DExIMA: Design Explorer for In-Memory Architectures

IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 27 March 2025, accepted 15 April 2025, date of publication 2 May 2025, date of current version 12 May 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3566614

RESEARCH ARTICLE

DExIMA: Design Explorer for In-Memory Architectures

ANDREA COLUCCIO[1], ALESSIO NACLERIO[1], (Graduate Student Member, IEEE), MARCO VACCA[1], GIOVANNA TURVANI[1], MARIAGRAZIA GRAZIANO[2], AND MAURIZIO ZAMBONI[1]

[1]Department of Electronics and Telecommunications, Politecnico di Torino, 10129 Turin, Italy
[2]Department of Applied Science and Technology, Politecnico di Torino, 10129 Turin, Italy

Corresponding author: Alessio Naclerio (alessio.naclerio@polito.it)

ABSTRACT In recent years, Computer-Aided Design (CAD) software have become indispensable tools for designing, testing, and synthesizing logic circuits. Commercial software provided by companies like Synopsys, Cadence, or Mentor dominate the Electronic Design Automation (EDA) field. Nevertheless, several Open Source tools are also available, and some of them aim at assisting the designer in emerging Beyond-von-Neumann computing paradigms, such as Logic-in-Memory (LiM). LiM is a promising architectural and technological solution to the von Neumann Bottleneck, i.e. the performance gap between the CPU and the memory in classical CPU-Memory structures. In literature, various approaches to the LiM paradigm have been proposed. This paper introduces the **D**esign **Ex**plorer for **I**n-**M**emory **A**rchitectures (DExIMA) tool, which has the ambitious aim of providing a well-defined design flow strategy for the development, validation and performance estimation of a wide range of LiM architectures. Currently, DExIMA focuses on Coarse-grain Logic-in-Memory (CGLiM) architectures, which integrate memory and computation elements at a coarse-grain level. Nevertheless, DExIMA encompasses a flexible architectural model and a modular performance estimation engine that can be adapted to LiM implementations where memory and logic elements are more finely integrated. Hence, DExIMA is a versatile tool offering an environment for testing and comparing different LiM solutions, empowering designers to explore novel approaches in LiM architecture design.

INDEX TERMS Computer aided design (CAD), electronic design automation (EDA), logic-in-memory (LiM), VLSI, SIMD, RISC-V.

I. INTRODUCTION

The technological evolution has brought notable improvements in Computer-Aided Design (CAD) software, requiring more and more computational capabilities and advanced features. In the Electronic Design Automation (EDA) field, a wide selection of tools are provided, assisting the designer in the synthesis and functional verification of Application-Specific Integrated Circuits (ASICs) and Field-Programmable Gate Arrays (FPGAs). Regarding ASICs, designers are aided in the so-called Very Large Scale Integration (VLSI) flow by several highly complex commercial and industrial EDA tools provided by Synopsys, Cadence and Mentor. In literature, open-source or academic EDA tools are available for simulating digital circuits described in Verilog/System Verilog (Verilator [1]), performing logic synthesis (ABC [2]) or producing the physical implementation of Random Access Memories (RAMs) (OpenRAM [3]).

Moreover, engineers have recently started designing specific software or CAD tools to assist designers in emerging computing paradigms. A promising paradigm currently spreading is Beyond von Neumann Computing (BvNC), which mainly aims at overcoming to overcome the limitations of CPU-centric architectures. A von Neumann architecture essentially consists of CPU and memory elements separated and constantly exchanging data. The processing is entirely executed inside the CPU, requiring the continuous movement of data, thus wasting power and computational time [4].

The associate editor coordinating the review of this manuscript and approving it for publication was Libo Huang.

BvNC proposes a solution to this problem that involves bringing the computing part as close as possible to the memory (Near-Memory Computing) or directly inside the memory array (In-Memory Computing or Logic-in-Memory).

Regarding Near-Memory Computing, solutions like 3D Stacked DRAMs and Hybrid Memory Cubes (HMC) [5], [6] have been proposed. HMC consists of several layers of DRAM vertically stacked and connected to a computational one using Through Silicon Vias. Consequently, the data path is reduced, resulting in increased performance. Modern tools implementing Near-Memory solutions are CLAPPS [7] or HMC-Sim [8]. CLAPPS, for instance, is developed in System C, and creates, simulates and generates synthesizable RTL of custom Hybrid Memory Cube solutions, exploiting the Gem5 [9] tool. A different implementation of the Near-Memory approach concentrates on Single Instruction Multiple Data (SIMD) co-processors. This approach is adopted in GP-SIMD [10] and Hybrid-SIMD [11]. GP-SIMD is a SIMD co-processor comprising multiple Processing Units (PUs) that share memory with the CPU and reside near the memory array. Each PU consists of a full adder, a logic block performing bitwise operations and four registers. Hybrid-SIMD is a BvNC architecture with an array-like structure consisting of Smart Rows, in which memory and computation elements coexist, and Standard Rows, which only provide memory capabilities. Hybrid-SIMD efficiently performs parallel computations. Therefore, by delegating part of the calculations to Hybrid-SIMD, it is possible to speed up the execution of several algorithms, while also reducing access to the memories and the associated energy.

In the Logic-in-Memory (LiM) field, the objective is to perform computations directly inside the memory array. Therefore, finer-level modifications to the internal structure of the array are carried out to allow for in-place logic operations. Some solutions rely on SRAM and DRAM memories [12], [13] implementing bitwise logic or arithmetic functions by sensing the voltage or currents on the bitlines (BLs). Others employ emerging technologies such as Resistive RAM (RRAM) [14], [15], Magnetic Tunnel Junctions (MTJs) [16], [17] or Phase Change Memories [18], that can be easily structured in crossbars. These devices show inherent memory and logic opportunities enabled by their intrinsic resistance, which can be controlled either for storage and computing purposes. Ferroelectric FETs (FeFETs) are emerging transistor-like devices that exhibit memory capabilities through the ferroelectric material layer inside their gate stack. They can be exploited for designing logic cells with storage capability or memory-like structures that perform boolean logic operations within the array. Tools are also proposed for In-Memory solutions. For example, Eva-CiM [19] provides a performance evaluation flow that provides results on the adoption of SRAM-based or FeFET-based arrays in a standard CPU-based system exploiting Gem5.

In this paper, the **D**esign **Ex**plorer for **I**n-**M**emory **A**rchitectures (DExIMA) tool [20] is presented. Its purpose is to provide designers with a well-defined methodology for the exploration of BvNC systems that encompasses development, validation and performance estimation techniques. DExIMA tool consists of two principal components, namely *DExIMA-CAD* and *DExIMA-Backend*. The first is implemented using Python, providing a Graphical User Interface (GUI) that incorporates a schematic capture. Its main purpose is to guide users through the entire design flow of the BvNC devices. The latter is a fast ad-hoc performance estimator implemented in C++. The current version of DExIMA focuses on a specific type of BvNC implementation that, within the scope of this paper, is referred to as *Coarse-grain Logic-In-Memory (CGLiM) Architectures*. The key characteristics of CGLiM architectures are inspired by architectures proposed by [11], [21], [22], and [23], in which standard CMOS-based memory and computational elements are exploited to create array-like structures allowing for highly parallel data processing. Although DExIMA currently targets the design of CGLiM architectures, the tool employs a highly flexible architectural model and a modular internal structure. Therefore, it is possible to use DExIMA to analyze different array-based architectures, including LiM systems in which single memory cells encompass both memory and logic capabilities. This is the case of LiM arrays based on SRAM and, especially, emerging Beyond-CMOS technologies, such as MTJs or FeFETs. As a result, future expansions of DExIMA will also allow for the inclusion of finer-grained LiM array structures within the tool design flow. To the best of our knowledge, DExIMA is the first tool suite that enables the architectural exploration of a wide range of Logic-in-Memory structures by addressing all aspects of their design. To summarize, the main features of DExIMA are:

1) the possibility to control each part of the LiM architecture design, from the lowest-level hierarchical blocks up to the top entity, including the control unit;
2) the automatic generation and simulation of synthesizable VHDL code describing the architecture;
3) the introduction of DExIMA-Backend, which provides a fast ad-hoc performance estimation engine currently based on the CMOS 45nm technology node. However, the support for other nodes can easily be integrated. Moreover, DExIMA-Backend has been designed with a modular structure to empower developers to craft performance estimation engines for LiM structures based on emerging technologies. A back-annotation process can be exploited to simulate the circuit and obtain a much more accurate power estimation by considering the switching activity values of each net;
4) a tool to directly evaluate the impact of the LiM solution inside a classical von Neumann context. This is fundamental to assessing the possible benefits of the LiM approach.

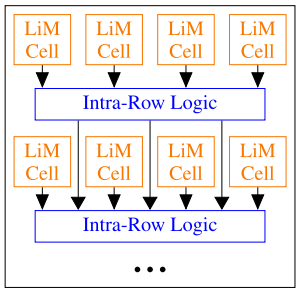

FIGURE 1. Coarse-grain Logic-in-Memory reference architecture model adopted by DExIMA.

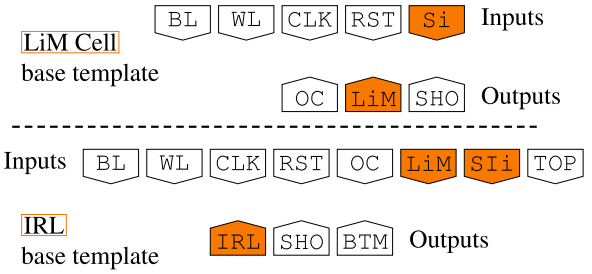

FIGURE 2. Base templates for the LiM cell and the IRL block.

The rest of the paper is organized as follows: section II illustrates the architectural model on which DExIMA works; section III explains the structure of the software and gives details on the fundamental steps executed in the LiM design; section IV discusses the procedure enacted for the validation of DExIMA-Backend; section V describes four examples of benchmarks that can be implemented with DExIMA and provides the performance results considering two systems: CPU-Memory (CPU-Mem), i.e. a classical von Neumann architecture and CPU-Memory-LiM (CPU-Mem-CGLiM), i.e. a beyond von Neumann architecture that considers the insertion of the designed LiM. In this section, DExIMA performance estimation capabilities are analyzed through a comparison with Synopsys Design Compiler; section VI concludes the paper.

II. DExIMA REFERENCE ARCHITECTURAL MODEL

As already introduced, DExIMA currently focuses on designing and analysing BvNC systems that can be modeled as Coarse-grain Logic-in-Memory architectures. In the following, the main characteristics of these architectures are discussed, and DExIMA's features that enable their description are highlighted.

A. COARSE-GRAIN LOGIC-IN-MEMORY ARCHITECTURAL MODEL

In Figure 1, DExIMA reference architectural model for the datapath of CGLiM architectures is shown. As previously mentioned, this model draws inspiration from the systems proposed in [11], [21], [22], and [23]. It consists of an array-like structure that comprises both memory and logic elements. Hence, logic operators are finely integrated within the storage array, which is a crucial characteristic of LiM implementations based on emerging beyond-CMOS technologies. At its core, DExIMA reference architectural model is based on two main basic blocks:

- the *LiM cell* is the fine-grained block, and it consists of a 1-bit memory element that can be connected to elementary logical circuits, usually made of a few logic gates. In CGLiM architectures, a D flip-flop serves as a memory element, while logic gates are implemented in standard CMOS technology. LiM cells are organized in rows (LiM rows) forming a memory array that also includes simple logic elements integrated within each storage cell.
- the *Intra-Row Logic* (*IRL*) is a more complex functional unit that is positioned between two consecutive LiM rows. If required by the implementation, more sophisticated logic blocks can be integrated into the IRL, such as adders, multipliers, shifters, etc.

The CGLiM architecture is organized as an array structure with rows of LiM cells and IRL units interleaved. As depicted in Figure 1, LiM cells can be connected to the subsequent IRL, and an IRL block can receive data from the IRL placed above in the CGLiM array. DExIMA allows users to design and optimize LiM cells and IRL units for the application or algorithm.

The main reason for selecting this architectural model is its high *flexibility*, which allows for adaptation to various LiM implementations. Moreover, it may also serve as a functional model for LiM systems based on emerging technologies, in which LiM cells naturally incorporate both logic and memory capabilities.

B. CGLIM CONTROL UNIT AND DESIGN TEMPLATES

In addition to the definition of the CGLiM datapath, a Control Unit must be introduced to correctly drive the circuit throughout the algorithm execution. As previously mentioned, DExIMA provides the user with high flexibility during the design phase, allowing for arbitrary definition of LiM cells and IRL units based on the application requirements. However, this flexibility increases the complexity of the Control Unit, as the degrees of freedom tend to be unpredictable. To address this challenge and refine the CGLiM design methodology while maintaining high generality, DExIMA provides a series of templates for this design phase. Templates encompass several types of input and output pins for both LiM cells and IRL. In Figure 2, LiM cell and IRL base templates are reported. Starting from the input pins, their functions are explained in the following:

- *Bitline* (BL) is the input data pin of the LiM cell.
- *Wordline* (WL) enables the row of LiM cells whose location is indicated by the input address.
- *Clock* (CLK) and *Reset* (RST) drive the synchronous circuits that, in this case, are the memory elements.
- *LiM cell selector(s)* (Si), *IntraRow selector(s)* (SIi) can be driven from the Control Unit according to the desired function. For instance, they can be connected to the

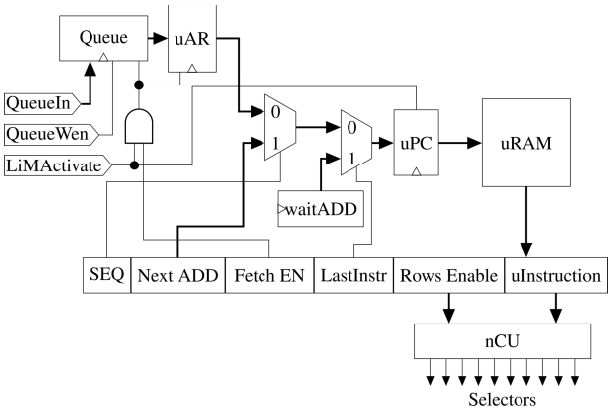

FIGURE 3. Control Unit and custom ISA fields for a DExIMA-generated architecture.

selector input of a multiplexer. However, they can also be used as data input pins.

- *Top* (TOP) is a special pin that allows the connection of an IRL unit to the previous one, as shown in Figure 1. In particular, the TOP pin of the i-th row is connected to the bottom pin (*BTM*) of the (i-1)-th row.

The output pins have the following meanings:

- *Output Cell* (OC) is usually connected to the output of the memory element of the LiM cell, and it is useful to read the data stored inside the memory element.
- *Output LiM* (LiM) is connected to the output of the logical part of the LiM cell.
- *Output IRL* (IRL) connects the output of the IRL block to the outside.
- *Shared Output* (SHO) is a single bus shared between all the rows of the CGLiM array. This output must be handled by a tri-state port to avoid conflicts.
- *Bottom* (BTM) is the output pin that allows the connection between two IRL units, as previously discussed for *TOP* pin.

OC and LiM signals are LiM cell outputs, but they can also be connected in input to the IRL unit, as they can be reused to perform other computations. Moreover, as depicted in Figure 2, some signals are highlighted in orange, meaning that other templates for the CGLiM structure can contain multiple instances of these elements. More complicated templates can have multiple selector inputs (S0,S1,S2, ...) or multiple LiM/IRL outputs (LiM0,LiM1, ...,IRL0,IRL1, ...). The user can select a template with the template selector tool provided by DExIMA.

After designing and quantifying needed LiM rows and IRL blocks, a specific Control Unit can be instantiated to orchestrate computation and data movement within the array. DExIMA offers a customizable Control Unit template capable of managing a variable amount of selectors and LiM rows. As done in [11], it has been decided to model the Control Unit as a micro-programmed machine since it provides a high degree of flexibility. A micro-programmed machine is composed of a micro-Program Counter (uPC) that addresses a memory called micro-RAM (uRAM), in which the instructions used to control the CGLiM array are defined. In particular, each CGLiM instruction has to define the values of selectors and which LiM rows have to be enabled. Hence, a custom Instruction Set Architecture (ISA) has been developed to suit various designs that can be implemented with DExIMA. In Figure 3, the high-level scheme of the micro-programmed Control Unit is shown, and the custom ISA fields are the following:

- *SEQ* is a single-bit field indicating the next address to choose between the micro address register (uAR), which is equal to the PC incremented by one, or the Next ADD field of the uRAM instruction.
- *Next ADD* indicates a specific address of the uRAM from which the next instruction must be retrieved. This feature can be exploited to support subroutines and loops in future tool advancements.
- *Fetch EN* is the fetch enable signal, asserted at the beginning of the program to sample the starting address. The *LiMActivate* signal is the start signal for the Control Unit, and it must be kept high for the duration of the algorithm execution.
- *LastInstr* is the last instruction flag that selects the wait address (waitADD) and stalls the Control Unit.
- *Rows Enable* is a signal whose parallelism is equal to the number of LiM rows, and it selects which row(s) is(are) enabled by that particular instruction.
- *uInstruction* contains the values of the selectors of the LiM template. Its parallelism is equal to the number of selectors employed in the design.

Rows Enable and uInstruction signals are sent to a nano-Control Unit (nCU) that performs an AND bitwise between the value of the selectors and the enable of each row, generating the actual Selectors signal. By leveraging the described custom ISA, the Control Unit enables SIMD-like operations by allowing simultaneous data processing across multiple LiM rows, significantly boosting computational throughput.

The described DExIMA reference architectural model essentially encompasses the array of LiM cells and IRL blocks, along with the Control Unit that orchestrates the computation and movement of data in the array itself. Due to the regular structure of the model, DExIMA can be mainly leveraged to design LiM implementations that are well-suited for dataflow applications. In these workloads, dynamic control flow is absent, while the focus is kept on data movement and processing. Moreover, these applications highly benefit from the SIMD-like computation enabled by the designed Control Unit.

In conclusion, supposing that the CGLiM architecture is integrated into a classic von-Neumann system encompassing CPU and memory, the following steps are required to accomplish the execution of an algorithm:

1) If input data for the algorithm is not already available in LiM rows, it must be moved from main memory to the CGLiM array by the CPU.
2) The needed instructions must be stored in the uRAM.

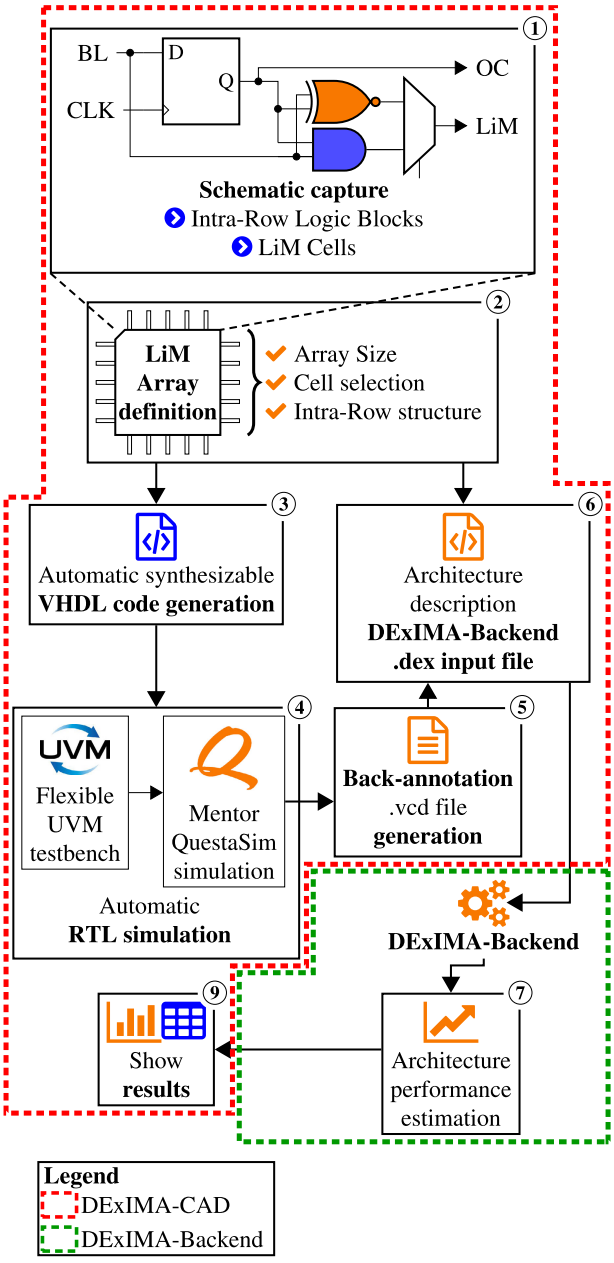

FIGURE 4. DExIMA high-level structure.

3) The Control Unit is started and handles computation and data movement within the LiM array. During this time, the CPU cannot access the LiM array locations.
4) In the end, results are stored in designated LiM rows that can be read by the CPU.

A possible methodology for CGLiM evaluation in a complete system is provided at the end of section III.

III. DExIMA STRUCTURE

In this section, an in-depth overview of the structure of DExIMA is provided. Figure 4 illustrates the internal organization of the tool. As shown in the mentioned Figure, two main sub-projects can be identified within DExIMA:

1) *DExIMA-CAD* encompasses functionalities that enable the description and simulation of the CGLiM array, and the tools that allow the designer to visualize its performance.
2) *DExIMA-Backend* is the C++ engine that estimates the performance of the CGLiM array related to the execution of the required application.

In the following, the design flow steps provided by both DExIMA-CAD and DExIMA-Backend are addressed. Then, the internal structure of DExIMA-Backend is further analyzed. Finally, an interesting feature of DExIMA is also presented, in which results obtained by DExIMA-Backend can be compared to ones retrieved by executing the same algorithm onto a classical RISC-V-based von Neumann system.

A. CGLIM DESIGN PHASES

The complete design flow enabled by DExIMA is presented below. The DExIMA-CAD and DExIMA-Backend stages and their interaction are addressed.

① *Schematic Capture*. This phase represents a crucial part of the CGLiM design, in which the LiM cells and the IRL units are defined. In Figure 5, the main window of DExIMA is reported. The CGLiM architecture design flow is structured hierarchically. By selecting the `LiM cells` tab, the user can start defining the finer-grained blocks (LiM cells), selecting the needed logic blocks and connecting them by exploiting the Schematic Capture. In particular, a LiM cell always consists of a memory element implemented as a D flip-flop for CGLiM architectures and some simple surrounding logic, if required. As suggested before, in future tool expansions, the memory element may also be extended to more optimized structures and different emerging technologies. The same design approach can be followed for the implementation of the IRL unit by selecting the `Intra-Row Logic` tab in the GUI. Both LiM cells and IRL have some default pins defined by the Template typology, as mentioned in section II-B. An example of LiM cell is reported in Figure 5: the cell comprises a simple memory element whose output is connected to the OC pin.

② *CGLiM array Definition*. Once LiM cells and IRL blocks are defined, users can access the `CGLiM Architecture` tab and instantiate the CGLiM array. The tool allows to define the structure of the array by providing the number of required rows and columns. After determining its size, each location of the matrix CGLiM array must be assigned a LiM cell previously designed in phase ①. IRL units between LiM rows must be assigned, too. This selection is accomplished using two tables, where the content of each location can be changed according to the LiM cell/IRL selected. An example of this process is shown in Figure 6(a). After defining the CGLiM

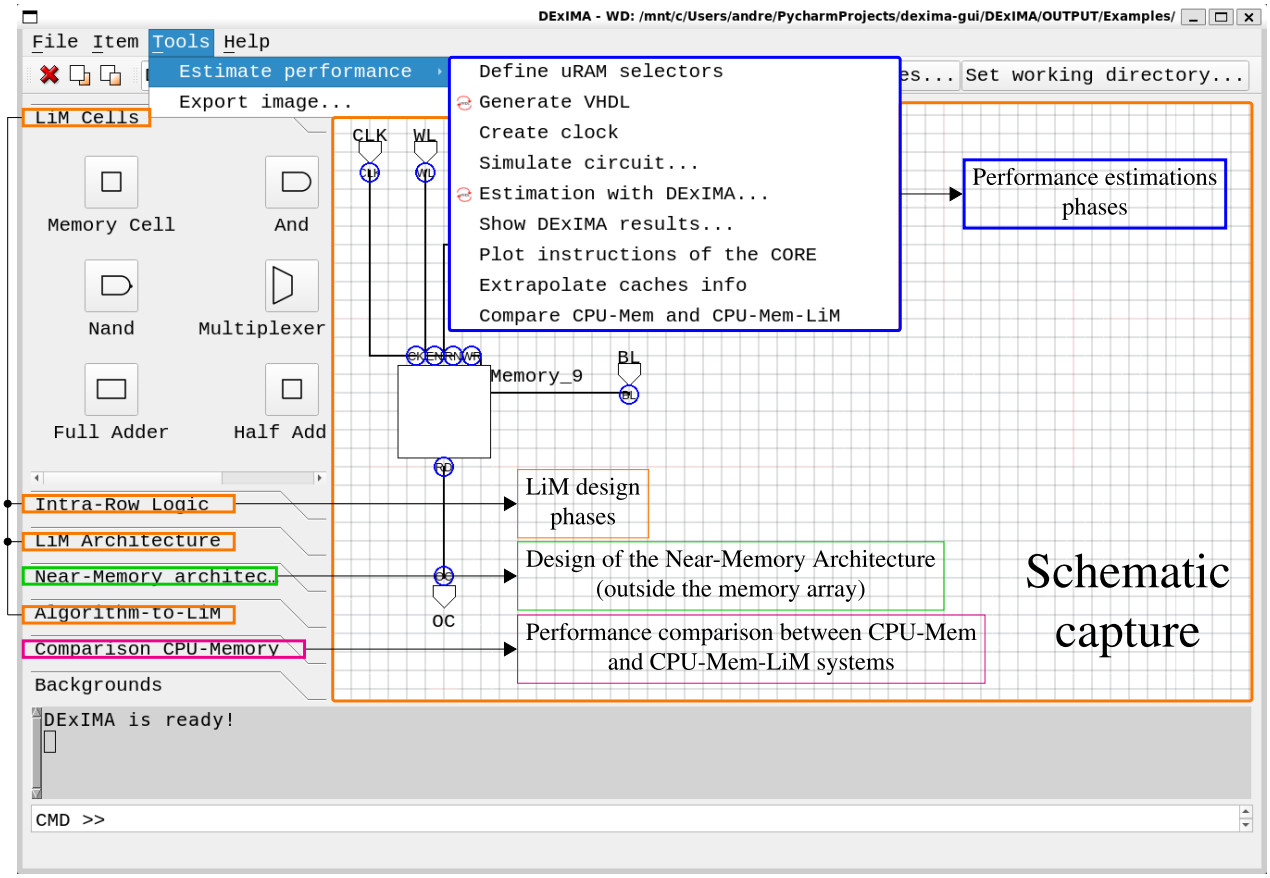

FIGURE 5. Main window of the DExIMA tool.

array, the program automatically instantiates the micro-programmed Control Unit needed to control the circuit.

③ *RTL Code Files Generation*. In this step, the designer can automatically generate a synthesizable VHDL description of the top-entity architecture, consisting of the CGLiM array and the micro-programmed Control Unit.

④ *Automatic RTL Simulation with UVM and Mentor QuestaSim*. The generated VHDL code can be simulated using QuestaSim. For this purpose, DExIMA generates a script to compile the code and runs the simulation of a UVM testbench. UVM is fundamental to standardize the tests of the CGLiM architectures. Independently on the CGLiM design, the testbench follows a procedure that encompasses two main steps. The first is called *data precharging*, and it is accomplished by a `uvm_sequence` class that sequentially writes data inside the array. The second is named *algorithm execution*, and it consists in activating the micro-programmed machine, thus starting to fetch instructions to be executed from the uRAM.

⑤ *Back-Annotation and .vcd File Generation*. At the end of the RTL simulation, two value-change-dump (*vcd*) files are generated. The first contains the waveforms of the signals related to all the internal circuits of the CGLiM architecture, such as LiM cells and IRL units. This information is highly relevant for determining the back-annotated power estimation using DExIMA-Backend. The second file contains the waveforms of the top-level entity signals, i.e. the ones that connect the micro-programmed Control Unit to the CGLiM array.

⑦ *Architecture Description with DExIMA-Backend input file*. After the simulation, DExIMA-Backend can be exploited to estimate the performance of the CGLiM architecture related to the algorithm execution. Hence, an intermediate and custom DExIMA-Backend input file with extension **.dex** is automatically generated. It contains the structural description of the CGLiM array and crucial information for performance estimations.

⑧ *Architecture Performance Estimation*. Once the DExIMA-Backend input file is ready, it is passed to the DExIMA-Backend, which is issued exploiting an embedded terminal. DExIMA-Backend estimates the performance and prints out the results of the CGLiM architecture inside a human-readable log file with extension **.dof** (DExIMA Output File).

⑨ *DExIMA Performance Results*. At this point, the user can visualize the results in bar plots or in tabular format.

B. DEXIMA-BACKEND

To evaluate the CGLiM architecture in terms of area, power and critical path delay, DExIMA is equipped with an ad-hoc

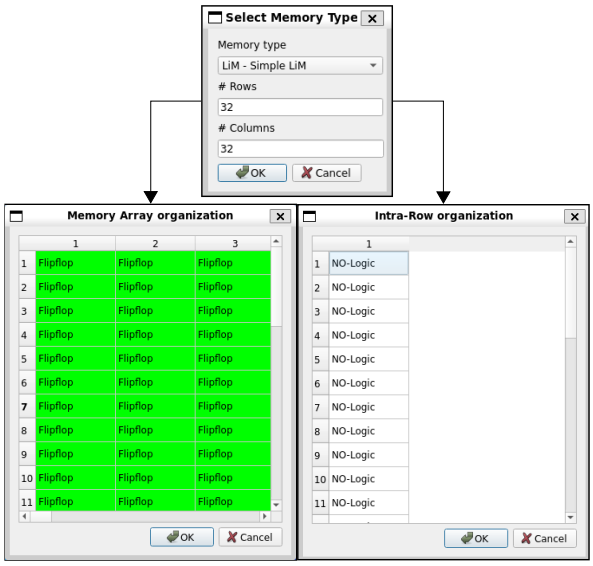

FIGURE 6. Procedural steps to design a CGLiM array. Example of CGLiM array made of LiM cells with a simple Flip Flop and no IRL unit.

estimator called DExIMA-Backend. This tool is realized in C++, and it can estimate the performance of CMOS-based designs, with the possibility to use different technology nodes. It presents a highly modular internal structure that, in future, may be exploited also to integrate other emerging technologies.

In Figure 7, the complete flow adopted by DExIMA-Backend is illustrated. Apart from the **.dex** file, DExIMA-Backend requires the descriptions of the Standard Cells as transistor-level circuits in Spectre format, as well as the technological parameters of the node, such as the on current, off current, gate capacitance, etc. At the beginning of the estimation phase, DExIMA-Backend parses the **.dex** file and creates a series of class instances related to the internal blocks of the CGLiM array. These instances can be single *Standard Cells* such as NOR, NOT, NAND, etc., *Composite Cells*, that are 1-bit blocks containing more than one Standard Cell, and *Multibit Blocks*, that represent complex structures with parallelism greater than 1 bit (such as multipliers, adders, etc.). Composite Cells and Multibit Blocks contain multiple Standard Cells. Hence, it has been decided to locate the computational core for the dynamic and static powers, the area, and the critical path inside the C++ class of Standard Cells. This design choice ensures modularity, as performance estimation of complex blocks always relies on the same core engine.

In the following, an example of an AND2 gate performance estimation is proposed, which schematic is shown in Figure 8.

1) DYNAMIC POWER MODEL ON NETS

Considering each net of the standard cell, the dynamic power equation is given by:

$$P_{dyn} = \frac{V_{dd}^2}{2 \times T_{ck}} \sum_{\forall \text{nets}_i} \left(C_{load_i} \times \alpha_i \right) = \frac{E_{dyn}}{T_{ck}} \qquad (1)$$

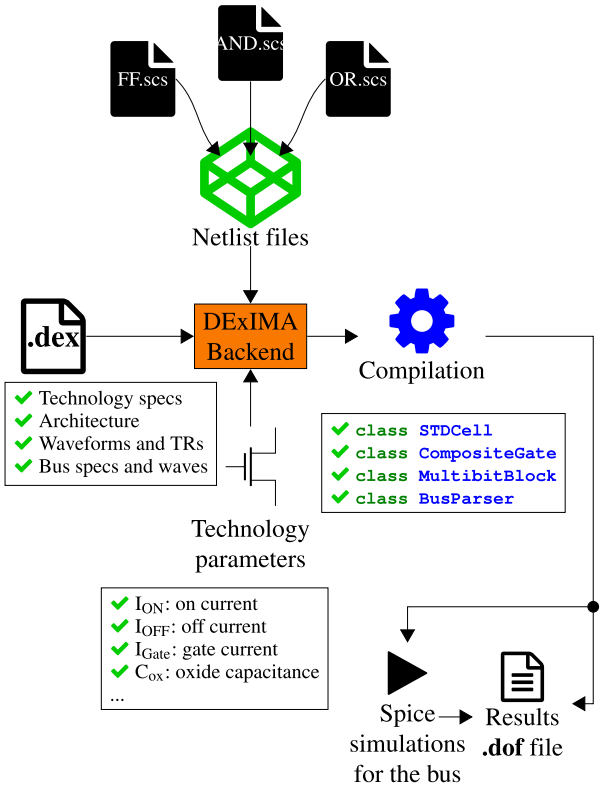

FIGURE 7. DExIMA-Backend estimation steps.

where V_{dd} is the supply voltage, C_{load_i} is the equivalent capacitance that is located on the considered net, α_i is the switching activity on the net and T_{ck} is the clock period. The V_{dd} and T_{ck} are defined from the technology node and the **.dex** file respectively, while C_{load_i} and α_i are computed. Considering a simple AND2 gate in Figure 8, C_{load_i} can be defined by considering the contributions of the source/drain and, possibly, gate capacitances. For instance, considering net O1 in the scheme, the resulting capacitance is given by $C_{Source_{P2}} + C_{Source_{P1}} + C_{Drain_{M1}} + C_{Gate_{P3}} + C_{Gate_{M3}}$ since they are all in parallel. The switching activity α_i of each node is calculated considering the back-annotation process, considering the exact waveform of each input/output provided in the **.vcd** file. The values of each input are propagated inside the cell and each MOS can be ON or OFF depending on the value on its gate. For each combination, DExIMA-Backend propagates the inputs and derives the net states: if the state of the net is different than the previous state, the number of toggles on that net increments. The value of α_i is then obtained by dividing the number of toggles on the net by the total number of toggles.

2) STATIC POWER MODEL

Leakage can be estimated considering the static contributions of the off currents and gate currents of each MOS. For example, considering the AND2 gate in Figure 8, each input combination is propagated inside the standard cell to evaluate the states of the transistors from which the current

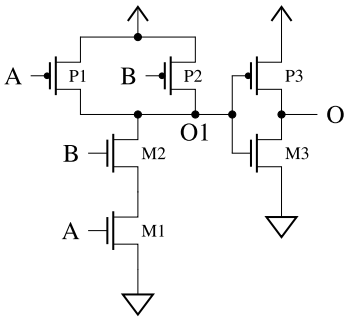

FIGURE 8. AND2 schematic.

contributions are derived. The gate current contribution is present in two cases for both PMOS and NMOS transistors: Gate = Vdd, Source = Gnd, Drain = Gnd or Gate = Gnd, Source = Vdd, Drain = Vdd. The off current instead flows when transistors are off, so the NMOS and PMOS have the $\text{Gate}_{\text{NMOS}} = \text{Gnd}$ and $\text{Gate}_{\text{PMOS}} = \text{Vdd}$, respectively. On the Source/Drain there should be a voltage potential greater than zero in module to have a current flow: this happens when Source = Vdd, Drain = Gnd and vice versa. Considering the combination A = 0, B = 1 the transistors assume the following configurations: (M1: Gate = Gnd, Source = Gnd, Drain = Vdd), (M2: Gate = Vdd, Source = Vdd, Drain = Vdd), (P1: Gate = Gnd, Source = Vdd, Drain = Vdd), (P2: Gate = Vdd, Source = Vdd, Drain = Vdd), (M3: Gate = Vdd, Source = Gnd, Drain = Gnd), (P3: Gate = Vdd, Source = Gnd, Drain = Vdd). The static current, in this case, is equal to:

$$I_{static_{\text{A}=0,\,\text{B}=1}} = I_{gate_{P1}} + I_{off_{M1}} + I_{gate_{M3}} \tag{2}$$

The same procedure is repeated for each input combination and the final static power is obtained as:

$$P_{static} = V_{dd} \times I_{leak} = V_{dd} \times \frac{1}{2^{\#\,\text{inputs}}} \times \sum_{\forall \text{combination}_i} I_{static_i} \tag{3}$$

3) SHORT CIRCUIT POWER MODEL

Estimating the short circuit power is not an easy task without performing SPICE simulations. The reason lies in the intrinsic complexity of correctly calculating the current flow between V_{dd} and Gnd during a commutation, considering all the possible input patterns. However, in the literature several analytical approaches are proposed, starting with the straight-forward inverter structure [24], [25] up to NAND and NOR gates with series/parallel transistors [26], [27]. DExIMA-Backend aims to be an approximated estimator, able to rapidly deliver performance results while maintaining a good accuracy compared to commercial tools like Synopsys Design Compiler. DExIMA-Backend implements the model proposed in [26], representing a good trade-off between accuracy and complexity. This model focuses on estimating a simple inverter case and, for more complicated gates, performs the transistor reduction to an equivalent inverter

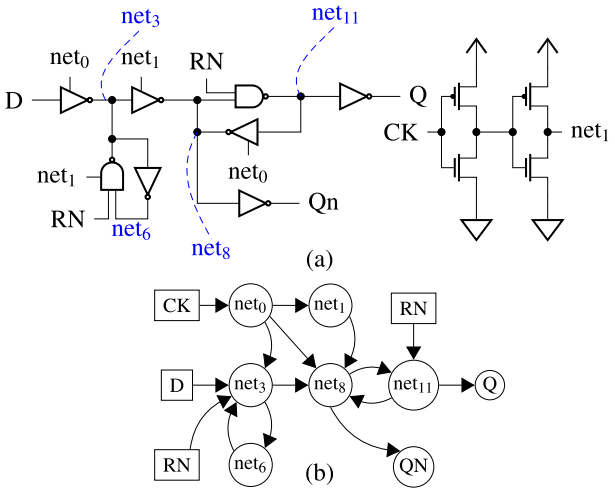

FIGURE 9. (a) Schematic of a Flip-Flop with the equivalent logic gates. (b) Equivalent graph of the Flip-Flop.

by computing the channel widths and lengths according to the transistors configuration. In DExIMA-Backend, the transistor reduction is performed by computing only the widths, maintaining equal lengths, as reported in Equation 4.

$$\frac{1}{W_{\text{series}}} = \sum_{i=0}^{N_{\text{series}}} \frac{1}{W_i} \quad ; \quad W_{\text{parallel}} = \sum_{j=0}^{N_{\text{parallel}}} W_j \tag{4}$$

4) AREA MODEL

The occupation of a single transistor is obtained by adding the channel length and the source/drain lengths and by multiplying them by the channel width. In a standard cell, the area is computed considering the sum of each transistor area plus an overhead given by the interconnections. The channel lengths are assumed to be equal to the minimum channel length of the technology. Regarding the widths, the minimum width that the technology can support is obtained multiplying the minimum channel length by the aspect ratio. However, in the netlists of the standard cells, the transistors widths can change due to the logical effort method. In the calculations, DExIMA-Backend considers the ratio between the actual transistor width and the minimum width of the technology.

5) DELAY MODEL

To estimate the delay, an equivalent graph of the standard cell is created. Considering for example a Flip-Flop cell represented in Figure 9, DExIMA-Backend interprets the schematic as an oriented graph, where the nodes are represented by the common outputs in the schematic and the edges correspond to a logic gate. The critical path of the standard cell is obtained by performing the longest path search on the equivalent graph from all possible inputs to all possible outputs, applying the formula in Equation 5.

$$\text{Delay}_{\text{path}_i} = \sum_{\forall \text{net} \in \text{path}_i} \frac{V_{dd} \times C_{load_{net}}}{I_{on}} \tag{5}$$

C. COMPARISON BETWEEN CPU-MEMORY AND CPU-MEMORY-CGLIM SYSTEMS

Once the CGLiM architecture has been designed and its performance estimated through DExIMA-Backend, a further crucial evaluation can be carried out using DExIMA. The tool offers a methodology to compare the execution of the same application between a classical von Neumann system (*CPU-Mem*) and one enhanced with the introduction of the designed CGLiM array (*CPU-Mem-CGLiM*). This methodology utilizes the C++-based Gem5 emulation platform [9], which enables efficient emulation of the overall system under consideration. As shown in Figure 10 (b-c), the CPU-Mem system consists of an in-order RISC-V CPU and two levels of cache, while the CPU-Mem-CGLiM one can be considered as a beyond-von-Neumann system that is structurally equivalent to CPU-Mem, except for the insertion of the developed CGLiM co-processor.

It is important to note that the CGLiM architecture is not directly integrated into the Gem5 emulation platform via a dedicated C++ model. However, it must be highlighted that the CGLiM architecture should be mainly exploited for its acceleration capabilities rather than functioning as a standard data memory in CPU-related computations. This simplifies its integration and interactions within the considered system, as it is conceptually equivalent to any other memory-mapped accelerator, thus allowing ignoring several issues that may arise, such as cache coherence. Hence, the interaction of the DExIMA-generated CGLiM array and the system can be inferred as explained in the following.

To interact with a CGLiM architecture generated by DExIMA, 3 main steps are required: ① loading data from the main system memory into the CGLiM array, and loading instructions into the uRAM, which is subsequently accessed by the Control Unit. Since memory locations within the array are memory-mapped, a pointer can be used to write data to the required addresses. ② Starting the execution of the algorithm for which the CGLiM array has been designed. On the system side, this should be equivalent to asserting a start bit inside a peripheral register of the accelerator. ③ Let the CPU wait for an interrupt triggered by the CGLiM array at the completion of the execution. As for ①, the movement of data can be emulated using standard memory transactions through C code directly, while ② and ③ would be only useful for retrieving the CGLiM array execution time, bit this is already evaluated through DExIMA simulations and BackEnd estimations. Thus, meaningful performance estimations can be made without directly integrating the CGLiM architecture into the Gem5-based emulation framework.

Referring to Figure 5, the emulation environment can be accessed with the `Comparison CPU-Memory` tab. As illustrated in Figure 10(a), it provides a text editor in which the user can define a C algorithm that will be automatically compiled and executed by Gem5 software. The steps to evaluate the CGLiM impact on a classical von Neumann architecture are the following:

1) *Definition of the CPU-Mem and CPU-Mem-CGLiM C codes.* In this phase, the user specifies the algorithm that will be executed by the RISC-V core of the CPU-Mem system. This algorithm has to be equivalent to the one carried out by the CGLiM array. However, another C code is needed for the CPU-Mem-CGLiM system to move data from caches to the CGLiM array, where the core of the computation is located. This data movement is accomplished through CPU instructions, and DExIMA automatically handles the generation of the C program needed to implement them.
2) *Compilation of the C codes.* By clicking on `Gem5` button in Figure 10(a), the C codes for the CPU-Mem and CPU-Mem-CGLiM architectures are automatically compiled with riscv-gnu-toolchain [28] and executed with Gem5. Important performance results are provided by the Gem5 tool, such as CPU execution time, total number of memory accesses, etc.
3) *Estimation of caches performance.* At this point, caches are characterized using Cacti [29]. By clicking on `Cacti by HP` in Figure 10(a), after providing relevant input arguments, such as the cache size, the associativity and the technology node, DExIMA executes Cacti. It provides useful output parameters, such as the access time, the read/write energy per access, etc.
4) *Estimation of CPU-Mem and CPU-Mem-CGLiM performance.* Exploiting data obtained by Gem5 simulation and Cacti evaluation, it is possible to estimate the overall performance of both C algorithms. However, the C program for the CPU-Mem-CGLiM system only models the movement of data between caches and CGLiM array. Therefore, DExIMA-Backend performance estimation must also be considered to have a complete evaluation of the CPU-Mem-CGLiM performance.

At this point, several options are available. By clicking on `Plot instructions of the CORE` in Figure 5, the previously defined C algorithms are profiled in terms of executed CPU instructions, and the most recurrent ones are reported: an example is shown in Figure 17 (b-c). `Extrapolate caches info` reports useful caches parameters such as the miss rate, the overall accesses per cache type, the total number of hits etc. Lastly, `Compare CPU-Mem and CPU-Mem-CGLiM` option compares the systems selecting a subset of figures of merit, that are the total memory accesses, the total caches access energy, the CPU execution time, CGLiM execution time and CGLiM energy consumption. These last two values come from DExIMA-Backend.

IV. DExIMA-BACKEND VALIDATION

Standard industrial software like Synopsys Design Compiler (DC) can provide advanced tools and algorithms that precisely synthesize and estimate the performance of CMOS-based circuits. However, to perform such complex routines,

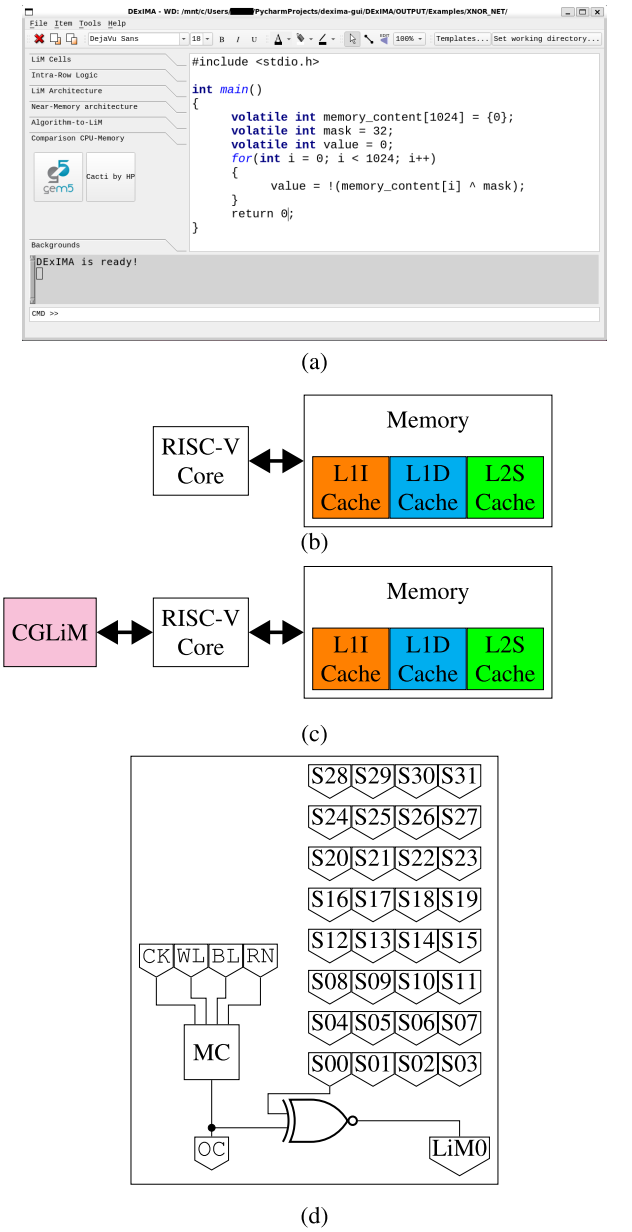

FIGURE 10. (a) Screenshot of the "Comparison CPU-Memory" tab. (b) CPU-Mem and (c) CPU-Mem-CGLiM systems used to estimate the CGLiM impact on a von Neumann architecture. (d) Basic XNOR LiM cell for the XNOR-Net algorithm.

an extensive and complete description of the technology is compulsory, starting from N and P transistors, moving to standard cell structure, layout and characterization. The description of transistors is usually carried out using SPICE files, which also include a vast list of parameters that depend on the model (MOS levels, BSIM, etc) [30] and on the technology node [31]. Once characterized, standard cells are automatically simulated and modeled. Results are grouped within a Liberty file, which is a standard representation of the power, area and delay of any cell belonging to a library. The Liberty file is employed by synthesis and Place&Route tools to provide precise indications about the circuit performance. Although companies or universities normally provide all the required files to perform synthesis and Place&Route, this process is time-consuming: a small modification of the starting transistor model requires the complete re-characterization of the entire library. Therefore, DExIMA-Backend is proposed to lighten this drawback by providing fast and approximated performance estimations of architectures, which can be mostly beneficial for emerging BvNC systems like LiM ones. Moreover, the modular organization of DExIMA-Backend ensures the possibility of integrating other technologies than standard CMOS in the future, further enlarging its exploration capabilities.

A. PROCEDURAL STEPS FOR VALIDATION

Although it aims to provide approximate evaluations, DExIMA-Backend should be validated against a standard EDA tool to verify the quality of the results it produces. The validation procedure of DExIMA-Backend consists of the following steps:

- *Selection of the reference library.* The starting point consists in choosing the reference technology library that is exploited for calculations in DExIMA-Backend. For this purpose, the FreePDK 45nm from North Carolina State University is used [32]. In this design kit, P and N MOS transistors are described with BSIM4 [33], which is a complex and extensive model that characterizes the parameters of the transistors.
- *Simplification of the technology model.* DExIMA-Backend performs static estimations of the circuit, meaning that simulating and predicting all transistor operating regions is not feasible. However, MOS parameters are strongly bias-dependent, meaning that some simplifications are needed. In particular, precisely estimating the MOS equivalent capacitances at the gate, source and drain is crucial for dynamic power and timing computations. By setting the parameter `capMod` to 0 in the P-N MOS models [33], the intrinsic capacitances are described using a simple piecewise model for each operating region. For `capMod = 0` and 50-50 charge partitioning in the channel (`xpart=0.5`), the intrinsic charges equations are extensively explained in [33] and reported in Figure 11 as reference. In the equations, k_1 is the first-order body bias coefficient; *VFBCV* is the flatband voltage for `capMod=0`; V_{gs}, V_{ds}, V_{bs} and V_{th} are the gate-source, drain-source and bulk-source voltages, respectively; ϕ_s is the surface potential; A_{bulk} models the bulk-charge effect and C_o is the gate capacitance expressed as $C_{ox} \times W_{eff} \times L_{eff}$. The equations for each intrinsic capacitance are obtained as the derivative of the charge at the source/drain/gate/bulk terminal with respect to the voltage at the considered terminal. For instance, $C_{gg} = \frac{\partial Q_g}{\partial V_{gs}}$. DExIMA-Backend evaluates C_{gg}, C_{dd} and C_{ss} equations in each MOS region, considering $V_{gs} = 0V, V_{ds} = V_{dd}$ in subthreshold and $V_{gs} = V_{dd}, V_{ds} = 0V$ in triode.

In saturation, once the partial derivatives are computed, there are no stdependences dependencies on the V_{gs} or V_{ds}. The values for each region are then averaged and added to the gate-source and gate-drain overlap capacitances.

- *Characterization of the simplified library.* The FreePDK 45nm library already contains the SPICE netlists of each standard cell, which can be used to characterize the cells with the modifications introduced before. For this purpose, Cadence Liberate is exploited to automatically generate the Liberty file descriptions of the cells and a detailed datasheet of the library.
- *Stardard Cells estimations: comparisons with Liberate datasheet.* In the datasheet generated by Liberate, important parameters, such as the internal dynamic energy, the timing, the leakage power, the capacitance of each pin and the truth table of the cells are reported. These results are useful to validate DExIMA-Backend in estimating the performance of the standard cells. In Table 1, a direct comparison between results obtained using DExIMA-Backend and Liberate is reported. In the datasheet, for each standard cell, the energy and timing are evaluated for each input and output transitions (input rising, input falling, output rising and output falling) and different capacitance load values, i.e. minimum, average and maximum load values (indicated respectively as Min, Avg and Max in Table 1). Choosing one combination as a reference, in Table 1, the minimum value for the dynamic energy and timing is considered, because the standard cells are evaluated with zero-load. As for the static power, the average value is taken. The absolute error is computed between DExIMA-Backend and Liberate, showing a maximum difference of 2.62 fJ for the dynamic energy, 2.92 ps for the timing and 77.09 nW for the static power. These discrepancies are mainly caused by the intrinsic differences between the tools: Liberate performs several simulations, estimating the performance of each transition and measuring the contributions, while DExIMA-Backend estimates the total energy with each node or net active at the same time. From these results, it is evident that precise estimations of the MOS intrinsic capacitances, I_{gate} and I_{off} are crucial to get accurate performance results.
- *Complex Designs: comparisons with Synopsys Design Compiler.* The comparison with the datasheet produced by Liberate is not sufficient to estimate the accuracy of DExIMA-Backend in more complex designs. Therefore, a comparison with Synopsys Design Compiler (DC) is carried out to evaluate the quality of DExIMA-Backend results for designs involving a higher number of standard cells. The Liberty file generated by Liberate is compiled with Synopsys Library Compiler and converted in a database file with extension **.db**. This file can be directly read from DC to perform the synthesis. In this phase, some architectures and blocks are realized in DExIMA-CAD, and the corresponding VHDL code is also

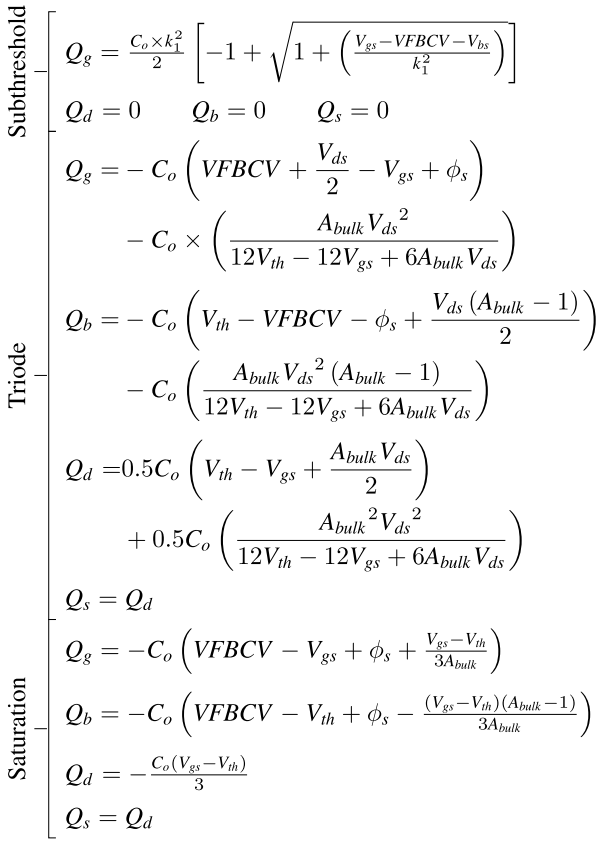

FIGURE 11. Equations of the intrinsic NMOS charges for each terminal and operating region.

generated. DExIMA-Backend is used to estimate the architectures performance, while the VHDL description is synthetized with DC. Hence, the results obtained by the two tools are directly compared. In particular, 17 blocks including array multipliers, registers, ripple-carry adders with 8, 16 and 64 bits parallelism and simple logic gates are considered: for each case, the values of DExIMA-Backend are compared with the Synopsys DC, thus evaluating the relative error in percentage. From this dataset, the mean, standard deviation and the corresponding normal distribution are evaluated, as shown in Figure 12. The mean value results to be in the range of [-8.7%, 5.6%] with a standard deviation ranging from 20.3% to 36.7%. A smaller standard deviation indicates a higher probability of having a relative error around the mean value that, for almost all the cases, is not far from 0%. These results can be considered valid, remembering that DExIMA-Backend should provide a rough and fast estimation of the performance. Once the CGLiM circuits are evaluated with the Backend, the user can directly synthesize the design.

V. BENCHMARKS

In this section, DExIMA is employed to design four different CGLiM architectures, respectively implementing an XNOR-Net, the Matrix-Vector multiplication,

TABLE 1. Standard cells performance comparison between Liberate and DExIMA-Backend.

Cell	Parameter	DExIMA Backend	Liberate	AE*
INV_X1	Dynamic Energy (fJ)	0.67	Min 0.7	0.03
	Timing (ps)	2.02	Min 2	0.02
	Static Power (nW)	74.01	Avg 86.13	12.12
AND2_X1	Dynamic Energy (fJ)	1.99	Min 1.5	0.49
	Timing (ps)	5.22	Min 7.4	2.18
	Static Power (nW)	145.96	Avg 138.22	7.74
NAND2_X1	Dynamic Energy (fJ)	1.73	Min 1.4	0.33
	Timing (ps)	3.21	Min 3.4	0.19
	Static Power (nW)	133.40	Avg 88.09	45.30
OR2_X1	Dynamic Energy (fJ)	2.09	Min 1.6	0.49
	Timing (ps)	4.80	Min 3.2	1.60
	Static Power (nW)	116.65	Avg 118.02	1.37
MUX2_X1	Dynamic Energy (fJ)	4.28	Min 2.1	2.18
	Timing (ps)	10.10	Min 11.6	1.50
	Static Power (nW)	216.77	Avg 182.79	33.98
XNOR2_X1	Dynamic Energy (fJ)	4.62	Min 2	2.62
	Timing (ps)	11.48	Min 14.4	2.92
	Static Power (nW)	262.55	Avg 185.46	77.09
XOR2_X1	Dynamic Energy (fJ)	4.89	Min 2.3	2.59
	Timing (ps)	3.99	Min 4.7	0.71
	Static Power (nW)	219.29	Avg 174.04	45.25

*AE stands for Absolute Error, obtained as $|x_{DExIMA} - x_{Liberate}|$.

the Mean-Variance calculation and the Bitmap Indexing algorithms. In all the proposed benchmarks, the CGLiM array is designed according to the algorithm to be

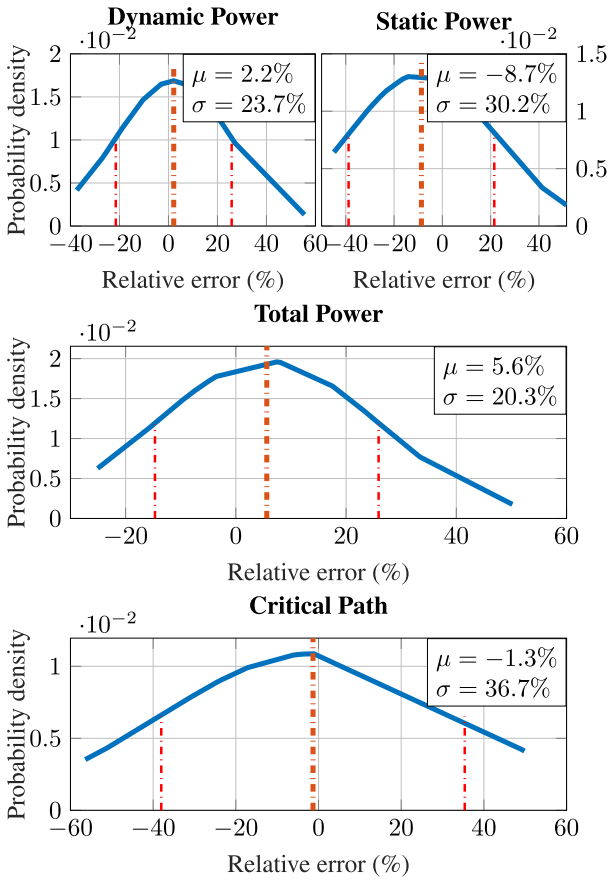

FIGURE 12. Normal distribution of the relative errors for Dynamic, Static, Total powers and timing.

carried out, from the definition of the LiM cells and IRL unitsto the description of the entire array. At last, the `Compare CPU-Mem and CPU-Mem-CGLiM` tool is exploited to compare the execution of the algorithms between the RISC-V-based classical CPU-Mem system and the CPU-Mem-CGLiM one.

A. XNOR-NET: A BINARY NEURAL NETWORK

The first proposed benchmark is a Binary Neural Network called XNOR-Net [34]. A Binary Neural Network approximates a classical Neural Network, aiming to reduce computation complexity. In XNOR-Net, weights and inputs are binarized, and they can assume only two values: ± 1. The convolution operation is approximated as a series of bitwise-XNOR and pop-counting operations. Pop-counting is the difference between the number of ones and zeros in a bit string. A generic i-th convolution operation is computed following the Equation 6

$$Conv_{XNOR} \approx \text{pop-count}\,[!(I_i \wedge W_0), \ldots, !(I_{i+N-1} \wedge W_{N-1})] \tag{6}$$

where N is the kernel window size. This algorithm is implemented in [21], which proposes two CGLiM arrays composed of basic cells with an XNOR gate, to compute the product, and ahalf adder, to compute the pop-count,

Algorithm 1 XNOR-Net Algorithm

```
volatile int memory_content[N] = 0;
volatile int weights[K] = {0x3,0x2,0x3e,...};
volatile int value = 0;
for int j = 0; j < K; j++ do
  for int i = 0; i < N; i++ do
    value = !(memory_content[i] ^ weights[j]);
  end for
end for
```

Algorithm 2 MVM Algorithm

```
for int i = 0; i < ROWS; i++ do
  for int j = 0; j < COLS; j++ do
    Y[i] = Y[i] + M[i][j] * V[j];
  end for
end for
```

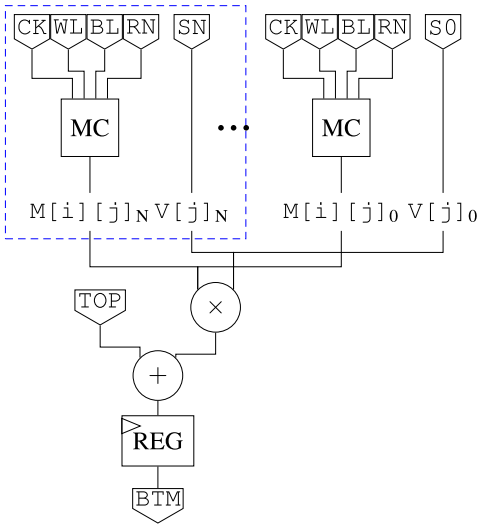

FIGURE 13. LiM cell (bordered in blue) and IRL circuits for the MVM algorithm. IRL contains a multiplier, an adder and a register.

respectively. In this paper, only the first array dedicated to the XNOR products is implemented. The basic LiM cell is captured from DExIMA-CAD and depicted in Figure 10 (d). The CPU-Mem code for the XNOR-Net is reported in Algorithm 1.

The algorithmic steps to implement Algorithm 1 on a CGLiM array are the following:

1) *Data precharging.* The values of the input matrix are binarized and saved in each LiM row such that the 0-th bit of LiM row 0 corresponds to the binarized pixel with coordinates(0,0) of the input image, the 1-st bit of LiM row 0 corresponds to the binarized pixel with coordinates (0,1), and so on. This step requires `N` clock cycles.
2) *Binary convolution computation.* The binarized weights are streamed inside the CGLiM array exploiting the selectors `S00-S31` shown in Figure 10(d). The array performs `N` parallel XNORs at the same time. This step requires `K` clock cycle.

B. MATRIX-VECTOR MULTIPLICATION

By exploiting Intra-Row Logic blocks, the user can implement more complex operations, such as the additions and multiplications required by the Matrix-Vector Multiplication (MVM) algorithm. The related pseudocode is reported in Algorithm 2.

For each line of the CGLiM array, the LiM cell consists of a D flip-flop-based memory cell (MC in the figure) that stores one bit of the input matrix `M[i][j]`,and an additional input pin called Sn, where n indicates the n-th bit. This input provides the value of the `V[j]` vector for each algorithmic step. The IRL unit comprises a multiplier and an adder. The first calculates the product `M[i][j]*V[j]`. The latter performs the addition of `M[i][j]*V[j]`, which is the value obtained by the multiplier in the same IRL, and `M[i][j-1]*V[j-1]`, which is retrieved from the upper IRL block in the CGLiM array. This connection is feasible since the TOP pin of the k-th row is connected to the BTM pin of the (k-1)-th row. Figure 13 depicts the LiM cell and the IRL blocks needed for the execution of the MVM algorithm. Referring to Algorithm 2, the steps executed by the MVM algorithm are the following:

1) *Data precharging.* Matrix elements are saved in each line of the CGLiM array so that `M[0], [0]` is contained in LiM row 0, `M[0], [1]` in LiM row 1, `M[0][COLS-1]` in LiM row COLS, `M[1], [0]` in LiM row COLS+1, and so on. This step requires ROWS × COLS clock cycles.
2) *Multiply and accumulate.* In this phase, the values of `V[j]` are provided to the CGLiM array for each clock cycle using `Sn`. In the first clock cycle, `V[0]` is streamed inside the array. The rows 0,16,...,240 containing `M[0], [0], M[1], [0], ..., M[ROWS-1][0]` are enabled to perform the product, which will be saved inside the register of the IRL. In the second clock cycle, `V[1]` is provided, so the rows 1,17,...,241 containing `M[0], [1], M[1], [1], ..., M[ROWS-1][1]`, respectively, are enabled. At the same time, the TOP signals that are connected to BTM of the previous rows get the value of the previous accumulation, which is summed together with the current product. The algorithm ends when all columns of the input matrix are processed, so when the last enabled set of rows is the ones containing `M[0][COLS-1], M[1][COLS-1], ..., M[ROWS-1][COLS-1]`. This step requires COLS clock cycles.

C. MEAN-VARIANCE

The last proposed benchmark is the Mean-Variance computation, implemented as in Algorithm 3.

Differently from the previous cases, the Mean-Variance calculation consists of sequential operations. Therefore,

Algorithm 3 Mean-Variance Algorithm

```
sum1, sum2, sum3 = 0;
for int i = 0; i < N; i++ do
   sum1 += X[1];
end for
mean = sum1/N;
for int i = 0; i < N; i++ do
   sum3 += X[i] - mean;
   sum2 += (X[i] - mean) * (X[i]- mean);
end for
variance = (sum2-sum3*sum3/N)/N;
```

operations to be performed at the i-th iteration depend on values computed at the (i-1)-th iteration. However, the user can still use DExIMA to design a CGLiM array that works sequentially. To accomplish the Mean-Variance algorithm, the array only features one IRL block positioned at the N-th LiM row that computes the values of sum1, sum2, sum3, mean and variance used in Algorithm 3. The LiM cell, represented in Figure 14, is composed of a memory element and a buffer tri-state on the shared output pin (SHO). Referring to Figure 14 and Algorithm 3, the algorithmic steps are the following:

1) *Data precharging.* The samples are stored inside the array, one sample for each LiM row. This step requires N clock cycles.
2) *Sum1 computation.* The `sum1` value is computed by enabling the `sum1` register and the selector `S0` one row at a time. This steps takes N clock cycles.
3) *Mean computation.* After all samples are scanned, the content of register `sum1` is right-shifted by k positions to accomplish a division by 2^k. In this case study, the number of samples N a power of 2, since the hardware division is a very complex and intensive task that should be delegated to the CPU. The right-shifter result is saved inside the `mean` register. This step requires 1 clock cycle.
4) *Sum3 and sum2 computations.* In the second for loop, `sum3` and `sum2` can be computed by scanning again the array and using the `sum1` register, which holds the values of `X(i) - mean`. The accumulation `sum3` is saved in the corresponding register. The content of `sum1` is used by the multiplier to obtain `(X(i)-mean)*(X(i)-mean)` and this result is then accumulated in `sum2` register. This step requires N clock cycles.
5) *Variance computation.* After `sum2` and `sum3` are ready, the IRL circuit starts the variance computation. The inputs of the multiplier are fed with `sum3` value and the multiplication result is right-shifted. Inside `sum2` register there will be `sum2-sum3*sum3/N`, which is again right-shifted and saved inside the `variance` register. This step requires 1 clock cycle.

Algorithm 4 Bitmap Indexing Algorithm

```
#define BITMAPS 15;
#define ELEMENTS 8;
unsigned char bmp[BITMAPS];
unsigned char temp = 0;
int onecounting_res = 0;
unsigned char bitwise_res = bmp[1] & (bmp[2] | bmp[3]);

for int i = 0; i < ELEMENTS; i++ do
   temp = bitwise_res >> i;
   temp = temp & 1;
   onecounting_res += temp;
end for
```

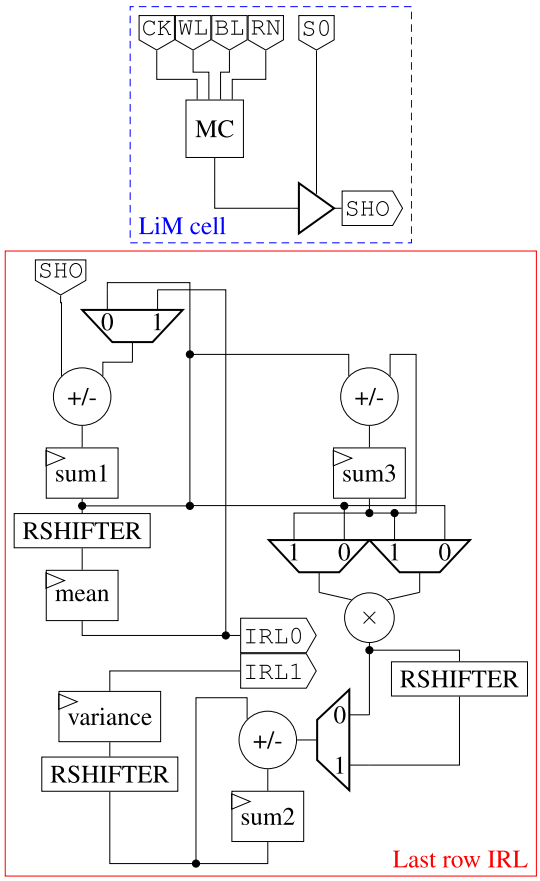

FIGURE 14. Mean-Variance LiM cells and IRL circuits. Selectors pins are not shown for clarity.

D. BITMAP INDEXING

Bitmap indexing (BMP) is a technique designed to accelerate database searches by transforming data representation, allowing for the use of bitwise operations to efficiently locate information within the database. In this method, each attribute in the database is represented by a single bit, which signifies the presence (1) or absence (0) of a specific characteristic in an element. The pseudocode of the algorithm considered for the implementation on a CGLiM architecture is reported in 4.

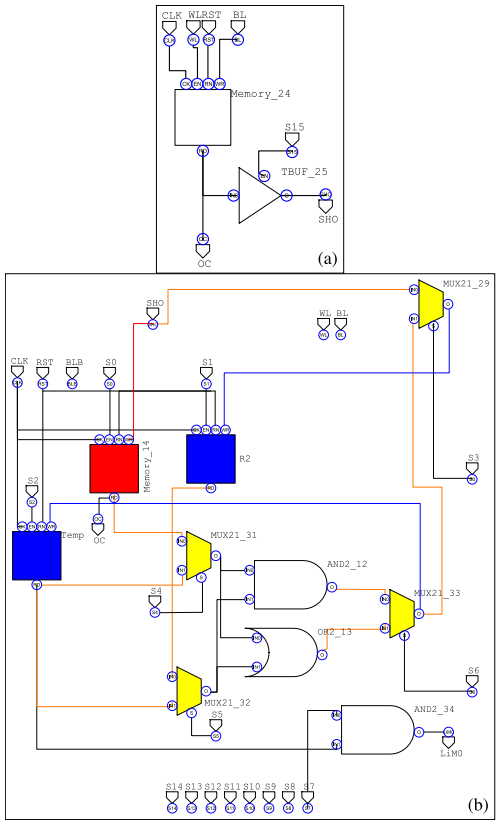

FIGURE 15. Bitmap Indexing algorithm LiM cells. (a) LiM cell with only storage capabilities. (b) LiM cell capable of performing the required bitwise operations.

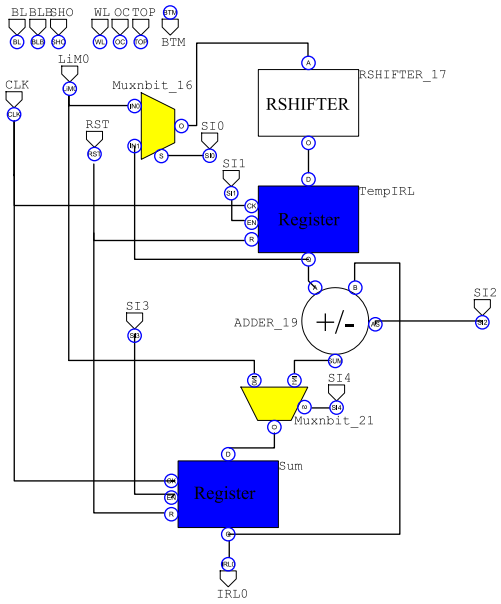

FIGURE 16. IRL unit of the CGLiM array performing the Bitmap Indexing algorithm.

In the proposed algorithm, 8 elements are present with the following 15 characteristics: category 1, category 2, type A, B, C, D, E, …, M. The following query is to be answered: "how many elements belong to category 2 and also to type A or type B?" Regarding the designed CGLiM array, the characteristics are stored in 15 LiM rows, and if an i-th element of the 8 analyzed possesses one of the above characteristics, the i-th bit on the corresponding row will be set to 1, otherwise 0. The CGLiM encompasses standard LiM memory cells and computation LiM cells. The former store data and, when selected appropriately, send data to the SHO bus through a tristate buffer. On the other hand, the latter also perform calculations along with the IRL units. The two types of LiM cell employed are reported in Figure 15, while the structure of the IRL block is shown in Figure 16.

1) *Data precharging.* Each of the 15 bitmaps is loaded into its related LiM row.
2) *Query execution.* The content of LiM row 2 (corresponding to type A) is moved into the D flip-flops of the computation LiM cells. The content of LiM row 3 (corresponding to type B) is moved into *R2* of the computation LiM cells. The bitwise OR is performed and the result is saved in *Temp*. Then, the content of LiM row 1 (corresponding to category 2) is brought into the D flip-flops of the computation LiM cells, and the AND operation is carried out. The described process requires 5 clock cycles. At this point, the IRL unit exploits AND and Right Shift operations to count the number of ones within the sequence of bits obtained from the previous steps. The amount of clock cycles required by this process depends of the size of the bitmaps and it is given by $\sum_{i=0}^{\text{bitmap-size}-1}(i+1)$

E. DEXIMA RESULTS EVALUATION

In this part, the results obtained for the proposed benchmarks are reported and discussed. For all the four tests, after the definition of the CGLiM array, the equivalent VHDL description has been generated, the clock period has been specified and its simulation has been carried out exploiting the UVM testbench. Subsequently, DExIMA-Backend has been employed to perform estimations based on *.vcd* files extracted from the simulation, thus providing results in graphical (for example as shown in Figure 17 (a) for the XNOR-Net algorithm) or tabular form. Furthermore, a comparison between CPU-Mem and CPU-Mem-CGLiM systems has been carried out by exploiting the `Compare CPU-Mem and CPU-Mem-CGLiM` tool. It reports the most recurrent instructions executed by the RISC-V core for both CPU-Mem and CPU-Mem-CGLiM systems. An example of this kind of results is illustrated in Figure 17 (b-c), and it refers to the XNOR-Net algorithm. The same tool proposes a comparison based on five figures of merits: total caches accesses, total caches energy, CGLiM energy, CGLiM algorithm execution time and CPU total execution time (memory read/write and computation). For all these metrics, a lower value is preferable. An example plot is shown in Figure 17 (d), always referring to the XNOR-Net algorithm. In the following, the experimental setup is defined for each algorithm:

TABLE 2. Performance values of each benchmark and comparison CPU-Mem and CPU-Mem-CGLiM.

Benchmark	XNOR-Net			MVM		Mean-Variance		BMP		Category
CGLiM size	1024			256		1024		16		Backend results
Parallelism	32			16		32		8		
Clock period (ns)	6.000			6.000		6.000		6.000		
Critical path (ns)	0.051			0.818		2.867		0.284		
Execution time (μs)	6.210			1.632		18.444		0.342		
Frequency (MHz)	166.667			166.667		166.667		166.667		
Area (mm^2)	0.285			0.386		0.290		0.002		
Dynamic Energy (nJ)	0.445			0.614		0.424		0.002		
Static Energy (nJ)	0.180			0.274		0.185		0.001		
Total Energy (nJ)	0.625			0.888		0.608		0.003		
Static Power (mW)	30.021			45.748		30.754		0.172		
Dynamic Power (mW)	74.111			102.310		70.649		0.349		
Total Power (mW)	104.132			148.058		101.403		0.566		
CPU Execution Cycles (Mticks)	CPU-Mem	1094.06	-89.3%*	88.97	-25.5%*	349.74	-66.6%*	53.48	-0.0%*	CPU-Mem/CPU-Mem-CGLiM
	CPU-Mem-CGLiM	116.76		66.28		116.76		53.47		
Caches Accesses (k)	CPU-Mem	526.53	-92.8%*	26.32	-42.7%*	154.34	-75.5%*	9.16	-0.1%*	
	CPU-Mem-CGLiM	37.80		15.07		37.80		9.15		
Caches Energy (uJ)	CPU-Mem	68.79	-92.6%*	3.52	40.8%*	20.24	-75.0%*	1.31	-2.5%*	
	CPU-Mem-CGLiM	5.06		2.08		5.07		1.28		
CGLiM Power-Delay product (uJ)	CPU-Mem-CGLiM	0.6467		0.2416		1.8703		0.0002		
CGLiM Execution cycles (ticks)		11.00		23.00		2050.00		41.00		
CPU + CGLiMExecution Time (μs) (@f_{cpu} = 1GHz)	CPU-Mem	1094.06	-88.8%*	88.97	-23.7%*	349.74	-61.3%*	53.48	0.6%*	
	CPU-Mem-CGLiM	122.97		67.91		135.20		53.81		

*Percentages indicate the reduction achieved in the CPU-Mem-CGLiM case.

- *XNOR-Net*: the designed CGLiM array performs the XNOR products similarly to the first layer of the Fashion-MNIST Convolutional Neural Network proposed in [21]. It receives in input a 32 × 32 matrix composed of 1 bit elements and a 5×5 kernel, producing 12 output values. The size of the CGLiM array has beendefined to be equivalent to `N` in Algorithm 1. Therefore, it features 1024 LiM rows of 32 bits each. Performance has been evaluated with 45nm CMOS technology using a clock period of 6ns and employing the back-annotation process. XNOR-Net results are shown in graphical format in Figure 17. In Figure 17 (b-c), it is possible to see the effectiveness of the CGLiM approach in reducing the von Neumann Bottleneck: apart from lightening the computational effortof the CPU, it also reduces the number of lw/sw instructions, thus limiting accesses to the memory. From the results in Figure 17 (d), CGLiM XNOR-Net implementation has a considerable impact on the system performance, thus improving both execution time and energy.
- *MVM*: in this case study, a matrix with ROWS = 16 and COLS = 16 is chosen. Therefore, the CGLiM array has been designed to have 256 rows of 16 bits each. Also in this case, performance has been evaluated with 45nm CMOS technology, defining a 6ns clock period, and exploiting the back-annotation process.
- *Mean-Variance*: a total number of 1024 samples (N) has been defined. Hence, the CGLiM array has been equipped with 1024 LiM rows. For this case study, the performance estimation has been carried out with the 45nm CMOS technology without the back-annotation process. When back-annotation is not performed, DExIMA asks the user to provide a value of the toggle rate (TR) to use for the dynamic power estimation. In this case, a TR of 0.5 has been used for each node in the design, emulating a worst-case scenario.
- *Bitmap Indexing*: as mentioned before, in this case study, the CGLiM array has been designed to accommodate for 15 bitmaps of 8 bits each. Hence, the array features 15 8-bit wide LiM rows. Previous benchmarks made use of a fairly large amount of data elements. On the contrary, the implemented BMP algorithm employs a small database (consisting of 15 elements), thus aiming to evaluate the performance impact of an extremely compact CGLiM architecture within a von Neumann architecture. Performance has been evaluated with 45nm CMOS technology, defining a 6ns clock period, and exploiting the back-annotation process.

The performance achieved by each benchmark are summarized in Table 2. It is divided in two sections, namely "Backend results" and "CPU-Mem/CPU-Mem-CGLiM". The former gathers all metrics obtained for the execution of algorithms onto CGLiM arrays exploiting DExIMA-Backend. The latter section provides several comparisons between CPU-Mem and CPU-Mem-CGLiM systems.

For all the implemented benchmarks, the CGLiM approach demonstrates its effectiveness in improving performance, allowing reductions in significant parameters, such as the number of cache accesses. As a consequence, in the analyzed cases, a substantial reduction in the associated energy

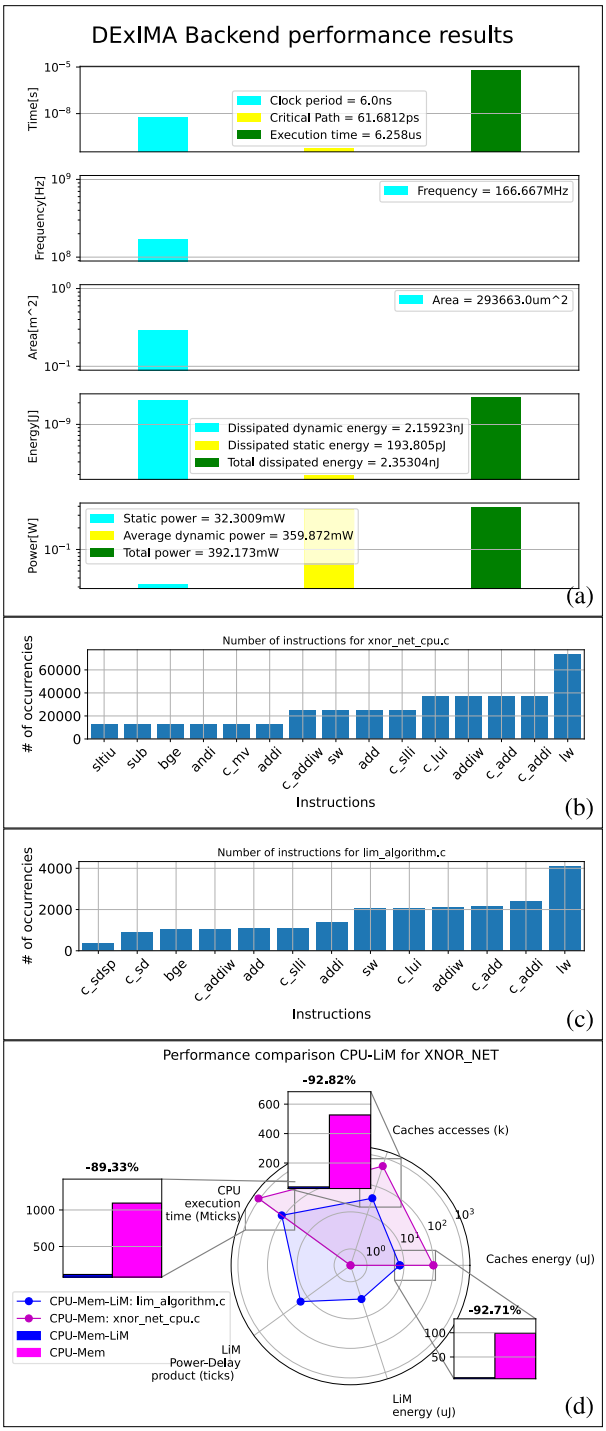

FIGURE 17. (a) DExIMA-Backend results for the XNOR-Net CGLiM array. (b-c) Instructions profiling for the XNOR-Net of the CPU-Mem and CPU-Mem-CGLiM architectures, respectively. (d) Comparison between CPU-Mem and CPU-Mem-CGLiM for the XNOR-Net (axes are in logarithmic scale).

consumption is achieved, thereby confirming the strength of the Beyond-von-Neumann approach. The least advantageous results have been obtained for the architecture implementing Bitmap Indexing, mainly due to the reduced complexity of the algorithm. Hence, as it could be expected, offloading a small computational workload to a CGLiM system has not brought significant advantages in terms of speed-up and energy savings. On the other hand, more promising results have been achieved for larger workloads, such as MVM, XNOR-Net and Mean-Variance algorithms. Among these, MVM presents the least favourable results. Since it is a very straightforward algorithm, the calculations exploited in Algorithm 2 for the CPU-Mem case do not have a significant impact on the system performance, so delegating the computation to CGLiM does not bring a notable advantage compared to XNOR-Net and Mean-Variance cases. However, the MVM execution on a dedicated CGLiM architecture still enables reductions in cache accesses and related energy, as well as reduction in CPU instructions. The CGLiM architecture implementing the XNOR-Net algorithm has turned out to be extremely advantageous, as it significantly reduces execution time by parallelizing operations of significantly, while also achieving energy savings compared to the classical von Neumann architecture.

A final comparison between CPU-Mem and CPU-Mem-CGLiM is proposed, and it considers the execution cycles needed by the two systems supposing a f_{clk} of 1GHz. However, in this comparison, it is supposed to have a slower external CGLiM array. Hence, the execution time for the two systems are obtained as:

$$T_{exe_{CPU-Mem}} = \text{CPU-Mem Ex. Cycles} \times \frac{1}{f_{clk}} \quad (7)$$

$$T_{exe_{CPU-Mem-CGLiM}} = \text{CPU-Mem-CGLiM Ex. Cycles} \times \frac{1}{f_{clk}} + T_{exe_{CGLiM}} \quad (8)$$

In the above equations, $T_{exe_{CPU-Mem}}$ and $T_{exe_{CPU-Mem-CGLiM}}$ correspond to the CPU execution cycles in the two cases, and they are obtained through Gem5 simulation. Also, $T_{exe_{CGLiM}}$ represents the CGLiM array execution time, and it is taken from the first section of Table 2. As shown in the same Table, for the considered cases and CPU frequency, although imposing a slower CGLiM array, results confirm that the adoption of a Beyond-von-Neumann approach could provide significant speed-up for some applications. Also in this scenario, the CGLiM implementation of the XNOR-Net algorithm shows the best speed-up, while BMP still suffers from its reduced workload complexity.

F. COMPARISON BETWEEN DEXIMA AND SYNOPSYS DESIGN COMPILER

In the comparative analysis presented here, the discrepancies between results obtained using DExIMA-Backend and Synopsys Design Compiler across XNOR-Net, MVM, and BMP architectures are examined. This assessment enables evaluating the deviation of DExIMA-Backend computational model when addressing more complex architectures. As it can be observed in Figure 18, DExIMA-Backend consistently overestimates performance, providing results up to 6.9 times worse than Design Compiler in the dynamic power case. This trend is largely attributable to differences in capacitance

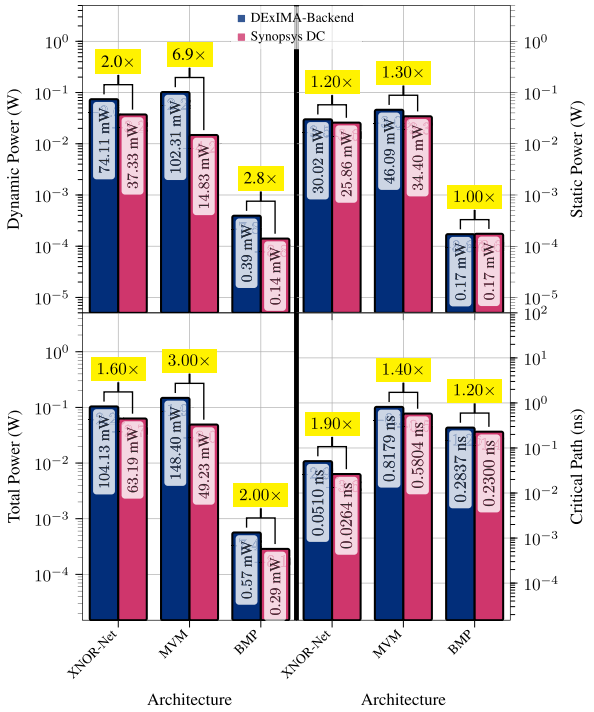

FIGURE 18. Comparisons on Critical Path Delay, Dynamic, Static and Total Power between DExIMA-Backend and Synopsys Design Compiler across the implemented CGLiM architectures.

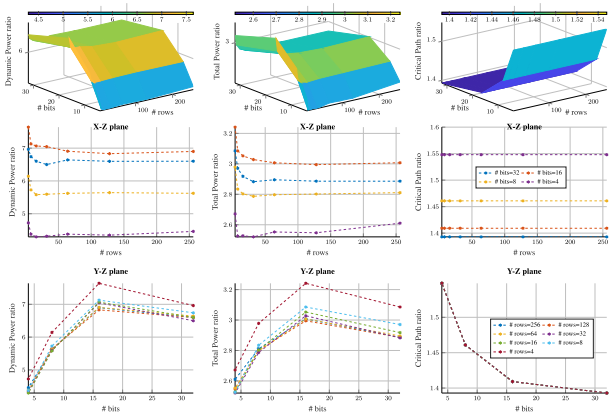

FIGURE 19. Comparison between DExIMA-Backend and Synopsys Design Compiler performance results for the MVM algorithm with varying parallelism and number of LiM rows in the array.

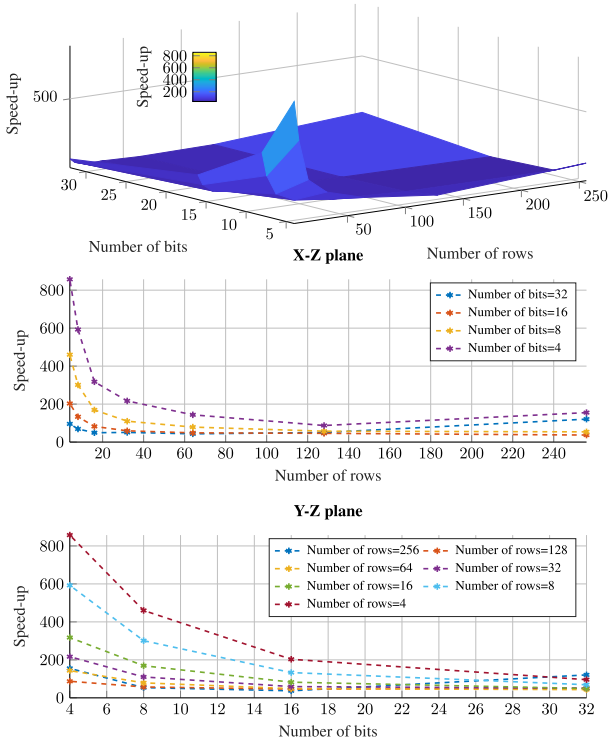

FIGURE 20. DExIMA-Backend vs Synopsys Design Compiler execution time speed-up ratio for the MVM algorithm.

models, methods used for approximating short-circuit power calculations and estimation of node switching activities. Furthermore, discrepancies in the identification of critical paths are also due to the approximate models used, which are heavily dependent on the capacitances of design nodes. However, static power values reported by DExIMA-Backend are consistent with those from Design Compiler for all the considered architectures.

Further analyses have been carried out to investigate the worst results, which are the ones obtained in the MVM case. Comparisons have been made between performance calculated by DExIMA-Backend and Synopsys Design Compiler for MVM architectures with varying parallelisms (from 4 bits to 32 bits) and array sizes (from 4 to 256 rows). As outcomes of these analyses, ratios between dynamic power, total power, and critical path obtained with DExIMA-Backend and Synopsys Design Compiler have been calculated, as depicted in Figure 19. As the number of rows in the array increases, the three evaluated ratios remain relatively constant, while variations are more pronounced with changes in parallelism. These ratios range between (4.3×, 7.6×), (2.52×, 3.24×), and (1.39×, 1.54×), respectively. The constant values obtained for power and timing metrics alongside the increasing number of LiM rows validates the effectiveness of DExIMA-Backend modeling strategies, as it correctly scales performance by size. On the other hand, parallelism variations cause discrepancies in the observed parameters. This result highlights the challenges encountered by the approximated computational model of DExIMA-Backend and the limitations of the tool in switching activities propagation. However, an additional noteworthy metric concerns the speed-up in the execution time required to provide the estimations achieved DExIMA-Backend compared to Synopsys Design Compiler. As shown in Figure 20, the speed-up varies from a maximum of approximately 800 times for smaller architectures to a minimum of about 37 times for more complex architectures (e.g., 16-bit MVM with 256 rows). Despite DExIMA-Backend known limitations, the obtained speed-ups with respect to Design Compiler corroborate DExIMA-Backend capabilities in delivering close-to-reality estimations in a short time. Specifically, for an array composed of 256

32-bit LiM rows, DExIMA-Backend requires approximately 60 seconds, whereas Synopsys Design Compiler takes about 120 minutes.

Although the presented results reveal several limitations of the developed tool, they should not be seen as a complete weakness of the framework. DExIMA provides valuable worst-case scenarios for the proposed beyond-von-Neumann architectures, which are useful in the comprehensive architectural exploration necessary before progressing to chip synthesis, Place&Route, and fabrication stages.

VI. CONCLUSION

This paper presents DExIMA, a CAD tool implemented in Python and C++that proposes a methodology for the exploration of BvNC systems, focusing on LiM approaches. DExIMA provides development, verification and performance estimation strategies that can be adapted to various LiM implementations. Currently, the tool specifically focuses on the CGLiM approach discussed in this paper. The design flow is structured hierarchically, progressing from from the definition of LiM cells and IRL units up to the top-level entity, which is the CGLiM array. All the subsequent steps are automatized by DExIMA, including the definition of the uRAM content, the RTL simulation, the performance estimation using an ad-hoc DExIMA-Backend and the comparisons with a standard RISC-V-based CPU-Memory system. As for the proposed benchmarks, DExIMA successfully demonstrated its flexibility in the implementation of different structures and algorithms. Moreover, the obtained results shows that the integration of a BvNC accelerator in a von Neumann system seems to guarantee a performance boost. Performance results obtained by DExIMA and Synopsys Design Compiler are also compared for several implemented architectures. Although the developed tool shows limitations mainly due to its approximated computational model, it ensures substantial speed-ups with respect to Design Compiler up to 800 times. DExIMA is proposed as an exploration tool for the evaluation of the impact of adopting a BvNC system. The flexibility of its architectural model and performance estimation engine could be that can be leveraged to incorporate various BvNC and LiM approaches within DExIMA evaluation flow. Therefore, future expansions of the tool will provide implementation and estimation strategies for LiM structures based on emerging technologies, in which memory and logic elements are more finely integrated. Future work will focus on establishing a seamless integration between DExIMA and Octantis [35], a tool designed to generate LiM architectures starting from input algorithms described in C code. By adapting the output format of Octantis to be compatible with DExIMA, designers will gain access to a higher-level entry point in the design flow for these architectures. This integration will allow designers to provide a C description of the target algorithm and leverage an automated engine to guide them through the architectural design space exploration process.

REFERENCES

[1] W. Snyder, "Verilator: Open simulation-growing up," in *Proc. DVClub Bristol*, 2013, pp. 1–30.

[2] R. K. Brayton and A. Mishchenko, "ABC: An academic industrial-strength verification tool," in *Proc. Int. Conf. Comput. Aided Verification*, Jan. 2010, pp. 24–40.

[3] M. R. Guthaus, J. E. Stine, S. Ataei, B. Chen, B. Wu, and M. Sarwar, "OpenRAM: An open-source memory compiler," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2016, pp. 1–6.

[4] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2017.

[5] B. Akin, F. Franchetti, and J. C. Hoe, "Data reorganization in memory using 3D-stacked DRAM," *ACM SIGARCH Comput. Archit. News*, vol. 43, no. 3S, pp. 131–143, Jan. 2016.

[6] J. Jeddeloh and B. Keeth, "Hybrid memory cube new DRAM architecture increases density and performance," in *Proc. Symp. VLSI Technol. (VLSIT)*, Jun. 2012, pp. 87–88.

[7] G. F. Oliveira, P. C. Santos, M. A. Z. Alves, and L. Carro, "A generic processing in memory cycle accurate simulator under hybrid memory cube architecture," in *Proc. Int. Conf. Embedded Comput. Systems: Architectures, Model., Simul. (SAMOS)*, Jul. 2017, pp. 54–61.

[8] J. D. Leidel and Y. Chen, "HMC-SIM: A simulation framework for hybrid memory cube devices," *Parallel Process. Lett.*, vol. 24, no. 4, Dec. 2014, Art. no. 1442002.

[9] N. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011.

[10] A. Morad, L. Yavits, and R. Ginosar, "GP-SIMD processing-in-memory," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 1–26, Jan. 2015.

[11] A. Coluccio, U. Casale, A. Guastamacchia, G. Turvani, M. Vacca, M. R. Roch, M. Zamboni, and M. Graziano, "Hybrid-SIMD: A modular and reconfigurable approach to beyond von Neumann computing," *IEEE Trans. Comput.*, vol. 71, no. 9, pp. 2287–2299, Sep. 2022.

[12] H. Jia, H. Valavi, Y. Tang, J. Zhang, and N. Verma, "A programmable heterogeneous microprocessor based on bit-scalable in-memory computing," *IEEE J. Solid-State Circuits*, vol. 55, no. 9, pp. 2609–2621, Sep. 2020.

[13] M. F. Ali, A. Jaiswal, and K. Roy, "In-memory low-cost bit-serial addition using commodity DRAM technology," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 1, pp. 155–165, Jan. 2020.

[14] O. Krestinskaya and A. P. James, "Binary weighted memristive analog deep neural network for near-sensor edge processing," in *Proc. IEEE 18th Int. Conf. Nanotechnol. (IEEE-NANO)*, Jul. 2018, pp. 1–4.

[15] X. Wang, M. A. Zidan, and W. D. Lu, "A crossbar-based in-memory computing architecture," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 67, no. 12, pp. 4224–4232, Dec. 2020.

[16] A. Roohi, S. Angizi, D. Fan, and R. F. DeMara, "Processing-in-memory acceleration of convolutional neural networks for energy-effciency, and power-intermittency resilience," in *Proc. 20th Int. Symp. Quality Electron. Design (ISQED)*, Mar. 2019, pp. 8–13.

[17] A. S. Rakin, S. Angizi, Z. He, and D. Fan, "PIM-TGAN: A Processing-in-Memory accelerator for ternary generative adversarial networks," in *Proc. IEEE 36th Int. Conf. Comput. Design (ICCD)*, Oct. 2018, pp. 266–273.

[18] G. Karunaratne, M. Le Gallo, G. Cherubini, L. Benini, A. Rahimi, and A. Sebastian, "In-memory hyperdimensional computing," *Nature Electron.*, vol. 3, no. 6, pp. 327–337, Jun. 2020.

[19] D. Gao, D. Reis, X. S. Hu, and C. Zhuo, "Eva-CiM: A system-level performance and energy evaluation framework for computing-in-memory architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 39, no. 12, pp. 5011–5024, Dec. 2020.

[20] A. Coluccio, "Exploration of beyond von Neumann computing to solve the memory-wall issue," Politecnico di Torino, Torino, Italy, Jul. 2023, pp. 1–321. [Online]. Available: https://iris.polito.it/handle/11583/2981487?mode=complete

[21] A. Coluccio, M. Vacca, and G. Turvani, "Logic-in-Memory computation: Is it worth it? A binary neural network case study," *J. Low Power Electron. Appl.*, vol. 10, no. 1, p. 7, Feb. 2020.

[22] G. Santoro, G. Turvani, and M. Graziano, "New Logic-In-Memory paradigms: An architectural and technological perspective," *Micromachines*, vol. 10, no. 6, p. 368, May 2019.

[23] M. Andrighetti, G. Turvani, G. Santoro, M. Vacca, A. Marchesin, F. Ottati, M. Ruo Roch, M. Graziano, and M. Zamboni, "Data processing and information classification—An in-memory approach," *Sensors*, vol. 20, no. 6, p. 1681, 2020. [Online]. Available: https://www.mdpi.com/1424-8220/20/6/1681
[24] O. Koufopavlou and L. Bisdounis, "Short-circuit energy dissipation modeling for submicrometer CMOS gates," *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, vol. 47, no. 9, pp. 1350–1361, Sep. 2000.
[25] H. J. M. Veendrick, "Short-circuit dissipation of static CMOS circuitry and its impact on the design of buffer circuits," *IEEE J. Solid-State Circuits*, vol. SSC-19, no. 4, pp. 468–473, Aug. 1984.
[26] S. R. Vemuru and N. Scheinberg, "Short-circuit power dissipation estimation for CMOS logic gates," *IEEE Trans. Circuits Syst. I, Fundam. Theory Appl.*, vol. 41, no. 11, pp. 762–765, Nov. 1994.
[27] S. Nikolaidis and A. Chatzigeorgiou, "Analytical estimation of propagation delay and short-circuit power dissipation in CMOS gates," *Int. J. Circuit Theory Appl.*, vol. 27, no. 4, pp. 375–392, Jul. 1999.
[28] *Riscv-collab/riscv-gnu-toolchain: GNU Toolchain for Risc-v, Including GCC*. Accessed: Dec. 2024. [Online]. Available: https://github.com/riscv-collab/riscv-gnu-toolchain
[29] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "CACTI 6.0: A tool to model large caches," *HP Laboratories*, vol. 27, p. 28, Jan. 2009.
[30] "Spectre circuit simulator components and device models reference," Cadence Des. Syst., Inc. (Cadence), San Jose, CA, USA, Tech. Rep. Product Version 19.1, Oct. 2020.
[31] (2015). *Itrs 2.0: International Technology Roadmap for Semiconductors*. [Online]. Available: http://www.itrs2.net/
[32] *Freepdk45 Technology Model*. Accessed: Dec. 2024. [Online]. Available: https://eda.ncsu.edu/freepdk/freepdk45/
[33] M. V. Dunga, X. Xi, J. He, W. Liu, K. M. Cao, X. Jin, J. J. Ou, M. Chan, A. M. Niknejad, and C. Hu, "Bsim4. 6.0 MOSFET model," Dept. of Electrical Engineering and Computer Sciences, Univ. California, Berkeley, CA, USA, Tech. Rep. 4.6.0, 2006.
[34] M. Rastegari, V. Ordóñez, J. Redmon, and A. Farhadi, "XNOR-net: Imagenet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.*, Jan. 2016, pp. 525–542.
[35] A. Marchesin, A. Naclerio, F. Riente, and M. Graziano, "Beyond von Neumann architectures: Exploring algorithmic opportunities via octantis," *IEEE Access*, vol. 12, pp. 120005–120022, 2024.

ANDREA COLUCCIO received the master's degree in electronic engineering, specialization in electronic systems from the Politecnico di Torino, in 2019, where he is currently pursuing the Ph.D. degree in electronic engineering. His research interests include logic-in-memory and Beyond von Neumann computing paradigms implemented with standard and emerging technologies.

ALESSIO NACLERIO (Graduate Student Member, IEEE) received the B.Sc. and M.Sc. degrees in electronic engineering from Politecnico di Torino, Turin, Italy, in 2019 and 2022, respectively, where he is currently pursuing the Ph.D. degree in electrical, electronics and communications engineering.

His main research interests include the investigation of beyond von-Neumann architectures and the development of techniques for automating the mapping of suitable algorithms on such architectures.

MARCO VACCA is currently an Associate Professor with the Department of Electronics and Telecommunications, Politecnico di Torino. His research interests include innovative and unconventional computer architectures and beyond-CMOS technologies, particularly magnetic devices, such as racetrack memories. His research branch is focused on the development of intelligent systems for industry and agriculture 4.0.

GIOVANNA TURVANI received the M.Sc. degree (Hons. and magna cum laude) in electronic engineering, in 2012, and the Ph.D. degree from Politecnico di Torino. She is currently an Assistant Professor with Politecnico di Torino. Her research interests include CAD tools development for non-CMOS nanocomputing, architectural design for field-coupled nanocomputing, and high-level device modeling for quantum computing and hardware systems for microwave imaging-based techniques for biomedical applications and food quality monitoring.

MARIAGRAZIA GRAZIANO received the Dr.Eng. and Ph.D. degrees in electronics engineering from Politecnico di Torino, Italy, in 1997 and 2001, respectively.

Since 2002, she has been an Assistant Professor with Politecnico di Torino. Since 2008, she has been an Adjunct Faculty with the University of Illinois at Chicago, and since 2014, she has been a Marie-Curie Fellow with London Centre for Nanoelectronics. She works beyond CMOS devices, circuits, and architectures for traditional and quantum processing systems.

MAURIZIO ZAMBONI received the Electronics Engineering and Ph.D. degrees from Politecnico di Torino, in 1983 and 1988, respectively, where he is currently a Full Professor. His research interests include multiprocessor architecture design, IC optimization for artificial intelligence, telecommunication, low-power circuits, and innovative beyond CMOS technologies.