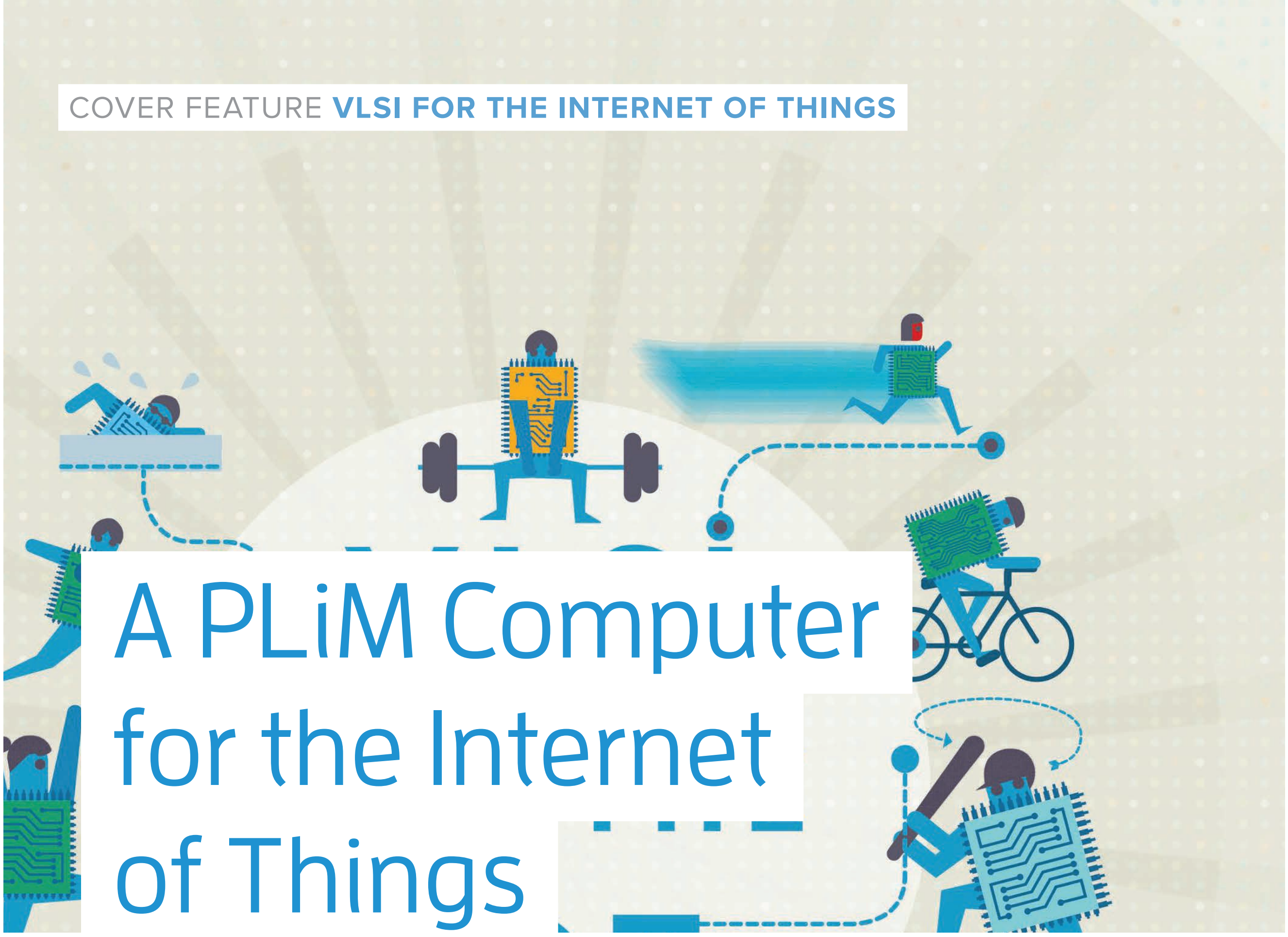


# A PLiM Computer for the Internet of Things

**Mathias Soeken,** EPFL

**Pierre-Emmanuel Gaillardon,** University of Utah

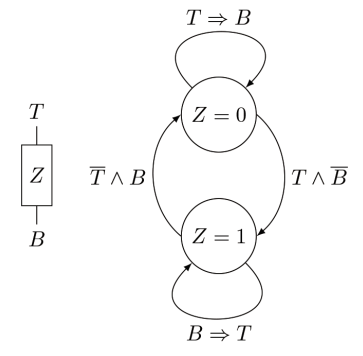
**Saeideh Shirinzadeh,** University of Bremen

**Rolf Drechsler,** University of Bremen and German Research Center for Artificial Intelligence

**Giovanni De Micheli,** EPFL

*Emerging applications are dramatically changing computer architecture requirements, with a shift toward big data that is processed using simple computations. A programmable logic-in-memory (PLiM) computer can allow memory cells to perform primitive logic operations and therefore compute without needing to communicate with a processing unit.*

The work of Hungarian-American mathematician John von Neumann uniquely influenced how we design computers. His "First Draft of a Report on the EDVAC," written in 1945 while von Neumann was commuting by train to Los Alamos, New Mexico, proposed a uniform memory that contains both data and instructions. Known today as the von Neumann architecture, this key concept has been continually improved over the past few decades, resulting in today's highly optimized and sophisticated memory hierarchies. The driving assumption behind this innovation has been that computation is complex and must be fast, and therefore memory needs to be readily available. Memory hierarchies allow fast access to small amounts of data and require longer times to access the larger memory located deeper in the hierarchy. Consequently, the fundamental assumption underlying today's computing architectures is only valid as long as computation



(a) (b)

$$Z \leftarrow TZ \vee \overline{B}Z \vee P\overline{B} = \langle T\overline{B}Z \rangle = \mathrm{RM}_3(T, B, Z)$$

(c)

| $T$ | $B$ | $Z$ | $Z'$ | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | $T \wedge \overline{B}$ |
| 0 | 1 | 0 | 0 | |
| 1 | 0 | 0 | 1 | |
| 1 | 1 | 0 | 0 | |
| 0 | 0 | 1 | 1 | $T \vee \overline{B}$ |
| 0 | 1 | 1 | 0 | |
| 1 | 0 | 1 | 1 | |
| 1 | 1 | 1 | 1 | |

(d)

**FIGURE 1.** Intrinsic majority operations: (a) a schematic of a resistive RAM (RRAM) cell with its internal state *Z* and electrodes *B* and *T*; (b) state machine illustrating how *Z* changes based on values for *B* and *T*; (c) transition relation for the state machine, resulting in the $\mathrm{RM}_3$ operation; and (d) truth table for the transition relation.

is dominant and not too much data is being processed.

Today, the requirements of emerging applications such as deep learning, data fusion, and the Internet of Things (IoT) are a challenge for the von Neumann architecture as the focus shifts to large amounts of data that are processed using comparably simple computations. At this point, improvements to the memory hierarchy cannot solve the problem, so a revolutionary change is necessary. In-memory computing is a promising candidate.[1,2] With this approach, memory cells can perform primitive logic operations and can therefore compute without needing to communicate with a processing unit. In addition, independent memory cells can perform their computations in parallel.

In this article, we propose a programmable logic-in-memory (PLiM) computer and demonstrate how it can help implement IoT applications. We show how to implement a Boolean majority operation with a single resistive RAM (RRAM) memory device (which can be used in industrial-scale applications),[3,4] build the PLiM computer from these devices, and program the PLiM computer. The underlying RRAM device switches its internal state based on its two terminals via inversion (complementation) and a majority-of-three operation. Consequently, for in-memory computing, this approach offers an assembly-level abstraction in terms of a natively implemented majority and complement operator. Therefore, we can use innovations in majority-based logic synthesis[5–9] to program the PLiM computer. Finally, because programs are data that are executed directly in memory, we can link applications by providing parts of the program's instructions from distributed devices. This innovative and new programming paradigm ideally matches the capabilities of in-memory computing in the IoT context.

## INTRINSIC MAJORITY OPERATIONS

Among other types of emerging nonvolatile memories, RRAMs are considered a leading candidate to implement memory arrays with higher density, lower power, and higher performance.[10] In addition to their memory properties, RRAMs can perform primitive Boolean logic operations.

To begin, let's review the basic Boolean switching primitive offered by RRAMs (see Figure 1). A RRAM is a two-terminal device with an internal resistive state that can be programmed depending on the voltage difference between the top electrode *T* and the bottom electrode *B*. Transition occurs whenever *T* and *B* are assigned different voltage. If $T = 0$ and $B = 1$ (that is, $V_{TB} < V_{\mathrm{prog}}$), the resulting low-resistance state is $Z = 0$. If $T = 1$ and $B = 0$ (that is, $V_{TB} > V_{\mathrm{prog}}$), then $Z = 1$. Here, $V_{\mathrm{prog}}$ is the memory technology's programming voltage, which for simplicity we assume is symmetric. The truth table in Figure 1d summarizes this behavior.

By denoting *Z* as the current resistance value and *Z′* as the resistance value after assigning signals to *T* and *B*, it is possible to express *Z′* as

$$Z' = \overline{Z}\left(T \wedge \overline{B}\right) \vee Z\left(T \wedge \overline{B}\right) = \left\langle T\overline{B}Z \right\rangle, \quad (1)$$

where $\langle xyz \rangle = xy \vee xz \vee yz$ is the Boolean three-input majority function that evaluates to true, if and only if at least two of its inputs are true. In the special case of Equation 1, one operand is negated, and for convenience, we define $\mathrm{RM}_3(T, B, Z) = \left\langle T\overline{B}Z \right\rangle$ for a three-input resistive majority. $\mathrm{RM}_3$ is universal and will be used as the PLiM's elementary computing operation.

## PLiM COMPUTER

The general philosophy underpinning the PLiM architecture addresses how to add computing capabilities (through bit-level $\mathrm{RM}_3$ instructions) to a regular dense memory array. Extra hardware is necessary to obtain a computer's abstraction without losing the standard memory functionality.

Figure 2 shows the PLiM computer architecture, which consists of a standard memory array with signals that are wrapped with the PLiM controller. This controller is a lightweight synchronous block that controls the memory array's access bus to allow a computation mode to run. The computation mode runs a sequential execution of a given set of instructions that represent a program. The program is stored on the memory array, and its output updates the memory array itself. The logic-in-memory (LiM) input controls the transition between the computation and memory modes.

PLiM revolves around one single instruction: $RM_3(A, B, C)$. The instruction takes three operands ($A$, $B$, and $Z$), applies the $RM_3$ majority operation with $A$ as the top electrode and $B$ as the bottom electrode, and updates the value of $Z$ accordingly. The single-instruction scheme simplifies the architecture as it is directly associated with the memory's intrinsic logic operation.

The architecture's source, destination, and processing unit is the memory block itself. Performing the instruction simply means loading the bit-level values of $A$ and $B$ from memory and applying them to $Z$. Also, the instruction itself is stored on the same memory block. Hence, to execute an instruction, the instruction is first loaded from memory, the operands are then loaded from memory, and the operands are finally applied to the destination. (Additional details about the $RM_3$ instruction encoding are available in earlier work.[11])

## PLiM COMPILER

We can now show how to compile arbitrary Boolean functions into $RM_3$ instruction streams. (These techniques were initially presented in earlier work.[12]) For the sake of convenience, we introduce the following commands, which are shorthand for several useful $RM_3$ instructions. Given registers $a$, $b$, and $z$, we define

- ZERO($z$): $z \leftarrow RM_3(0, 1, z) = \langle 00z \rangle = 0$
- ONE($z$): $z \leftarrow RM_3(1, 0, z) = \langle 11z \rangle = 1$
- BUF($a$, $z$): ZERO($z$); $z \leftarrow RM_3(a, 0, z) = \langle a10 \rangle = a$
- NOT($a$, $z$): ZERO($z$); $z \leftarrow RM_3(1, a, z) = \langle 1\bar{a}0 \rangle = \bar{a}$
- RM($a$, $b$, $z$): $z \leftarrow RM_3(a, b, z)$

In these commands, $z$ is the modified register. Also, note that the commands ZERO, ONE, and RM require exactly one $RM_3$ instruction, whereas the other two require two instructions.

Next, we show how to use majority-inverter graphs (MIGs)[9] to translate Boolean functions into a sequence of $RM_3$ instructions that compute the functions. The left side of Figure 3a illustrates the idea using a small MIG. It consists of two nodes and four primary inputs $x_1$, $x_2$, $x_3$, and $x_4$. We want to compute the function of the single primary output $y$. We consider primary inputs to be environment variables that cannot be modified by PLiM instructions. Because none of the primary inputs can be overridden, we need a free RRAM to which we can write the result in order to compute the output of node 1. Also, we need to get rid of one of the inverters, because the $RM_3$ operation expects exactly one input to be inverted. The two commands—NOT($x_3$, $z_1$); RM($x_1$, $x_2$, $z_1$)—compute the value of node 1 and require three $RM_3$ instructions and one RRAM cell $z_1$, which stores the node's output. For the remainder of this article, we will use $z$ variables to refer to RRAM cells.

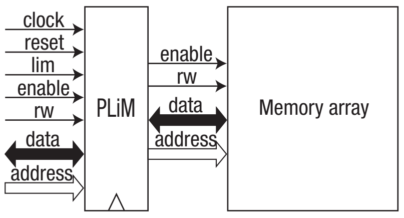


FIGURE 2. Programmable logic-in-memory (PLiM) computer. The architecture consists of a standard memory array and a lightweight controller for data access and controlling whether the memory behaves as a default memory or performs in-memory computations.

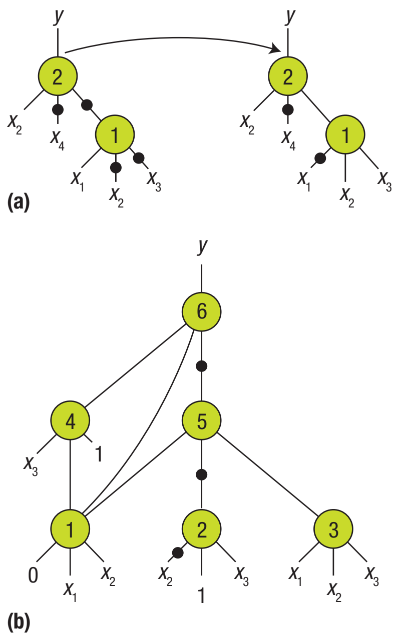


FIGURE 3. Using majority-inverter graphs (MIGs) to translate Boolean functions into a sequence of $RM_3$ instructions. (a) Rewriting an MIG involves using inverter propagation, which can lead to better starting points for PLiM program compilation. (b) The order in which nodes are processed in the MIG affects the PLiM program's quality.

NOT($x_1$, $z_1$)
BUF($x_2$, $z_2$)
1 ▻ RM(0, $z_1$, $z_2$)
BUF($x_3$, $z_3$)
2 ▻ RM(1, $x_2$, $z_3$)
NOT($x_2$, $z_4$)
BUF($x_3$, $z_5$)
3 ► RM($x_1$, $z_4$, $z_5$)
NOT($x_3$, $z_6$)
ONE($z_7$)
4 ▻ RM($z_2$, $z_6$, $z_7$)
5 ▻ RM($z_2$, $z_3$, $z_5$)
6 ▻ RM($z_7$, $z_5$, $z_2$)

**(a)**

BUF($x_2$, $z_1$)
1 ▻ RM($x_1$, 1, $z_1$)
ONE($z_2$)
2 ▻ RM($x_3$, $x_2$, $z_2$)
NOT($x_2$, $z_3$)
BUF($x_3$, $z_4$)
3 ▻ RM($x_1$, $z_3$, $z_4$)
5 ▻ RM($z_1$, $z_2$, $z_4$)
BUF($x_3$, $z_2$)
4 ▻ RM($z_1$, 0, $z_2$)
6 ▻ RM($z_1$, $z_4$, $z_2$)

**(b)**

**FIGURE 4.** Two PLiM programs constructed from the MIG in Figure 3b: (a) 19 $RM_3$ instructions and seven RRAM cells and (b) 15 $RM_3$ instructions and four RRAM cells. The difference in the programs is due to the choice of node-traversal order and the mapping of the nodes' children to $RM_3$ operands. The $i$ ▻ indicates the computation of node $i$.

In the next step, we can compute node 2 with a similar sequence: NOT($z_1$, $z_2$); RM($x_2$, $x_4$, $z_2$). Again, this requires three $RM_3$ instructions and one additional RRAM cell $z_2$.

In total, the PLiM program requires six instructions and two RRAMs. Several MIGs exist that realize the same function, and we can obtain one from the other by applying rewriting rules.[9] We can illustrate the effect by applying the inverter propagation

$$\overline{\langle x_1, \bar{x}_2, \bar{x}_3 \rangle} = \langle \bar{x}_1, x_2, x_3 \rangle \qquad (2)$$

to node 1.

The right side of Figure 3a illustrates the resulting MIG. This MIG can be translated into the PLiM program BUF($x_3$, $z_1$); RM($x_2$, $x_1$, $z_1$); RM($x_2$, $x_4$, $z_1$). This program only requires four $RM_3$ instructions and one RRAM cell. In addition to inverter propagation, the other rewriting rules from the axiomatic set of MIG manipulation rules can lead to MIGs for which we can find better PLiM programs. Furthermore, the MIG rewriting algorithms to optimize for PLiM compilation differ from rewriting algorithms that target area or delay optimization in conventional logic synthesis.

Even without rewriting the MIG, we can obtain different PLiM programs by simply changing the order in which the nodes are traversed and by changing the children of each node to be used as the operand in the $RM_3$ instruction. Nodes can be traversed as long as they follow a topological order from the primary inputs to the primary outputs. Operands can be selected arbitrarily because the majority operation is fully symmetric. However, the choice of a traversal order and operand mapping can have a significant impact, so we are interested in finding a good one. Figure 3b illustrates this effect.

Based on the MIG in Figure 3b, several different PLiM programs can be constructed using the technique we just discussed. Figure 4 shows two example PLiM programs. For the longer program (Figure 4a), which consists of 19 $RM_3$ instructions and 7 RRAM cells, we used the traversal order 1, 2, 3, 4, 5, 6. The shorter program (Figure 4b) was created using the traversal order 1, 2, 3, 5, 4, 6. The latter program consists of 15 $RM_3$ instructions and only 4 RRAM cells. However, the traversal order is not the only cause of this improvement. When selecting which child to map to which operand in the $RM_3$ instruction, constants in particular allow some freedom. We can always invert a constant to match the polarity requirement of operand $B$ or the $RM_3$ instruction—for example, as we did for node 1. In the program in Figure 4a, we chose to invert input $x_1$, requiring a NOT command and one additional RRAM. Those can be avoided when using the constant 0 as an inverted constant 1, as we did in the Figure 4b program.

Figure 5 shows our experimental results for the PLiM compiler. We applied the PLiM compiler to MIGs for instances in the EPFL (École Polytechnique Fédéderale de Lausanne) benchmark suite (lsi.epfl.ch/benchmarks). Figure 5a gives the number of $RM_3$ instructions, and Figure 5b gives the number of instructions. The blue bars correspond to an approach in which we directly translated the MIGs to PLiM programs following a node-traversal order on node indexes and selecting operands from left to right. The red bars show the effect after MIG rewriting but still using the naive node-traversal order and operand selection. Finally, the brown bars show the effect after MIG rewriting and taking into consideration heuristics for better node traversal and operand selection. These results show that MIG rewriting strongly affects the number of instructions, but not necessarily the number of RRAMs. However, when taking the node-traversal heuristics into account, the number of RRAMs decreases, but there is little gain in the number of instructions.

The PLiM computer we describe here is a low-power platform that is capable of implementing the IoT applications of tomorrow. In-memory computing is a better fit for IoT applications than conventional von Neumann architectures because it can deal with large amounts of data using comparably simple computations. PLiM computers and programs allow a new paradigm of computing, where programs are sequences of $RM_3$ instructions that send the data from one PLiM computer to another. These PLiM programs can be partial and distributed, where each IoT device provides its part to the computation. This model allows a high degree of configurability.

As part of our ongoing research efforts, we are evaluating the physical design of a PLiM computer and more advanced programming models.

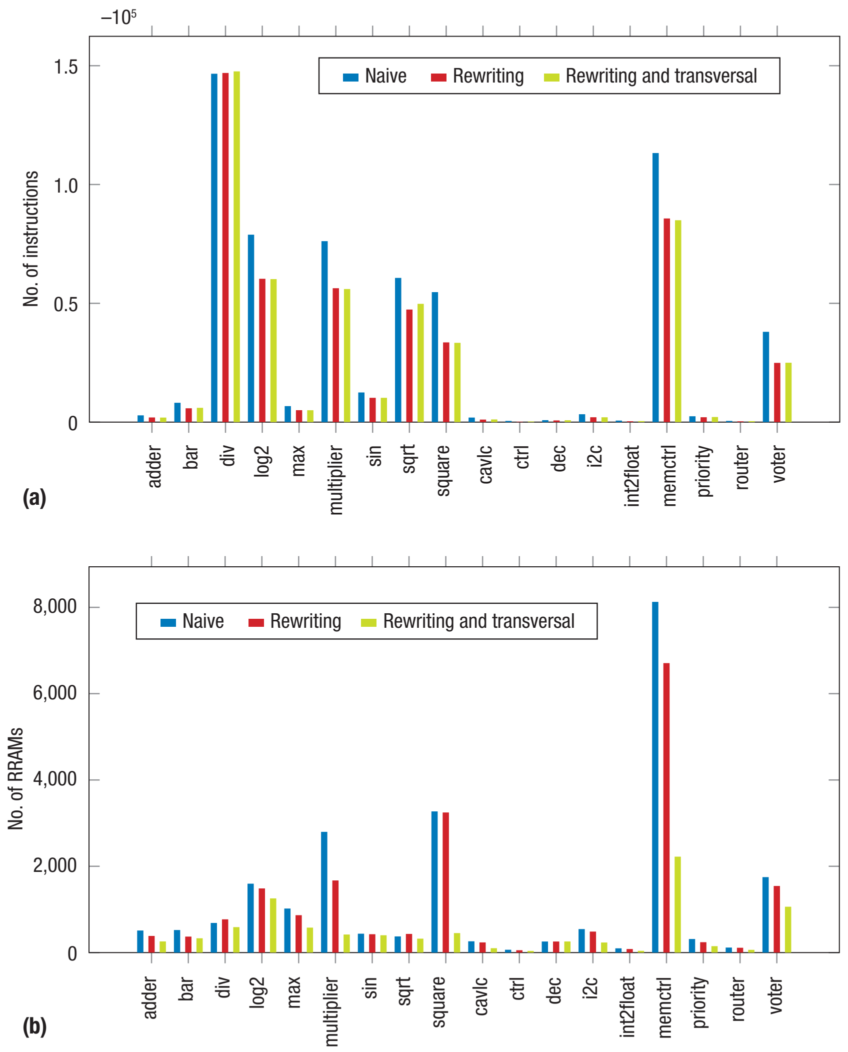


**FIGURE 5.** Experimental results after compiling MIGs into PLiM programs: (a) number of instructions and (b) number of RRAMs. The blue bars show a configuration in which the MIGs were translated naively without taking rewriting or node-traversal heuristics into account. The red bars show the results after rewriting, and the brown bars show the results after rewriting and node-traversal heuristics.

## ACKNOWLEDGMENTS

We thank Luca Amarù and Giulia Meuli for fruitful discussions. This research was supported by H2020-ERC-2014-ADG 669354 CyberCare, the Swiss National Science Foundation (200021-169084 MAJesty and 200021-146600), the University of Utah SEED grant 51900298, and the University of Bremen's graduate school SyDe (System Design), funded by the German Excellence Initiative.

## REFERENCES

1. M. Chang et al., "Designs of Emerging Memory Based Non-volatile TCAM for Internet-of-Things (IoT) and Big Data Processing: A 5T2R Universal Cell," *Proc. Int'l Symp. Circuits and Systems* (ISCAS 16), 2016, pp. 1142–1145.
2. M. Ueki et al., "Low-Power Embedded ReRAM Technology for IoT Applications," *Proc. Symp. VLSI Circuits* (VLSI Circuits 15), 2015, pp. 108–109.
3. R. Fackenthal et al., "A 16Gb ReRAM with 200MB/s Write and 1GB/s Read in 27nm Technology," *Proc. IEEE Int'l Solid-State Circuits Conf.* (ISSCC 14), 2014, pp. 338–339.
4. S. Sheu et al., "A 4Mb Embedded SLC Resistive-RAM Macro with 7.2ns Read-Write Random Access Time and 160ns MLC-Access Capability," *Proc. IEEE Int'l Solid-State Circuits Conf.* (ISSCC 11), 2011, pp. 200–202.
5. S.B. Akers Jr., "Synthesis of Combinational Logic Using Three-Input Majority Gates," *Proc. 3rd Ann. Symp. Switching Circuit Theory and Logical Design* (SWCT 62), 1962, pp. 149–158.
6. H.S. Miller and R.O. Winder, "Majority-Logic Synthesis by

Geometric Methods," *IRE Trans. Electronic Computers*, vol. 11, no. 1, 1962, pp. 89–90.

7. R. Lindaman, "A Theorem for Deriving Majority-Logic Networks within an Augmented Boolean Algebra," *IRE Trans. Electronic Computers*, vol. 9, no. 3, 1960, pp. 338–342.
8. R. Zhang, P. Gupta, and N.K. Jha, "Majority and Minority Network Synthesis with Application to QCA-, SET-, and TPL-Based Nanotechnologies," *IEEE Trans. CAD of Integrated Circuits and Systems*, vol. 26, no. 7, 2007, pp. 1233–1245.
9. L.G. Amaru, P.-E. Gaillardon, and G. De Micheli, "Majority-Inverter Graph: A Novel Data Structure and Algorithms for Efficient Logic Optimization," *Proc. Design Automation Conf.* (DAC 14), 2014, article no. 194.
10. H.P. Wong et al., "Metal-Oxide RRAM," *Proc. IEEE*, vol. 100, no. 6, 2012, pp. 1951–1970.
11. P.-E. Gaillardon et al., "The Programmable Logic-In-Memory (PLiM) Computer," *Proc. Design, Automation and Test in Europe* (DATE 16), 2016, pp. 427–432.
12. M. Soeken et al., "An MIG-Based Compiler for Programmable Logic-In-Memory Architectures," *Proc. Design Automation Conf.* (DAC 16), 2016, article no. 117.

## ABOUT THE AUTHORS

**MATHIAS SOEKEN** is a scientist at EPFL (École Polytechnique Fédéderale de Lausanne). His research interests include the many aspects of logic synthesis and formal verification. Soeken received a PhD in computer science and engineering from the University of Bremen. He is a member of IEEE and ACM. Contact him at mathias.soeken@epfl.ch.

**PIERRE-EMMANUEL GAILLARDON** is an assistant professor in the Electrical and Computer Engineering Department at the University of Utah. His research interests include reconfigurable logic architectures and digital circuits exploiting emerging device technologies and novel electronic design automation techniques. Gaillardon received a PhD in electrical engineering from the University of Lyon. He is a Senior Member of IEEE and a member of ACM. Contact him at pierre-emmanuel.gaillardon@utah.edu.

**SAEIDEH SHIRINZADEH** is a PhD student in the Group for Computer Architecture at the University of Bremen's Institute of Computer Science. Her research interests include multiobjective optimization, evolutionary computation, logic synthesis, and in-memory computing. Shirinzadeh received an MSc in electrical engineering from the University of Guilan. Contact her at s.shirinzadeh@uni-bremen.de.

**ROLF DRECHSLER** is a professor and the head of the Group for Computer Architecture at the University of Bremen's Institute of Computer Science. He is also the director of the Cyber-Physical Systems Group at the German Research Center for Artificial Intelligence (DFKI). Drechsler's research interests include the development and design of data structures and algorithms, with a focus on circuit and system design. He received a PhD in computer science from J.W. Goethe University Frankfurt am Main. Drechsler is an IEEE Fellow. Contact him at drechsler@uni-bremen.de.

**GIOVANNI DE MICHELI** is a professor and director of the Institute of Electrical Engineering at EPFL. He is also a program leader of the Nano-Tera.ch program. De Micheli is a recipient of the IEEE Computer Society Harry Goode Award and the European Design and Automation Association (EDAA) Lifetime Achievement Award. He is a Fellow of IEEE and ACM, a former president of the IEEE Circuits and Systems Society, a former IEEE Division 1 director, and a member of Academia Europaea. Contact him at giovanni.demicheli@epfl.ch.