The *Programmable Logic-in-Memory* (PLiM) Computer

Pierre-Emmanuel Gaillardon*†, Luca Amarú†, Anne Siemon‡, Eike Linn‡, Rainer Waser‡,
Anupam Chattopadhyay§, Giovanni De Micheli†

*Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT, USA
†Integrated Systems Laboratory, EPFL, Lausanne, Switzerland
‡Institut für Werkstoffe der Elektrotechnik II (IWE II) & JARA-FIT, RWTH Aachen, Germany
§School of Computer Engineering, NTU, Singapore
pierre-emmanuel.gaillardon@utah.edu

***Abstract*—Realization of logic and storage operations in memristive circuits have opened up a promising research direction of in-memory computing. Elementary digital circuits, e.g., Boolean arithmetic circuits, can be economically realized within memristive circuits with a limited performance overhead as compared to the standard computation paradigms. This paper takes a major step along this direction by proposing a fully-programmable in-memory computing system. In particular, we address, for the first time, the question of controlling the in-memory computation, by proposing a lightweight unit managing the operations performed on a memristive array. Assembly-level programming abstraction is achieved by a natively-implemented majority and complement operator. This platform enables diverse sets of applications to be ported with little effort. As a case study, we present a standardized symmetric-key cipher for lightweight security applications. The detailed system design flow and simulation results with accurate device models are reported validating the approach.**

I. Introduction

Multitude of emerging *Non-Volatile Memories* (NVM) are receiving widespread research attention as candidates for high-density and low-cost storage. NVMs store information as an internal resistive state, which can be either a *Low Resistance State* (LRS) or a *High Resistance State* (HRS) [1]. Among the different types of NVMs, redox-based *Resistive RAM* (RRAM) is considered a leading candidate due to its high density, good scalability, low power and high performance. Industrial demonstrations have recently been presented [2], [3], showcasing the viability of fast, high-density memory arrays.

A different and arguably more tantalizing aspect of RRAMs is their ability to perform primitive Boolean logic operations. The possibility of in-memory computing significantly widens the scope of the commercial applications. To undertake a logic computation, RRAM-based switches are needed. *Bipolar Resistive Switches* (BRS) [4] are easy-to-fabricate devices, consisting of a switching layer sandwiched between two metal electrodes. BRS have SET and RESET voltages of opposite polarities. BRS can be used in ultra-dense passive crossbar arrays. However, such passive crossbar arrays suffer from the formation of parasitic currents [5], which create *sneak paths* and strongly limit the size of the array. This problem can be alleviated by either adding a bipolar rectifying element to the BRS or constructing a *Complementary Resistive Switch* (CRS), which connects two BRS anti-serially [6]. Both BRS and CRS can natively perform *logical implication* [7]–[9] and can be used to implement any Boolean operations. As we will show later in this paper, the logical capabilities of RRAM devices go beyond the logic implication with an expressive implementation of *majority-oriented logic*. Recognizing the potential usage of in-memory computing, a few experimental as well as simulation-based results on primitive computing blocks have been reported in recent literature [10]–[13]. However, the studies lack in two aspects. First, in-memory computing often requires the generation of complex signal sequences and the corresponding control logic overhead is not considered in the practicality and performance analysis. Second, very few studies have so far connected the immense potential of such platforms to a practical end-user application.

In this paper, we address both of these problems by proposing a fully-programmable in-memory computing system and by evaluating its potential to map a standardized security kernel. We envision the proposed architecture as a low-cost add-on to standalone memory arrays that enables the implementation of computing tasks directly within the memory. Our contributions are as following:

- We propose a novel computer architecture constructed with resistive memories, called *Programmable Logic-in-Memory* (PLiM) computer architecture. This new programmable platform allows us to compute information on large resistive memory arrays. The PLiM computer architecture consists of a lightweight controller that complements a standard memory array. The controller executes simple instructions from the memory array and operates the memristive elements in logic mode.
- We validate the behavior of the system by implementing PRESENT [14], a lightweight security ISO primitive on the PLiM computer architecture. Using electrical simulations, we show that a full-crypto operation can be implemented in-memory with an energy of 5.88 pJ per block encryption and a throughput of 120.7 kbps, achieving a performance level compatible with standard silicon implementations [14].

The presented architecture is particularly interesting in the context of *Internet of Things* (IoT) [15], where low-cost distributed memory systems capable of operating at low-power on their own content may be the basis for new computing paradigms. Though we are taking a specific example of

application and technology in this paper, it must be noted that our proposed architecture is open to adaptation for diverse application and technology scenarios.

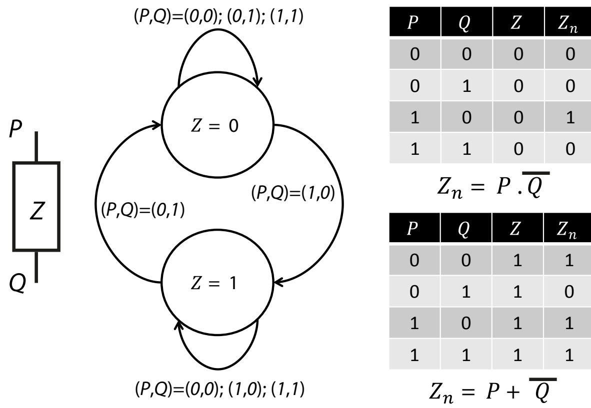
The remainder of this paper is organized as follows. In Section II, we discuss the realization of majority-based operations using RRAM and introduce the PLiM controller and its supported instruction. We also validate its working principle using electrical simulations fitted on experimental kinetic data. In Section III, we focus on the implementation of the PRESENT cipher on the presented architecture. We comment on the design methodology used to realize its individual elements and we evaluate the performance of a full cipher operation. In Section IV, we draw some conclusions.

II. Programmable Logic-In-Memory Extension for Memristive Arrays

In this section, we first focus on the realization of majority-based logic operations with a single memory cell. Then, we introduce a memory architecture that is able to support both standard memory behavior and computation within the memory elements. Finally, we validate the basic principle of the proposed system by showing memristive electrical simulation of a single instruction.

A. Intrinsic Majority Operations

Bipolar Resistive Switches and *Complementary Resistive Switches* are devices with two terminals, denoted P and Q. Their internal resistance state, Z, can be modified by applying a positive or a negative voltage V_{PQ}. The functionality of BRS/CRS can be summarized by a state machine, as shown in Fig. 1. Further details can be found in [8]. Transition occurs only for the conditions $P = 0, Q = 1$, i.e., $V_{PQ} < 0$ so $Z \to 0$ and $P = 1, Q = 0$, i.e., $V_{PQ} > 0$ so $Z \to 1$.



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. 1. Majority operations with BRS/CRS devices

By denoting Z as the value stored in the switch and Z_n the value stored after the application of signals on P and Q, it is possible to express Z_n as the following:

$$\begin{aligned} Z_n &= (P.\overline{Q}).\overline{Z} + (P + \overline{Q}).Z \\ &= P.Z + \overline{Q}.Z + P.\overline{Q}.\overline{Z} \\ &= P.Z + \overline{Q}.Z + P.\overline{Q}.Z + P.\overline{Q}.\overline{Z} \\ &= P.Z + \overline{Q}.Z + P.\overline{Q} \\ &= M_3(P, \overline{Q}, Z) \end{aligned}$$

where M_3 is the majority Boolean function with 3 inputs. A 3-input majority Boolean function is evaluated to be *true* if at least 2 of its inputs are *true*. This basic operation will be used in the rest of the paper as the elementary computing operation achievable by the switches. We refer to this operation as 3-input *Resistive Majority* (RM_3), such as: $RM_3(P, Q, Z) = M_3(P, \overline{Q}, Z)$.

B. PLiM Computer Architecture

Resistive memory elements can implement 3-input majority operations. This gives the opportunity to perform basic computations directly within a standalone memory array. Considering the large amount of memory cells available in standalone memories, it is possible to exploit massive parallelism by performing concurrent operations on a large set of memories. However, such in-memory computing requires complex control as many concurrent data has to be read/write simultaneously on the memory array. Conversely, it is possible to implement in-memory computation with a lightweight control by considering only serial operations on the memory array. Operating only a unique operation at a given time drastically simplifies the control schemes and ensures limited additional implementation costs. We choose this second approach in the paper.

We introduce in Fig. 2 a novel memory array architecture that can preform both standard memory and logic-in-memory operations. The memory is based on traditional multi-bank memory architecture. In addition to the standard RAM circuitries, we add a *Programmable Logic-in-Memory* (PLiM) controller block. This block controls the different operations on the array between either a standard memory mode or a computation mode. The controller consists in a simple control *Finite State Machine* (FSM) and few registers.

C. Operations of Programmable Logic-in-Memory Controller

The PLiM controller works as a simple processor core, reading instructions from the memory array and performing computing operations (majority) within the memory array. The control FSM diagram is given in Fig. 3. When LiM=0, the controller is *off* and the whole array works as a standard RAM system. When LiM=1, the circuit starts performing computation. The FSM initializes all the work registers and starts fetching an instruction. Resistive memories implement natively majority functions. Therefore, we focus on a unique instruction performing a majority operation on a single bit. The instruction format consists of the address of the first operand, the address of the second operand and the destination address of the results. Its format is @A, @B, @Z. Single-bit operands A and B are then read from the memory array, and logic operation is performed during the write operation to the memory location Z. To do so, we apply to the top electrode (P) the signal A and to the bottom electrode (Q) the signal B. The new value stored in the node Z is then $Z_n = M_3(A, \overline{B}, Z)$. When the write operation is completed, the program counter is incremented, and a new cycle of operation is triggered. Note that the PLiM controller uses the same addressing and read/write peripheral circuitries than the standard RAM mode. Therefore, the complexity of the controller is limited to the

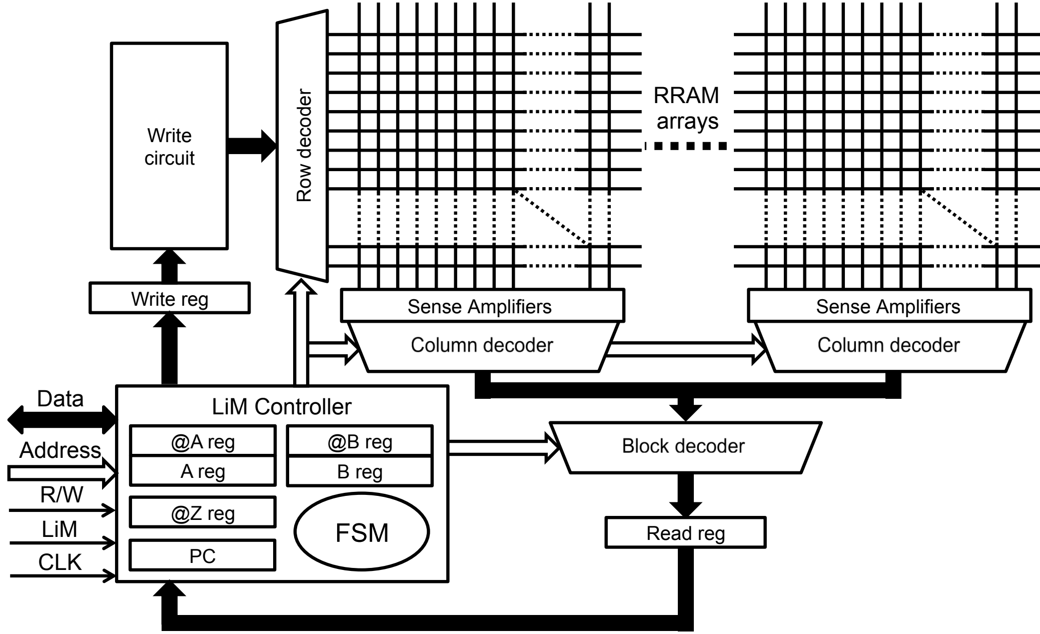


Fig. 2. Multi-bank Resistive Memory Architecture with Programmable Logic-in-Memory (PLiM) Controller module.

control FSM and few registers. With this architecture, the program and data are loaded in the memory array and a bit-level addressing is required. Program code generation is discussed in the following sections.

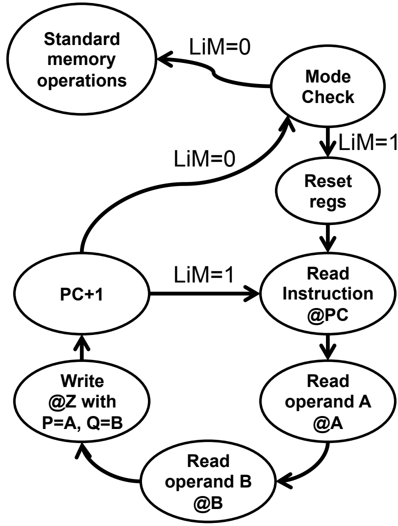


Fig. 3. Programmable Logic-in-Memory Controller FSM. All transitions are synchronous. Instruction format @A, @B, @Z.

D. Elementary Logic Operations on the Memristive Array

Boolean AND and OR operations can be emulated using the *Resistive Majority* operator. A majority operator reduces to AND/OR logic when one operand is set to constant 0 and 1, respectively. We present the following exemplary machine code for the PLiM controller module in order to perform C = A.B and C = A+B. The operations are directly performed on the storage of C. C is pre-programmed to either 0 or 1 depending on the Boolean operation. Note that direct addressing is used for constants for which the @ sign is not used.

AND

1: 0, 1, @C; //C=0 **2**: 0, 1, @B_{inv}; //B_{inv}=0
3: 1, @B, @B_{inv}; //$B_{inv}=\overline{B}$ **4**: @A, @B_{inv}, @C; //C=A.B

OR

1: 1, 0, @C; //C=1 **2**: 0, 1, @B_{inv}; //B_{inv}=0
3: 1, @B, @B_{inv}; //$B_{inv}=\overline{B}$ **4**: @A, @B_{inv}, @C; //C=A+B

A bit-level addressing is required for operations that manipulate and rearrange bits within a word. For example, the following machine code implements a 1-bit left rotate on a 4-bit array **Z** = $Z_3Z_2Z_1Z_0$, with X and Y as auxiliary locations:

1-BIT LEFT ROTATE

01: 0, 1, @X; **02**: 1, @Z_3, @X; // X = $\overline{Z_3}$
03: 0, 1, @Y; **04**: 1, @Z_2, @Y; // Y = $\overline{Z_2}$
05: @Z_2, @Y, @Z_3; // **Z** = $Z_2Z_2Z_1Z_0$
06: 0, 1, @Y; **07**: 1, @Z_1, @Y; // Y = $\overline{Z_1}$
08: @Z_1, @Y, @Z_2; // **Z** = $Z_2Z_1Z_1Z_0$
09: 0, 1, @Y; **10**: 1, @Z_0, @Y; // Y = $\overline{Z_0}$
11: @Z_0, @Y, @Z_1; // **Z** = $Z_2Z_1Z_0Z_0$
12: 0, 1, @Y; **13**: 1, @X, @Y; // Y = $\overline{X}$ = Z_3
14: @Y, @X, @Z_0; // **Z** = $Z_2Z_1Z_0Z_3$

E. RM_3 Instruction Simulation

We validate the presented RM_3 by running electrical simulations on a resistive memory array.

We consider a simple 4×4 bits memory array built using a dynamic *Valence Change Mechanism* (VCM) model fitted on experimental kinetics data and a bipolar rectifying selector. Full details about the compact model are available in [17]. Each memory cell implements the basic RM_3 operator. Here, the input Z corresponds to the resistive state, while P and Q

are the input voltages applied to either bottom or top electrode to perform the operation. The memory array is programmed using a $V/2$ scheme. More precisely, a logic 1 is realized by a voltage pulse of 2.3 V and a logic 0 is encoded by a voltage pulse of -2.3 V. Therefore, the height of the programming pulses is 4.6 V. Unselected lines are kept to ground. In a readout phase, the presence of a 5 μA current is considered as output `1`, while its absence is considered as `0`.

The first three words of the memory contain a RM_3 instruction that will operate on the data stored on the 4th word. The memory array is initialized as follow, and our example consists in applying $RM_3(A, B, Z)$ with A=bit 0 of word 0x03=1, B=bit 3 of word 0x03=0 and Z=bit 1 of word 0x03=0.

Initial state		Final state
0x00: 1 1 0 0	Instruction memory	0x00: 1 1 0 0
0x01: 1 1 1 1		0x01: 1 1 1 1
0x02: 1 1 0 1		0x02: 1 1 0 1
0x03: 0 1 0 1	Data memory	0x03: 0 1 1 1

The instruction is operated in 3 steps, as illustrated in Fig. 4.
Step 1: The $RM_3(A, B, Z)$ instruction is dumped from memory by reading the operand addresses: $@A$=1100, $@B$=1111, $@Z$=1101. These addresses follow a bit-level addressing as follows: $@A$=[Address of the word containing A (2bits), Position of the bit A within the word].
Step 2: The operands A and B are read. As $@A$=1100, A is read from the LSB of the word 0x03 (A = 1). Similarly, $@B$= 1111, i.e., MSB of word 0x03, and B = 0.
Step 3: The final destination Z is written by fixing A = 1 on wl_3 and B = 0 on bl_3, leading to a write operation. The final value of Z is then verified.

We therefore operated on Z the operation $RM_3(1, 0, 0) = M_3(1, 1, 0)$=1.

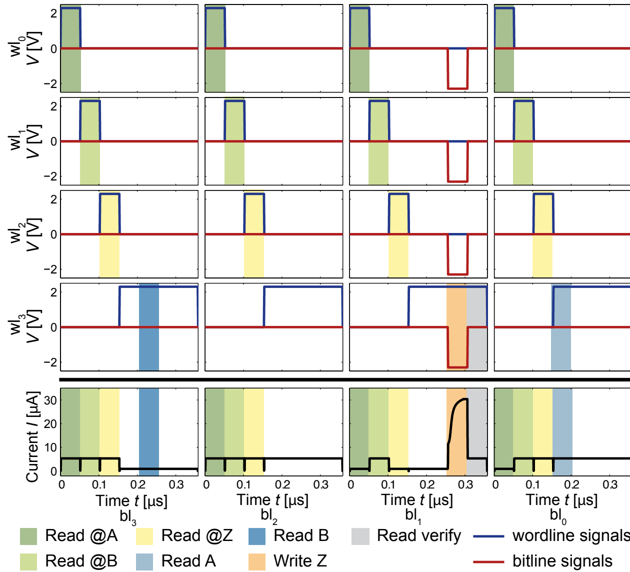


Fig. 4. Simulation of *3-input Resistive-Majority* (RM_3) operation with A_0=1, B_0=0, and Z_0=0. Each operation is highlighted by a colored background.

III. Case Study: Implementing PRESENT on PLiM Array

In this section, we discuss the implementation of the PRESENT cipher on the proposed architecture. After introducing the necessary background about PRESENT, we comment on the realization of the individual components along with the PLiM code snippet. We finally evaluate the implementation of a full cipher operation in terms of cycles and energy.

A. Background: PRESENT Block Cipher

As an ISO-standardized block cipher for lightweight cryptography, PRESENT [14] is considered as a meaningful application for the proposed architecture. The cipher encrypts a 64-bit plaintext with an 80 or 128-bit key provided by the user. We restrict the following discussion to the 80-bit key and only to the encryption module. A PRESENT encryption consists of 31 rounds, through which multiple operations are performed on the 64-bit plaintext and finally produces a 64-bit ciphertext. The rounds modify the plaintext, which is referred as *STATE* internally. The operation of the cipher components is briefly reviewed in the following. Readers may kindly refer to [14] for further details.

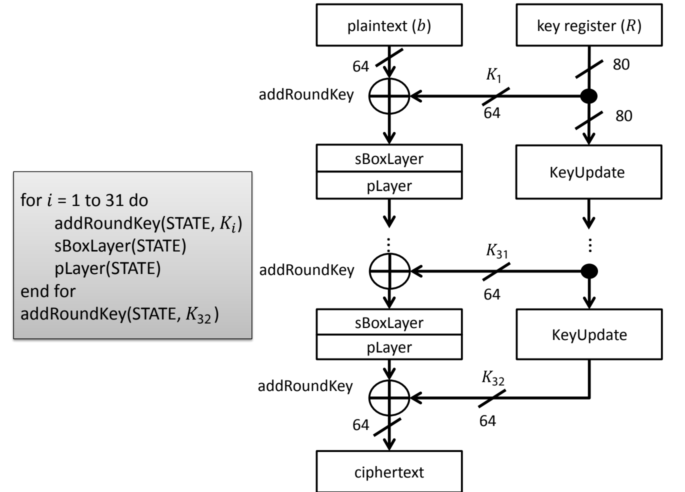


Fig. 5. PRESENT Block Cipher: Encryption

The algorithmic flow for PRESENT is shown in the Fig. 5. PRESENT consists of three different sub-blocks operating on the text to cipher: *addRoundKey*, *sBoxLayer*, *pLayer*, and one sub-block operating on the key: *KeyUpdate*. In the following, we discuss the PLiM implementation of the different blocks.

B. Operations on STATE

For each round i ($1 \leq i \leq 32$) of the cipher, the current 64-bit $STATE$ ($b_{63} \cdots b_0$) is processed in 3 steps:

1) addRoundKey: First, the state is processed by the round-specific key $K_i = k_{63}^i \cdots k_0^i$. The round-specific key K_i is extracted from the user-specified 80-bit key R, by simply selecting the 64 leftmost bits. The *addRoundKey* operation is defined by the bit-wise operation: $b_j = b_j \oplus k_j^i$. Therefore, it involves a 64-bit XOR operation between the STATE and the

leftmost bits of the key. We implement it by 64 1-bit XOR instructions between the bit-addressed operands.

1-BIT XOR

1: 0, 1, @Z; // Z=0	**2**: @A, 0, @Z; // Z=A
3: 0, @B, @Z; // Z=A.$\overline{B}$	**4**: 0, 1, @C; // C=0
5: @B, 0, @C; // C=B	**6**: 0, @A, @C; // C=B.$\overline{A}$
7: @Z, 0, @C; // C=A⊕B	

2) sBoxLayer: Then, the state is processed by the 4-bit to 4-bit S function. The S function operates $\mathbb{F}_2^4 \rightarrow \mathbb{F}_2^4$ and is defined in [14]. the input and output values of the S-Box function respectively. To be processed, the 64-bit state is divided in sixteen 4-bit words that are individually processed by the S operator.

The S operator can be interpreted as a 4-input, 4-output combinational Boolean function. The combinational network is mapped to the proposed programmable memristive array system using a two-step approach. Note that the proposed approach is general enough for implementing any arbitrary Boolean logic on the platform. The implementation is done via the two following steps:

1) The Boolean function is converted and optimized into an internal data-structure in form of majority-inverter dataflow network.
2) The internal data-structure is mapped to the RM_3 instructions.

The first step exploits a recently introduced majority-based logic optimization package [16], while the second step corresponds to a one-to-one mapping of the nodes onto the supported instruction.

To illustrate the proposed approach, we provide an example for a part of the S operator. More precisely, assuming the primary inputs to the S-box being pi_0, pi_1, pi_2, pi_3 and the primary outputs being po_0, po_1, po_2, po_3, we focus on the generation of the output po_0.

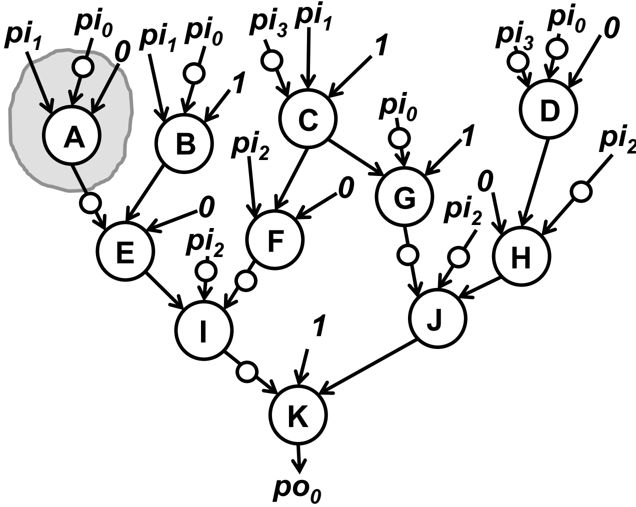


Fig. 6. Portion of the S operator majority-inverter data flow graph corresponding to po_0 generations.

The majority-inverter dataflow network corresponding to po_0 is depicted in Fig. 6. The generation of po_0 requires 11 majority operations. Each majority operation is mapped to a set of RM_3 instructions. For instance, the portion highlighted in grey on the network corresponds to the operation $M_3(pi_1, \overline{pi_0}, 0)$. This is translated into 2 RM_3 instructions that respectively store an initial value 0 and perform the expected majority operation on a given memory position.

The procedure is iterated for all the nodes in the dataflow network. The corresponding code snippet for the generation of po_0 is shown below:

S **operator -** po_0

01: 0, 1, @X1;	**02**: @pi_1, @pi_0, @X1; // X1=N_A
03: 1, 0, @X2;	**04**: @pi_1, @pi_0, @X2; // X2=N_B
05: 1, 0, @X3;	**06**: @pi_1, @pi_3, @X3; // X3=N_C
07: 0, 1, @X4;	**08**: 1, @pi_0, @X4; // X4 = $\overline{pi_0}$
09: 0, 1, @X5;	**10**: @X4, @pi_3, @X5; // X5=N_D
11: 0, @X1, @X2;	// X2=N_E
12: 1, 0, @X1;	**13**: @pi_1, @pi_3, @X1; // X1=N_C
14: 0, 1, @X4;	**15**: 1, @pi_2, @X4; // X4=$\overline{pi_2}$
16: 0, @X4, @X3;	// X3=N_F
17: 1, @pi_0, @X1;	// X1=N_G
18: 0, @pi_2, @X5;	// X5=N_H
19: @X4, @X3, @X2;	// X2=N_I
20: @X4, @X1, @X5;	// X5=N_J
21: 1, @X2, @X5;	// X5=N_K = po_0

The total S operator requires a total of 38 cycles for its operation.

3) pLayer: Finally, the 64-bit state is processed by the permutation function P. The permutation P is described in [14].

The permutation P moves every individual bits to a new unique bit positions. This requires bitwise manipulation easily implementable with the bit-level granularity of the instruction.

C. Operations on KEY

For each round i, the key register is updated using the *KeyUpdate* procedure. The key update procedure is decomposed on three elementary operations.

1) Left Rotate Operation: First, the key register is rotated by 61 bit positions to the left: $[R_{79}R_{78}\cdots R_1R_0] = [R_{18}R_{17}\cdots R_{20}R_{19}]$.

The 61-bit rotation operation on the left is implemented using a similar procedure than the 1-bit left rotate introduced earlier.

2) S-box Operation: Then, the S function is applied to the bits 76 to 79: $[R_{79}R_{78}R_{77}R_{76}] = S[R_{79}R_{78}R_{77}R_{76}]$.

The S operator is applied on the bits 76 to 79 using a similar procedure than for the cipher text discussed above.

3) XOR Operation: : Finally, a bit-wise XOR operation is performed between the bits 19 to 15 and the round counter i: $[R_{19}R_{18}R_{17}R_{16}R_{15}] = [R_{19}R_{18}R_{17}R_{16}R_{15}] \oplus i$.

Between every round, the round counter i is incremented. The incrementer, which is invoked at every round, is implemented using a simple toggle-cell adder procedure suited for RRAM computation [17]. The following code snippet illustrates the procedure for a 2-bit word length:

2-BIT ADDER

01: 0, 1, @C; // C=C_{in}=0
02: 0, 1, @X; **03**: 1, @B_0, @X; // X=$\overline{B_0}$
04: @A_0, @X; @C; // C = C_1
05: 0, 1, @S_0; **06**: @A_0, @B_0, @S_0; // S_0=S'$_0$
07: @B_0, @C, @S_0; // S_0 = S_0
08: 0, 1, @X; **09**: 1, @B_1, @X; // X=$\overline{B_1}$
10: @A_1, @X; @C; // C = C_2
11: 0, 1, @X; **12**: 1, @C_1, @X; // X=$\overline{C_1}$
13: @C, @X; @S_1; // S_1 = C_1
14: @A_1, @B_1, @S_1; // S_1=S'$_1$
15: @B_1, @C, @S_1; // S_1 = S_1

The XOR operations between the 5-bit counter value i and the bits 19 to 15 of the key R are performed individually using the bit-level XOR procedure discussed earlier.

D. Performance of PRESENT Cipher on PLiM Computer

We estimate the performance of the PRESENT implementation on the memristive array. We consider a 8 Gb memory array arranged in 16-bit words. Therefore, the whole memory can be accessed by 32-bit addresses.

We also make the assumption of a mature RRAM technology [18]. More precisely, we consider a write time of 1 ns and a write energy of 0.1 fJ/bit. Table I summarizes the number of RM_3 instructions and *Read/Write* (R/W) cycles required by the different operations of the PRESENT cipher.

TABLE I
PRESENT IMPLEMENTATION PERFORMANCES

Operation	Instructions (#RM_3)	Cycles (#R/W)	Energy (pJ)	Throughput (kbps)
Key copy	80	720	-	-
Cipher copy	64	576	-	-
AddRoundKey	448	4032	-	-
sBoxLayer	608	5472	-	-
pLayer	64	576	-	-
KeyUpdate	760	6840	-	-
PRESENT Block	58 872	455 184	5.88	120.7

Each RM_3 instructions operates the following micro-operations on the memory array: Read @A (32 bits), Read @B (32 bits), Read @Z (32 bits), Read A (1 bit), Read B (1 bit), Write @Z (1 bit). This corresponds to 9 R/W cycles on the considered machine, i.e., with 16-bit words. The total number of RM_3 instructions for the encryption of a 64-bit cipher text is 58872. The large amount of operations comes from the fact that we merge both logic and memory operations and, therefore, many memory accesses are required. Nevertheless, considering that R/W access on the memory array are fast, the total throughput reachable by the system is 120.7 kbps, making it comparable to silicon implementations [14]. It is worth pointing out that the large number of memory accesses to the memory do not create a "bottleneck", as all the operations stay within the memory system. Finally, the total energy required for one block encryption operation is 5.88 pJ. The presented results are particularly promising in the context of IoT, where a low-cost non-volatile memories could be used to perform low-performance low-power consumption on their own content.

IV. CONCLUSION

In this paper, we introduced a fully-programmable in-memory computing system, which exploits memristive devices to perform both the storage and computing operations. For the first time, this paper addresses the question of the computation control unit, by proposing a novel *Programmable Logic-in-Memory* (PLiM) computer architecture. The PLiM computer architecture consists of a lightweight controller that complements a standard memory array. Assembly-level programming abstraction is achieved with a natively implemented majority and complement operator. This platform enables diverse sets of applications to be ported with little effort and adds computation capabilities to a memory array at a very limited overhead. The validity and performance of the proposed approach have been evaluated with accurate device models by considering the implementation of a standardized cryptographic system on the array. A full-crypto operation can be implemented in-memory with an energy of 5.88 pJ per block encryption and a throughput of 120.7 kbps.

ACKNOWLEDGEMENT

This work has been supported by the European Research Council grant ERC-2009-AdG-246810, the Swiss National Science Foundation project number 200021_146600 and the German Research Foundation (DFG) grant LI 2416/1-1.

REFERENCES

[1] G. W. Burr *et al.*, "Overview of candidate device technologies for storage-class-memory," *IBM J. R&D*, 52(4/5), 2008.
[2] R. Fackenthal *et al.*, "A 16Gb ReRAM with 200MB/s Write and 1GB/s Read in 27nm Technology," *ISSCC Tech. Dig.*, 2014.
[3] S.-S. Sheu *et al.*, "A 4Mb Embedded SLC Resistive-RAM Macro with 7.2ns Read-Write Random-Access Time and 160ns MLC-Access Capability," *ISSCC Tech. Dig.*, 2011.
[4] H.-S. P. Wong *et al.*, "Metal-Oxide RRAM," *Proc. of the IEEE*, 100(6), 2012.
[5] C. Kügeler, R. Rosezin, E. Linn, R. Bruchhaus, R. Waser, "Materials, technologies, and circuit concepts for nanocrossbar-based bipolar RRAM," *Appl. Phys. A - Mater. Sci. Process.*, 102(4), 2011.
[6] E. Linn, R. Rosezin, C. Kügeler, R. Waser, "Complementary resistive switches for passive nanocrossbar memories," *Nature Materials*, 9, 2010.
[7] J. Borghetti *et al.*, "Memristive switches enable stateful logic operations via material implication," *Nature*, 464, 2010.
[8] E. Linn, R. Rosezin, S. Tappertzhofen, U. Böttger, R. Waser, "Beyond von Neumann–logic operations in passive crossbar arrays alongside memory operations," *Nanotechnology*, 23(305205), 2012.
[9] S. Kvatinsky, E. G. Friedman, A. Kolodny, U. C. Weiser, "Memristor-Based Material Implication (IMPLY) Logic: Design Principles and Methodologies," *IEEE TVLSI*, 22(10), 2014.
[10] J. J. Yang, D. B. Strukov, D. R. Stewart, "Memristive devices for computing," *Nat. Nanotechnol.*, 8, 2013.
[11] E. Lehtonen, M. Laiho, "Stateful Implication Logic with Memristors," *Nanoarch Tech. Dig.*, 2009.
[12] T. You *et al.*, "Exploiting Memristive $BiFeO_3$ Bilayer Structures for Compact Sequential Logics," *Adv. Func. Mat.*, 24(3357), 2014.
[13] T. Breuer *et al.*, "Low-current and high-endurance logic operations in $4F^2$-compatible TaO_x-based complementary resistive switches," *SNW Tech. Dig.*, 2014.
[14] A. Bogdanov *et al.*, "PRESENT: An Ultra-Lightweight Block Cipher," *CHES Tech. Dig.*, 2007.
[15] E. A Lee *et al.*, "The Swarm at the Edge of the Cloud," *IEEE Design & Test*, 31(3):8-20, 2014.
[16] L. Amarú, P.-E. Gaillardon, G. De Micheli, "Majority-Inverter Graph: A Novel Data-Structure and Algorithms for Efficient Logic Optimization," DAC Tech. Dig., 2014.
[17] A. Siemon, S. Menzel, A. Chattopadhyay, R. Waser, and E. Linn, "In-Memory Adder Functionality in 1S1R Arrays, *IEEE ISCAS*, 2015.
[18] Emerging Research Devices (ERD) report, International Technology Roadmap for Semiconductors (ITRS), 2013