Logic Design using Memristors: An Emerging Technology

(Embedded Tutorial)

Saeideh Shirinzadeh* Kamalika Datta† Rolf Drechsler*‡
*Department of Mathematics and Computer Science, University of Bremen, Bremen, Germany
†Department of Computer Science and Engineering, National Institute of Technology Meghalaya, India
‡Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
Email: kdatta@nitm.ac.in, {s.shirinzadeh,drechsler}@uni-bremen.de

***Abstract*—This paper provides an introduction to memristor, which is considered as the fourth circuit element along with resistor, inductor and capacitor. Memristors possess some unique properties, i.e. it can change the resistance under voltage control and can retain its value even after the voltage is withdrawn. Another property of memristors is their small feature size which makes them useful for design of ultra-compact memory systems. In addition, the resistive switching property of memristors allows to execute logic primitives and thus can also be used for implementing logic functions using various logic design styles studied in this paper. The paper also discusses memristor fabrication, circuit models, methods for implementing logic functions, and the various computing methodologies that can be used viz. near-memory computing and in-memory computing.**

***Index Terms*—Memristor, IMPLY, MAGIC, in-memory computing, logic synthesis**

I. Introduction

The memristor has been considered by scientists as the fourth fundamental circuit element after resistor, capacitor and inductor. Chua predicted the existience of memristors in 1971 [1], as a circuit element that directly relates magnetic flux and charge. The current-voltage characteristic of a memristor exhibits a pinched hysteresis loop, that allows switching of resistive states. In 2008, a group of researchers in the HP Labs were successful in fabricating a device [2] using a TiO_2 material doped with oxygen vacancy, which has similar resistive switching properties. The memristor has the unique property that its resistance value can be changed by applying a suitable voltage across it, and the resistance value does not change even if the voltage is withdrawn. Subsequently, various other research groups successfully demonstrated the fabrication of devices with similar properties [3], [4]. One of the biggest advantages that memristors offer is their extremely small size as compared to conventional MOS transistors. In fact, a memristor can be fabricated with feature size as small as 9 nm^2 [5].

Due to their ability to memorize past states, memristors can be used to build high capacity non-volatile resistive memory systems. Because of their extremely small sizes and regular structure, they can be very conveniently fabricated in a crossbar array. Over and above applications of memristors in memory systems, several works have been reported where they have been used for the implementation of logic functions and interconnections. There are several design styles for implementing logic functions using memristors, two of which are briefly stated below.

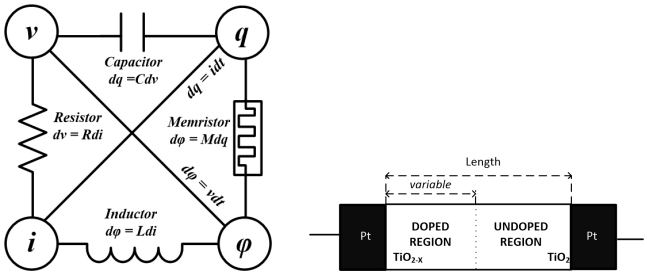
a) In memristor IMPLY logic [6], memristors can be used to realize the material implication operation ($A \rightarrow B = A' + B$). The initial values of A and B, and also the result are stored as resistance values in memristors. To implement a complex function, several IMPLY operations may need to be done in a particular sequence.
b) The MAGIC design style [7] uses only memristors to implement logic gates, where the inputs are applied as resistance values. MAGIC gate realizations can be mapped to crossbar arrays, and offer flexibility and scalability.
c) Recently, a number of works have been carried out on memristor crossbar arrays [8], [9], where a two-dimensional grid of memristors is created. By suitably initializing the memristors to known states, and applying suitable voltages to the rows and columns, logic functions can be computed as determined by the current flowing through the array. One problem with this approach is the existence of sneak paths in the crossbar, through which unwanted currents may start flowing resulting in erroneous output. However, because of the highly dense nature of the crossbar, this approach holds great potential.

II. Memristor: Fabrication and Modeling

The memristor represents a non-linear relationship between electrical charge and magnetic flux [1], as shown in Fig. 1(a). In 2008, Strukov et al. fabricated a memristor device at HP Lab. They used a TiO_2 material sandwiched between two platinum electrodes, with one of the regions doped with oxygen vacancy (TiO_{2-x}) (see Fig. 1(b)). The doped region has a lower resistivity than the undoped region. By applying a suitable voltage across the device, the doped region can be expanded or contracted thereby resulting in a change in resistance. When the voltage is withdrawn, the states of the oxygen vacancy carriers remain unchanged, and thus the device can remember or memorize its last resistance value. In addition to TiO_2, several other materials have also been explored by researchers.

A. Memristor Modeling

Memristors can be used in non-volatile storage applications, as well as for implementing logic operations. In typical logic



(a) (b)

Fig. 1. (a) Relationship between fundamental circuit elements. (b) Schematic diagram of TiO_2 memristor

operations, logic values are represented as distinct resistive states of the memristor. The resistive state of a memristor can be switched by applying voltages of suitable polarity and magnitude across it. For analyzing memristor based circuits, various circuit models for memristors have been proposed. This allows a designer to simulate the circuit designs using standard circuit simulation tools and analyze their performance. Some of the simulation models that have been proposed are:

1) Linear ion drift model [2]
2) Simmons tunnel barrier model [10]
3) Threshold adaptive memristor model (TEAM) [11]
4) Voltage threshold adaptive memristor model (VTEAM) [12]

One of the first memristor models to be proposed is the linear ion drift model, which is based on the simplified view of the HP memristor as shown in Fig. 1(b). In this model, the memristor is viewed as a combination of two variable resistors in series, one corresponding to the doped region and the other to the undoped region. The width of the doped region w is referred to as the *state variable*, and determines the conductivity of the memristor.

The following equations describe the drift-diffusion velocity and the time varying voltage in this model:

$$\frac{dw}{dt} = \frac{\mu_v R_{on} i(t)}{D} \quad (1)$$

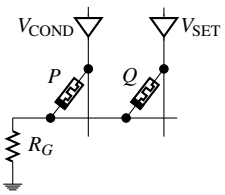
$$v(t) = \left(\frac{w(t)}{D}R_{on} + \left(1 - \frac{w(t)}{D}R_{off}\right)\right) i(t) \quad (2)$$

where D is the width of the memristor, μ_v is the average ion mobility of the TiO_2 region, and $w(t)$ is the thickness of the doped region as a function of time t, also called the state variable. R_{on} is the resistance when the width of the doped region $w(t)$ is D, and R_{off} is the resistance when $w(t)$ is 0. The total memristance of the device is given by:

$$M(q) = R_{off}\left(1 - \frac{\mu_v R_{on}}{D^2} q(t)\right) \quad (3)$$

where $v(t)$, $i(t)$ and $q(t)$ respectively denote the voltage, current and total charge flowing through the device at time t.

To overcome the limitations of the model when w approaches the boundaries of the device, and also to introduce non-linearities in ion drift, various window functions have been proposed, like Joglekar's window function [13], Biolek's window function [14], etc. The other memristor models directly incorporate non-linear behavior in the ion drift phenomena and as such are more accurate but also more computation intensive. Some of the recently proposed models like TEAM [11] and VTEAM [12] are widely used by researchers.



(a)

p	q	q'
0	0	1
0	1	1
1	0	0
1	1	1

(b)

Fig. 2. IMPLY operation. (a) Implementation of IMPLY using memristors. (b) Truth table for IMPLY ($q' \leftarrow p\,\text{IMPLY}\,q = \overline{p} + q$) [15].

III. Logic Synthesis with Memristors

In this section, we study two design styles for memristors, i.e., *Material Implication* (IMPLY) [15] and *Memristor-Aided LoGIC* (MAGIC) [7]. For the IMPLY style, we survey the use of known logic representations including *Binary Decision Diagram* (BDD), *And-Inverter Graph* (AIG), and *Majority-Inverter Graph* (MIG) for logic synthesis using memristors.

A. IMPLY Design Style

In [15], it was shown that *Material Implication* (IMPLY) can be executed using memristive switches. IMPLYtogether with FALSE operation, i.e., assigning the output to logic 0, makes a complete set to express any Boolean function [15]. This enables to synthesize arbitrary logic functions on memristive crossbars.

Fig. 2(a) shows the implementation proposed in [15] for an IMPLY gate [15]. The implementation needs two memristors denoted by P and Q which are connected to a load resistor R_G. The gate is controlled by three voltage levels V_{SET}, V_{COND}, and V_{CLEAR}. V_{CLEAR} and V_{SET} can be independently applied to the memristor to set it to 0, i.e., the FALSE operation, and 1, respectively. To perform IMPLY, two voltage levels V_{SET} and V_{COND} should be simultaneously applied to P and Q. The interaction of devices under the aforementioned voltage controls executes IMPLY according to truth table shown in Fig. 2(b) [15].

1) BDD based Synthesis: The starting point for BDD based synthesis is to realize the logic primitive designating each graph node, i.e., a 2-to-1 multiplexer (MUX), with memristors. Fig. 3 shows the MUX realization proposed for this purpose in [16]. This realization executes the MUX function within six steps performed in five memristors. Memristors S, X, and Y store the inputs and the two others, A and B, are required for the IMPLY operations. The corresponding computational steps are as follows:

1: $S = s, X = x, Y = y, A = 0, B = 0$
2: $a \leftarrow s\,\text{IMPLY}\,a = \bar{s}$
3: $a \leftarrow y\,\text{IMPLY}\,a = \bar{y} + \bar{s}$
4: $b \leftarrow a\,\text{IMPLY}\,b = y \cdot s$
5: $s \leftarrow x\,\text{IMPLY}\,s = \bar{x} + s$
6: $b \leftarrow s\,\text{IMPLY}\,b = x \cdot \bar{s} + y \cdot s$

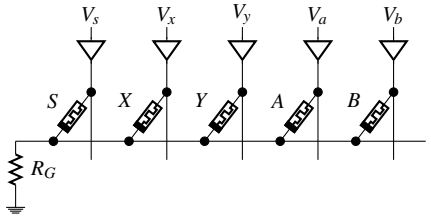


Fig. 3. The realization of MUX with memristors using IMPLY style [16]

The first step initializes the memristors with input variables or 0. Each of the remaining steps include a single IMPLY operation which evaluates the MUX function in step 6.

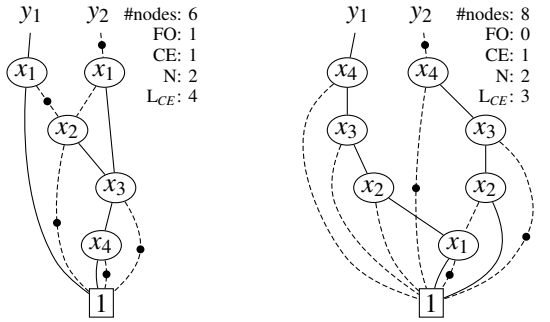
The methodology to evaluate a BDD can be sequential [16], i.e., node by node, or parallel [16], [17], [18], i.e., level by level, which is studied in this paper. For a given Boolean function, the parallel approach maps the corresponding BDD representation to a netlist of memristors using the MUX realization shown in Fig. 3. The approach starts computing a BDD from the bottom level of the graph and allocates one crossbar row to each node of the level. All the nodes in the level are computed in parallel in separate rows which means six time steps for a level. Thus, the total number of steps to evaluate the whole BDD is six times the number of BDD levels. The memristors storing the results of the computed level are then directly used as input memristors for the consecutive level. According to MUX realization, the number of memristors required for computing by this approach is equal to five times the maximum number of nodes in any BDD level.

Table I shows the number of steps and memristors required for implementing a BDD on a memristive crossbar. As the table shows, the presence of complemented edges and fanouts also increases the costs. Every complemented edge in the BDD needs to be inverted by an IMPLY operation (see step 2), which requires one extra memristor and time step. The required IMPLY operations for all complemented edges in a level can be performed in parallel, and thus for any level possessing ingoing complemented edges only one extra step is required. Moreover, to avoid data distortion the values of nonconsecutive fanouts should be copied to be used at fanout targets. The copy operation can be performed simultaneously in the first initialization step but still needs an extra memristor.

To lower the number of memristors and computational steps simultaneously, i.e., finding a trade-off between both cost metrics, a bi-objective optimization algorithm can be used. A genetic algorithm was proposed for this purpose (see [17], [19]). The general framework of the algorithm is based on NSGA-II (*Non-dominated Sorting Genetic Algorithm*) [20] which has been experimentally proven useful for solving NP-complete problems such as BDD optimization. Fig. 4 shows a BDD representing a function with four input variables and two outputs before and after optimization. As shown in the figure, both of the number of memristors and computational steps have decreased in the optimized BDD, while the number of nodes has increased. According to Table I, this improvement is due to the reductions in the number of nonconsecutive fanouts and levels with complemented edges.

TABLE I
THE COST METRICS OF IMPLY DESIGN STYLE FOR DIFFERENT LOGIC REPRESENTATIONS

Metric	**Definition\Value**	
$\mathbf{N}_i$	No. of nodes in the i^{th} level	
$\mathbf{CE}_i$	No. of ingoing complemented edges in the i^{th} level	
$\mathbf{RE}_i$	No. of ingoing regular edges in the i^{th} level	
FO	Maximum no. of nonconsecutive fanouts in any BDD level	
D	The depth of the graph	
$\mathbf{L}_{CE}$	No. of levels with ingoing complemented edges	
$\mathbf{L}_{RE}$	No. of levels with ingoing regular edges	
M	No. of memristors	
S	No. of steps	
BDD	$\#M = \max_{0\le i\le D}(5\cdot N_i + CE_i) + FO$	$\#S = 6\cdot D + L_{CE}$
AIG	$\#M = \max_{0\le i\le D}(3\cdot N_i + RE_i)$	$\#S = 3\cdot D + L_{RE}$
MIG	$\#M = \max_{0\le i\le D}(6\cdot N_i + CE_i)$	$\#S = 10\cdot D + L_{CE}$



(a) Initial
$\#M = 12$, $\#S = 28$

(b) Optimized
$\#M = 11$, $\#S = 27$

Fig. 4. Cost metrics of logic synthesis with memristors for an arbitrary BDD, (a) before (Initial), and (b) after optimization (Optimized)

2) AIG based Synthesis: A similar procedure presented for BDDs can be used for AIG based synthesis using memristors. Realization of a NAND gate using the IMPLY operation has been proposed in [15] as per the following.

1: $X = x, Y = y, A = 0$
2: $a \leftarrow x$ IMPLY $a = \bar{x}$
3: $a \leftarrow y$ IMPLY $a = \bar{x} + \bar{y}$

For crossbar implementation, the realization above can be used together directly for nodes with complemented edges while it needs a negation for regular edges. Using the parallel evaluation method explained before, the nodes of each AIG level are evaluated simultaneously, such that the employed memristors can be reused for the successive levels. After computing each level, the memristors in a level are automatically updated with the results of the IMPLY operations. Then, the memristors are used as the inputs of the upper level and this procedure is continued until the target function is evaluated.

Crossbar implementation of an AIG requires as many NAND gates as the maximum level size, i.e., the number of nodes in the level, over the entire graph. The major part of the corresponding

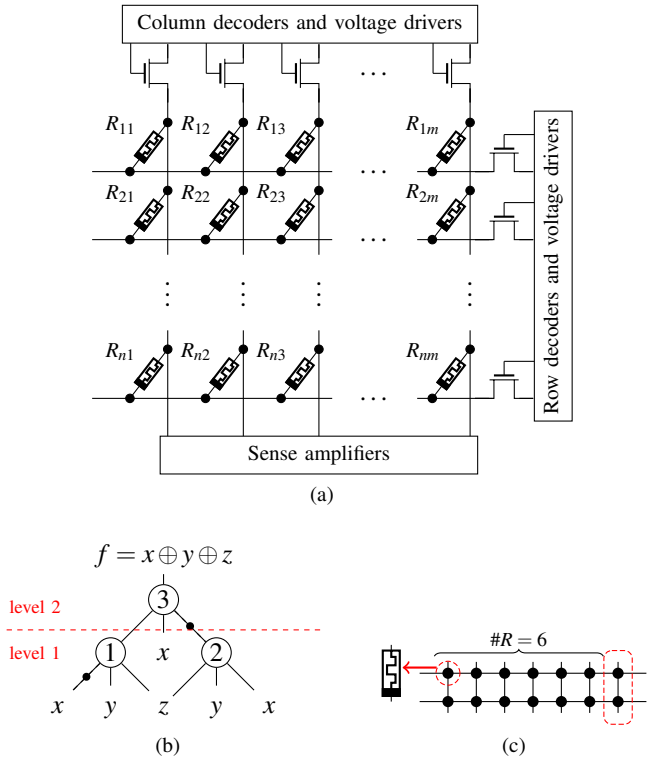


Fig. 5. Standard memristive crossbar for the presented synthesis approach. (a) MIG representing a three bit XOR gate, and (b) upper-bound crossbar for its crossbar implementation.

number of memristors and time steps for implementation is three times the number of required NAND gates and three times the number of AIG levels, respectively (see Table I). The values in the table also include additional memristors needed for the required NOT operations, i.e., the regular edges in the realization.

The costs shown in Table I can be reduced with respect to the number of time steps or devices, addressing the latency and area of the resulting implementations. ABC [21] provides commands for rewriting AIGs to more efficient ones. To lower the number of required memristors, the command *dc2* can be used to lower the number of nodes in the graph.

Latency of the designs can also be reduced before mapping them to their corresponding netlist of memristors by the ABC command *if -x -g*. The command minimizes the depth of the AIG which is indeed the most significant term in the number of time steps because of being multiplied by a factor of three. Applying any of these commands iteratively can considerably reduce the targeted cost metric.

3) MIG based Synthesis: The realization for the majority gate using the IMPLY operation was proposed in [22] and is as the following:

01: $X = x, Y = y, Z = z$ $A = 0, B = 0, C = 0$	**06:** $c \leftarrow y$ IMPLY $c = \overline{x+y}$
02: $a \leftarrow x$ IMPLY $a = \bar{x}$	**07:** $c \leftarrow z$ IMPLY $c = \overline{x \cdot z + y \cdot z}$
03: $b \leftarrow y$ IMPLY $b = \bar{y}$	**08:** $a = 0$
04: $y \leftarrow a$ IMPLY $y = x + y$	**09:** $a \leftarrow b$ IMPLY $a = x \cdot y$
05: $b \leftarrow x$ IMPLY $b = \bar{x} + \bar{y}$	**10:** $a \leftarrow c$ IMPLY $a = x \cdot y + y \cdot z + x \cdot z$.

The realization needs ten IMPLY steps and six memristors. However, it is worth mentioning that a majority based design style beyond the scope of this paper was proposed in [23] and was shown more efficient when using MIGs [22].

The cost metrics for the parallel evaluation of MIG based approach using IMPLY are given in Table I. The values in the table are obtained similarly to the explanation given for BDDs and AIGs but according to the realization of majority gate. The complete set of logic axioms to optimize an arbitrary MIG to a logically equivalent MIG with smaller number of nodes or levels was proposed in [24]. In [18], several MIG optimization algorithms have been proposed which can be employed to lower the cost metrics of crossbar implementation with respect to both area and latency.

Fig. 5(a) shows a standard multi-row/column memristive crossbar required for presented synthesis approach. For the sake of clarity, we present step-by-step implementation of an example MIG shown in Fig. 5(b). The MIG has a maximum level size of 2 which needs an upper bound of 12 (2×6) memristors placed in two rows besides one more for the ingoing complemented edges (see Fig. 5(c)). As Table I suggests, the computation needs 22 steps, 2×10 for the two levels and two more steps for the complemented edges. The implementation steps are listed below:

Initialization:	$R_{ij} = 0$;
1: Loading variables for level 1:	$R_{11} = x, R_{12} = y, R_{13} = z$; $R_{21} = x, R_{22} = y, R_{23} = z$;
2: Negation for node 1:	$R_{17} \leftarrow x$ IMPLY $R_{17} : R_{17} = \bar{x}$;
3-11: Computing level 1:	node 1: $R_{14} = M(\bar{x}, y, z)$; node 2: $R_{24} : M(x, y, z)$;
12: Loading variables for level 2:	$R_{11} = x, R_{12} = M(x, y, z), R_{13} = M(\bar{x}, y, z)$ $R_{14} = R_{15} = R_{16} = R_{17} = 0$;
13: Negation for node 3:	$R_{17} \leftarrow R_{12}$ IMPLY R_{17} : $R_{17} = \overline{R_{12}} = \overline{M}(x, y, z)$;
14-22: Computing level 2 (root node):	$R_{14} = M(M(\bar{x}, y, z), x, \overline{M}(x, y, z))$;

The names R_{i1} to R_{i6} in the implementation steps above respectively designate the resistance states of the memristors shown by X, Y, Z, A, B, and C used for the realization of the majority gate. For initialization, all of the memristors in the entire crossbar are cleared. Since IMPLY needs all the variables to be stored in the same horizontal line, there may be a need to have several copies of primary inputs or intermediate results at different rows. This is shown in step 1, where each variable of nodes 1 and 2 are loaded into two memristors in both rows. Step 2 computes the complemented edge, i.e., showed by a dot, of node 1 in the seventh memristor considered for this case at the end of first row, R_{17}. Steps 3-11 compute both nodes at level one which updates R_{14} and R_{24} by the results of the computations. Step 12 loads the inputs of level 2 which includes a single node, i.e., the root node. The complemented edge originating at node 2 is negated in step 13, and then the root node is computed in step 22.

B. MAGIC Design Style

MAGIC is the acronym for memristor aided logic [7], which is a stateful logic design style where the resitance values represent the logic states. The input(s) and output values are stored in different memristors. All basic gates (NOT, AND,

OR, NOR, NAND) can be implemented using the MAGIC design style, as illustrated in Fig. 6.

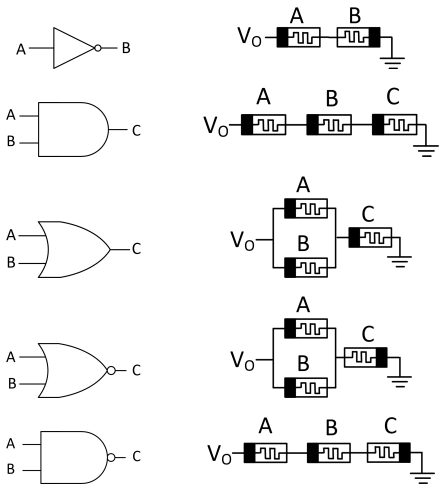


Fig. 6. Basic gates realized using the MAGIC design style.

A MAGIC gate operation requires two sequential steps:

i) In the first step the output memristor is initialized to a known value (either 0 or 1). For non-inverting gates like AND/OR it is initialized to 0, while for inverting gates like NOT/NOR/NAND it is initialized to 1.

ii) In the second step a suitable voltage V_0 is applied to the input memristor(s). The voltage across the output memristor depends upon the logical state of the input and output memristor, and switches accordingly.

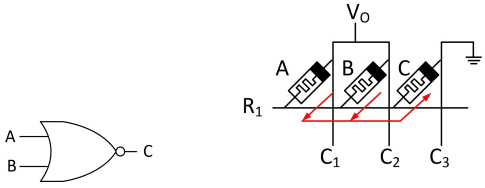


Fig. 7. 2-input MAGIC NOR gate and equivalent circuit mapped in crossbar

Though all gates can be implemented using the MAGIC design style; however, only NOR and NOT gate can be mapped to memristor crossbars. A crossbar consists of horizontal and vertical nanowires, where the memristors are fabricated at the junctions. Fig. 7 shows the crossbar mapping of a 2-input NOR gate. For synthesizing larger boolean functions, the functions are first represented in terms of NOR and NOT gates, and then various mapping techniques [25], [26] are used to map the gates to the crossbar.

IV. Near Memory and In-Memory Computing: Challenges and Implementation Issues

A. In-Memory Computing

As explained in Section III, both of the two discussed styles for logic-in-memory computing result in sequences of computational steps. Accordingly, lowering the number of steps as well as the number of memristors are considered important objectives for in-memory computing architectures [27]. For this purpose, usually memristors storing the intermediate results are reused for computation in the successive steps to avoid usage of extra steps or devices [28]. The successive switching of some memristors compared to the other devices in a memristive crossbar results in an unbalanced write traffic. Considering the fact that the resistive switching devices have basically low write endurance, i.e., in the best cases about 10^{10} [29] to 10^{11} write counts [30], this problem can lower the reliable lifetime of the entire crossbar architecture.

Using IMPLY for synthesis, unbalanced distribution of writes happens due to the lack of commutativity property, which results in higher write traffic in the memory cell storing the output of the operation. For example in [15], an implementation for the NAND gate requiring two memristors and three time steps was proposed. The implementation needs to switch one memristor, the so-called work device, at each of the three steps while the other memristor is switched only once for initialization which is called input memristor. Similarly, in [31], a synthesis approach has been proposed which considers only two work memristors besides N input devices, where N is the number of input variables of the Boolean function. In this case, the work devices wear out fast and therefore lower the lifetime of the design. In such implementations, the write traffic can be distributed more evenly only by allocating extra devices to replace those with high write counts. This also costs additional steps to copy the contents of memristors which is often avoided for the sake of efficiency.

B. Near-Memory Computing

In [26] Thangkhiew et al. presented a near-memory mapping scheme, where various adder circuits were mapped to memristive crossbar array using the MAGIC design style. The term *near-memory* signifies that the input memristors need to be configured prior to the evaluation as opposed to in-memory computing, where the inputs are already present *in memory*. Broadly two mapping techniques are discussed, serial and parallel. In serial mapping the gates are evaluated one by one, and in parallel mapping, several gates are evaluated in a single time step. For carrying out the operations, several voltages [25] need to be applied to the rows/columns of the crossbar as:

a) A voltage V_{set} applied in column i and GND to row j initializes the memristor at (i, j) to 1.

b) A voltage V_{clear} applied in column i and GND to row j initializes the memristor at (i, j) to 0.

c) A voltage V_o applied to the columns corresponding to inputs of a gate, and GND to the column corresponding to the output, performs the NOR gate operation.

d) During a NOR gate operation, a voltage V_{iso} applied to a row i disables the gate operation in row i.

To reduce the hardware cost, a level-wise mapping approach is presented in which the gates are mapped level by level. To illustrate the parallel and level-wise mapping let us consider the circuit shown in Fig. 8(a).

The snapshot for the evaluation of gates in level 1 is shown in Fig. 8(b). It can be observed that in level 1 the inputs (A,C

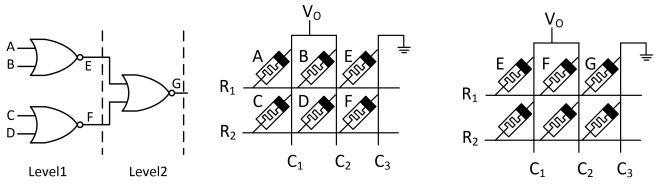


Fig. 8. (a) Evaluation of Level 1. (b) Evaluation of Level 2

and B, D) and output (E, F) of the two gates are aligned in columns. First the gate input values are all initialized to 0. Then some of the inputs are selectively initialized to 1. For instance, in Fig. 8(b), if memristor A is at logic 1 and C is at logic 0, then V_{set} is applied to C_1 and V_{iso} is applied to R_2. After these initialization steps the evaluation of level 1 is performed by applying V_o and GND to the columns. Next, the outputs from level 1 are read and stored in buffers. In level 2, all the cells in the crossbar are cleared by performing the reset operation. Then the inputs are read from the buffer and the memristors are again initialized. Then the evaluation is carried out as shown in Fig. 8(c).

V. Conclusion

A brief introduction to memristors and its applications in logic design has been discussed in this paper. After introducing the IMPLY design style, three scalable synthesis approaches using BDDs, AIGs, and MIGs are discussed in some detail. Finally, the MAGIC design style is discussed and various issues in crossbar mapping and evaluation are also highlighted.

References

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

[2] D. B. Strukov, G. S. Snider, D. R. Steward, and R. S. Williams, "The missing memristor found," *Nature*, vol. 453, no. 7191, pp. 80–83, May 2008.

[3] N. Duraisamy, N. M. Muhammad, H. C. Kim, J. D. Jo, and K. H. Choi, "Fabrication of Tio2 thin film memristor device using electro-hydrodynamic inkjet printing," *Thin Solid Films (Elsevier)*, vol. 520, pp. 5070–5074, 2012.

[4] E. M. Gale, A. Adamatzky, and B. L. Costello, "Fabrication and modelling of titanium dioxide memristors," in *RSC Younger Members Symposium*, 2012. [Online]. Available: http://eprints.uwe.ac.uk/17057

[5] A. Sinha, M. S. Kulkarni, and C. Teuscher, "Evolving nanoscale associative memories with memristors," in *Proc. of 11th IEEE Conference on Nanotechnology*, Aug 2011.

[6] S. Kvatinsky, G. Satat, N. Wald, E. G. Friedman, A. Kolodny, and U. C. Weiser, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," *IEEE Trans. VLSI Syst.*, vol. 22, no. 10, pp. 2054–2066, Oct 2014.

[7] S. Kvatinsky, D. Belousov, S. Liman, G. Satat, N. Wald, E. Friedman, A. Kolodny, and U. Weiser, "MAGIC – Memristor-Aided Logic," *IEEE Trans. Circuits Syst. II*, vol. 61, no. 11, pp. 895–899, Nov 2014.

[8] S. N. Truong and K. S. Min, "New memristor-based crossbar array architecture with 50% area reduction and 48% power saving for matrix-vector multiplication of analog neoromorphic computing," *Journal of Semiconductor Technology and Science*, vol. 14, no. 3, pp. 356–363, June 2014.

[9] A. Velasquez and S. K. Jha, "Automated synthesis of crossbars for nanoscale computing using formal methods," in *Proc. IEEE/ACM Intl. Symp. on Nanoscale Architectures*, 2015.

[10] J. G. Simmons, "Electric tunnel effect between dissimilar electrodes separated by a thin insulating film," *Journal of applied physics*, vol. 34, no. 9, pp. 2581–2590, 1963.

[11] S. Kvatinsky, E. G. Friedman, A. Kolodny, and U. C. Weiser, "TEAM: threshold adaptive memristor model," *IEEE Trans. Circuits Syst. I*, vol. 60, no. 1, pp. 211–221, 2013.

[12] S. Kvatinsky, M. Ramadan, E. G. Friedman, and A. Kolodny, "VTEAM: A general model for voltage-controlled memristors," *IEEE Trans. Circuits Syst. II*, vol. 62, no. 8, pp. 786–790, Aug 2015.

[13] Y. N. Joglekar and S. J. Wolf, "The elusive memristor: properties of basic electrical circuits," *European Journal of Physics*, vol. 30, no. 4, p. 661, 2009.

[14] D. Biolek, V. Biolkova, and Z. Biolek, "SPICE model of memristor with nonlinear dopant drift," *Radioengineering*, 2009.

[15] J. Borghetti, G. Snider, P. Kuekes, J. Yang, D. Stewart, and R. Williams, "Memristive switches enable stateful logic operations via material implication," *Nature*, vol. 464, no. 7290, pp. 873–876, 2010.

[16] S. Chakraborti, P. Chowdhary, K. Datta, and I. Sengupta, "BDD based synthesis of Boolean functions using memristors," in *IDT*, 2014, pp. 136–141.

[17] S. Shirinzadeh, M. Soeken, and R. Drechsler, "Multi-objective BDD optimization for RRAM based circuit design," in *IEEE International Symposium on Design and Diagnostics of Electronic Circuits & Systems*, 2016, pp. 46–51.

[18] S. Shirinzadeh, M. Soeken, P.-E. Gaillardon, and R. Drechsler, "Logic synthesis for RRAM-based in-memory computing," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, 2017.

[19] S. Shirinzadeh, M. Soeken, and R. Drechsler, "Multi-objective BDD optimization with evolutionary algorithms," in *Genetic and Evolutionary Computation Conference*, 2015, pp. 751–758.

[20] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, 2002.

[21] Berkeley Logic Synthesis and Verification Group, "ABC– A System for Sequential Synthesis and Verification," *http://www.eecs.berkeley.edu/ alanmi/abc/*, 2005.

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

[23] P. Gaillardon, L. G. Amarù, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli, "The programmable logic-in-memory (PLiM) computer," in *Design, Automation & Test in Europe*, 2016, pp. 427–432.

[24] L. G. Amarù, P.-E. Gaillardon, and G. De Micheli, "Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization," in *Design Automation Conference*, 2014, pp. 194:1–194:6.

[25] N. Talati, S. Gupta, P. Mane, and S. Kvatinsky, "Logic Design Within Memristive Memories Using Memristor-Aided loGIC (MAGIC)," *IEEE Trans. Nanotechnol.*, vol. 15, no. 4, pp. 635–650, July 2016.

[26] P. L. Thangkhiew, R. Gharpinde, P. V. Chowdhary, K. Datta, and I. Sengupta, "Area efficient implementation of ripple carry adder using memristor crossbar arrays," in *IEEE International Design and Test Symposium*, 2016, pp. 142–147.

[27] M. Soeken, P. Gaillardon, S. Shirinzadeh, R. Drechsler, and G. De Micheli, "A PLiM computer for the internet of things," *IEEE Computer*, vol. 50, no. 6, pp. 35–40, 2017.

[28] S. Shirinzadeh, M. Soeken, P. Gaillardon, G. De Micheli, and R. Drechsler, "Endurance management for resistive logic-in-memory computing architectures," in *Design, Automation & Test in Europe*, 2017, pp. 1092–1097.

[29] H. Y. Lee, Y. S. Chen, P. S. Chen, P. Y. Gu, Y. Y. Hsu, S. M. Wang, W. H. Liu, C. H. Tsai, S. S. Sheu, P. C. Chiang, W. P. Lin, C. H. Lin, W. S. Chen, F. T. Chen, C. H. Lien, and M. J. Tsai, "Evidence and solution of over-reset problem for HfOX based resistive memory with sub-ns switching speed and high endurance," in *IEEE International Meeting on Electron Devices*, 2010, pp. 19.7.1–19.7.4.

[30] Y. B. Kim, S. R. Lee, D. Lee, C. B. Lee, M. Chang, J. H. Hur, M. J. Lee, G. S. Park, C. J. Kim, U. I. Chung, I. K. Yoo, and K. Kim, "Bi-layered RRAM with unlimited endurance and extremely uniform switching," in *Symposium on VLSI Technology*, 2011, pp. 52–53.

[31] E. Lehtonen, J. Poikonen, and M. Laiho, "Two memristors suffice to compute all Boolean functions," *Electronics Letters*, vol. 46, pp. 230–231, 2010.