Memristive Logic-in-Memory Implementations: A Comparison

Pietro Inglese, Elena Ioana Vatajelu, Giorgio Di Natale

▶ **To cite this version:**

Pietro Inglese, Elena Ioana Vatajelu, Giorgio Di Natale. Memristive Logic-in-Memory Implementations: A Comparison. 16th International Conference on PRIME, Jul 2021, Online, Germany. hal-03370877

HAL Id: hal-03370877
https://hal.science/hal-03370877

Submitted on 8 Oct 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - NonCommercial 4.0 International License

Memristive Logic-in-Memory Implementations: A Comparison

Pietro Inglese, Elena Ioana Vatajelu, Giorgio Di Natale
Univ. Grenoble Alpes, CNRS, Grenoble INP*, TIMA, Grenoble, France
{pietro.inglese,ioana.vatajelu,giorgio.di-natale}@univ-grenoble-alpes.fr

Abstract — The technology evolution addresses the demand for faster computers. Despite the achieved speed-up in terms of memory and computation performances, the communication between the memories and the processor remains a bottleneck of today's computers. The Computation in Memory (CiM) paradigm aims at solving this problem by moving the computation directly inside the memory, eliminating thus the need for data transfer between memory and processor. Among the available CiM implementations, this study focuses on the Logic-in-Memory (LiM) solutions, i.e., digital operations to accelerate Boolean Logic. This work provides a comparison among the most prominent LiM solutions in terms of required memory resources (i.e., number of memristors) and number of operations.

Keywords — Emerging technologies, Logic-in-Memory, Memristors, Non-Volatile memories

I. INTRODUCTION

Among the most important challenges faced by today's computing systems are the memory wall (caused by the uneven evolution of processing speed and memory access times and bus data transfer) and the energy efficiency. Emerging non-volatile memories are widely studied today as means to maximize energy efficiency mainly due their ability to reduce the static power consumption. These memories include Resistive Random-Access Memories (RRAMs), Phase Change Memories (PCMs), and Spin-Transfer Torque Magnetic RAMs (STT-MRAMs). Another perceived advantage of emerging non-volatile memories resides in their physical capabilities which can be exploited to perform logic or arithmetic operations directly inside the memory array, therefore providing a solution to bypass the memory wall issue.

Several solutions to mitigate the memory wall by reducing the data movement between memory and CPU have been proposed. They can be classified in 3 main categories: (i) Computation Near Memory [1], where the memory core is placed as close as possible to the CPU, permitting to have a shorter bus and therefore decrease the latency, (ii) Computation via LUT, that exploits a Look-Up Table storing pre-calculated operations, and the (iii) Computation in Memory (CiM) which exploits memory technologies (both classical and emerging) to perform calculations directly within the memory array. The latter is considered the most efficient, since it is more flexible than the Computation via LUT and completely eliminates the need for data transfer via buses.

Depending on the exploited physical memory device characteristic, CiM can allow to perform analog or digital computations. Analog computations are mainly used to perform additions and matrix multiplications mainly to design accelerators for machine learning. Digital computations are used to accelerate Boolean logic. In this scenario, a part or the whole classical Arithmetic Logic Unit (ALU) embedded in the processor is actually implemented directly within memory. This is referred to as Logic-in-Memory (LiM) and it is the main focus of this paper.

In the last five years, the number of research papers dealing with this topic on different levels of abstraction has increased exponentially. In this context, research is focused on the design of LiM architectures, the development of LiM-compatible instructions set, the methods for system integration and development of the programming model for LiM integration in computing systems. Nevertheless, the actual status of the research is fragmented and the reproduction of the reported results, along with the choice of an implementation to be adopted, are not trivial. For instance, [2] presents a review on in-Memory Computing, focusing on the memories enabling it and on its applications, [3] offers a classification of the in-Memory Computing solutions, while [4] describes several adder implementations. Besides the fragmentation, the use of existing electrical models and their parameters is not always supported by physical measurements on real devices.

While these comparative surveys show the characteristics of existing solutions, a thorough analysis of the implementations of Boolean functions is still missing. Therefore, in this work we present a study of the existing LiM implementations in order to perform a fair comparison in terms of resources and efficiency. More in particular, we present a thorough study of simple Boolean functions implemented in-memory resulting in a comprehensive comparison of LiM solutions in terms of required number of memristors and number of operations.

The remainder of this paper is organized as follows. Section II presents the Logic-in-Memory solutions and describes the basic functioning and the primitive logics enabled. Section III presents an analysis of the described solutions through the basic Boolean logic blocks and a full adder implementation. Finally, Section IV concludes the paper.

II. BACKGROUND

This section describes basic memory array modifications to enable LiM and describes some selected LiM solutions.

In the traditional use of a memory array, a memory cell is selected by means of address decoders, it is written into by the write driver and read from with the help of the sense amplifiers. The voltage levels required to enable the operations on the memory cell are set by the voltage regulators. One memory array communicates with the processor or other memory arrays by means of bus connections. In order to enable the LiM operations, several changes need to be implemented to the memory array or/and to its peripheral circuitry. In this context, the peripheral circuitry consists of standard memory periphery (sense amplifier, write driver, address decoders). Moreover, in some instances, additional logic is added to enable computation.

Several LiM proposals exist in literature, some are general, and can be used with any memory technology, others take advantage of the device physics and are only suitable to be implemented in a specific technology. In addition, some of the

* Institute of Engineering Univ. Grenoble Alpes

existing LiM solutions are designed to implement specific logic functions [5]–[11] henceforth called "primitive operations", while others propose solutions for the implementation of any Boolean function [12].

The existing LiM solutions can be classified depending on the way the inputs are stored (the memory content, i.e., stateful logic, or an electrical signal, i.e., non-stateful logic) and depending on where the operations are performed (in the memory array, or in the periphery). In this context, three main LiM classes can be distinguished. They are described in the following and their characteristics are summarized in Figure 1.

A. *Stateful Logic in LiM Array*

The operations are performed within the memory array and the data are stored as memory content. This solution is proposed only for memristive crossbar (1R-RRAM) arrays. The input data are stored within the memory array, and the output (computation result) is obtained as memory content within the memory array. Inputs and outputs are coded as the resistive states of the storing memristors.

In order to enable LiM operations within the memory array, several conditions need to be respected: (1) the memory cells containing the input data and the memory cells to store the result of the computation must share the same row (column); (2) access to multiple memory cells should be enabled; (3) specific control voltages (different than the memory read/write voltages) to be applied for the completion of logic operations. As a consequence, the write driver, the voltage regulator and the address decoders of standard memory array have to be modified to enable LiM.

LiM solutions pertaining to this class include: *Memristor-Aided Logic - MAGIC* [5], with NOT and NOR as primitive operations, *FELIX* [6], [7] with OR, NAND and MIN as primitive operations, *IMPLY* [8], [9] with Boolean implication as primitive operation, and Stateful Three-Input Logic [10] with ORNOR3 (i.e., input1 OR (input2 NOR input3)) as primitive operation.

B. *Stateful Logic in LiM Array and its Periphery*

The operations are performed within the memory array periphery or by means of additional logic and the input data are stored as memory content. This solution can be used with any memory technology. The input data are stored within the memory array, while the output (computation result) is obtained as a voltage (or current) outside of the memory array.

In order to enable LiM operations within the periphery of the memory array, several conditions need to be respected: (1) the memory cells containing the input data must share the same column; (2) access to multiple memory cells should be enabled by modifying the address decoders; (3) the sense amplifiers should be modified such that different references are allowed. We refer to this class of solutions as Logic in Periphery (LIP).

The *MRIMA* architecture [12] is based on the Logic In Periphery: it exploits re-configurable Sense Amplifiers (SAs) to perform arithmetic and logic operations on STT-MRAM. All Boolean functions can be implemented with this solution by resorting to additional combinational gates.

C. *Non-Stateful Logic in LiM Array and its Periphery*

The operations are performed within the memory array and by using additional logic, and the data are coded partially as memory content and partially as voltage levels. This solution can be used with resistive technology only. It uses two types of input data: (1) memory content, (2) voltage level, while the output (computation result) is obtained as memory content within the memory array.

In order to enable this type of LiM operation several conditions need to be respected: (1) specific control voltages (different from the memory read/write voltages) to be applied for the completion of logic operations, (2) specific registers to store the inputs to be given as voltage levels. As a consequence, the write driver, the voltage regulator and the address decoders of standard memory array have to be modified to enable LiM.

An implementation of this solution is *PLiM* [11], which implements, as primitive operation, a special case of majority voter, where one of the inputs is negated (a.k.a., *Resistive majority*).

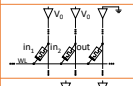
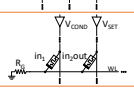
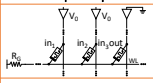
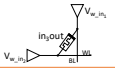
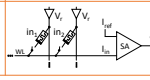
	CiM Solution	#mem pts	Gate	Remarks	Technology	Operations
LIM Array Stateful Logic	MAGIC (NOR2)	3		input non-destructive, parallelizable	Memristive crossbar (1R-RRAM)	1. Initialize R_{out} 2. Apply proper voltages to R_{in}, R_{out} 3. Obtain output in R_{in_out}
	IMPLY A → B	2		input destructive, parallelizable		
	Stateful Three-Input Logic (ORNOR3)	3		input destructive		
LIM Array + Periphery – Non Stateful Logic (Hybrid inputs)	PLiM (RMAJ3)	3 (1 in the array, 2 external)		input destructive, requires input pre-processing	Memristive crossbar (1R-RRAM)	1. Read inputs R_{in} and convert to voltages 2. Apply the voltages to R_{in_out} 3. Obtain output in R_{in_out}
LIM Array + Periphery Stateful Logic	Logic in Periphery	2		input non-destructive, additional step to store output in memory	SRAM 1T1R-RRAM STT-MRAM	1. Apply voltage on multiple rows 2. Obtain output (via SA) 3. [Store output in memory]

Figure 1 – Primitive logic gates: Column 2 (CiM Solution) lists the considered LiM solutions and the corresponding primitive operations. The number of memory cells needed to implement a 2-input (1-bit) primitive operation is summarized in Column 3 (#mem pts), while the schematic of the "primitive operation" gate for each solution is illustrated in Column 4 (Gate). Column 7 (Operations) lists the algorithm executed to obtain the result of the primitive operation. The executed operation can be input-destructive or not (see column 5, Remarks). An input-destructive operation is an operation that changes the value of the inputs after it is executed.

III. PROPOSED METHOD AND RESULTS

The goal of this work is to provide a comprehensive comparison of existing LiM solutions and understand their implementation complexity. The analysis has been performed on basic Boolean functions, in order to be as generic as possible and to provide the designer an indication of implementation complexity and cost of each LiM solution. In addition, this study can give an indication of which LiM solution is more suitable for a target application, depending on the most used Boolean functions.

In order to achieve a fair comparison among all solutions, we mapped all the 0-input logic functions (TRUE and FALSE), 1-input logic functions (COPY, NOT), 2-input logic functions (NOR, OR, NAND, AND, XNOR, XOR, NIMPLY, IMPLY) and the Full Adder as 3-input logic function, by using the primitive operations of *MAGIC* (and its extensions), *IMPLY* (and *ORNOR3*) and *PLiM* solutions. The full adder has as inputs A, B and C_{IN} while the outputs are S and C_{OUT}.

$$C_{OUT} = ((A + B)' + (B + C)' + (C + A)')'$$
$$S = (((A' + B' + C')' + ((A + B + C)' + C_{OUT})')')'$$

Tables I and II summarize the results of this study. Table I shows the mapping of all considered Boolean functions on LiM primitive operations. For each LiM implementation, the primitive operations are written in blue. Each row contains the mapping of the Boolean function defined in the first column.

	LIM Array *Stateful Logic*					*LIM Array + Periphery* *Non Stateful Logic (Hybrid inputs)*
	MAGIC-based		IMPLY-based			PLiM
	MAGIC	FELIX	IMPLY	IMPLY (non-destructive)	ORNOR3	RMAJ
Primitives	**MAGIC NOT**(in1,out) **MAGIC NOR**(in1,in2,out)	**FELIX OR**(in1,in2,out) **FELIX NAND**(in1,in2,out) **FELIX MIN**(in1,in2,in3out)	**IMPLY_IMPLY**(in1,in2out):		**ORNOR3**(in1out,in2,in3)	**RMAJ**(in1,in2,in3out)
COPY	**MAGIC COPY**(in1,f1,out): 1-2) MAGIC_NOT(in1,f1) 3-4) MAGIC_NOT(f1,out)	**FELIX COPY**(in1,f1,out): 1) Write0(f1) 2-3) FELIX_OR(in1,f1,out)	**IMPLY COPY**(in1,f1,out): 1-2) IMPLY_NOT(in1,f1) 3-4) IMPLY_NOT(f1,out)			**RMAJ COPY**(in1,out): 1) Write0(out) r1) Read(in1) 2) RMAJ(in1,0,out):
NOT	**MAGIC NOT**(in1,out): 1) Write0(out) 2) MAGIC(in1,out)		**IMPLY NOT**(in,out): 1) Write0(out) 2) IMPLY(in,out)			**RMAJ NOT**(in1,out): 1) Write0(out) r1) Read(in1) 2) RMAJ(1,in1,out)
NOR	**MAGIC NOR**(in1,in2,out): 1) Write0(out) 2) MAGIC(in1,in2,out)		**IMPLY NOR**(in1,in2,out): 1-3) IMPLY OR(in1,out,in2) 4-5) IMPLY_NOT(in2,out)	**IMPLY NOR ND**(in1,in2,f1,out): 1-7) IMPLY_OR_ND(in1,in2,out,f1) 8-9) IMPLY_NOT(f1,out)	**ORNOR3 NOR**(in1,in2,out): 1) Write0(out) 2) ORNOR3(out,in1,in2)	**RMAJ NOR**(in1,in2,f1,out): 1-4,r3) RMAJ OR(in1,in2,out,f1) 5-6,r4) RMAJ_NOT(f1,out)
OR	**MAGIC OR**(in1,in2,f1,out): 1-2) MAGIC_NOR(in1,in2,f1) 3-4) MAGIC_NOT(f1,out)	**FELIX OR**(in1,in2,out): 1) Write0(out) 2) FELIX(in1,in2,out)	**IMPLY OR**(in1,f1,in2out): 1-2) IMPLY_NOT(in1,f1) 3) IMPLY IMP(f1,in2out)	**IMPLY OR ND**(in1,in2,f1,out): 1-4) IMPLY_COPY(in2,f1,out) 5-6) IMPLY_NOT(in1,f1) 7) IMPLY(f1,out)	ORNOR3 OR ND(in1,in2,out): 1-2) ORNOR3_NOR(in1,in2,f1) 3-4) IMPLY_NOT(f1,out)	**RMAJ OR**(in1,in2,f1,out): 1) Write1(out) 2-3,r1) RMAJ_NOT(in2,f1) r2-r3) Read(in1,f1) 4) RMAJ(in1,f1,out)
NAND	**MAGIC NAND**(in1,in2,f1,f2,out): 1-8) MAGIC_AND(in1,in2,out,f1,f2) 9-10) MAGIC_NOT(f2,out)	**FELIX NAND**(in1,in2,out): 1) Write1(out) 2) FELIX(in1,in2,out)	**IMPLY NAND**(in1,in2,out): 1-2) IMPLY_NOT(in2,out) 3) IMPLY(in1,out)			**RMAJ NAND**(in1,in2,f1,out): 1-4,r4) RMAJ_AND(in1,in2,out,f1) 5-6,r5) RMAJ_NOT(f1,out)
AND	**MAGIC AND**(in1,in2,f1,f2,out): 1-2) MAGIC_NOT(in1,f1) 3-4) MAGIC_NOT(in2,f2) 5-6) MAGIC_NOR(f1,f2,out)	**FELIX AND**(in1,in2,f1,out): 1-2) FELIX_NAND(in1,in2,f1) 3-4) MAGIC_NOT(f1,out)	**IMPLY AND**(in1,in2,f1,out): 1-3) IMPLY_NAND(in1,in2,f1) 4-5) IMPLY_NOT(f1,out)			**RMAJ AND**(in1,in2,f1,out): 1) Write0(out) 2-3,r1) RMAJ_NOT(in2,f1) r2-r3) Read(in1,f1) 4) RMAJ(in1,f1,out)
IMP	**MAGIC IMPLY**(in1,in2,f1,out): 1-2) MAGIC_NOT(in1,out) 3-6) MAGIC_OR(out,in2,f1,out)	**FELIX IMPLY**(in1,in2,f1,out): 1-2) MAGIC_NOT(in1,f1) 3-4) FELIX_OR(f1,in2,out)	**IMPLY IMPLY**(in1,in2out): 1) IMPLY(in1,in2out)	**IMPLY IMPLY ND**(in1,in2,f1,out): 1-4) IMPLY_COPY(in2,f1,out) 5) IMPLY(in1,out)		**RMAJ IMPLY**(in1,in2,out): 1) Write1(out) r1-r2) Read(in1,in2) 2) RMAJ(in2,in1,out)
NIMP	**MAGIC NIMPLY**(in1,in2,f1,f2,out): 1-2) MAGIC_NOT(in1,f1) 3-4) MAGIC_NOR(f1,in2,out)		**IMPLY NIMPLY**(in1,in2,out): 1) IMPLY(in1,in2) 2-3) IMPLY_NOT(in2,out)	**IMPLY NIMPLY ND**(in1,in2,f1,out): 1-5) IMPLY_ND(in1,in2,out,f1) 6-7) IMPLY_NOT(f1,out)		**RMAJ NIMPLY**(in1,in2,f1,out): 1-2,r2) RMAJ_IMPLY(in1,in2,f1) 3-4,r3) RMAJ_NOT(f1,out)
XOR	**MAGIC XOR**(in1,in2,f1,f2,out): 1-2) MAGIC_NOT(in1,f1) 3-4) MAGIC_NOT(in2,out) 5-6) MAGIC_NOR(f1,out,f2) 7-8) MAGIC_NOR(in1,in2,f1) 9-10) MAGIC_NOR(f1,f2,out)	**FELIX XOR**(in1,in2,out): 1-2) FELIX_OR(in1,in2,out) 3) FELIX_NAND(in1,in2,out)	**IMPLY XOR**(in1,in2,f1,f2,out): 1-4) IMPLY_COPY(in1,f1) 5-8) IMPLY_COPY(in2,f2) 9) IMPLY(in1,f2) 10) IMPLY(in2,f1) 11-12) IMPLY_NOT(f1,out) 13) IMPLY(f2,out)		**ORNOR3 XOR**(in1,in2,f1,f2,out): 1-2) IMPLY_NOT(in1,f1) 3-4) IMPLY_NOT(in2,out) 5-6) ORNOR3_NOR(f1,out,f2) 7-8) ORNOR3_NOR(in1,in2,f1) 9-10) ORNOR3_NOR(f1,f2,out)	**RMAJ XOR**(in1,in2,f1,out): 1-2,r1) RMAJ_COPY(in1,f1) r2) Read(in2) 3) RMAJ(0,in2,f1) 4-5,r2) RMAJ_COPY(in2,out) r3) Read(in1) 6) RMAJ(0,in1,out) r4) Read(f1) 7) RMAJ(f1,0,out)
XNOR	**MAGIC XNOR**(in1,in2,f1,f2,out): 1-10) MAGIC_XOR(in1,in2,out,f1,f2) 11-12) MAGIC_NOT(f2,out)	**FELIX XNOR**(in1,in2,f1,out): 1-3) FELIX_XOR(in1,in2,f1) 4-5) MAGIC_NOT(f1,out)	**IMPLY XNOR**(in1,in2,out): 1-13) IMPLY_XOR(in1,in2,out,f1,f2) 14-15) IMPLY_NOT(f2,out)		**ORNOR3 XNOR**(in1,in2,f1,f2,out): 1-10) ORNOR3_XOR(in1,in2,out,f1,f2) 11-12) IMPLY_NOT(f2,out)	**RMAJ XNOR**(in1,in2,f1,out): 1-7,r4) RMAJ_XOR(in1,in2,out,f1) 8-9,r5) RMAJ_NOT(f1,out)
Full Adder	**MAGIC FA**(in1,in2,cin,f1,f2,f3,f4,s,cout): 1-2) MAGIC_NOR(in1,in2,f1) 3-4) MAGIC_NOR(in2,cin,s) 5-6) MAGIC_NOR(f1,s,cout) 7-8) MAGIC_NOT(cout,f1) 9-10) MAGIC_NOR(cin,in1,s) 11-12) MAGIC_NOR(f1,s,cout) 13-14) MAGIC_NOR(in1,in2,f1) 15-16) MAGIC_NOT(f1,s) 17-18) MAGIC_NOR(s,cin,f1) 19-20) MAGIC_NOR(f1,cout,f2) 21-22) MAGIC_NOT(in1,f1) 23-24) MAGIC_NOT(in2,f3) 25-26) MAGIC_NOR(f1,f3,s) 27-28) MAGIC_NOT(s,f1) 29-30) MAGIC_NOT(cin,f3) 31-32) MAGIC_NOR(f1,f3,f4) 33-34) MAGIC_NOR(f2,f4,f1) 35-36) MAGIC_NOT(f1,s)	**FELIX FA**(in1,in2,cin,f1,s,cout): 1-3) FELIX_XOR(in1,in2,f1) 4-6) FELIX_XOR(f1,cin,s) 7-9) FELIX_COPY(cin,cout,f1) 10) FELIX_MIN(in2,in2,f1) 11-12) MAGIC_NOT(f1,cout)	**IMPLY FA SUM**(in1,in2,cin,f1,f2,out): 1-2) Write0(f2), Write1(out) 3-11) IMPLY_OR(in1,f1,f2), IMPLY_OR(in2,f1,f2), OR(cin,f1,f2) 12-16) IMPLY_AND(f2,out,f1,out) 17) Write0(f2) 18-22) IMPLY_OR(in1,f1,f2), IMPLY(in2,f2), IMPLY(cin,f2) 23-27) IMPLY_AND(f2,out,f1,out) 28) Write0(f2) 29-33) IMPLY(in1,f2), IMPLY(in2,f2), IMPLY_OR(cin,f1,f2) 34-38) IMPLY_AND(f2,out,f1,out) 39) Write0(f2) 40-44) IMPLY(in1,f2), IMPLY(cin,f2), IMPLY_OR(in2,f1,f2) 45-49) IMPLY_AND(f2,out,f1,out) **IMPLY FA COUT**(in1,in2,cin,f1,f2,cout): 1-2) Write0(f2), Write1(cout) 3-8) IMPLY_OR(in1,f1,f2), IMPLY_OR(in2,f1,f2) 9-13) IMPLY_AND(f2,cout,f1,cout) 14) Write0(f2) 15-20) IMPLY_OR(in1,f1,f2), IMPLY_OR(cin,f1,f2) 21-25) IMPLY_AND(f2,cout,f1,cout) 26) Write0(f2) 27-32) IMPLY_OR(in2,f1,f2), IMPLY_OR(cin,f1,f2) 33-37) IMPLY_AND(f2,cout,f1,cout)		**ORNOR3 FA**(in1,in2,f1,f2,f3,f4,f5,sum,cin,cout): 1-7) Write0(sum,cout,f1,f2,f3,f4,f5) 8) IMPLY(in1,f1) 9) IMPLY(in2,f2) 10) ORNOR3(cout,f1,f2) 11-12) Write0(f1,f2) 13) ORNOR3(sum,in1,in2) 14) ORNOR3(f1,sum,cout) 15) IMPLY(cin,f2) 16) ORNOR3(cout,f2,sum) 17) Write0(sum) 18) IMPLY(f2,f3) 19) IMPLY(f1,f4) 20) ORNOR3(f5,f3,f2) 21-22) Write0(f1,f3) 23) ORNOR3(f1,f2,f4) 24) ORNOR3(sum,f5,f1)	**RMAJ FA**(in1,in2,cin,f1 sum,cout): 1-2,r1) RMAJ_COPY(cin,cout) 3-4,r2) NOT(in2,f1) r3,r4) Read(in1,f1) 5)- RMAJ(in1,f1,in3out) 6-7,r5) RMAJ_COPY(cin,sum) r6,r7) Read(in1,in2) 8) RMAJ(in1,in2,sum) r8) Read(cin) 9) RMAJ(in2,cin,sum)

Table I – Mapping of all considered Boolean functions on LIM primitive operations

	Input			#memristors						#steps					
2 bit	0 0 1 1 0 1 0 1			MAGIC	FELIX	IMPLY	IMPLY destr.	ORNOR3	RMAJ3	MAGIC	FELIX	IMPLY	IMPLY destr.	ORNOR3	RMAJ3
Gate	**Output**	**#in**	**#out**												
TRUE (write 1)	1 1 1 1	0	1	1 + 0	1 + 0	1 + 0	1 + 0	1 + 0	1 + 0	1	1	1	1	1	1
FALSE (write 0)	0 0 0 0	0	1	1 + 0	1 + 0	1 + 0	1 + 0	1 + 0	1 + 0	1	1	1	1	1	1
in1 (COPY)	0 0 1 1	1	1	2 + 1	2 + 1	2 + 1	2 + 1	2 + 1	2 + 0	4	3	4	4	4	2 + 1
NOT *in1*	1 1 0 0	1	1	2 + 0	2 + 0	2 + 0	2 + 0	2 + 0	2 + 0	2	2	2	2	2	2 + 1
in1 NOR *in2*	1 0 0 0	2	1	3 + 0	3 + 0	3 + 1	3 + 0	3 + 0	3 + 1	2	2	9	5	2	6 + 4
in1 OR *in2*	0 1 1 1	2	1	3 + 1	3 + 0	3 + 1	2 + 1	3 + 1	3 + 1	4	2	7	3	4	4 + 3
in1 NAND *in2*	1 1 1 0	2	1	3 + 2	3 + 0	3 + 0	3 + 0	3 + 0	3 + 1	10	2	3	3	3	6 + 5
in1 AND *in2*	0 0 0 1	2	1	3 + 2	3 + 1	3 + 1	3 + 1	3 + 1	3 + 1	6	4	5	5	5	4 + 3
in2 IMP *in1*	1 0 1 1	2	1	3 + 1	3 + 1	3 + 1	2 + 0	3 + 1	3 + 0	6	4	5	1	1	2 + 2
in2 NIMP *in1*	0 1 0 0	2	1	3 + 1	3 + 1	3 + 1	3 + 0	3 + 1	3 + 1	4	4	7	3	3	4 + 3
in1 XOR *in2*	0 1 1 0	2	1	3 + 2	3 + 0	3 + 2	3 + 2	3 + 2	3 + 1	10	3	13	13	10	7 + 4
in1 EQUAL *in2* (XNOR)	1 0 0 1	2	1	3 + 2	3 + 1	3 + 2	3 + 2	3 + 2	3 + 1	12	5	15	15	12	9 + 5
	Input														
3 bit	0 0 0 0 1 1 1 1 0 0 1 1 0 0 1 1 0 1 0 1 0 1 0 1														
	Output														
FA (sum)	0 1 1 0 1 0 0 1	3	2	5 + 4	5 + 1	5 + 2	5 + 2	5+5	5 + 1	36	12	49+37	49+37	24	9 + 10
FA (c_out)	0 0 0 1 0 1 1 1														

Table II – number of memristors and number of operations per Boolean function

For clarity, a unique syntax is used for all cells of Table I:

```
LiM_Boolean (used memory cells)
i-j)  LiM_Boolean(used memory cells)
```

where the first line defines the name of the Boolean function implemented in a specific LiM, together with the used memory cells for inputs and outputs; the following lines describe the algorithm used to map that function on primitive operations, underlying the number of steps required for its execution (*i-j*). In the case of majority voter requiring additional registers, the algorithm contains extra read operations marked as r_i.

For each function, a number of memristors are used to store the inputs (in_i), the output (*out*), and intermediate results (f_i). The intermediate results are used to solve complex mapping algorithm where several operations are executed in sequence, and they are stored in so-called functional memristors. In case of destructive operations (i.e., *IMPLY*), the content of one of the input memristors is overwritten by the output (noted as $in_i out$ in the table).

To validate the solutions, we have developed a script which checks for the correctness of each Boolean function mapped on LiM primitive operations.

Table II summarizes, for each Boolean function:

its truth table (column 2);

number of inputs and outputs of the function (columns 3 and 4);

for each LiM solution: number of used memristors, in the form #(input and output) + #functional (columns from 5 to 10);

for each LiM solution: number of operations needed to perform the computation (columns from 11 to 16). For the *PLiM* implementation, the steps are indicated as the memory cycles + the reading operations.

IV. CONCLUSIONS

In this paper we have presented an extensive study of the most prominent LiM solutions and provided a comparison in terms of required memory resources (i.e., number of memristors) and number of operations to implement basic Boolean functions.

The obtained results show big discrepancies among LiM solutions in the number of steps to execute the operations. For instance, the XOR requires many more steps if implemented with *IMPLY* logic compared to *FELIX*. These results reflect the complexity of each operation but do not directly translate into an estimation of the actual execution time. This is due to the fact that, due to physical and electrical characteristics of the memristive devices, the timing of each operation can vary significantly.

REFERENCES

[1] M. Gokhale *et al.*, "Processing in memory: the Terasys massively parallel PIM array," *Computer*, vol. 28, no. 4, pp. 23–31, Apr. 1995.

[2] A. Sebastian *et al.*, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, pp. 1–16, Mar. 2020.

[3] H. A. D. Nguyen *et al.*, "A Classification of Memory-Centric Computing," *J. Emerg. Technol. Comput. Syst.*, vol. 16, no. 2, p. 13:1-13:26, Jan. 2020.

[4] P. L. Thangkhiew *et al.*, "Efficient implementation of adder circuits in memristive crossbar array," in *TENCON 2017 - 2017 IEEE Region 10 Conference*, Nov. 2017, pp. 207–212.

[5] S. Kvatinsky *et al.*, "MAGIC—Memristor-Aided Logic," *IEEE Trans. Circuits Syst. II*, vol. 61, no. 11, pp. 895–899, Nov. 2014.

[6] N. Peled *et al.*, "X-MAGIC: Enhancing PIM Using Input Overwriting Capabilities," in *2020 IFIP/IEEE 28th International Conference on Very Large Scale Integration (VLSI-SOC)*, Oct. 2020, pp. 64–69.

[7] S. Gupta *et al.*, "FELIX: fast and energy-efficient logic in memory," in *Proceedings of the International Conference on Computer-Aided Design*, San Diego California, Nov. 2018, pp. 1–7.

[8] E. Lehtonen and M. Laiho, "Stateful implication logic with memristors," in *2009 IEEE/ACM International Symposium on Nanoscale Architectures*, San Francisco, CA, USA, Jul. 2009, pp. 33–36.

[9] S. Kvatinsky *et al.*, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 10, pp. 2054–2066, Oct. 2014.

[10] A. Siemon *et al.*, "Stateful Three-Input Logic with Memristive Switches," *Sci Rep*, vol. 9, no. 1, pp. 1–13, Oct. 2019.

[11] P.-E. Gaillardon *et al.*, "The Programmable Logic-in-Memory (PLiM) Computer," in *Proceedings of the 2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2016, pp. 427–432.

[12] S. Angizi *et al.*, "MRIMA: An MRAM-based In-Memory Accelerator," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, pp. 1–1, 2019.