisfying assignment of the instance.

Every MAJ3 node in the MIG representation is expressed as a set of clauses compatible with the SAT solver. To encode the ReRAM micro-operations as a SAT instance, every functional node of the ReSG is expressed by a set of clauses. After encoding the MIG and ReSG into respective SAT instances, we define a *miter* to check the equivalence between a MIG and a ReSG.

A miter is a circuit structure composed of a set of 2-input XOR gates in the first level and an OR gate in the second level. By applying the input assignments to both the circuits (viz. golden and reference), the inequality

Table 1 Experimental results

Equivalent cases

Benchmark		MIG			ReSG			
Name	PI/PO	#Nodes	#Clauses	t_1 (s)	#Nodes	#Clauses	t_2 (s)	time (s)
sym10	10/1	79	80	0.003	114	343	0.007	0.062
t481	16/1	25	51	0.002	36	109	0.005	0.063
c6288	32/32	1867	1899	0.025	2347	7073	0.095	*
c1908	33/25	296	738	0.006	388	1189	0.018	10.636
c432	36/7	95	233	0.003	124	379	0.009	2.013
c499	41/32	292	762	0.006	356	1100	0.017	9.541
c3540	50/22	824	1989	0.013	1159	3499	0.075	**

*c6288 (time) = 22270.671 s, **c3540 (time) = 15600.673 s

Non-equivalent cases

Benchmark		MIG			ReSG			
Name	PI/PO	#Nodes	#Clauses	t_1 (s)	#Nodes	#Clauses	t_2 (s)	time (s)
sym10	10/1	79	80	0.003	115	346	0.007	0.062
t481	16/1	25	51	0.002	37	112	0.006	0.063
c6288	32/32	1867	1899	0.025	2379	7169	0.11	*
c1908	33/25	296	738	0.006	385	1180	0.018	10.333
c432	36/7	95	233	0.003	131	400	0.009	2.331
c499	41/32	292	762	0.006	388	1196	0.021	10.673
c3540	50/22	824	1989	0.013	1151	3475	0.073	**

*c6288 (time) = 22273.673 s, **c3540 (time) = 15600.333 s

between the corresponding outputs are checked by the XOR operations. In case of multi-output circuits, all the outputs of XOR gates are combined by the OR operation. If the OR returns a value 1, it means at least one XOR gate evaluates to 1, and the MIG and ReSG representations are non-equivalent; otherwise, they are equivalent.

4 Experimental Evaluation

We present the experimental results in this section. All the benchmarks are obtained from ISCAS and IWLS. We have implemented our proposed scheme of constructing the ReSG, checking equivalence (i.e. miter structure) and generating clauses in Python 3.6. For checking equivalence based on Boolean Satisfiability, we have used *Z3* solver [9]; however, any standard SAT solver could have been used for this purpose. All the experiments have been run on a 2.8 GHz machine with a dual core processor and an 8 GB RAM.

Table 1 summarizes the obtained results. The upper part of Table 1 reports the cases where the MIG representation and the corresponding micro-operations (or ReSG) are functionally *equivalent*. The average run-time (t_1) for generating clauses from MIGs is very few CPU seconds. Similarly, the average time (t_2) for ReSG generation and formation of the respective clauses is also a few CPU seconds. The SAT-solver obtains the solutions very quickly for all the considered benchmarks except *c6288* and *c3540*, for which the run-times are higher. Actually, these two benchmarks have significantly larger number of clauses as compared to the other benchmarks, resulting in higher run-time. The proposed method must also detect the non-equivalence between a given MIG and the corresponding micro-operations when the latter is erroneous. For this validation, we modify the micro-operations by randomly inserting or deleting operations, while keeping the MIG representation unchanged. As expected, the SAT-solver indicates that the MIG and the modified micro-operations are functionally *non-equivalent*. The lower part of the table shows such non-equivalent cases. The run-times of equivalent and non-equivalent cases are approximately the same because we inserted/deleted a few nodes at the output ends of the ReSGs.

5 Conclusions

An automated approach to verify the micro-operations generated from majority-based mapping on ReRAM crossbar against the original functional specification has been presented in this paper. The intermediate ReSG data structure is used for direct generation of the clauses, which are then fed to the verification tool. The method has been found to correctly verify the generated micro-operations by running on several benchmark functions. As a future work, the mapping approach can be improved for crossbar circuits by applying various optimizations. In addition, adequate measures to address the sneak-path issue in crossbars shall be addressed.

6 Literature

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

[2] S. Chakraborti, P. Chowdhary, K. Datta, and I. Sengupta, "BDD based synthesis of boolean functions using memristors," in *Proc. Intl. Design and Test Symp. (IDT)*, pp. 136–141, 2014.

[3] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Fast logic synthesis for RRAM-based in-memory computing using majority-inverter graphs," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 948–953, 2016.

[4] A. Zulehner, K. Datta, I. Sengupta, and R. Wille, "A staircase structure for scalable and efficient synthesis of memristor-aided logic," in *Asia and South Pacific Design Automation Conference*, p. 237–242, 2019.

[5] S. Froehlich and R. Drechsler, "Generation of verified programs for in-memory computing," in *Digital System Design (DSD-2022)*, pp. 815–820, 2022.

[6] A. Deb, K. Datta, M. Hassan, S. Shirinzadeh, and R. Drechsler, "Automated equivalence checking method for majority based in-memory computing on ReRAM crossbars," in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, pp. 489–494, 2023.

[7] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (plim) computer," in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 427–432, 2016.

[8] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2014.

[9] L. d. Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340, Springer, 2008.