Automated Equivalence Checking Method for Majority based In-Memory Computing on ReRAM Crossbars

Arighna Deb*, Kamalika Datta†, Muhammad Hassan†, Saeideh Shirinzadeh†§, Rolf Drechsler†‡
*School of Electronics Engineering, KIIT DU, Bhubaneswar, India
†German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany
§Fraunhofer Institute for Systems and Innovation Research (ISI), Karlsruhe, Germany
‡Group of Computer Architecture, University of Bremen, Bremen, Germany
airghna.debfet@kiit.ac.in, {kamalika.datta, muhammad.hassan, saeideh.shirinzadeh}@dfki.de, drechsler@uni-bremen.de

ABSTRACT

Recent progress in the fabrication of *Resistive Random Access Memory* (ReRAM) devices has paved the way for large scale crossbar structures. In particular, in-memory computing on ReRAM crossbars helps in bridging the processor-memory speed gap for current CMOS technology. To this end, synthesis and mapping of Boolean functions to such crossbars have been investigated by researchers. However the verification of simple designs on crossbar is still done through manual inspection or sometimes complemented by simulation based techniques. Clearly this is an important problem as real world designs are complex and have higher number of inputs. As a result manual inspection and simulation based methods for these designs are not practical.

In this paper for the first time as per our knowledge we propose an automated equivalence checking methodology for majority based in-memory designs on ReRAM crossbars. Our contributions are twofold: first, we introduce an intermediate data structure called *ReRAM Sequence Graph* (ReSG) to represent the logic-in-memory design. This in turn is translated into *Boolean Satifiability* (SAT) formulas. These SAT formulas are verified against the golden functional specification using Z3 *Satifiability Modulo Theory* (SMT) solver. We validate the proposed method by running widely available benchmarks.

KEYWORDS

Boolean Satifiability (SAT), In-Memory Computing, ReRAM crossbar, Verification

ACM Reference Format:
Arighna Deb*, Kamalika Datta†, Muhammad Hassan†, Saeideh Shirinzadeh†§, Rolf Drechsler†‡. 2023. Automated Equivalence Checking Method for Majority based In-Memory Computing on ReRAM Crossbars. In *28th Asia and South Pacific Design Automation Conference (ASPDAC '23), January 16–19, 2023, Tokyo, Japan.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3566097.3567842

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPDAC '23, January 16–19, 2023, Tokyo, Japan
© 2023 Association for Computing Machinery.
ACM ISBN 978-1-4503-9783-4/23/01…$15.00
https://doi.org/10.1145/3566097.3567842

1 INTRODUCTION

Resistive Random Access Memory (ReRAM) or *memristor* [5] 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 [12]. There have been several attempts for efficient mapping and evaluation of Boolean functions on such crossbars [4, 11, 13–15]. Some of the most widely explored approaches are based on material implication (IMPLY) [3], memristor-aided-logic (MAGIC, with NOR and NOT realizations) [9], and majority logic operation (MAJ) [7].

To ensure the functional correctness of the micro-operations, traditionally manual inspection sometimes complemented by simulation based techniques are widely used. These methods are used to compare the micro-operations against the golden function specification. Manual inspection methods can be employed to very small designs, while simulation based techniques are limited to verification for a subset of input combinations. Also traditional methods for equivalence checking cannot be directly applied to ReRAM based crossbars (See section 3 for details). This is clearly a problem. With the increased complexity of larger in-memory designs, the possibility for errors in crossbars may grow, which emphasizes the need for advanced verification techniques to prove the functional correctness of the crossbar mappings. Recently, some initial efforts have been done to verify the in-memory programs [6]. However, the work does not verify the instructions at the micro-operations level. This clearly emphasizes the need for a systematic advanced functional verification technique to ensure the correctness of the micro-operations.

In this paper for the first time to the best of our knowledge, we propose an automated equivalence checking methodology for majority based in-memory designs on ReRAM crossbar. In particular, we systematically verify the micro-operations performed on the crossbar against the golden functional specification as *Boolean Satifiability* or *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 ReSGs 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 [1, 8].

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 motivation and outline

the proposed 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, ReRAM crossbar, and logic operations that can be performed on the crossbars. We also briefly discuss about the SAT.

2.1 Resistive Random Access Memory (ReRAM)

A ReRAM is an emerging memory device which consists of an oxide layer sandwiched between two metal electrodes (p, q) in a *Metal-Insulator-Metal (MIM)* structure as shown in Fig. 1(a). Such a device can be switched between a low resistance state (LRS or logic 1) and a high resistance state (HRS or logic 0) by applying a voltage of proper magnitude and polarity to the device's terminals. The behavior of a 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. Depending on the behavior of the device, we can state that a ReRAM device inherently realizes a Boolean function $f(p, q, r) = p\overline{q} + pr + \overline{q}r$ [7].

Fig. 1(c) shows the logic symbol of a ReRAM device. Several such devices are typically laid out in a compact fashion in crossbar 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* and a horizontal wire (or q terminal) is called a *wordline*.

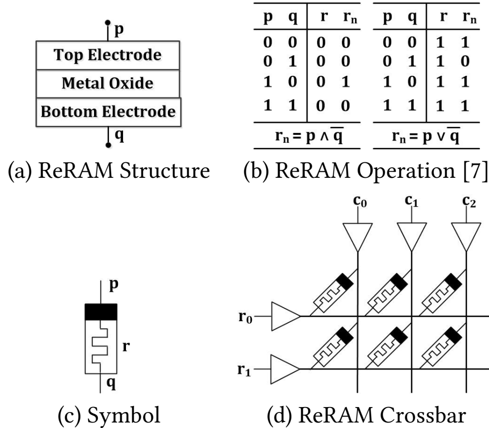


Figure 1: ReRAM device and crossbar structure

To realize an arbitrary Boolean functions in crossbar, we must sequentially execute several operations called *micro-operations* in the crossbar depending on the given function representation, e.g. *Majority-Inverter Graph* (MIG) [2]. The micro-operations in the crossbar are performed by traversing each node in the corresponding MIG. Each node in a MIG, called MAJ3 realizes a 3-input majority function of the form $f(a, b, c) = ab + ac + bc$.

Example 1. *Consider a full adder function 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* Majority Inverter Graph *(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 entire MIG in a breadth-first manner and realize node m1 as a sequence of operations that include (1) realization of* $\overline{b}$ *in crossbar located at row 1(r1) and column 0(c0) (i.e. 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 MIG are realized as a set of operations realizing the desired full adder functionality. The complete crossbar micro-operations realizing full adder are shown in Fig. 3a. Further, the execution of each micro-operation leading to the realization of sub-function in each ReRAM device located at each row and column of the crossbar is depicted in Fig. 3b. A more detailed explanation of the micro-operations is provided in Section 3.2.*

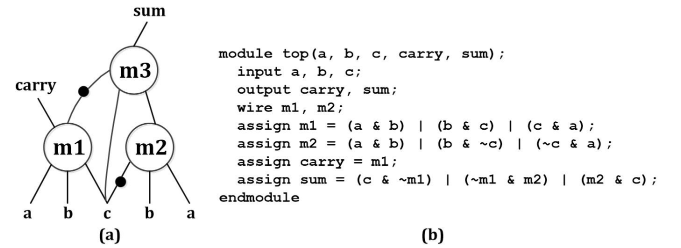
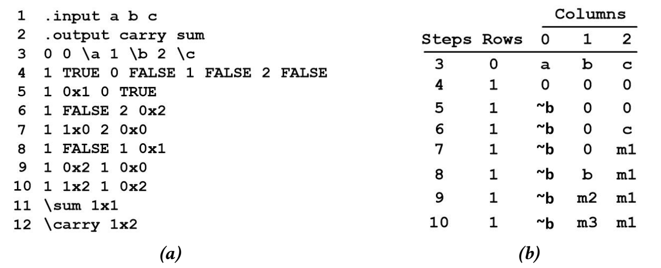


Figure 2: Full adder: (a) MIG, (b) Equivalent verilog code



```
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 a 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*). Often, 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. For example, a Boolean function $F = (\overline{a} + \overline{b})(\overline{a} + \overline{c})(\overline{b} + \overline{c})$ is satisfied for an assignment $a = b = 0, c = 1$. SAT solvers are commonly used in the industry for verification and equivalence checking. In our work, we use *Z3* solver at the backend [10].

3 PROPOSED AUTOMATED EQUIVALENCE CHECKING METHODOLOGY

In this section we first present the motivation and general idea, then discuss the proposed crossbar file format, the intermediate data structure to represent the sequence of operations, and the verification methodology.

3.1 Motivation

Like traditional CMOS design, we need to ensure that a ReRAM crossbar for a given set of micro-operations indeed realizes the desired functionality. As mentioned before manual inspection and simulation based methods cannot be applied to functions with larger input. In this context, equivalence checking plays a significant role. In particular, the equivalence checker determines whether the function description (the traditional logic network) and the ReRAM micro-operations on crossbar realize the same functionality.

However, unlike traditional combinational circuit design, ReRAM crossbars realize the desired functionality sequentially, thereby yielding a situation where traditional equivalence checkers cannot be applied directly to the ReRAM crossbar. While combinational circuits implement the sub-functions without considering the current states of the logic gates, the realization of any sub-function on ReRAM crossbar depends on the present state of the device (as illustrated previously in Example 1). The present state decides the subsequent steps to be carried out to implement the desired functionality on the crossbar. This leads to a situation where the sequential states of the devices need to be monitored during any equivalence checking.

It may be noted that no equivalence checker for ReRAM crossbars exist yet that considers the domain-specific characteristics and issues. This leads to the question: how do we verify whether the generated ReRAM micro-operations correctly realize the original (MIG) functional specification? The solution to this question further leads to the specific query: how do we monitor the present and next states of the ReRAM devices? For a small function like the full adder, it may be possible to answer these questions manually through step-by-step evaluation. However, for large functions the process becomes too complex to be carried out manually. Hence there is a need to develop automated systematic solutions to this problem. This is the precise aim of the present paper.

3.2 Crossbar Micro-Operations File Format

The synthesis tool generates the sequence of micro-operations for the crossbar to realize a given function. As mentioned earlier, every MAJ3 operation can be carried out by applying suitable voltages on the wordline and the bitline(s) of the crossbar. Basically, three things need to be specified as discussed below.

a) The initial crossbar locations for the primary inputs of the function. The general format is:
```
<row> <col1> \<PI1> <col2> \<PI2> ...
```
where <row> indicates the wordline, <col1>, <col2>, etc. indicate the bitlines, and <PI1>, <PI2>, etc. indicate the primary input names as defined in the input Verilog file.

b) The crossbar locations where the primary outputs of the function are finally available. The general format is:
```
\<PO1>: r1xc1 \<PO2>: r2xc2
```
where $r1$, $r2$ indicate the wordlines and $c1$, $c2$, etc. indicate the bitlines where the output variables <PO1>, <PO2>, etc. will get stored.

c) The sequence of MAJ3 operations to be carried out on the crossbar. The general format to specify one (or more) parallel MAJ3 operations on wordline <row> is:
```
<row> <val> <col1> <val> <col2> <val> ...
```
where <col1>, <col2>, etc. indicate the bitlines, and <val> indicates a voltage value to be applied. The value of <val> can be either TRUE or FALSE or the value of a crossbar cell specified as rxc; it represents the voltage to be applied to the wordline or the bitline as specified in the preceding line.

The complete crossbar micro-operations for the full adder is shown in Fig. 3. The first two lines indicate that the input variables are a, b and c, and the output variables are $carry$ and sum. The third line indicates that the input variables a, b, c are loaded in the cells 0x0, 0x1 and 0x2 respectively. The next seven lines specify MAJ3 operations on the crossbar, in the following order:

1) Apply TRUE in wordline 1, and FALSE in bitlines 0, 1 and 2. This resets the cells 1x0, 1x1 and 1x2 to logic 0.
2) Apply the value of cell 0x1 in wordline 1 and TRUE in bitline 0. This stores the value of $\overline{b}$ in cell 1x0.
3) Apply FALSE in wordline 1 and value of cell 0x2 in bitline 2. This copies the value of c in cell 1x2.
4) Apply the value of cell 1x0 in wordline 1 and value of cell 0x0 in bitline 2. This stores the value of MAJ3(a,b,c) in cell 1x2. This is the value of $carry$ ($m1$).
5) Apply FALSE in wordline 1 and value of cell 0x1 in bitline 1. This copies the value of b in cell 1x1.
6) Apply the value of cell 0x2 in wordline 1 and value of cell 0x0 in bitline 1. This stores the value of MAJ3(a,b,c') in cell 1x1. This is the value of $m2$.
7) Apply the value of cell 1x2 in wordline 1 and value of cell 0x2 in bitline 1. This computes the majority operation MAJ3(m1',m2, c) and stores it in cell 1x1.

The last two lines specify that the output sum is in cell 1x1, and the output $carry$ is in cell 1x2.

3.3 ReRAM Sequence Graph (ReSG)

Every line in the micro-operation file format provides the values applied to the wordline and bitline of the ReRAM device, but does not specify the current state of the device. This makes the generation of Boolean functions from the micro-operation file format a difficult task. To overcome this difficulty, we structure the micro-operations in such a manner that makes the generation of Boolean functions simpler while ensuring the sequential nature of ReRAM operations. This necessitates the design of an intermediate data structure called

ReRAM Sequence Graph (ReSG). It is defined as a directed acyclic graph $H = (V, E)$ composed of four types of vertices, and represents the micro-operations in the crossbar. The first and second types of vertices have no incoming edges and represent primary inputs and two constant inputs (logic 0 and logic 1), respectively. The third type of vertices has no outgoing edges and represents primary output (or terminal) nodes. The fourth type has three incoming edges and an outgoing edge, and represents the function $f(p, q, r) = p\overline{q} + pr + \overline{q}r$. These non-terminal function nodes have 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. More formally, a ReSG is defined as follows.

Definition 1. *A ReRAM Sequence Graph (ReSG) over the primary input variables* $X = \{x_0, x_1, \cdots, x_{n-1}\}$ *and primary output variables* $Y = \{y_0, y_1, \cdots, y_{m-1}\}$ *is a directed acyclic graph* $H = (V, E)$ *with*

a) *a finite set of nodes* $V = (V_X \cup V_{CI} \cup V_h \cup V_Y)$, *where* $V_X = \{v_{x_0}, v_{x_1}, \cdots, v_{x_{n-1}}\}$ *are primary input nodes,* $V_{CI} = \{v_T, v_F\}$ *are constant input nodes, True (logic 1) and False (logic 0),* $V_h = \{v_{h_0}, v_{h_1}, \cdots, v_{h_l}\}$ *are non-terminal nodes inherently realizing the ReRAM functionality, and* $V_Y = \{v_{y_0}, v_{y_1}, \cdots, v_{y_{m-1}}\}$ *are terminal nodes representing primary outputs,*

b) *an edge* $e \in E$ *between a source node* $u \in V$ *and a target node* $v \in V$ *is either a regular edge* p *or a regular edge* r *or a complement edge* q, *i.e.* $e = \{(u, (v \times t)) \mid u, v \in V, u \notin V_Y, v \notin V_X, v \notin V_{CI}\}$, *where* t *denotes whether the edge is a regular edge* p $(t = +1)$ *or* r $(t = -1)$ *or a complement edge* q $(t = 0)$.

The size of a ReSG is measured by the number of functional nodes that depends on the number of ReRAM operations in the crossbar file. We now explain the process of translating the crossbar file into a functionally equivalent ReSG.

Suppose a set of ReRAM operations is represented as $M = \{m_1, m_2, \cdots, m_k\}$ where any operation m_i is denoted as:

$$m_i = <Q_j> <val(Q_j)> <P_s> <val(P_s)>$$

where $val(Q_j)$ and $val(P_s)$ denote the values at row Q_j and column P_s respectively. The ReSG is obtained by traversing each operation m_i and mapping it to an equivalent functional node v_i such that the regular edges p and r connect source nodes u_s and u_i respectively to the target node v_i, whereas the complement edge q connects the source node u_j to the target node v_i. Herein, the nodes $u_s, u_j, u_i \in V$ indicate the value of column P_s, value of row Q_j and the present state of the functional node v_i, respectively.

The above idea is further explained with an example.

Example 2. *Consider the crossbar operations for the full-adder shown in Fig. 3 to be transformed into a functionally equivalent ReSG. The transformation begins by traversing the listing line-by-line from top to bottom and from left to right. The complete ReSG representing the full adder realization on ReRAM crossbar is shown in Fig. 4. Note that the regular incoming edges,* p *and* r *of a functional node are highlighted in red and green colors, respectively, while the incoming complement edge of a functional node is denoted by a dashed line in blue color. For the line 3, the primary input and two constant input nodes are inserted at* ***level 1*** *of the ReSG as shown in Fig. 4. For the line 4, we insert three nodes at* ***level 2*** *with regular edges* r *connected to the constant input node* False *as shown in Fig. 4. Once these initial operations are translated into suitable nodes in ReSG, we then 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.*

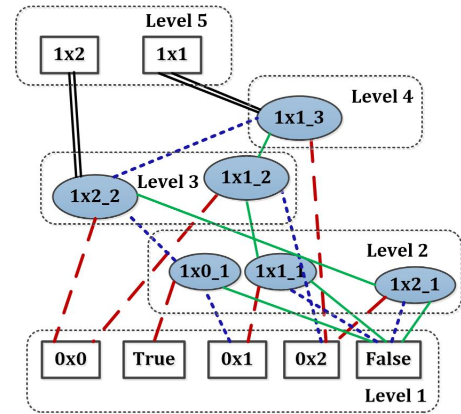


Figure 4: ReRAM Sequence Graph (ReSG)

We label the ReSG functional nodes with the corresponding locations of the ReRAM crossbar and a number separated by an underscore (_). The numbers indicate the sequence number of the operations being executed sequentially on the same ReRAM devices. For example, the functional 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 at 1x1, and so on.

3.4 Overall Verification Methodology

The proposed verification methodology is depicted in Fig. 5, where it is considered that a given function specification is represented as a *Majority-Inverter Graph* (MIG). The MIG data structure is represented in Verilog form, and is considered as the golden representation of the function in the context of the present work. A ReRAM mapping tool analyzes the MIG representation and generates a set of micro-operations for evaluation on the crossbar. The micro-operations are represented as ReSG, and represents the reference representation of the function. An equivalence checker based on a SAT solver finally determines whether the golden representation and the reference representation are equivalent or not.

The general idea of the SAT-based equivalence checking is to encode the problem as a Boolean satisfiability instance to be solved by the SAT solver. In the present context, if the solver returns

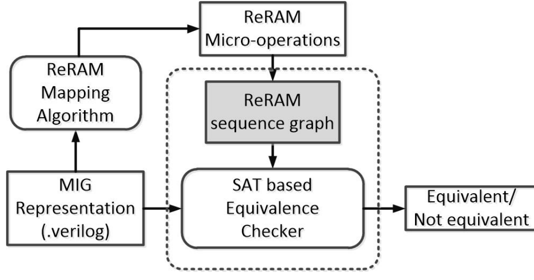
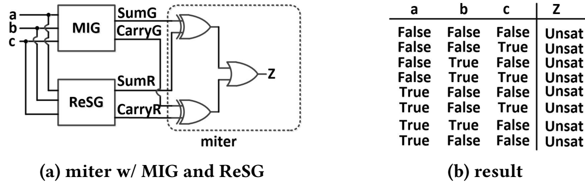


Figure 5: Proposed verification methodology



a	b	c	Z
False	False	False	Unsat
False	False	True	Unsat
False	True	False	Unsat
False	True	True	Unsat
True	False	False	Unsat
True	False	True	Unsat
True	True	False	Unsat
True	False	False	Unsat

(a) miter w/ MIG and ReSG **(b) result**

Figure 6: SAT formulation and outcome

unsatisfiable, then the golden and the reference representations are equivalent. Otherwise, a counter-example is extracted from the satisfying assignment of the instance.

More precisely, every MAJ3 node in the MIG representation is expressed as a set of clauses that can be directly processed by the SAT solver. To encode the ReRAM micro-operations into a SAT instance, the ReSG is generated as discussed in the previous section, where every functional node of the ReSG can directly be expressed by a set of clauses. After encoding the MIG and ReSG structures into respective SAT instances, we define a *miter* to check the equivalence between a MIG and a ReSG.

A miter is a circuit structure which is 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 (i.e. golden and reference), the inequality between the corresponding outputs are checked by the XOR operations. In case of multi-output circuits, all the outputs of XOR gates are observed by the OR operation. If the OR operation returns a value 1, it means at least one XOR gate evaluates to 1 indicating that the MIG and ReSG are non-equivalent. Otherwise, they are equivalent. Note that, the added miter structure is only used to determine the circuit equivalence, and do not alter the actual functionality of the MIG and ReSG.

Example 3. *The miter structure for two representations of full adder containing two output lines is shown in Fig. 6(a). For all the input assignments, the OR operation evaluates to 0 (i.e. Unsat as shown in Fig. 6(b)) indicating that the MIG and ReSG for full adder are equivalent.*

4 EXPERIMENTAL EVALUATION

This section summarizes the experimental results. All the benchmarks were obtained from ISCAS and IWLS [1, 8]. 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 [10]. All the experiments have been run on a 2.8 GHz machine with a dual core processor and an 8 GB RAM.

Table 1 shows the obtained results. The first column provides the details of the benchmark, i.e. name of the benchmark and the number of Primary Inputs (PI) as well as the number of Primary Outputs (PO). The next column reports the number of nodes (*#Nodes*) in the MIG representation of the respective benchmark, the number of resulting clauses (*#Clauses*) and the time to obtain the clauses (t_1). The next column provides the total number of micro-operations (*#Ops*), the number of nodes (*#Nodes*) in the corresponding ReSG, the number of resulting clauses (*#Clauses*) and their generation time (t_2). The final column shows the time to check the equivalence (*SAT solver time*) between MIG and ReSG. All the times are shown in CPU seconds.

Note that the table has two parts: equivalent and non-equivalent. The benchmarks are divided into small (where $PI + PO \leq 20$) and larger (where $PI + PO > 20$). The upper part of Table 1 reports that the MIG representation and the corresponding micro-operations (or ReSG) are functionally equivalent. The average run-time for generating clauses from MIGs is very few CPU seconds. Similarly, the average time taken to obtain the ReSG from the given crossbar file and then to generate the respective clauses is also a few CPU seconds. For the small and large benchmarks, the SAT-solver obtains the solution (i.e., equivalent (Unsat) or non-equivalent (Sat)) very quickly except two large benchmarks, *c6288* and *c3540*, for which the run-times are higher. This is due to the fact that those benchmarks (i.e. *c6288* and *c3540*) have significantly large number of clauses as compared to all the other large benchmarks, resulting in higher run-time. In our future work we will try to incorporate some optimization technique to improve the mapping methodology, which might in turn helps to reduce the number of clauses and eventually the run-time.

The proposed verification method must also detect the non-equivalence between a given MIG and the corresponding micro-operations when the latter is erroneous. For this purpose, we have modified the micro-operations by randomly inserting or deleting operations in the crossbar file, while keeping the given 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 Table 1 shows the results for non-equivalent cases, where all the columns remain same as that of equivalent cases except the third column (ReSG). Since we modify the micro-operations, the number of operations, ReSG nodes, clauses reported in the third column of lower part of the Table 1 differ from that of the third column in upper part of the same table. Overall, our proposed automated verification approach can identify the equivalence or non-equivalence between the MIG and its corresponding crossbar micro-operations.

5 CONCLUSION

We present an automated method for verifying whether the crossbar micro-operations generated from majority-based mapping for ReRAM circuits is equivalent to the original functional specification. To the best of the knowledge of the authors, this is the first

Table 1: Experimental results for small and large benchmarks

Equivalent cases

	Benchmark			MIG			ReSG				
	Name	PI	PO	#Nodes	#Clauses	t_1 (s)	#Ops	#Nodes	#Clauses	t_2 (s)	SAT solve time (s)
small	exam1	3	1	6	13	0.002	12	9	28	0.004	0.039
	rd32	3	2	3	11	0.002	10	6	20	0.003	0.041
	exam3	4	1	10	21	0.002	19	16	49	0.004	0.049
	xor5	5	1	12	25	0.002	22	19	58	0.004	0.033
	rd53	5	3	20	59	0.006	39	34	105	0.012	0.141
	con1	7	1	8	18	0.002	15	12	37	0.004	0.053
	con2	7	1	9	19	0.002	16	13	40	0.004	0.053
	rd73	7	3	34	99	0.006	63	58	177	0.016	0.179
	newtag	8	1	9	19	0.002	16	13	40	0.004	0.042
	newill	8	1	20	43	0.002	31	28	85	0.007	0.089
	rd84	8	4	43	127	0.009	79	73	223	0.021	0.258
	9sym	9	1	60	131	0.003	91	88	265	0.006	0.059
	max46	9	1	132	302	0.004	181	178	535	0.009	0.152
	sym10	10	1	79	80	0.003	117	114	343	0.007	0.062
	sao2	10	4	141	297	0.008	220	214	646	0.025	0.258
	t481	16	1	25	51	0.002	39	36	109	0.005	0.063
large	c6288	32	32	1867	1899	0.025	2381	2347	7073	0.095	22270.671
	c1908	33	25	296	738	0.006	415	388	1189	0.018	10.636
	c432	36	7	95	233	0.003	133	124	379	0.009	2.013
	c499	41	32	292	762	0.006	390	356	1100	0.017	9.541
	c3540	50	22	824	1989	0.013	1183	1159	3499	0.075	15600.673

Non-equivalent cases

	Benchmark			MIG			ReSG				
	Name	PI	PO	#Nodes	#Clauses	t_1 (s)	#Ops	#Nodes	#Clauses	t_2 (s)	SAT solve time (s)
small	exam1	3	1	6	13	0.002	11	8	25	0.004	0.039
	rd32	3	2	3	11	0.002	9	5	17	0.003	0.041
	exam3	4	1	10	21	0.002	18	15	46	0.004	0.049
	xor5	5	1	12	25	0.002	21	18	55	0.004	0.033
	rd53	5	3	20	59	0.006	40	35	108	0.012	0.141
	con1	7	1	8	18	0.002	14	11	34	0.004	0.053
	con2	7	1	9	19	0.002	17	14	43	0.004	0.053
	rd73	7	3	34	99	0.006	64	59	180	0.016	0.179
	newtag	8	1	9	19	0.002	15	12	37	0.004	0.042
	newill	8	1	20	43	0.002	30	27	82	0.007	0.089
	rd84	8	4	43	127	0.009	80	74	226	0.021	0.258
	9sym	9	1	60	131	0.003	92	89	268	0.006	0.059
	max46	9	1	132	302	0.004	182	179	538	0.009	0.152
	sym10	10	1	79	80	0.003	118	115	346	0.007	0.062
	sao2	10	4	141	297	0.008	222	216	652	0.026	0.258
	t481	16	1	25	51	0.002	40	37	112	0.006	0.063
large	c6288	32	32	1867	1899	0.025	2413	2379	7169	0.11	22273.673
	c1908	33	25	296	738	0.006	412	385	1180	0.018	10.333
	c432	36	7	95	233	0.003	140	131	400	0.009	2.331
	c499	41	32	292	762	0.006	422	388	1196	0.021	10.673
	c3540	50	22	824	1989	0.013	1175	1151	3475	0.073	15600.333

attempt to develop a systematic approach in this regard. The intermediate data structure ReSG helps in direct generation of the clauses, as required by the verification method. Experimental results validate that the method is able to correctly verify the generated micro-operations. This further helps in improving our confidence on whether the crossbar mapping is indeed generating functionally equivalent micro-operations corresponding to a given function. As a future work, optimizations can be applied to improve the mapping methodology for crossbar circuits, which in turn can reduce the time for verification.

ACKNOWLEDGEMENT

This work was supported by the German Research Foundation (DFG) within the Project PLiM (DR 287/35-1) and in part by the German Federal Ministry of Education and Research (BMBF) within the project AUTOASSERT under contract no. 16ME0117.

REFERENCES

[1] C. Albrecht. 2005. *IWLS 2005 Benchmarks.* Technical Report.

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

[3] J. Borghetti et al. 2010. Memristive Switches Enable Stateful Logic Operations via Material Implication. *Nature* 464 (2010), 873–876.

[4] S. Chakraborti, P.V. Chowdhary, K. Datta, and I. Sengupta. 2014. BDD based Synthesis of Boolean Functions using Memristors. In *Proc. Intl. Design and Test Symp. (IDT)*. 136–141.

[5] L. Chua. 1971. Memristor – The Missing Circuit Element. *IEEE Trans. on Circuit Theory* CT-18, 5 (1971), 507–519.

[6] S. Froehlich and R. Drechsler. 2022. Generation of Verified Programs for In-Memory Computing. In *Digital System Design (DSD-2022) (Accepted)*.

[7] P.-E. Gaillardon, L. Amarú, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli. 2016. The Programmable Logic-in-Memory (PLiM) computer. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*. 427–432.

[8] M.C. Hansen, H. Yalcin, and J.P. Hayes. 1999. Unveiling the ISCAS-85 benchmarks: a case study in reverse engineering. *IEEE Design Test of Computers* 16, 3 (1999), 72–80.

[9] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser. 2014. MAGIC—Memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs* 61, 11 (2014), 895–899.

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

[11] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler. 2016. Fast logic synthesis for RRAM-based in-memory computing using Majority-Inverter Graphs. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 948–953.

[12] D.B. Strukov, G.S.Snider, D.R. Stewart, and R.S. Williams. 2008. The Missing Memristor Found. *Nature* 453 (2008), 80–83.

[13] N. Talati, S. D. Gupta, P. S. Mane, and S. Kvatinsky. 2016. Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC). *IEEE Trans. on Nanotechnology* 15 (2016), 635–650.

[14] P. L. Thangkhiew, R. Gharpinde, and K. Datta. 2018. Efficient mapping of Boolean functions to memristor crossbar using MAGIC NOR gates. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 8 (2018), 2466–2476.

[15] A. Zulehner, K. Datta, I. Sengupta, and R. Wille. 2019. A Staircase Structure for Scalable and Efficient Synthesis of Memristor-Aided Logic. In *Asia and South Pacific Design Automation Conference*. 237–242.