Verification of In-Memory Logic Design using ReRAM Crossbars

Kamalika Datta[*§], Arighna Deb[†], Fatemeh Shirinzadeh[§], Abhoy Kole[*], Saeideh Shirinzadeh[*‡], Rolf Drechsler[*§]
[*]German Research Centre for Artificial Intelligence (DFKI), Bremen, Germany
[†]Kalinga Institute of Industrial Technology, Bhubaneswar, India
[‡]Fraunhofer Institute for Systems and Innovation Research(ISI), Karlsruhe, Germany
[§]Institute of Computer Science, University of Bremen, Germany
E-mail: kamalika.datta@dfki.de, airghna.debfet@kiit.ac.in, shirinfa@uni-bremen.de, abhoy.kole@dfki.de,
saeideh.shirinzadeh@dfki.de, drechsler@uni-bremen.de

***Abstract*—Resistive Random Access Memory (ReRAM) technologies enable the development of innovative architectures for in-memory computing. Many logic design styles, like Imply, Magic or Majority, have been explored for mapping Boolean functions to ReRAM crossbars. However, little attention has been given to the verification of the mapping process. Simulation based approaches can be used to check the functional correctness of smaller designs, but only formal verification techniques can ensure completeness for larger designs. Some initial works in this area have been proposed, which specifically focus on the verification of micro-operations using majority-based logic design. However, these techniques cannot be directly applied to other logic design styles, like Imply or Magic. This necessitates the design and exploration of more general verification techniques for logic-in-memory using ReRAM crossbars, and opens up the scope for further investigation. In this paper, we provide an overview of existing verification techniques for logic-in-memory designs, and also discuss directions for future work.**

I. Introduction

Various technologies for logic design have evolved over the years. Also, various innovations have been proposed to tackle technological challenges, e.g. processor-memory bottleneck in computing. Architectures for in-memory computing have been discussed in recent years to address this specific problem. Among the various alternatives, architectures based on *Resistive Random Access Memory (ReRAM)* hold much promise due to their compact form factor, low power consumption and compatibility with existing fabrication technology [14].

Many in-memory logic designs are explored using ReRAM crossbars, viz. Imply, Magic and Majority [4], [10], [12], [13], [15]–[17]. These methods rely on some representation of Boolean functions for the purpose of crossbar mapping. Many works have focused on synthesis and mapping of functions to ReRAM crossbars, but much less emphasis has been given to the process of verification.

Here we consider the problem of verification of the micro-operations generated during synthesis and mapping against the golden functional specification. We traditionally use simulation-based methods to verify the functional correctness of the designs. However, only formal verification techniques can ensure completeness for larger designs. Some initial works have been carried out to formally verify the in-memory programs [6]. In [5], the authors propose an automated equivalence checking technique to verify the micro-operations generated using Majority-based logic. However, this is targeted to Majority-based designs, and as such cannot be directly applied to other logic design styles. This opens up room for further research in this area. In this paper, we provide an overview of in-memory design verification using ReRAM crossbar. We provide the overall verification methodology and in particular discuss the method based on Majority-based logic. Experimental results are reported for various benchmarks. Finally, we discuss about the open question and challenges.

II. Background and Related Work

A. Logic Design Styles

ReRAM devices can execute several universal logic primitives, including *Material Implication (IMPLY)* (Fig. 1), *Memristor-Aided LoGIC (MAGIC)* (Fig. 2), and *Resistive Majority Operation (MAJ)* (Fig. 3). IMPLY [3] was the first logic operation that was shown to be executable in resistive switches. An IMPLY gate employs two ReRAM devices that are connected to a load resistor through a nanowire. This provides a stateful logic structure so that the result of the operation is stored as resistance values in the output ReRAM device [3]. In [9], MAGIC was presented, which allows to compute a NOR operation within ReRAM crossbar. The execution of a MAGIC gate consists of two stages. It employs previously initialized input devices and implements NOR within an output device initially storing a known logic value. In [7], it was shown that the MAJ operation with a negated input can be performed in ReRAM devices. According to this operation, the resistive state of an ReRAM device is switched from its current value r to $r' = p\bar{q} + pr + \bar{q}r$, where p and $\bar{q}$ represent the values applied to its top and bottom terminals, respectively.

B. Related Works

Efficient synthesis and mapping of Boolean function on ReRAM crossbar are promising alternatives to address the processor-memory speed gap problem. However, the verification of the crossbar mapping is an important issue, because manual inspection and simulation techniques are not practical when a complex design with many inputs is required. In this regard, SAT-based equivalence checking methods for ReRAM crossbars were introduced in [5], [6]. In [5], an automated equivalence checking methodology for Majority-based in-memory designs on ReRAM crossbar is proposed, in

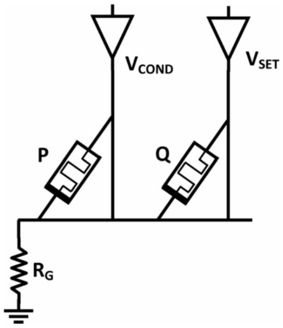


Fig. 1: IMPLY gate [3]

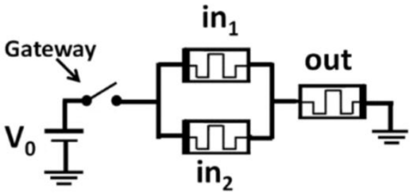
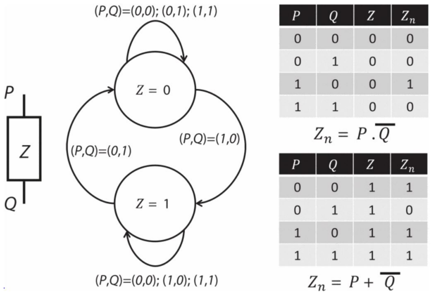


Fig. 2: MAGIC nor gate [9]



P	Q	Z	Z_n
0	0	0	0
0	1	0	0
1	0	0	1
1	1	0	0

$Z_n = P.\overline{Q}$

P	Q	Z	Z_n
0	0	1	1
0	1	1	0
1	0	1	1
1	1	1	1

$Z_n = P + \overline{Q}$

Fig. 3: Majority operations [7]

which a Boolean SAT formula is developed for checking the microoperations on the crossbar against the golden functional specification, e.g. *Majority Inverter Graphs (MIG)* [2]. In [6], the authors propose a method for verifying the functional equivalence between MIGs and a novel HDL-program supporting ReRAM operations.

As of date, very few methods have been proposed to verify crossbar mapping of functional specifications, this also holds for Majority-based designs. However, no such works have been reported so far that encompass the other logic design styles.

III. Verification of in-memory Logic Design

Any implementation of logic synthesis is prone to errors either due to imperfections in the algorithms or in the methodology for implementation. It is therefore important to verify a design before it is deployed for field use. Broadly there are two approaches to verification, using simulation or using formal verification. The latter approach is more comprehensive and is also the focus of the present work.

Various algorithms have been proposed for synthesis and crossbar mapping of functions for in-memory computing. A number of different logic design approaches like Imply, Magic, Majority, etc. have been explored in this regard. Each of these approaches requires distinctly different crossbar mapping and evaluation technique, which necessitates a method-specific micro-operation format for crossbar mapping. As such no general micro-operation format is available to date that can encompass all the different approaches.

The general flow for the verification of in-memory synthesis and mapping techniques is shown in Fig. 4. Given a functional specification, it is first transformed into a method-specific intermediate representation, which is then converted into equivalent crossbar micro-operation. The intermediate representation can be a NOR-gate netlist, Imply-gate netlist or Majority-gate netlist.

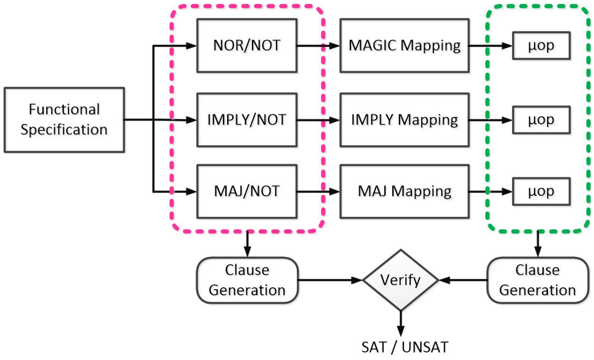


Fig. 4: General flow of the verification process

A. General Crossbar File Format

When a function is synthesized into a technology-mapped netlist (e.g. MAJ-NOT netlist for Majority-based synthesis), the next step is to map the corresponding operations to a ReRAM crossbar for evaluation. It is necessary to express these operations in a standard format, so that the design automation tools become inter-operable with a standard interface.

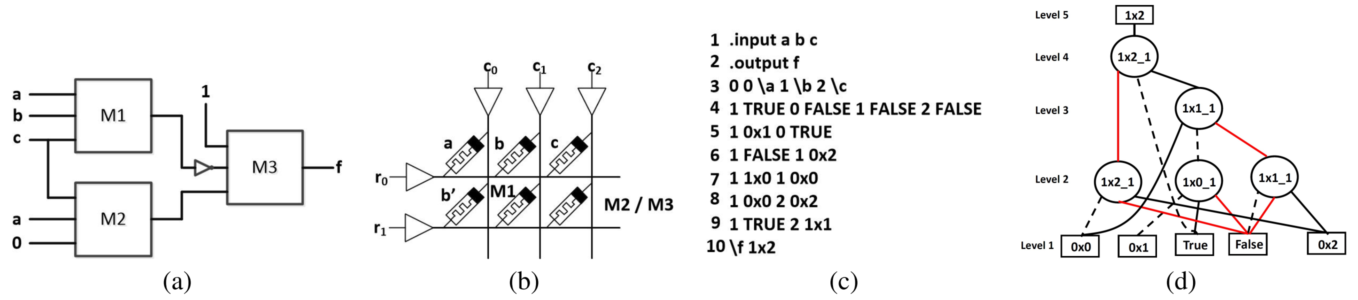
For instance, a 3-variable Majority operation denoted as $MAJ(p, q, r) = p.q + q.r + p.r$ can be evaluated on a ReRAM device in row i and column j by initializing the device with a resistive state corresponding to r, and applying p and $\bar{q}$ on row i and column j respectively. Following the convention of [5], this can be formally denoted as:

$$\texttt{i}\ \langle val\_p \rangle\ \texttt{j}\ \langle val\_qbar \rangle$$

Similarly, if we want to carry out multiple MAJ operations on row i of the crossbar on cells in columns $j_1, j_2, \ldots, j_n$, where the first operand p is common for all the operations, we apply p on row i, and the corresponding second operands $q_1, q_2, \ldots, q_n$ on the columns $j_1, j_2, \ldots, j_n$; respectively. It is assumed that the third operands $r_1, r_2, \ldots, r_n$ are loaded as resistive states in the corresponding devices prior to the operation. This can be formally denoted as [5]:

$$\texttt{i}\ \langle val\_p \rangle\ j_1\ \langle val\_qbar1 \rangle\ \texttt{...}\ j_n\ \langle val\_qbarn \rangle$$

An example MAJ-NOT netlist for a 3-variable function is shown in Fig. 5(a), and the corresponding crossbar mapping of the variables and operations in Fig. 5(b). The complete micro-operation sequence for the function is shown in Fig. 5(c), as per the convention of [5]. The micro-operation format as proposed in [5] only permits digitally encoded voltage values to be applied in the rows and columns during the evaluation process. Thus a value of 0 may indicate ground level, while a value of 1 may indicate a positive voltage (e.g. 1.0V). One



```
.input a b c
.output f
0 0 \a 1 \b 2 \c
1 TRUE 0 FALSE 1 FALSE 2 FALSE
1 0x1 0 TRUE
1 FALSE 1 0x2
1 1x0 1 0x0
1 0x0 2 0x2
1 TRUE 2 1x1
\f 1x2
```

(a) (b) (c) (d)

Fig. 5: (a) A MAJ-NOT netlist, (b) Crossbar mapping, (c) Complete sequence of micro-operations, (d) ReSG for the function

drawback of this format is that it does not allow a user to specify parallel MAJ operations in more than one row or column of the crossbar.

Also, the same file format cannot be directly used for other logic design approaches due to the following:

a) In the Imply logic style, multiple voltage levels like V_{cond} and V_{set} need to be applied on the columns, and the row has to be electrically isolated (i.e. floating).
b) In the Magic design style, a different voltage level V_o need to be applied to some of the columns, while the row has to be electrically isolated.
c) Parallel evaluations on multiple rows is not supported in [5]. Also, there is no provision for applying an isolation voltage V_{iso} along rows and columns.

B. Verification Methodology

The overall verification flow is as shown in Fig. 4. The actual process of verification is carried out by SAT solver or SMT solver, and hence the problem must be transformed into a form that can be directly processed by the tool.

Given the specification of a function in some format, the function can be transformed into a suitable intermediate form for further processing. This intermediate form is typically expressed in the form of a *Conjunctive Normal Form (CNF)* that can be directly accepted by the verification tool. For instance, in Majority-based verification discussed in [5], the intermediate form used is the MAJ/NOT netlist of the function.

For mapping the specification for evaluation, suitable micro-operations for crossbar operation have to be generated. For instance, in [5], the ReRAM micro-operations is converted into an intermediate form, viz. the *ReRAM Sequence Graph (ReSG)*. The ReSG representation for the MAJ/NOT netlist of Fig. 5(a) is shown in Fig. 5(d). The ReSG representation is then converted into a form compatible with the verification tool. Finally, a SAT solver verifies whether the two representations (viz. the clauses corresponding to the function netlist, and those generated from the ReSG) are equivalent. It may be noted that for other logic design styles (like Imply or Magic), some representation different from ReSG may be required for storing the information.

IV. Experimental Evaluation

We have conducted our experiments on some of the well-known benchmarks that are taken from ISCAS [8] and IWLS [1]. All the implementations including our proposed scheme of constructing the ReSG, checking equivalence (i.e. miter structure) and generating clauses are performed in Python 3.6. We have used the Z3 solver [11] for checking equivalence between the SAT clauses generated from the golden (i.e. MIG) and the reference (i.e. ReSG) representations. All the experiments are run on a 2.8 GHz machine with a dual core processor with 8GB RAM.

Table I summarizes the obtained results. The first three columns provide the details of the benchmark, i.e. name of the benchmark and the number of *Primary Inputs* (PI) as well as primary outputs (PO). The next three columns report the number of nodes (*#Nodes*) in the MIG representation of the respective benchmark, the number of resulting clauses (*#Clauses*) and the time taken to obtain the clauses (t_1). The next four columns provide the total number of micro-operations (*#Ops*), the number of nodes in the ReSG (*#Nodes*), the number of resulting clauses (*#Clauses*) and their generation time (t_2). The final column shows the time to check the equivalence between MIG and ReSG. All the times are shown in CPU seconds.

The table has two parts: equivalent and the non-equivalent. The benchmarks are divided into small (where $PI+PO \leq 20$) and large (where $PI + PO > 20$). The upper part of Table I reports the cases where MIG representations and the corresponding micro-operations (or ReSG) are functionally equivalent. The results demonstrate that our proposed scheme requires very few CPU seconds to complete the entire process that includes clause generation from MIGs and ReSGs, and obtaining solutions – equivalent (UNSAT) or non-equivalent (SAT) from resulting clauses. To generate the erroneous cases, we have modified the micro-operations by randomly inserting or deleting operations in the crossbar micro-operation sequence, 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 I shows the results for non-equivalent cases, where all the columns remain same as that of equivalent cases except the third column (ReSG) (due to the insertion or deletion of some micro-operations). Overall, the proposed approach can correctly identify the equivalence or non-equivalence between the MIG and its corresponding crossbar micro-operations.

TABLE I: Experimental results

Equivalent cases

	Benchmark			MIG			ReSG				
	Name	PI	PO	#Nodes	#Clauses	t_1 (s)	#Ops	#Nodes	#Clauses	t_2 (s)	SAT solve time (s)
	rd32	3	2	3	11	0.002	10	6	20	0.003	0.041
	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
small	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
	parity	16	1	24	73	0.003	75	72	217	0.02	0.077
	t481	16	1	25	51	0.002	39	36	109	0.005	0.063
	c6288	32	32	1867	1899	0.025	2381	2347	7073	0.095	4.031
	c1908	33	25	296	738	0.006	415	388	1189	0.018	0.399
	c432	36	7	95	233	0.003	133	124	379	0.009	0.140
	c499	41	32	292	762	0.006	390	356	1100	0.017	0.376
large	c3540	50	22	824	1989	0.013	1183	1159	3499	0.075	1.099
	c880	60	26	347	803	0.006	409	469	1169	0.017	0.266
	c5315	178	123	1376	3342	0.02	1548	1728	4398	0.059	1.621
	c7552	207	108	1384	3309	0.026	1687	1824	4629	0.06	2.684
	c2670	233	140	812	1451	0.009	751	986	1973	0.035	0.644

Non-equivalent cases

	Benchmark			MIG			ReSG				
	Name	PI	PO	#Nodes	#Clauses	t_1 (s)	#Ops	#Nodes	#Clauses	t_2 (s)	SAT solve time (s)
	rd32	3	2	3	11	0.002	9	5	17	0.003	0.041
	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
small	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
	parity	16	1	24	73	0.003	74	71	214	0.02	0.071
	t481	16	1	25	51	0.002	40	37	112	0.006	0.063
	c6288	32	32	1867	1899	0.025	2413	2379	7169	0.11	5.124
	c1908	33	25	296	738	0.006	412	385	1180	0.018	0.367
	c432	36	7	95	233	0.003	140	131	400	0.009	0.134
	c499	41	32	292	762	0.006	422	388	1196	0.021	0.367
large	c3540	50	22	824	1989	0.013	1175	1151	3475	0.073	1.078
	c880	60	26	347	803	0.006	408	468	1166	0.017	0.265
	c5315	178	123	1376	3342	0.02	1547	1727	4396	0.057	1.618
	c7552	207	108	1384	3309	0.026	1686	1823	4626	0.06	2.67
	c2670	233	140	812	1451	0.009	750	985	1970	0.031	0.64

V. Conclusions and Open Questions

In this paper we provide an overview of the verification of ReRAM crossbar mapping for in-memory computing. It has been recognized that formal verification methodology for this problem is important because it provides completeness as compared to simulation based methods. A general SAT based verification methodology is suggested to cover the various logic design styles used for in-memory computing using ReRAM. The importance of having a generalized micro-operation format for crossbar operations is also discussed and some of the requirements are identified. The existing file format is very specific to one particular logic design style and cannot be used for other logic design styles. The issue of intermediate representations is also discussed. Although formal verification methods are complete, they are time consuming and do not scale well for large functions. To this end more scalable verification strategies may need to be developed. Exploiting the inherent MAC realizations of the memristive crossbars in developing an in-memory SAT solver can be explored as a future work. In literature only Majority-based verification methodology exists, and we aim to cover other logic design styles like Imply and Magic as well in future.

Acknowledgement

This work was supported by the German Research Foundation (DFG) within the Project PLiM (DR 287/35-1, DR 287/35-2 and SH 1917/1-2).

REFERENCES

[1] C. Albrecht. IWLS 2005 benchmarks. Technical report, June 2005.

[2] L. Amaru, P.-E. Gaillardon, and G. De Micheli. Majority-inverter graph: A new paradigm for logic optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(5):806–819, 2015.

[3] J. Borghetti, G. S. Snider, P. J. Kuekes, J. J. Yang, D. R. Stewart, and R. S. Williams. Memristive switches enable stateful logic operations via material implication. *Nature*, 464(7290):873–876, 2010.

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

[5] 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 *ASP-DAC*, pages 489–494, 2023.

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

[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)*, pages 427–432. Ieee, 2016.

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

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

[10] F. Lalchhandama, M. Sahani, V. M. Srinivas, I. Sengupta, and K. Datta. In-memory computing on resistive ram systems using majority operation. *Journal of Circuits, Systems and Computers (Accepted)*, 31(4), 2022.

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

[12] 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)*, pages 948–953, 2016.

[13] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler. Logic synthesis for RRAM-based in-memory computing. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 37(7):1422–1435, 2018.

[14] D.B. Strukov, G.S.Snider, D.R. Stewart, and R.S. Williams. The missing memristor found. *Nature*, 453:80–83, 2008.

[15] N. Talati, S. D. Gupta, P. S. Mane, and S. Kvatinsky. Logic design within memristive memories using memristor-aided logic (MAGIC). *IEEE Trans. on Nanotechnology*, 15:635–650, 2016.

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

[17] D. N. Yadav, P. L. Thangkhiew, and K. Datta. Look-ahead mapping of Boolean functions in memristive crossbar array. *Integration*, 64:152–162, 2019.