Politecnico di Torino

ScuDo
Scuola di Dottorato ~ Doctoral School
WHAT YOU ARE, TAKES YOU FAR

Doctoral Dissertation
Doctoral Program in Electronics Engineering (35^{th}cycle)

Exploration of Beyond von Neumann Computing to solve the Memory-Wall issue

By

Andrea Coluccio
******

Supervisor(s):
Prof. M. Graziano, Supervisor

Doctoral Examination Committee:
Prof. Alberto Bosio, Full Professor, INL – Ecole Centrale de Lyon, Lyon, France
Prof. Giovanni Finocchio, Associate Professor, Università degli studi di Messina, Dipartimento di Scienze matematiche e informatiche, scienze fisiche e scienze della terra, Italy

Politecnico di Torino
2023

Declaration

I hereby declare that, the contents and organization of this dissertation constitute my own original work and does not compromise in any way the rights of third parties, including those relating to the security of personal data.

Andrea Coluccio
2023

* This dissertation is presented in partial fulfillment of the requirements for **Ph.D. degree** in the Graduate School of Politecnico di Torino (ScuDo).

I would like to dedicate this thesis to all the people who have been with me on this journey, supporting me and giving me the motivation to face the difficulties.

Abstract

The impressive growth in complexity of transistor technology has been the driving force behind modern electronics. Many applications (e.g., Neural Networks), which have become increasingly popular over the years, require processing enormous datasets quickly, placing stringent requirements on the hardware. Many computer architectures employed today are mainly based on a Central Processing Unit (CPU) and memories: the CPU executes the instructions composing programs, takes data from memory, and, once the processing is over, stores back the outcomes in the memory. As a result, these CPU-Memory structures, known as von Neumann architectures, require frequent data exchanges, wasting time and power. In addition, CPU and memory have not followed the same trend, resulting in an increasingly wider performance gap that requires the CPU to wait for memory constantly. This problem, known as Memory-Wall, is the most significant bottleneck preventing modern systems from keeping up with the performance demands of the latest applications. Therefore, a complete redefinition of the computing paradigms is required to overcome the limitations of von Neumann structures. A possibility lies in Beyond von Neumann Computing (BvNC), where part of the computational elements is moved close or even inside the memory, aiming to reduce the data traffic and execute tasks in parallel, achieving energy and time savings. However, developing new computing methods often requires a comprehensive rethinking of the design paradigm: for this reason, researchers have developed specialized software and CADs to assist designers with new computing paradigms or technologies. These tools generally specialize in one or more types of BvNCs and state-of-the-art architectural templates and focus either on simulations, performance estimations, or both. This thesis work presents a tool known as Design Explorer for In-Memory Architectures (DExIMA). Differently from existing tools, the idea is to define the architecture with high flexibility, going through the whole design flow up to automatic simulation, performance estimation, and comparison with von Neumann architectures. DExIMA maintains architectural-level descriptions, so estimations can be done on any technology if implemented inside the tool. The framework allows the designer to develop BvNC architectures in a simple and guided manner by providing a schematic editor, supported implementation of algorithms and control units, Register Transfer Level (RTL) simulation, circuit performance estimation with DExIMA-Backend (an ad-hoc tool implemented in C++), and comparisons with von Neumann architectures using the Gem5 and Cacti by HP tools. At the end of the design flow, DExIMA will give an indication of the performance obtained in the BvNC case to understand how effective this type of solution is compared to a classical implementation. DExIMA also provides the architecture's RTL code that can be synthesized with classical EDA tools. Each step is guided by DExIMA, equipped with a PyQT5-based Graphical User Interface that implements all the previous routines. On the user side, the effort is significantly reduced and consists of defining the algorithm and the architecture. Different benchmarks are proposed that confirm the effectiveness of BvNC and show how, with DExIMA, the user has the possibility to control every step of the design with simplicity, considerably speeding up the whole procedure. The contributions of this work include a study of the state-of-the-art on BvNC proposals, an overview of existing EDA tools applied to BvNC, proposals of standalone BvNC architectures with demonstrated effectiveness, implementation of the DExIMA tool capable of modeling BvNC structures and estimating performance, validation of DExIMA compared to commercial EDA tools, and evaluation of the tool's versatility through the implementation of various benchmarks and analysis of their impact on von Neumann architectures.

Contents

List of Figures

List of Tables

Part I

Background and previous works

⚠

At the end of this document, a glossasy listing acronyms and abbreviations used in this thesis can be found.

Chapter 1

Introduction

Summary

This chapter introduces the concept of Beyond von Neumann Computing, a new computing approach that aims to solve the problems associated with classical architectures based on central processing units and physically separated memories. These architectures are widely used because of their extreme versatility, but they suffer from performance bottlenecks that limit their effectiveness. The section 1.1, introduces the topic and shows the limitations in von Neumann architectures, providing the rationale for why a new computation approach is needed. A study of the state of the art of these architectures is made in the subsection 1.1.1, reporting the most significant cases and providing a categorization of the Beyond von Neumann approach, which appears to be widely used and employed. The subsection 1.1.2 provides an example of an application in which Beyond von Neumann architectures, namely neural networks, with a few examples of architectural implementations provided in the literature. As will be shown in later parts, performing the design of these types of architectures can be extremely complex and tedious, so in this introductory chapter, an overview of tools developed to simplify the design procedures is given. All this is addressed in section 1.2. Finally, in section 1.3, the reason why Beyond von Neumann architectures are so important is presented. Quantitative state-of-the-art results are presented, showing the positive impact this approach has on computing architectures.

1.1 State-of-the-art

Nowadays, electronic devices have rapidly spread in many aspects of human life. Computers, in particular, gained notable improvements over time, becoming fundamental devices for various applications. These applications (such as Machine Learning algorithms or neural networks) have become more and more complex over the years, requiring a massive amount of data (or datasets) to be processed in no time, imposing strict requirements on the hardware or a complete redefinition of the computing paradigms [22]. For this reason, the driving force of the architectural design has focused on integrating more transistors in a single chip by reducing their sizes, especially the channel length, or by investigating new technologies or manufacturing processes like the consolidated Fin-Field Effect Transistors (FinFET) or Gate-All-Around FETs (GAAFET) [23]. However, integrating more transistors will no longer be sufficient to stem the continuous demand for computing power in the near future because of the end of Moore's Law [24, 25]. Moore's Law states that transistors in an Integrated Circuit (IC) double every two years. Up to the present, the silicon industries have followed this trend. Transistor size reduction and technological evolution reduced power and allowed to fit more transistors in the same area. Unfortunately, this trend is about to end, probably reaching a plateau by 2025 [26, 27]. The cause is mainly due to the physical limitations of the channel length and the intrinsic difficulty in controlling the current flow in deep nanoscale transistors. Two directions are possible to overcome CMOS limitations and improve performance: exploring emerging technologies and new computing paradigms and architectures.

The most common architectures used nowadays are based on a Central Processing Unit (CPU), where most operations occur. Each program, application, or task is automatically decomposed in a series of simpler operations (or instructions), that are executed by the CPU, producing results in output. These results and instructions are saved inside memories. These CPU-memory architectures are the most diffused ones and constitute all modern computers. Two main structures are used: the von Neumann architecture [28–30], in which there is only one memory that contains instructions and data into different address spaces, and the Harvard architecture [28], in which data and instruction memories are separated. In both cases, the computational part is located in the CPU, which executes calculations extracting data from memory and saving the results again into the memory itself. This organization achieves a

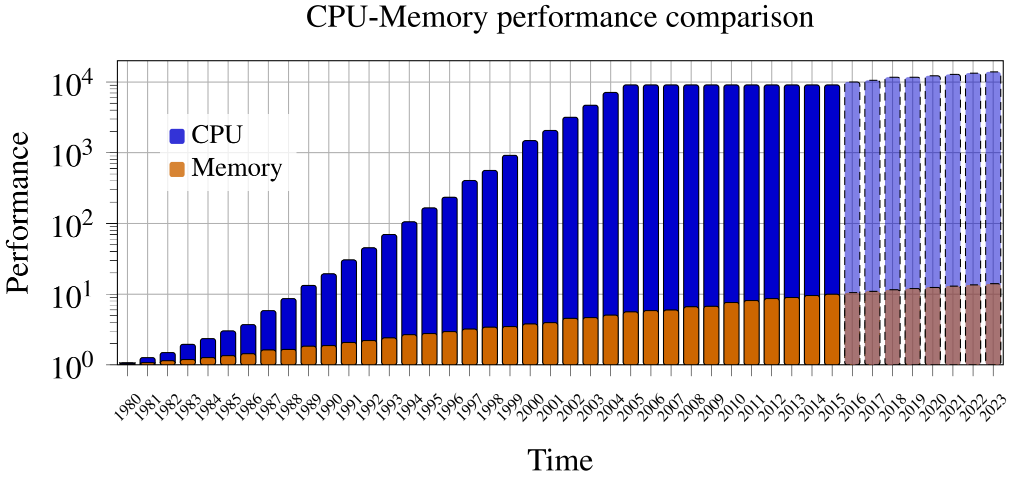


Fig. 1.1 CPU-memory performance comparison. Non-shaded data are taken from [1]. The shaded bars are obtained through predictions and comparisons with existing qualitative graphs in the literature, particularly found in the reference [2].

high degree of flexibility and supports a wide variety of algorithms that can be easily written and compiled for such architectures. However, von Neumann and Harvard architectures are affected by a huge bottleneck: the Memory-Wall [31–33]. CPUs are becoming more efficient and faster, but the memories cannot follow the same trend, implying a performance reduction of the overall system [1]. This effect is confirmed in Fig. 1.1, where the performance of the CPU and the memory are reported over time. The gap between the two elements has started to grow exponentially since the 1980s. Referring to the von Neumann architectures, this effect is also called von Neumann Bottleneck [34], which is particularly evident in memory-intensive applications. Several techniques have been exploited to reduce the effects of the Memory-Wall. For example, Out-Of-Order (OOO) scheduling, concurrent instructions execution during a memory operation, caching (which consists of inserting several levels of small and very fast SRAMs close to the CPU), and prefetching [35] try to recover the latency of the main memory. However, this bottleneck still persists even in the most recent CPU-centric architectures. For this reason, in recent years, new design concepts have been born that specifically aim to reduce or solve the Memory-Wall.

1.1.1 Beyond von Neumann concept

One promising technique to solve the Memory-Wall consists of the Beyond von Neumann Computing (BvNC) [36]. The idea of the BvNC focuses on moving part of the computation inside or close to the memory, consequently reducing the data traffic. However, embedding calculations In-Memory is a challenging task and usually requires a redefinition of the existing memory design paradigms or the employment of exotic emerging technologies.

BvNC can be distinguished into two categories: if the computation is performed inside the memory array, the methodology is called Logic-in-Memory (LiM) Computing, otherwise Near-Memory Computing. LiM approaches are generally based on using different memory technologies. In standard CMOS, the most frequently adopted solutions are based on SRAM [37–41] and DRAM [5, 42] arrays. By enabling multiple wordlines (WLs) at the same time and sensing the voltage potentials on the bitlines (BLs), the BL voltages will depend on the contents of the enabled cells, so by choosing appropriate threshold voltages, the sense amplifiers' outputs will coincide with logic functions. Considering the example in Fig. 1.2 (a), proposed in [3], by precharging RBLF and RBLT to VDD and by enabling both A and B lines simultaneously, the resulting voltage on the bitlines will be proportional to the NOR and AND operation. The sense amplifiers are equipped with additional logic to perform different operations. Similarly, the BL current can also be used to perform a current-based computation. The BL current is proportional to the conductance on the path, so it depends on the state of the transistors inside the cell [43]. Moreover, SRAM-based computing is an advantageous technique exploited in [44] to empower cache memories, allowing logical computations within the array and arithmetic operations with a small Near-Memory computational unit. Content Addressable Memories (CAMs) can also be used to perform calculations since they execute the XOR (or XNOR, depending on the CAM type) operation between the search line (SL) and the cell content. The result will be written on the Match Line (ML), which is initially precharged to VDD: if there is no match, the XOR result is 0, and the ML will be discharged to 0. CAMs, as the name suggests, are addressed for data to perform search operations and are divided into two categories, namely Binary CAMs (BCAMs) and Ternary CAMs (TCAMs), which possess 10 and 16 transistors, respectively. BCAMs take care of searching for the exact word within the array, while TCAMs can also handle don't care cases [45]. In detail, the CAM functionality

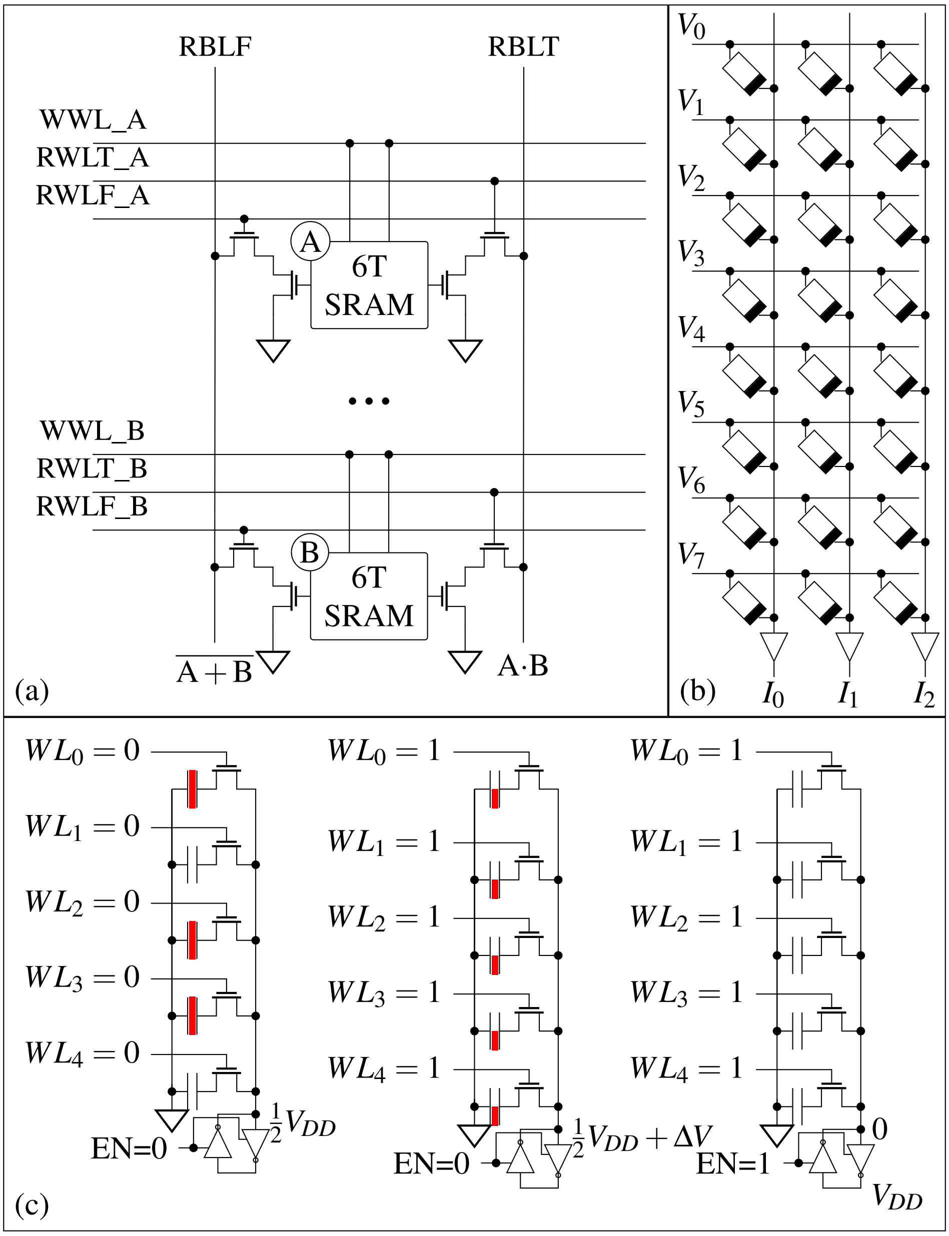


Fig. 1.2 Examples of Logic-in-Memory computing using (a) SRAM-based [3], (b) memristor-based [4] and (c) DRAM-based architectures [5].

relies on several XNOR(XOR)-bitwise operations between cell contents and SLs, followed by word-level AND that produces the search result on the ML [46]: this working principle is adopted in works like [47, 48]. Moreover, taking advantage of this important feature, CAMs can also be used for other in-memory operations: in [45], a memory is constructed mimicking the functionality of CAM, but also extending its operations to AND and NOR between two or more words within the array. Specifically, the authors propose a modified version of classical CAM memory using only 6 transistors (i.e., equivalent to SRAM) but still allowing search operations and reducing the area of the array. This memory can be dynamically reconfigured into three modes: BCAM, TCAM and SRAM, respectively. Taking advantage of different storage modes, the words can be stored by columns in the BCAM and TCAM modes, exploiting classical bitlines to perform logic operations, while they are stored by rows when using the memory in SRAM mode. In order to perform search operations, the authors divided the WLs of cells into right WL and left WL to which search line and negated search line are applied, respectively. When there is a match between the data in the cell and the value on the SL, the bitline and the bitline bar remain high. Otherwise, at least one of them is grounded. For each bitline, a modified sense amplifier is placed, which compares the voltage value with a reference voltage. Finally, each pair of sense amplifiers are ANDed to identify the match. In order to write the data in the column, the authors propose a data writing scheme based on the left and right WLs. Unlike the BCAM mode, TCAM instead needs a pair of columns to store a single word, as an additional bit is needed to represent the don't care value, thus "01" don't care, "00" and "11" for 0 and 1, respectively. In fact, the value "01" will always produce a match, whatever the value of the input search string is. Taking advantage of the BCAM mode of operation, it is also possible to perform bitwise AND operations between two or more rows; by enabling multiple right WLs, AND values can be directly obtained on sense amplifiers; similarly, a NOR operation can be obtained by enabling only left WLs. Another example of a BvNC implementation based on the CAM approach is the one presented in [46], where a methodology is developed to perform additions directly within CAM, which is implemented with different technologies: traditional SRAM and Ferroelectric FET (FeFET). The authors then propose a CAM-based architectural solution for searching within a homomorphically encrypted database, which involves the insertion of additional logic near the memory array that is responsible for computing the sum operation. Moreover, to perform the accumulation, the architecture is provided with in-place copy buffers that allow the rewriting of the sum result directly inside the memory. Another example is the work proposed in [49], where the authors propose an implementation of the Nearest Neighbor search algorithm using a CAM, which performs the computation of the hamming distance directly within the array. This time, the CAM is based on nanoelectromechanical switch technology, a new promising technology for the proposed implementation.

Regarding the DRAM-based computing, a solution is proposed in [5] and shown in Fig. 1.2 (c). The working principle is similar to the SRAM one, but this time the sensing is performed on the capacitors inside the DRAM cells. The computation starts by precharging the BL on $\frac{VDD}{2}$, keeping all the WLs disabled. Then, the WLs are enabled, and the capacitors inside the cells share their charges on the BL, which voltage potential results to be shifted by ΔV, proportional to the contents of the enabled cells. Finally, the sense amplifiers are enabled, and the resulting ΔV voltage will be translated to VDD or 0V, realizing the 5-inputs majority voting function (MAJ5). The majority voting logic can be exploited to compute the 1-bit sum by performing a MAJ3(A,B,Cin) to compute the Cout and a MAJ5(A,B,Cin,$\overline{\text{Cout}}$,$\overline{\text{Cout}}$). Another important LiM approach consists of the use of emerging resistive technologies, very popular in the LiM field, such that they become a promising alternative to CMOS arrays. Technologies like Magnetic Tunnel Junction (MTJ) [50–52, 36, 53], memristor [54–57] and Phase Change Memory (PCM) [36, 58, 59] are often used instead of CMOS-based memory cells. They are capable of storing data and performing computations within the array, maintaining low operating power and high efficiency simultaneously. Such devices store data through their intrinsic resistance, which can assume a high or low value depending on the applied voltage or current: high resistance (R_H) is usually associated with logic '0', while low resistance (R_L) with logic '1'. Usually, the computation is performed by exploiting Kirchhoff's current laws, similar to current-based computation in standard technologies. In the example shown in Fig. 1.2 (b) and proposed by [4], a memristor-based crossbar structure is used both as memory and computation array. The voltages are applied to one end of the memristor devices, which acts as a resistor and provides a current that is proportional to the value of the resistance. The current is sensed by the sense amplifier, which outputs the corresponding value. By enabling multiple wordlines simultaneously, the current will be proportional to the equivalent resistance on the path, performing analog multiply-and-accumulate operations that are useful in applications like neural networks.

The categories explained so far can execute simple logical or arithmetical computations in a highly parallelized fashion. More complex operations must be delegated to an external processing unit: this happens in Near-Memory computing architectures, such as Hybrid Memory Cubes (HMCs), where a processing unit is put very close to the memory. Usually, HMCs are based on DRAM memories, and the intrinsic cell structure is kept unaltered. DRAM arrays are organized in vertically stacked layers connected utilizing Through Silicon Vias (TSVs) to a logical one [60]. Using TSVs shortens the data path and achieves a very high bandwidth. The concept of Near-Memory and In-Memory computing can also be extended to Single-Instruction-Multiple-Data (SIMD) accelerators as in [61, 47, 62], where SIMD processing units are put very close or inside a memory array. In [47], authors presented an associative processor that replaces the last level of cache, enabling both SIMD computation and storage capabilities at the same time, and [61] realizes a SIMD unit closely connected to a cache, improving parallel computations and enabling re-usability of the design. The architecture is called GP-SIMD [61], optimized to run algorithms made up of highly parallelizable functions that process a vast amount of data (e.g., Machine Learning algorithms). Apart from efficiently performing parallel workloads, GP-SIMD uses a common core to handle also sequential activities, making use of Reduction Tree Interconnections. The GP-SIMD infrastructure solves data synchronization using a regular CPU and a SIMD co-processor that shares two-dimensional memory, providing a collection of bit-serial processing units near the memory array, each associated with a different memory row. These bit-serial processing units (PUs) with In-Memory-like operations contain a full adder (FA), a logic function bit block, and four 1-bit registers. Together with a modified SRAM structure, the GP-SIMD forms a huge computational memory with SIMD capabilities. The CPU can also launch the SIMD co-processor in a non-blocking way to complete a job while performing tasks that cannot be accelerated by the GP-SIMD in an efficient manner. This co-processor has two components. The Reduction Tree Network connects all processor units directly linked to the shared memory array in the datapath and a microprogrammed sequencer, driven by the CPU, that is used to operate the SIMD array and initiate parallel processes. The CPU processes the remaining sequential sections once the sequencer finishes the requested tasks.

Summarizing, BvNC can be cathegorized in [63, 13]:

1. Near-Memory: where a portion of the computing blocks is relocated in the neighborhood of memory. Belonging to this category, there are WIDE-IO2 and 3D stacked DRAM memories [64], HMCs and frameworks like GP-SIMD [61].

2. Logic-in-Memory (LiM): three different families belong to this category, differing essentially in the type of computation performed. (1) Computation is performed by physically inserting logic elements inside the memory cell

and LiM array. These elements carry out sparse, distributed, and parallel computation. (2) Memory arrays are themselves used as computation elements. Memory is addressed and provides the result of the computation. Look-Up Tables and CAMs belong to this family. (3) Memory arrays, through analog operations, perform computations on data. These arrays can be SRAM, DRAM, or Non-Volatile Memories (NVM) especially based on emerging resistive technology.

A note about MemComputing

One approach that has been gaining popularity is MemComputing (https://www.memcpu.com/). Like the approaches reported earlier, processing and storage occur in the same physical location [65]. Taking inspiration from the human brain model of computation, MemComputing-based machines are able to self-organize, building the solution pathway themselves, in fact, calculations are performed on devices that exhibit properties of temporal non-locality, thus having memory. Memristors [66] (but also Memcapacitors and Meminductors), for example, are mainly used in this type of approach because, in addition to storing data, they are capable of performing calculations that, as mentioned earlier, are based on current/voltage measurements, which can vary because of their inherent ability to reprogram their intrinsic state values (resistance, capacitance or inductance). MemComputing machines have essential differences from classical von Neumann machines, and their main characteristics are intrinsic parallelism, i.e. all the MemComputing units (or MemProcessors) work simultaneously; functional polymorphism, i.e., the ability to compute different functions without changing the network topology; and information overhead, i.e., the ability of a network of interconnected and interacting MemProcessors to store more information than the same number of non-interacting MemProcessors. Because of its computational capabilities, MemComputing can be used to solve non-polynomial (NP) problems in polynomial time. The number of MemProcessors in solving NP problems in polynomial time can increase exponentially or linearly if they take advantage of the information overhead property, as demonstrated in the article [65]. These kinds of solutions are proposed in works such as [65], which implements the algorithm of the subset sum problem, i.e., locating a subset of integers that form exactly a given sum value. The authors propose a detailed explanation and mathematical modeling of the MemComputing approach, providing an ideal implementation with a DCRAM (Dynamic Computing Random Access Memory), which is a memory composed of MemCapacitors [67], which has an infinite set of states and thus implements analog rather than digital memory cells. Computations are implemented within memory by simply taking advantage of row and column activations; by applying the activation signal to the different rows, the MemProcessors change their state according to the chosen operation, thus promoting massively parallel computations with the application of a small number of input control signals by implementing, in this specific case, sum or, similarly, data movement operations within the array. By taking advantage of these operations, the DCRAM array is able to implement the subset sum problem algorithm in n-1 iterations. However, this approach requires an exponential number of MemProcessor, as it does not exploit the information overhead. Therefore, a second solution to the subset sum problem [68] is proposed, exploiting the Discrete Fourier Transform (DFT) algorithm. Considering a subset of integers G = {a1, a2, ..., an} and the following formula:

$$g(x) = -1 + \prod_{j=1}^{n} \left(1 + e^{i2\pi a_j x}\right) \tag{1.1}$$

And by expanding the products as $e^{i2\pi x \sum_{j \in P} a_j}$, the function $g(x)$ contains all the sums of all possible subsets of G. Now applying the Fourier transform to the function $g(x)$ as:

$$F(f_h) = \frac{1}{N} \sum_{k=1}^{N} g(x_k) e^{i2\pi f_h x_k} \tag{1.2}$$

The spectrum of this function will exhibit peaks at values of f_h, where f_h are the values of the sums and the amplitude of the harmonics is equal to the number of subsets of G. This type of implementation is definitely different from those presented so far, in fact it does not include the use of a memory array to implement computations, but still exploits the concept of nontemporal locality to perform NP algorithms in polynomial time. Another attractive MemComputing-based solution to an NP-Complete algorithm is the one proposed in [66], in which the authors implement a memristor-based MemComputing architecture that solves the Ant Colony problem, consisting of finding the most convenient path within a maze. The maze can be reduced to a graph, and a similar method to the one used by the ants is used to find the path. The flow of ants, in fact, can be analogized to a stream of electric current flowing through memristors that, in turn, model the physical system. If, for example, there are two paths leading from point A to a point B, where

$L_2 = 2L_1$, the equivalent model will be two parallel branches of memristors, where in the first branch, there will be only one memristor, and in the second branch there will be two memristors in series. By applying a constant current to this circuit, the conductance of the devices, as time changes, will change by modeling the amount of pheromones within the two paths, all done in a single algorithmic step.

A nexus with classical logic MemComputing is a completely different way of performing computation based on a physical system that exploits temporal non-locality, so it is a much broader physical concept that can also be applied in classical logic as we currently know it. As shown in [69], an OR port can be transformed into a MemComputing OR port by transforming its digital inputs into continuous variables (e.g., voltages and currents) and, along with them, also "memory variables" are associated. The OR gate, in the digital domain, has 4 different equilibrium states that are analogous to local minimum points in the analog world. By associating memory variables, it is possible to transform these local minimum points into saddle points, where MemComputing performs computations in search of equilibrium. Logic gates used in MemComputing are therefore named Self-Organized Logic Gates (SOLGs) [70], and are devices whose terminals can be either input or output, so the current can flow in both directions. Within a SOLG, Dynamic Correction Modules (DCMs) ensure the Boolean satisfiability of the logic gate. By combining multiple SOLGs, a SOLC (Self-Organized Logic Circuit) [71] can be constructed, and because of the ability of the current to flow in both directions, a SOLC can "run in reverse," solving problems that are simpler in the opposite direction (e.g., Prime Factorization). Starting with SOLCs, Self-Organized Algebraic Gates (SOAGs) [72] were developed, which are based on the same principle of operation as SOLCs. However, this time they self-organize to model algebraic expressions, aiming to solve problems such as Integer Linear Programming. In [72], the authors compared a solution implemented with SOAGs using the MemCPU solver with Gurobi 8.0, which is a commercial solver of mathematical programming problems based on classical computation, showing that the MemComputing approach was able to find much better solutions, regardless of size and structures, and, for some of these, MemCPU's convergence was extremely faster than Gurobi 8.0 (5 minutes vs. > 1 hour). The MemComputing paradigm turns out to be very efficient in solving NP problems due to the use of memory variables and the ability of the circuit to reorganize itself. This concept, however, differs from what has been covered so far in that one does not necessarily have a memory array in which computation takes place, but instead exploits the memory that certain devices possess to perform computations: quoting the article [69], "By memory, however, I don't mean storage, but rather time nonlocality - the ability of a physical system to remember its past dynamics in order to perform necessary tasks." The concept of BvNC covered in this thesis refers to computation devices composed of memory elements or arrays connected to computation logic to keep as much data as possible thereby reducing communication with the CPU.

1.1.2 BvNC application example: neural networks

The BvNC technique is used most often in the acceleration of neural networks (NN). A NN is a computational model capable of carrying out highly difficult assignments. It is made up of "neurons," which are the fundamental building elements; by arranging these "neurons" in an interconnected network, the NN can make decisions and potentially learn from its mistakes [73].

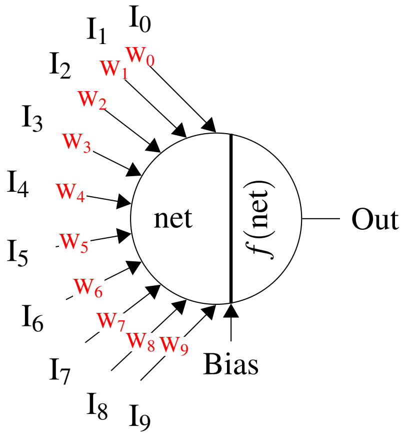


Fig. 1.3 Schematic of a neuron. Example with 9 inputs.

The diagram shown in Fig. 1.3 provides a good illustration of an artificial neuron. It is composed of two primary components, which are *net*, which is in charge of computing the weighted sum, and $f(\text{net})$, which is an activation function applied to the output. In most cases, the term *net* is expressed as follows:

$$net = \sum_{i=0}^{N} I_i \times W_i + \text{Bias} \tag{1.3}$$

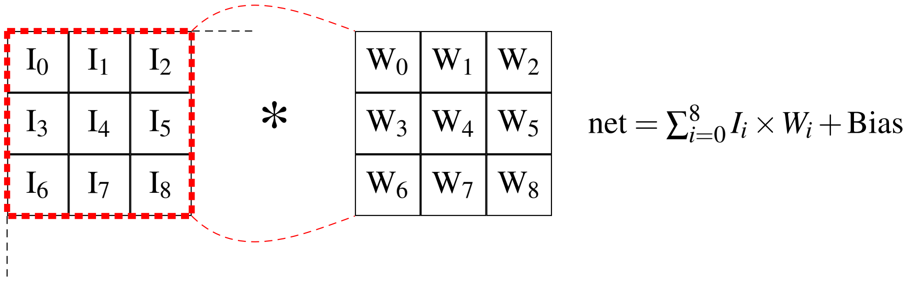


Fig. 1.4 Convolution computation example with a 3×3 kernel.

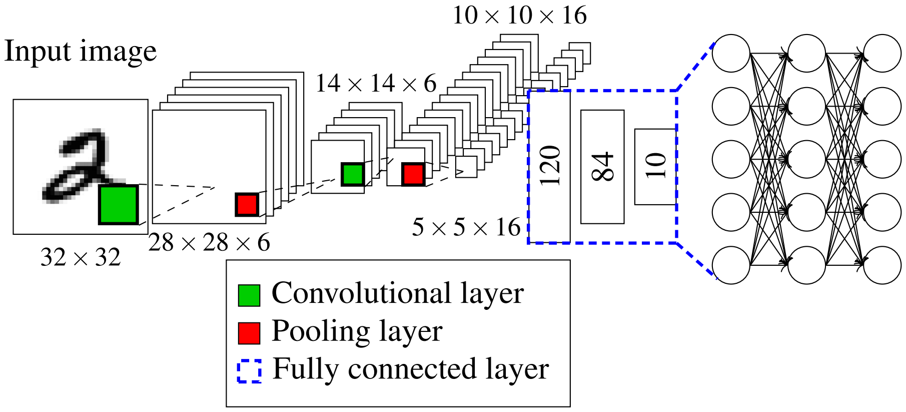


Fig. 1.5 Structure of LeNet 5 CNN [6], composed of 2 convolutional, 2 pooling and 3 fully connected layers and their sizes are indicated in the model.

Where I_i is the input value, W_i is the corresponding weight and Bias is an additive term. Weights and biases can be adjusted to achieve the desired output with a procedure called training. Usually, the activation function is non-linear. The most important activation functions are Rectified Linear Unit (ReLU), hyperbolic tangent (tanh), and the sigmoid function [74]. NNs are made up of layers composed of a set of arranged neurons and the most common structures that can be found in the literature are Convolutional Neural Networks [6, 75] (CNNs) and Multi-Layer Perceptrons (MLP). In CNNs, convolutional layers perform the convolution operation of the input feature map (IFMAP) with a set of weights called kernels. An example of a convolution computation is depicted in Fig. 1.4: the parameters to take into account in a convolution are the weights, the input feature map to be convolved, and the stride. After the first convolution, the kernel window is moved by a step equal to stride, and a new convolution can start. As it is possible to notice by this tiny example, the convolution computation perfectly matches the equation of the neuron reported in Equation 1.3, in which usually an activation function is applied to normalize the results obtained. A real example of a CNN is the LeNet 5 [6], in which all the convolutional layers have the same kernel size of 5×5 pixels. The first one produces 6 output feature maps (OFMAPs), meaning that the same IFMAP is convolved with 6 different kernels. The second convolutional layer instead produces 16 OFMAPs, starting from 6 IFMAPs: for each input, 16 kernels produce 16 outputs, so 16 from the first IFMAP, 16 for the second IFMAP and so on, implying a total number of OFMAPs equal to 6×16. These considerations bring to the general formulation of convolution in a convolutional layer, derived from [76, 13]:

$$y_o(j,i) = Bias_o + \sum_{c_{in}=0}^{\#C_{in}-1} \sum_{k=0}^{W_y-1} \sum_{p=0}^{W_x-1} W_{o,c_{in}}(k,p) \times I_{o,c_{in}}(j \times s + k, i \times s + p) \tag{1.4}$$

Where i, j are the indexes for the OFMAP corresponding pixel, c_{in} is the input channel index, $\#C_{in}$ the total number of input channels, W_x, W_y are the sizes of the kernel matrices indicating the number of rows and columns, respectively, o refers to the OFMAP considered, s is the stride and lastly p, k are the indexes of the kernel. Pooling layers have similar behavior to convolutional layers. In literature, different kinds of pooling layers are used, such as max or average pooling [77]. Similarly to convolution, they perform the maximum (or the average) of the selected input pixels and return only one value, performing the so-called subsampling operation. Pooling, specifically max pooling, is widely used to reduce the size of the CNN, preventing overfitting behaviors. In the example depicted in Fig. 1.5, the pooling kernel size is 2×2 for all the cases. Fully Connected (FC) layers are MLP subnetworks included in the CNN to perform the classification operation, usually made of neurons fully interconnected straightforwardly, as shown in Fig. 1.5.

NN are complex and computation-intensive models that can be power-hungry: implementing them on low-energy budget systems like embedded contexts can be challenging [78]. Therefore NN binary approximations are proposed in the literature, trying to reach good trade-offs between complexity and accuracy. Binary Neural Networks are particularly used in LiM-like architectures. Usually, Binary Neural Networks introduce normalization layers: one of the most used is the Batch Normalization (BatchNorm) [79] that is very useful to recover a portion of the accuracy lost from the binarization procedure [80]. Recalling its equation from [79]:

$$\tilde{X} = \frac{X - \mu}{\sqrt{\sigma^2 + \varepsilon}} \times \gamma + \beta \tag{1.5}$$

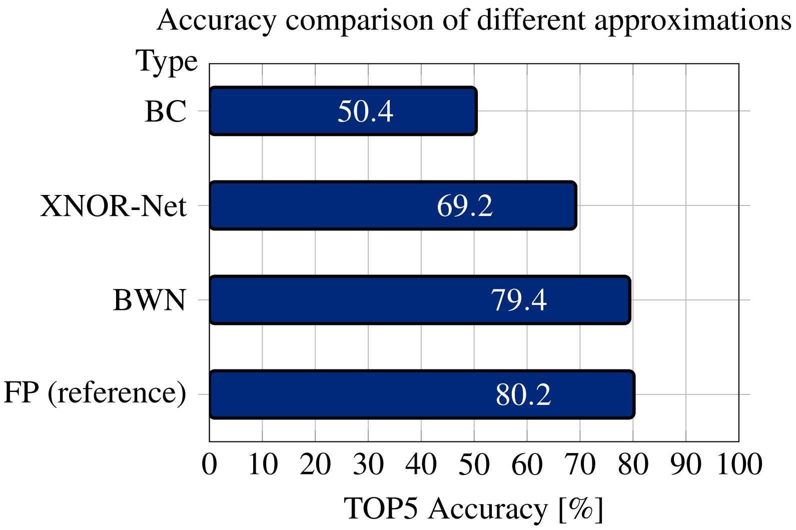


Fig. 1.6 TOP5 accuracy comparison between different binary approximations [7]

Where μ, σ are the batch mean and batch variance, while γ, β are correction variables. These four variables are trainable, meaning that during the training procedure, they are modified in order to increase accuracy. The value of ε is usually added to the variance to avoid 0 division if the variance is 0.

In [7], an interesting comparison between some possible Binary Neural Network approximations is proposed, also introducing XNOR-Net as a possible alternative: the values are recalled in Fig. 1.6. In the plot, TOP5 is intended as the accuracy classification rate to hit 1 out of 5 most probable classes. The Binary Neural Networks are compared with the original floating point implementation (FP) of AlexNet neural network [75], and accuracies are reported for each case.
All the weights are binarized in the approximation considered, meaning that $w \approx w_b \in \{-1, 1\}$ where w_b is the binarized weight value. Some binarization techniques are now briefly recalled from [7].

- Binary Weight Network binarizes only weights of the NN, keeping the activations and the inputs at full precision. By binarizing only weights, the convolution operation can be performed only with adds and subtractions, avoiding multiplication as reported in Equation 1.6.

$$Conv_{out,BWN} = X * w + Bias \approx \alpha(X * w_b) + Bias \tag{1.6}$$

An extra factor α is multiplied by the convolution result in order to compensate for precision losses:

$$\alpha = \frac{\sum_{i=0}^{N} \|w_i\|}{N} \tag{1.7}$$

Where w_i is the considered full precision weight and N is the number of weights. Binary Weight Network represents a very good alternative to reduce NN's complexity. However, it requires full precision inputs and activations.

- XNOR-Net binarizes both weights and inputs. The convolution result is obtained by performing the binary convolution, which is multiplied by a correction factor α (the same in Equation 1.7) and a matrix **K**. **K** is obtained as shown in Equation 1.8.

$$\mathbf{K} = \overbrace{\frac{\sum_{c_{in}=0}^{\#C_{in}-1} |X(:,:,c_{in})|}{\#C_{in}}}^{\text{First term}} * \overbrace{\begin{bmatrix} \frac{1}{W_x \times W_y} & \frac{1}{W_x \times W_y} & \cdots \\ \frac{1}{W_x \times W_y} & \frac{1}{W_x \times W_y} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}}^{\text{Second term}} \tag{1.8}$$

In Equation 1.8, the first term indicates the absolute punctual sum of the multiple IFMAPs divided by the number of input channels, and so the number of IFMAPs. The second term is a regular matrix of $W_x \times W_y$ size, which is made of the same term repeated in all the positions. Finally, the XNOR-Net convolution can be rewritten as:

$$Conv_{out,XNOR-Net} \approx (X_b \circledast w_b) \cdot \mathbf{K} \times \alpha \tag{1.9}$$

Where X_b is the binarized input, $\circledast$ is the binary convolution, $\cdot$ is puntual multiplication between matrices and $\times$ is the simple product. In [7], it is demonstrated that the binary convolution can be performed by considering the XNOR pop-counting of binary inputs/weights. XNOR truth table matches the multiplication if -1 is mapped to logic '0' and +1 is logic '1'. Pop-counting computes the difference between the number of 1s and 0s of the input sample.

- Binary Connect (BC) [81] binarizes both inputs and weights without applying any correction factor to the final convolutional equation, implying less recognition accuracy as shown in Fig. 1.6.

BvNC approach is especially applied in NNs' implementations. Some consider binary approximations by choosing an implementation based on emerging technologies. Some works like [82, 19] are based on MTJ technology while [83, 14] used RRAM. In each of these works, the resistive element is used to perform simple logical operations based on the current sensing technique. In [82, 19, 84, 85], several Binary Convolutional Neural Networks (BCNNs) implementations are discussed: they achieve very good results in terms of energy and power, thanks to the intrinsic low power nature of the MTJ and RRAM devices. In [83], a Binary Neural Network design based on SRAM array is proposed, where the logic parts in charge of doing the computations are disposed below the memory array. The memory parts allow to store the required parameters for the NN computation (like weights and biases), and the logic parts perform the results for the next layer of the NN, which are useful for addressing the following memory part, forming an alternation between memory-logic. This architecture achieves very good performance in terms of energy and speed thanks to its pipeline-like structure. In [14], the NN is mapped in a Wide-IO2 DRAM, using TSVs as high-speed communication link obtaining remarkable results in terms of execution time.

1.2 Electronic Design Automation applied to BvNC

All of the designs presented so far have been realized and simulated using tools like SPICE (for transistor-level circuit design and simulation), synthesizers (from the Register-Transfer Level functional model to the standard cell netlist description), Register-Transfer Level (RTL) simulators, and Place&Route software. This section delivers an overview of various tools, providing an understanding of the complexity and tools involved in the design process. An emphasis is also placed on state-of-the-art tools developed expressly for the BvNC approach.

1.2.1 Overview of the standard tools

Technological evolution has brought notable improvements in Computer-Aided Design (CAD) software, requiring more computational capabilities and advanced features. In Electronic Design Automation (EDA) field, for example, a wide selection of tools are provided, assisting the designer in the synthesis and functional verification of Application-Specific Integrated Circuits (ASICs), Field-Programmable Gate Arrays (FPGAs) or, differently, in the design of Printed Circuit Boards (PCBs) [86]. Regarding the ASIC world, the most advanced tools start from the implementation and simulation of transistor-level circuits using engines like SPICE, then going to the direction of the RTL description using languages like VHDL, Verilog, or System Verilog, functional verification by means of RTL simulation and, once everything works as expected, finishing with the synthesis and Place&Route phase of the final circuit, obtaining what is called the Graphical Data System II (GDS-II) description. This process is called Very Large Scale Integration (VLSI) flow. In VLSI flow, several very complex commercial and industrial tools are used. A possible list of tools in a typical VLSI process starts with Cadence Virtuoso and Spectre for transistor-level simulation and design, Synopsys Design Compiler/Cadence Genus for the RTL synthesis, Mentor QuestaSim/Synopsys VCS/Cadence Incisive or NC-Verilog for the functional simulation and Cadence Innovus for the Place&Route and GDS-II implementation of the circuit [87–89]. In the end of the Place&Route phase, a netlist made of standard cells is obtained, and it is functionally equivalent to the original RTL. Then, the netlist can be simulated again with the intrinsic delays of the standard cells and the back-annotated switching activity to check the circuit functionality and to estimate the performance precisely. In the literature, it is possible to find other EDA tools that belong to open source or academic category. Examples can be ABC [90], a software for logic synthesis and verification that makes use of And-Inverter Graphs (AIGs) logic transformations; Cacti by HP [91], that estimates the performance of modern memory systems; Verilator [92], a Verilog/SystemVerilog simulator similar to Synopsys VCS able to achieve overwhelming speed-ups in particular applications; OpenRAM [93], a memory compiler framework for the physical implementation of Random Access Memories (RAMs); Ngspice [94], a CLI-based SPICE simulator and many others.

1.2.2 Tools for BvNC

Unfortunately, performing the design of BvNC architectures is extremely complex because it requires, for every minimal change within the architecture or algorithm to be implemented, a revisit from scratch of the entire design, starting with the structure, the type of BvNC, the interface, and ending with the functional simulation, performance estimation, and circuit implementation. For this reason, in recent years, engineers have started to design very specific software or CADs, intending to assist the designers in emerging computing paradigms or technologies. For example, modern tools implementing Near-Memory solutions are CLAPPS [95], or HMC-Sim [96]. CLAPPS, for instance, is based on System C and creates, simulates, and generates synthesizable RTL of custom Hybrid Memory Cube solutions. Furthermore, it relies on Gem5 [97], a very accurate and robust simulator of computer-system architectures. Another example is Sim^{2}PIM [18] (Simple Simulator for PIM Devices), which integrates and simulates any Near-Memory architecture based on 3D stacked DRAMs with the host processor and memory hierarchy. In this tool, an algorithm written in C is compiled and simulated on the Near-Memory part. For In-Memory architectural solutions, several tools are proposed, such as Eva-CiM [8] that, given a program, the technology, the processor architecture, and the memory array characteristics, is able to provide performance results on SRAM or FeFET arrays. The tool considers data dependencies and relies on the instruction dependency graph (IDG) model that has been supplemented with memory access data. The analyzer is incorporated into Gem5 [97]. The architectural performance estimations rely on McPAT [98], on which authors incorporated the In-Memory module. Eva-CiM quantifies the system speedup and analyzes the energy savings of the In-Memory approach, demonstrating the reduction in memory accesses and lower computational demands on the host. Other remarkable tools are NVSim [99], which focuses on Non-Volatile In-Memory technologies like Spin Transfer Torque-Magnetic RAMs (STT-MRAM), PCMs, RRAMs and NAND-Flash; PIMSim [100] that integrates both Non-Volatile technologies and DRAM Hybrid Memory Cubes. Similarly to Cacti, DESTINY [101] is a modeling tool for 2D/3D memories created using SRAM, RRAM, STT-RAM, PCM, and embedded DRAM (eDRAM) and 2D memories developed using spin-orbit torque RAM (SOT-RAM), domain wall memory (DWM), and Flash memory. DESTINY enables modeling Multi-Level Cell (MLC) designs for NVMs in addition to Single Level Cell (SLC) designs for all of these memories. DESTINY is useful for performing design-space exploration across multiple dimensions, such as optimizing for a target (e.g., latency, area, or energy-delay product) for a given memory technology, selecting the appropriate memory technology or fabrication method (e.g., 2D vs. 3D) for a given optimization target. Another interesting work is the one developed in [102], which extends two other existing works, respectively Accelergy [103], which is an architectural performance estimator, and Timeloop [104], which is a specific infrastructure for exploring accelerators for deep neural networks. The tool focuses on In-Memory architectures implemented with memristor technology. The arrays estimated in [102] perform MAC operations. This tool's great advantage is allowing the designer to have performance estimations of each component belonging to the architecture (so ADCs, buffers, memory arrays, etc.) without having to go through the whole tedious design phase of BvNC architectures. Following a similar approach, MNEMOSENE [105] is another tool that takes care of compiling, mapping, generating LiM-compatible instructions, and estimating the performance of the chosen algorithm on the memory array. It has a compiler built in C++ that translates the algorithm into a new instruction set specifically for handling each component of the LiM design (peripheral circuits, arrays, etc.). The performance estimation is implemented in System C. This is only part of the tools proposed in the literature that focus on the LiM approach and, more generally, on BvNC. The goal of the tools mentioned above is to provide a simulative framework, in some cases also capable of estimating the performance of BvNC devices, with the intention of quantifying the impact of these innovative architectures. In the next part, some of the most significant results of the state-of-the-art BvNC tools and architectures are reported and discussed.

1.3 Promoting BvNC: results and discussions

From the analysis and considerations so far, the BvNC approach is very promising and increasingly used in the literature. However, what are the results obtained, and to what extent does BvNC bring benefits? Analyzing the effect of BvNC architectures on the von Neumann bottleneck is surely the best way to give validity to the concept. If BvNC architectures are able to reduce the effects of constant processor-memory communication, then the BvNC approach may be a good solution. This part reports some results and case studies from previous works in the literature.

In Eva-CiM [8], an estimation of the LiM impact in a processor-memory system is provided. The authors show how the inclusion of a LiM module based on SRAM-type implementation within a memory hierarchy has a positive impact on the total energy and speed-up of the system. LiMs used in Eva-CiM can perform bitwise operations, including AND, NOR, XOR, and addition. The study is carried out for several benchmarks, which are Naive Bayes (NB), Decision Tree (DT), Support Vector Machine (SVM), Linear Regression (LiR), K-means (KM), Longest Common

Subsequence (LCS), MPEG-2 decode (M2D), Breadth-First Search (BFS), Depth-First Search (DFS), Betweennes Centrality (BC), Shortest Path (SSSP), Connected Component (CCOMP), Page Rank (PRANK) and some algorithms from the SPEC 2006 benchmark-suite (Astar, h264ref, hmmer, Mcf). Results are reported in Fig. 1.7. In all cases, through LiM, the authors demonstrated energy and speed-up benefits for

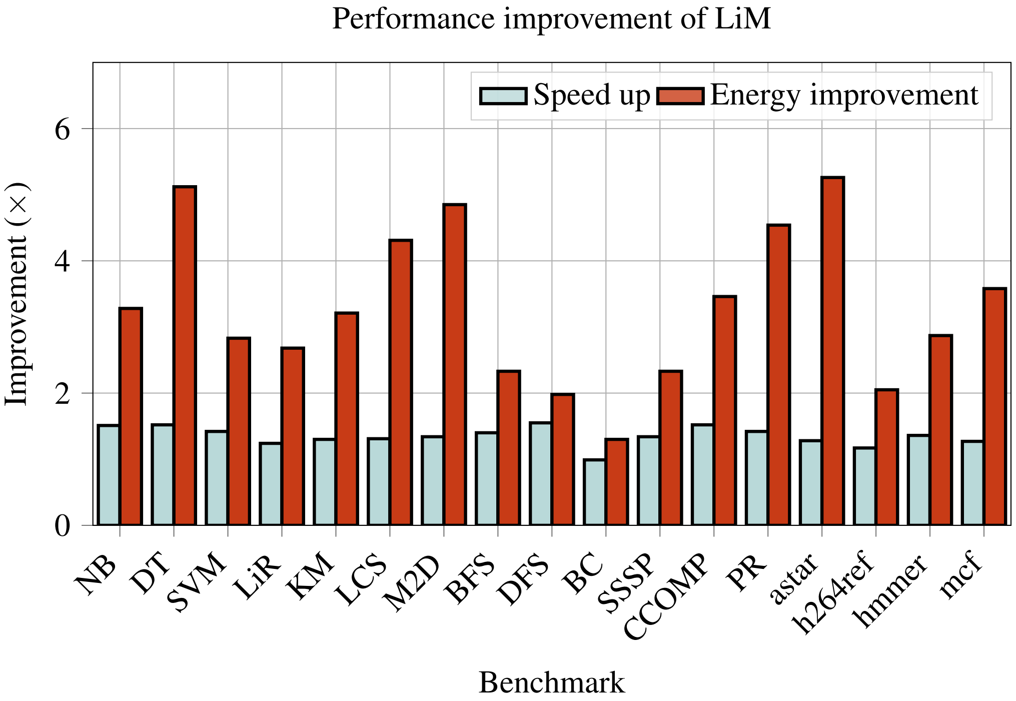


Fig. 1.7 Speed-up and energy improvements between non-LiM and LiM solutions. Data are taken from [8].

selected applications, promoting Eva-CiM and demonstrating the effectiveness of the LiM approach.

Sim^{2}PIM [18] implements different versions of BvNC architectures. The first is a DRAM memory that exploits the concept of shared charge to perform bitwise calculations on bitlines [106], called ComputeDRAM. The second is a memristor-implemented memory that accelerates a Convolutional Neural Network algorithm [107], called ISAAC. On the other hand, the last one implements a 3D stacked memory [108], called RVU. These architectures differ in structure and in the applications they can implement. The authors simulated the three systems with different benchmarks that best fit the chosen memory architecture. For the ComputeDRAM solution, the algorithm chosen is Bitmap Indexing (BMP), which consists of searching for elements having certain characteristics within a database through simple bitwise operations. The data are represented as strings of bits, where zeros indicate the lack of a feature, while ones vice versa. ISAAC, on the other hand, being set up for the computation of Convolutional Neural Networks, underwent an algorithm consisting of Matrix-Vector Multiplication (MVM). The calculation operation takes place on a matrix of size 4x4x64. Finally, RVU is employed to perform the operations involved in linear algebra equations (in this case, multiplication, addition, and subtraction), using vectors of size 8kB operating on 8MB of data. The authors demonstrated the effectiveness of the Sim^{2}PIM tool in accurately modeling different BvNC architectures, showing data such as execution time and utilization rates. In addition, with the results obtained with the tool, summarized in Table 1.1, another of the strengths of the LiM approach and, more generally, of BvNC is revealed. These architectures are particularly useful when employed as accelerators for specific data-intensive and parallel applications, reducing CPU effort and moving part of the computation core to the BvNC architecture.

Table 1.1 Results of execution cycles and usage percentage for BMP, MVM and Linear Algebra algorithms on the architectures used in Sim^{2}PIM. The numerical values are extracted from the graph in Figure 8 on page 9 of the Sim^{2}PIM article [18].

Architecture	**Algorithm**	**Cycles**		**% BvNC**	**% Host**
		Host	**BvNC**		
ComputeDRAM [106]	BMP	20	78	79.66	20.34
ISAAC [107]	MVM	560	188	25.17	74.83
RVU [108]	Linear Algebra	856	36	4.07	95.93

The work presented in [102] provides estimations of the performance impact of each component belonging to the memristor-based LiM design. These data are particularly interesting in getting an idea of each element's area occupancy and energy consumption, which should not be underestimated at the design stage. In the example proposed by the authors, 80 tiles having a size of 64x64 memristor are evaluated. The results proposed in [102] show that the impact of memristor arrays from the energy point of view accounts for only 17% of the total. The remaining 83% is caused by ADC/DACs, Near-Memory logic accumulation, and input buffers. There is a similar trend from an area perspective, where memristor arrays account for only 1.5% of the total area, and the vast majority is instead occupied by ADCs (73.5%).

MNEMOSENE [105] provides performance estimations on the linear algebra GEMM benchmark proposed in Polybench [109]. The memristor array is responsible for performing MAC operations. The study is carried out on several technologies, including PCM, RRAM and STT-RAM providing, for each of them, data on the energy contribution of the various components of the design (crossbar, drivers, ADC, Sample&Hold, digital Near-Memory units), execution time with different numbers of ADCs, and the latency contribution of each simulation phase for LiM and Near-Memory components (setup, read, execute and Near-Memory processing) at different clock frequencies. Interestingly, the authors demonstrate that, with higher frequencies, the impact of the latency of the analog computation part performed by the array (execute stage) increases because the latency of the analog circuits becomes a predominant contribution with respect to the others. Despite this, the execution phase has a significantly low duration compared to the other phases (max. 20%), demonstrating the LiM's effectiveness in performing calculations.

Considering now only architectural solutions, several approaches proposed in the literature demonstrate the strengths of BvNC. One of these is DIVA (Data IntensiVe Architecture) [110], which proposes an SRAM-based Near-Memory architecture in which the memory array is directly connected to the logical computation layer. The authors also suggest modifications needed to implement the memory architecture on a DRAM array in the discussion. Unlike 3D Stacked approaches, DIVA implements memory and computation units on the same plane, thus treating memory as an accelerator. The architecture consists of 32 memory-computation chips linked by an interconnected network that allows data to be moved from one array to another without involving the host processor. The interface with the DIVA memory follows the Synchronous DRAM (SDRAM) standard so that from the outside, this appears to be the same as a standard memory. Inside the DIVA unit, there is a translation layer between the SDRAM protocol, needed to drive the modules. Within a node of the DIVA architecture, apart from a scalar datapath and a pipelined control logic, there is a unit called WideWord that operates on words composed of very long bit strings (i.e., 256 bits) that allows it to do, in addition to classical arithmetic operations, a whole range of useful functions for data manipulation (e.g., permutation, data transfer, or selective execution). The DIVA architecture has been tested with several benchmarks, operating on large datasets or implementing image-processing algorithms. In order to simulate DIVA, the authors developed a simulator called DSIM that implements all the features of the DIVA architecture. The host processor is a MIPS 1000 with two-level caches. In the paper, the authors show a comparison in terms of speed-up between a host without a DIVA module and one with, obtaining values averaging 3.3 times on the considered benchmarks (which are template matching, corner turn, conjugate gradient, transitive closure, natural join, neighborhood, pointer random walk, and OO7), confirming again the strengths of the BvNC approach.

In RIME [111], the authors propose another memristor LiM architecture capable of implementing floating-point calculations. Computation is done starting from the bitwise logic implementable by RRAM devices, thus minority function, NAND and NOR. The concept of a memristor-implemented logic function has been employed in other work in the past, as explained in [112], such as IMPLY [113] and MAGIC [55], and consists of appropriately driving a group of memristors placed on the same memory row. Some of these are used as "input" memristors, while others are used as "output" memristors. The memristors have two terminals, one positive and one negative. When the voltage applied between the positive and negative is greater than the absolute value of the memristor's ON threshold voltage, its state changes from OFF to ON. Conversely, if a voltage is applied between the negative and positive terminals and it is greater than the OFF threshold. For the calculation of a NOR, for example, n-input memristors are connected in parallel: the positive terminals are connected to the positive terminal of another memristor used as output, while the negative terminals are connected to the ground. The negative terminal of the output memristor is connected to an input voltage. Suppose all parallel memristors are in a low resistance state (i.e., logic '1'). In that case, it means that the voltage between the negative and positive terminals of the output memristor is high enough to switch to a high resistance state. From the realization of basic gates, one can then move on to implementing full adders through the execution of multiple NOR and NAND operations, requiring 6 clock cycles with this scheme. One can implement architectures such as an integer or a floating point multiplier from the full adder building block. These architectures were simulated with SPICE, while the control logic part was synthesized with Synopsys Design Compiler and compared with a von Neumann-like architecture. The results of area (3523 μm^2 vs. 12975 μm^2), energy (7 pJ vs. 10326 pJ), and latency (2ns vs. 3793ns) for a single floating point multiplication are all favorable in the case of the von Neumann multiplier, highlighting the negative performance impact of serial execution of individual logic functions of the LiM case. These comparisons, however, are made without considering the contribution of memory accesses, which is very impactful from the standpoint of latency and energy for the von Neumann-like architecture. The authors also show that considering multiple floating point multiplications in parallel, the memristor implementation outperforms compared to a parallel von Neumann implementation.

The impact of the BvNC approach can also be evaluated against other standard architectural solutions, such as GPUs. In fact, GPUs are particularly affected by the von Neumann bottleneck, which is why the authors propose a 3D stacked DRAM Near-Memory accelerator specifically for image-processing applications, called iPIM [114]. To control iPIM, the authors have created an ad-hoc instruction set that enables operations on the memory array, and calculations on the logic die, including operations on vectors/scalars, 2D access patterns on memory (for image-processing applications), data movement, control flow, and synchronization. Within the logic die, there are also memory elements (such as scratchpad memories) to avoid structural (resource) hazards and share data among the various processing elements. The performance evaluation of iPIM was done with the architecture as a standalone object in mind. However, it can still be integrated with the processor using a standard bus such as AMBA or PCIe. For 3D RAM modeling, Cacti-3DD [115] was used, while the logic part was synthesized. iPIM achieved an average speed-up of 11× on the benchmarks analyzed in the paper, compared to an Nvidia Tesla V100 GPU, an advantage essentially due to the large memory bandwidth and memory-intensive nature of the proposed algorithms. The advantages of iPIM are not only related to execution time but also energy, with savings averaging up to 79.49%.

Belonging to the category of Near-Memory computing, the work presented by researchers in pPIM [116] is based on the use of LiM arrays used as Look-Up Tables combined with a DRAM memory array. The use of LUTs brings many advantages, including lower dynamic power consumption due to less switching of logic circuits, the possibility of employing approximate computing, and reconfiguring their functions. The proposed architecture is used to accelerate machine learning applications, including Convolutional Neural Networks and Deep Neural Networks, by integrating within the computation cores MAC units and units for computing activation functions. LUTs perform computations on 8-bit words and have 256 entries (or function words). This architecture is organized as a 2D memory composed of multiple sub-arrays: the logical computation elements explained earlier are inserted between one sub-array and another. Performance comparison is made with several standard and non-standard architectures, including an Intel Knights Landing CPU and an Nvidia Tesla P100 GPU and other works belonging to the BvNC category in the state-of-the-art. The power of the pPIM architecture is found to be 98.6% lower than that of the GPU and 98.4% lower than that of the CPU. The throughput, on the other hand, increases dramatically for pPIM, showing a gain of about 9 times and 33 times for the GPU and CPU, respectively.

1.4 Conclusions

From the considerations so far, based on prior work presented in the state-of-the-art, the BvNC approach is promising and effective in reducing the effects of the von Neumann bottleneck. From works reported in the literature, there is a general tendency to quantify the impact of BvNC architectures with certain figures of merit, including speed-up (and thus execution time) and overall system energy. These figures of merit are often compared with von Neumann-like architectures. In general, they are expected to have optimal values in the BvNC case because they are architectures capable of reducing memory accesses, thus avoiding data transport to the computational unit. Furthermore, one of the strong aspects of BvNC is the locality of computation; in addition to reducing communication with external logic units, data can be transmitted to the In-Memory/Near-Memory logic with extremely high bandwidth, thus reducing the execution time of an algorithm. With this in mind, this thesis aims to analyze the BvNC approach (and more specifically LiM), proposing case studies, the design flow adapted for the LiM structures investigated, and demonstrating the goodness of the results obtained and the validity of the LiM concept. As will become clear from the discussion of the proposed case studies, it is definitely complex to design a LiM architecture. For this reason, a tool called DExIMA, dedicated to architectural exploration in the LiM domain, was developed starting from these architectural models. DExIMA, unlike other tools proposed in the literature, gives the designer high flexibility, maintaining an architectural-level description of the LiM. DExIMA makes it possible to carry out the design of LiM cells and internal memory logic, realize the top-level architecture, simulate, estimate performance, and make direct comparisons with von Neumann architectures, allowing the designer to evaluate the effectiveness of the LiM studied for a given application.

Chapter 2

Previous works and architectural models

Summary

This chapter describes LiM architectures and case studies proposed in the past in our working group. It is of fundamental importance to define a design approach for LiM architectures, which, as will be reported later, can be distinguished into two categories: Application Specific and General Purpose. Furthermore, the proposed case studies highlight a recurring pattern in the architectural structure of LiMs for both General Purpose and Application Specific, thus defining the architectural model implemented within DExIMA.

2.1 General-Purpose and Application Specific

How can a LiM be designed? Two possible approaches can be followed: the Application Specific, so starting from an algorithm, the LiM is designed with the needed blocks and hardware to accelerate that particular application, or the General-Purpose, in which LiM is designed to accommodate more calculations as possible. In previous works developed during this doctoral program, both approaches have been followed, bringing to the realization of different architectural solutions.

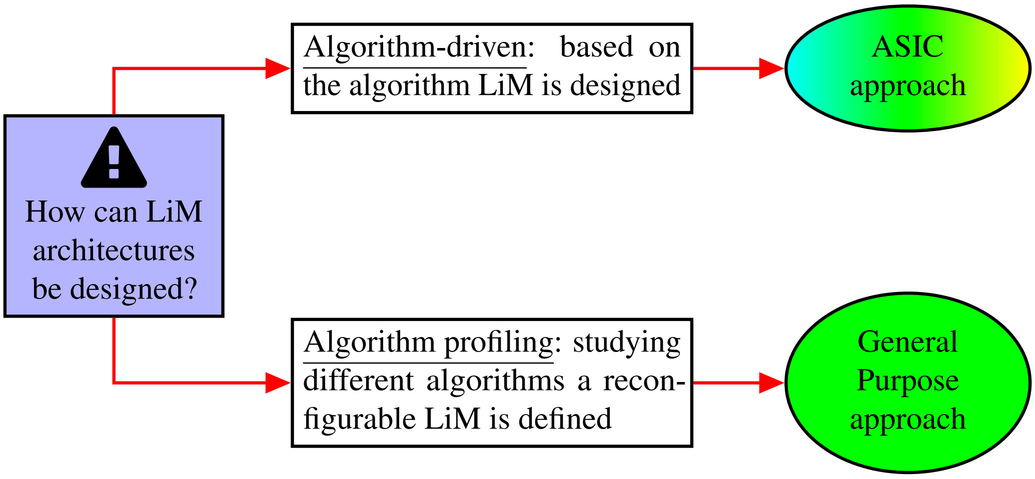

How can LiM architectures be designed?
Algorithm-driven: based on the algorithm LiM is designed
ASIC approach
Algorithm profiling: studying different algorithms a reconfigurable LiM is defined
General Purpose approach

Fig. 2.1 LiM design approaches.

2.1.1 Application Specific implementations

An overview of the Application Specific LiM architectures developed in this thesis work is shown in Fig. 2.2. Their aim is to accelerate a specific algorithm by parallelizing as much as possible calculations; specifically, the RISC-VLiM, which LiM Cell is depicted in Fig. 2.2 (a), accelerate simple bitwise operations, while the LiM solutions in Fig. 2.2 (b-c) implement two neural networks: a fixed point and binary solutions, respectively.

RISC-VLiM: a modified RISC-V data memory empowering in-situ calculations [9, 10]

RISC-V is an Instruction Set Architecture (ISA) that was initially created to help research and teaching in computer architecture. RISC-V ISA has grown in popularity over the years because it is an open standard suitable for hardware implementation in any technology (such as ASIC or FPGA) and is highly customizable. The RISC-V ISA is structured using standard and non-standard extensions. Each implementation must provide the basic integer ISA between the standard extensions, offering a limited number of instructions, which are adequate to give a decent target for compilers, assemblers, linkers, and operating systems, and are comprehensive enough to construct a software toolchain skeleton. Around the RISC-V integer basis, it is possible to construct more specialized processor ISAs [117]. The other additions to the fundamental ISA provide additional architectural features that en-

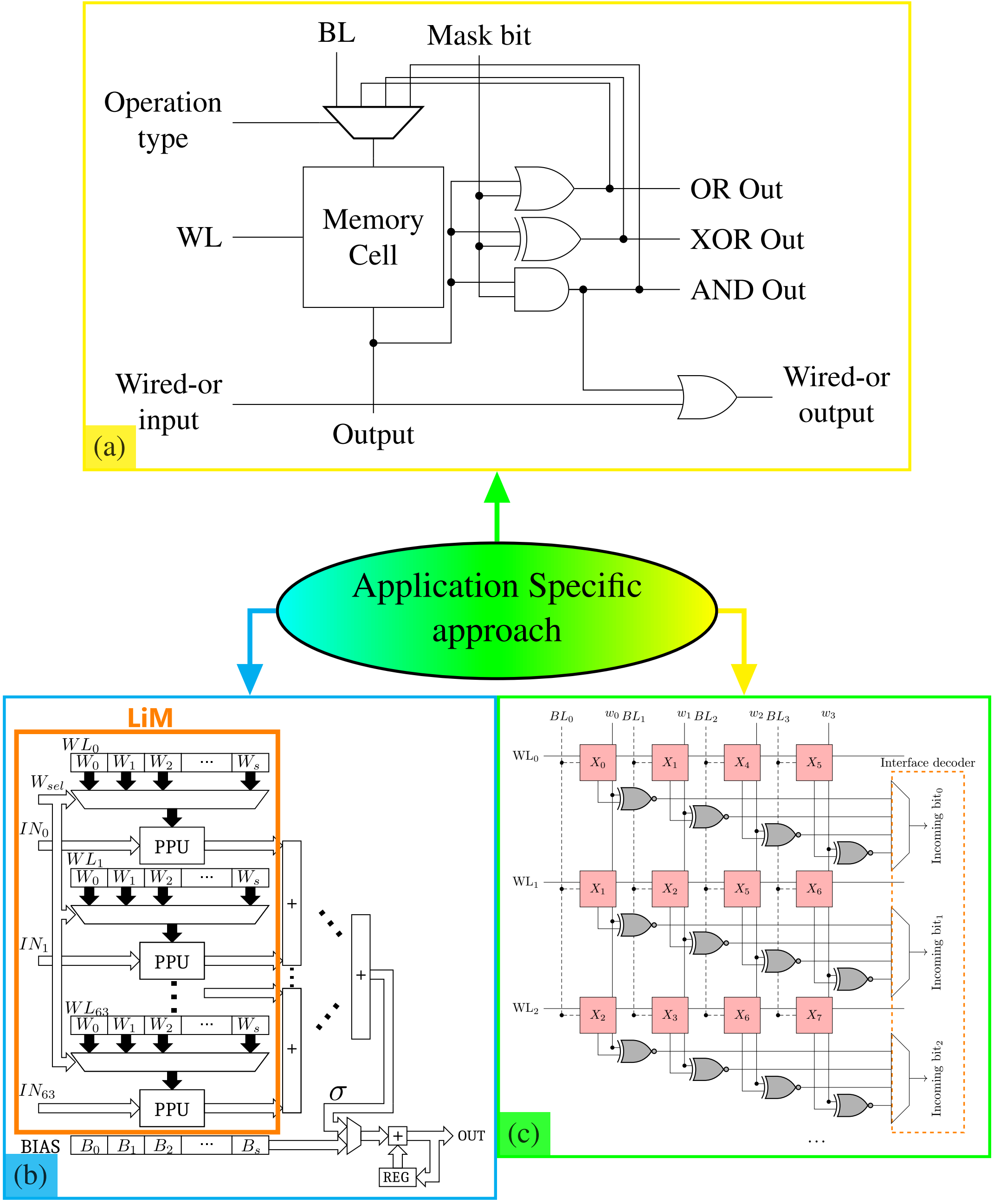


Fig. 2.2 Overview of the Application Specific implementations. (a) RISC-VLiM data memory. [9, 10] (b) Fixed point LiM implementation of a neural network. [11, 12] (c) XNOR-Net binary neural network implemented in LiM. [13]

hance code density and performance. This computer system was selected for this LiM investigation primarily because of its adaptability: the given memory model is replaced with a LiM model with additional logic inside to support new In-Memory operations. The framework relies on the existing interface between the processor and the memory with new instructions to control the new LiM, enlarging the decoder of the Instruction Fetch stage (IF) and defining a new immediate type for handling the different formats of the LiM instructions, consequently modifying the sign-extension block. The RISC-VLiM framework is openly available at the link https://github.com/vlsi-nanocomputing/risc-v-lim-architecture.

The RISC-V core has a single memory for both instructions and data, so the instruction and data sections share the same physical space with no specific separation. The compiler is responsible for managing the separation between them. As with the normal RISC-V memory model, the new one is a dual-port memory so that fetch and load-store operations may occur during the same clock cycle, and two decoders corresponding to the two ports provide access to all memory locations. In addition, the constructed LiM includes some logic around and inside the memory array. The new LiM is capable of performing simple load/store operations, bitwise operations (AND, OR, XOR) between a given range of memory locations and an input mask, and minimum/maximum computation of a vector stored inside the memory. To do this, the memory cell is enlarged with some extra logic operators, as shown in Fig. 2.2 (a): the additional OR port is needed for max and min computation, while the multiplexer is needed to feed back the operation result directly to the cell itself. The bitwise operations require only one clock cycle to be accomplished.

Minimum-Maximum computation LiM computes the maximum and minimum on unsigned values using a very simple procedure. In the maximum search, the algorithm begins by performing AND bitwise between the content of the selected rows and an external mask initially set to "10...0". Then, by comparing the values of the MSBs of the AND results, if there are words with MSB equal to 1, the corresponding words with MSB values of 0 are eliminated, meaning that these numbers do not represent the maximum. Otherwise, all the values are kept for the successive iteration. The procedure continues by setting the mask to "01...0" and checking the MSB-1-th bit and so on, requiring iterations as many times as the number of bits in words under consideration. Minimum computation works in the same way, but this time the exclusion is performed on words when the considered bit is set to 1 rather than 0. Extra logic blocks are required, both inside the LiM Cell and outside the memory array (Near-Memory architecture). In particular, the Near-Memory architecture produces a 32-bit mask with just one bit set in each clock cycle. The AND gate inside the LiM Cell receives the bit cell content and the input mask as inputs, and its output is fed to the LiM Cell's additional OR gate, which performs a 32-bit wired-or operation between the AND gates' outputs on the same row. Then, the results of the wired-or are analyzed: only words with a wired-or bit of '1' will be examined. If not, such rows will be omitted from the comparison. When calculating the minimum, the reverse approach is used. The information about the memory rows that must be kept for the next iteration is saved in the Enabled Words register and, at the end of the process, the rows that this register has kept correspond to the memory words with the maximum/minimum value.

To execute all words in parallel, the whole process needs N+1 clock cycles, where N is the number of bits. The extra cycle is necessary to initialize the Enabled Words data, and N cycles are required to evaluate N-bit words.

Interface Bus In the new design, the connection between the CPU and the new memory posed the greatest challenge. Unlike the typical memory-processor interface, the LiM needs information regarding the sort of operation to conduct differently from a basic memory. Therefore, a system that supports the new LiM operations while maintaining the regular RISC-V core's memory interface is selected. This is done to prioritize the core's adaptability and reusability on various platforms. By writing a word to a particular memory region, the LiM configures itself to execute a particular LiM operation: this allows us to support any kind of in-memory activity without modifying the RISC-V bus. To perform a LiM operation, the CPU will conduct a store to this memory location to program the memory, so any subsequent load or store will be interpreted based on the memory's preset behavior.

RISC-V compiler ISA expansion To implement these additional capabilities, the RISC-V ISA includes the following new instructions:

- **STORE_ACTIVATE_LOGIC**. This new instruction programs the LiM to function in a given mode by writing to a special memory address. The instruction format includes details about the operation type and the range size. The implemented operation types are NONE, AND, OR, XOR, MAX, and MIN.
- **LOAD_MASK**. The base integer ISA load instruction does not provide the option to transmit the input mask to memory. The primary purpose of the

LOAD_MASK instruction is to get the input mask from the Register File. The read value will be written to memory through the write data bus. This instruction must always be placed after activating the in-memory logic operations.

- **STORE**. If the memory is programmed as LiM, the value received from the Register File corresponds to the input mask in the case of a logic store. In contrast, if the LiM is used as a standard memory, this instruction is interpreted as a regular store.

To increase the flexibility of the RISC-VLiM system, similar to what is done in [118], the original RISC-V compiler was changed to include the new custom instructions. Using the RISC-V Opcodes (https://github.com/riscv/riscv-opcodes) tool in conjunction with the RISC-V-GNU-Toolchain (https://github.com/riscv-collab/riscv-gnu-toolchain), the additional instructions were added. To do this, the file opcodes-rv32i inside the riscv-opcodes repository is modified, including the new custom LiM instructions. For instance, the sw_active_or instruction is described as follows:

```
sw_active_or rd rs1 imm12 14..12=3 6..2=0x0E 1..0=3
```

Listing 1 Declaration of a new instruction inside the opcodes-rv32i file.

The fields of an instruction specify, from left to right, the instruction's name, the operands (rd: destination register; rs1: source register 1; imm12: 12-bits immediate), and the remaining bit values. The function field (bits 14 to 12) is 3, the opcode field is 0x0E (bits 6 to 2), and the LSBs are equal to 3 (bits 1 down to 0). Subsequently, by executing the `parse_opcodes` program, a header file including MASK, MATCH, and DECLARE_INSN directives for the new custom instruction is created, as shown in Listing 2.

```
#ifndef RISCV_ENCODING_H
#define RISCV_ENCODING_H
/*...*/
#define MATCH_SW_ACTIVE_OR 0x303b
#define MASK_SW_ACTIVE_OR  0x707f
/*...*/
DECLARE_INSN(sw_active_or, MATCH_SW_ACTIVE_OR, MASK_SW_ACTIVE_OR)
/*...*/
```

Listing 2 Generated RISCV_ENCODING for sw_active_or instruction.

After setting RISCV-GNU-TOOLCHAIN with the `-with-arch=rv32ima` and `-with-abi=ilp32` options, two files in the riscv-binutils directory are updated. The code created by the riscv-opcodes tool (Listing 2) is copied into `./include/opcodes/riscv-opc.h`. The other is `./opcodes/risc-opc.c`, where the code lines in Listing 3 are added, indicating the instruction's name, class, operands, match, and mask values.

```
/*...*/
const struct riscv_opcode riscv_opcodes[] =
{
/* name, xlen, isa, operands, match, mask, match_func, pinfo.  */
{"sw_active_or",0, INSN_CLASS_I, "d,s,j",MATCH_SW_ACTIVE_OR,
 ↪  MASK_SW_ACTIVE_OR, match_opcode, 0 },
/*...*/
```

Listing 3 Modified riscv-opc.c file, with sw_active_or custom instruction.

Inserting LiM operations in a C program Once the toolchain is complete, the user may declare the custom assembly instruction directly inside the C code using the `asm volatile` statement, as shown in Listing 4. The destination register (rd) of the custom LiM instruction specifies the total number of active memory lines, i.e., how many rows perform the LiM operation. In the source register 1 (rs1), the configuration address is supplied, i.e., the reserved address used to configure the memory function (which in this instance is an XOR), hence correctly establishing the selection value of the LiM Cell multiplexer (see Fig. 2.2 (a)). The immediate value specifies the offset to be added to the value of rs1. One can observe that there is a store operation between the two custom LiM instructions: after the memory is configured to execute a LiM operation from the corresponding memory location with a range specified by rd, the specified logic function is executed, and the result is saved in the corresponding memory lines. In the end, `sw_active_none` restores the standard memory function after the LiM section.

```c
//activate the xor operation on N_ACTIVE_LINES rows
asm volatile("sw_active_xor %[rd], %[rs1], 0"
    : [rd] "=r" (N_ACTIVE_LINES)
    : [rs1] "r" (cnfAddress), "[rd]" (N_ACTIVE_LINES)
    );
//store operation to run the xor in-memory
(*ofmap)[0][0] = bWeight;
//restore the normal function of the memory.
asm volatile("sw_active_none %[rd], %[rs1], 0"
    : [rd] "=r" (zero)
    : [rs1] "r" (cnfAddress), "[result]" (zero)
    );
```

Listing 4 Fragment of C code for the XNOR-Net algorithm.

Benchmarks Using the ideal model of the LiM, the initial findings are gathered to examine the performance gains that the new memory architecture would provide over a von Neumann system. The new ISA extensions can reduce the number of memory accesses during program execution, hence speeding up RISC-V's execution time. A series of programs are built and simulated to demonstrate the efficacy of the RISC-VLiM framework. The testing phase starts by programming a C code and compiling it using a RISC-V compiler: in this phase, no special LiM instructions are included in the C program. The output is the program.hex file containing machine instructions. Then, the RTL simulation of the RISC-V system starts, in which the LiM is used as a standard memory. After that, the C program is written again, this time incorporating the custom LiM instructions using `asm volatile` statements. The outcomes of both simulations are then compared, considering different algorithms having the following characteristics:

- *High data demand.* LiM is ideal for applications with a large data demand since it reduces data transportation to/from memory.
- *Data manipulation with supported LiM operations.* The implemented LiM is capable of performing a limited number of operations without a CPU. To take use of LiM capabilities, a selection of algorithms using these operations is required.

Based on these assumptions, the chosen algorithms are the following:

Table 2.1 Simulation results comparison

Algorithm	Memory (cc)	LiM (cc)	Speed-up (cc)	Speed-up (%)
bitwise.c	416	332	84	20.2
max_min.c	479	381	98	20.5
bitmap_search.c	453	454	-1	-0.2
aes128_arkey.c	554	529	25	4.5
transport_cost.c	1920	1698	222	11.6
xnor_net.c	464765	461316	3449	0.7

- *bitmap_search.c.* An index search technique for bitmaps. A bitmap index is a unique kind of data structure that utilizes bitmaps to expedite the processing of stored data.

- *aes128_arkey.c.* This method implements a portion of the Advanced Encryption Standard, the AddRoundKey step, executed eleven times over the whole encryption process. AddRoundKey consists of element-by-element XOR operations between the bytes of the states matrix and the key matrix. The outcome is the next state matrix utilized in the remaining encryption-related procedures.

- *transport_cost.c.* Transport problem is an algorithm that minimizes the cost of delivering a commodity from several origins or sources to multiple destinations.

- *xnor_net.c.* XNOR-Net is a Binary Neural Network model that reduces computation complexity by approximating weight and inputs to just two values (-1,+1) depending on their signs. -1 corresponds to logic ’0’ while +1 corresponds to logic ’1’. This approximation simplifies convolution into bitwise XNOR and pop-counting operations. In a bitstream, pop-counting is the difference between the number of ones and zeros. The proposed LiM enables concurrent XOR computation while the CPU does pop-counting. Due to the lengthy simulation period, just one convolutional layer with an input of 28x28 pixels (the size of the MNIST dataset) and a filter of 5x5 is implemented.

Table 2.1 compares the execution time of a certain algorithm for all the versions implemented, where cc stands for clock cycles.

In almost all the proposed programs, the RISC-V processor with LiM instructions exhibits an improvement, in some cases up to 20%. Due to the fact that the programs are optimized to take use of the new LiM instructions, the improvement is rather significant. In terms of execution time, the XNOR-Net scenario shows a decrease of 3449 clock cycles, despite a minor increase in speed bacause of its small dimensions. However, by increasing the size of the neural network by inserting several convolutional and fully connected layers, LiM significantly reduces the execution time because of parallel execution. In general, real-world applications must deal with enormous volumes of data, so the new LiM ISA additions would perform much better in this context. In Table 2.3, a comparison is made between the Memory and LiM cases about the number of memory accesses. As can be seen, memory operations decrease for almost all benchmarks, with the exception of the bitmap search. This benchmark was created with a small sample size, resulting in inefficiency for LiM. However, even with bitmap search, the trend is favorable because, for more sophisticated algorithms (such as max/min computation or XNOR-Net), the LiM paradigm reduces the influence of the Memory-Wall by leveraging parallel processing and decreasing CPU-Memory communication.

Evaluating the LiM impact Evaluating the impact of LiM architectures in a standard structure is fundamental. This can be done by assessing the performance of both the standard and the LiM solutions in terms of power, area, energy, and execution time. The traditional digital design flow for the standard memory case was used. The flow begins with the description of the architecture in HDL language, then moves on to the synthesis with Synopsys Design Compiler, and finally concludes with Place&Route with Cadence Innovus. For the LiM case, the design flow starts with the definition of the custom cell, implemented at the transistor-level, and then realized with a custom layout. The process starts with Cadence Virtuoso, where the transistor-level LiM cell is realized and simulated. After verifying the functionality, the cell layout view is built and simulated, including the parasitic effects. Next, the layout view undergoes Design Rule Check (DRC), Layout-Vs-Schematic (LVS), and Parasitics EXtraction (PEX). After PEX, a Spectre netlist is generated, which describes the LiM Cell with transistors and parasitic components. The Spectre netlist is then elaborated by Cadence Liberate, which generates a Liberty file with cell power, area, and timing of the cell. The Liberty file is compatible with Synopsys Design Compiler, which is used for performing the synthesis. The final step consists

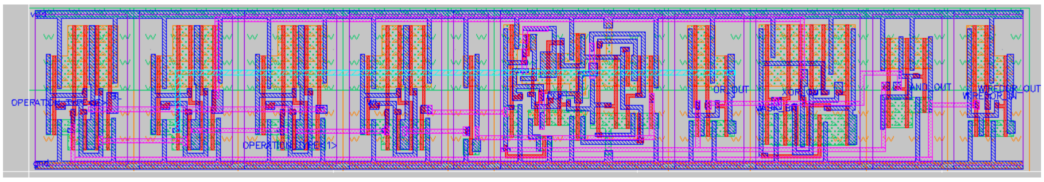


Fig. 2.3 Layout view of the LiM Cell

Table 2.2 Estimated Post Place&Route performance of the RISC-V core, standard (MEMORY), and LiM memory with a 4kB size.

Parameter	**MEMORY**	**LiM**	**RISC-V Core**
Power (mW)*	452.77	252.09	17.65
Area (μm^2)	432777.7	1610215.1	146709.6
Critical path (ns)	1.816	2.534	1.002

* Power levels are calculated using the worst-case scenario of maximal switching activity, i.e. no back annotation procedure and a clock period of 3ns.

in creating the Abstract view of the LiM cell and Library Exchange Format (LEF) file production with Cadence Abstract: with the Abstract view, it is possible to proceed to the Place&Route phase with Cadence Innovus. The LiM Cell layout is provided in Fig. 2.3. Standard and LiM flows are based on the 45nm CMOS technology of FreePDK. After the LiM process, the Liberty file is compiled using Library Compiler by Synopsys, and the newly created custom library is available for usage.

At this phase, the power, area, energy, and critical path of LiM and standard solutions are compared: the outcomes of a full memory array constructed of LiM Cells following synthesis and Place&Route processes are compared to those achieved in a normal memory scenario. Both memories contains 1024 rows with 32 bits of parallelism and employ a D flip-flop with enable (DFFEn) as a basic cell. Employing a D flip flop as a fundamental memory element is not ideal, but the goal is to understand the relative performance difference between conventional memory and a memory containing logic components. The results are shown in Table 2.2. Because of the greater number of logical components needed in the design, the area, and critical path are worst in the LiM case; however, power is reduced in the LiM scenario because the optimal arrangement of the LiM Cell (given in Fig. 2.3) keeps near parts close together. Even when the Place&Route in the conventional memory case is done without flattening the hierarchy, the LiM architecture is more efficient because the interconnections are shorter than in normal memory, resulting in power savings. However, it is important to underline that in this situation, power decreases, but this may not be the case with different LiM architectures. A comparison of dissipated energy is used to assess the effect of the LiM paradigm, in fact energy takes into account both the architecture's power consumption and the algorithm's execution time, providing a comprehensive assessment of system performance. Only the energy consumption of the memory is considered for the examined instances (MEMORY and LiM), which is calculated as follows:

$$\begin{aligned}\text{Energy} = \text{Power} \times \text{Number of memory operations} \times \\ \times \text{Clock period}\end{aligned} \tag{2.1}$$

The number of memory operations for each architecture can be found in Table 2.3, with the clock period set to 3ns.

Table 2.3 Number of memory operations comparison between Memory and LiM cases

Algorithm	Number of memory operations (LW+SW)		Reduction (%)
	MEMORY	LiM	
bitwise.c	114	89	21.9
max_min.c	126	85	32.5
bitmap_search.c	164	166	-1.2
aes128_arkey.c	144	130	9.7
transport_cost.c	336	286	14.9
xnor_net.c	65091	63942	1.8

Table 2.4 Memory energy comparisons using a clock period of 3ns.

Algorithm	MEMORY (nJ)	LiM (nJ)	Energy reduction (%)
bitwise.c	154.85	67.31	56.5
max_min.c	171.15	64.28	62.4
bitmap_search.c	222.76	125.54	43.6
aes128_arkey.c	195.60	98.32	49.7
transport_cost.c	456.39	216.29	52.6
xnor_net.c	88413.82	48357.42	45.3

As indicated in the Table 2.4, LiM decreases energy consumption by at least $\sim 43\%$ due to both the considerable influence of the custom layout on power consumption and the overall reduction of memory operations, which is shown in the Table 2.3. This comparison demonstrates how the LiM paradigm applied with CMOS technology increases system performance, especially for algorithms that can be accelerated by a LiM solution, showing a decrease in both energy and execution time.

A more complex case: fixed-point implementation of a neural network [11, 12]

WINNER (Weight In Memory Neural Network Embedded RAM) is the proposed name for a particular LiM implementation of a fixed-point neural network that is described in this study. The word "neuron" refers to a computational building component that carries out the task of performing the sum-of-products of the inputs with the weights that are associated with them. Along with simple multiplication and sum operations, activation functions are often applied to the neuron's output to model complex problems that require the introduction of non-linearities into the neural network. Some of the most widely used are the Rectified Linear Unit (ReLU), sigmoid and hyperbolic tangent. Because neurons are mostly made up of multiply-and-accumulate (MAC) PEs, the number of these PEs has to be maximized to reach acceptable Frame-per-Second (FPS) values, but also considering trade-offs with power and area. AlexNet [75] is the chosen neural network for this implementation, which structure is reported in Table 2.5. WINNER implements 384 neurons which can be used to perform the convolution operation with the AlexNet kernels. However, in AlexNet, the maximum kernel size is 9216 ($6 \times 6 \times 256$, the dimension of the first fully connected input), meaning that a single neuron should have at least 9216 fixed point inputs, which is impracticable. A trade-off between the number of inputs and parallelization must be considered. In order to decrease complexity, the number of contemporary inputs is set to 64, and for layers needing more than 64 contemporary inputs, the method is serialized, requiring $\#steps_{neuron} = \#inputs_{neuron}/64$ to be finished. The reference architecture is shown in Fig. 2.2 (b).

Architecture overview Fig. 2.2 (b) illustrates the portion of the architecture where the neural computation is conducted. It includes all 384 neurons, each having 64 input bytes and 384 output bytes. Data are represented on 8 bits using fixed-point notation. Neurons take many resources to be implemented; as a result, each neuron has 64 "WordLines" (WL_i in Fig. 2.2 (b)), where each "WordLine" is a 2595-byte register holding the values of the weights. Multiplying the size in bytes of these registers by 64 "WordLines" and 384 neurons yields the total number of stored weights inside the

Table 2.5 AlexNet structure with the necessary number of steps for the reference architecture consisting of 64 inputs per neuron.

Layer	Kernel size	$\#inputs_{neuron}$	$\#steps_{neuron}$	$\#steps_{layer}$	TOT_{steps}
CONV 1	11x11x3	363	6	3025	18150
CONV 2	5x5x48	1200	19	729	13851
CONV 3	3x3x256	2304	36	169	6084
CONV 4	3x3x192	1728	27	169	4563
CONV 5	3x3x192	1728	27	169	4563
FC 1	4096	9216	144	11	1584
FC 2	4096	4096	64	11	704
FC 3	1000	4096	64	3	192
TOT	-	-	-	-	52902

architecture. Multiplexers in Fig. 2.2 (b) are implemented as wired-or to decrease the size of the design, while Parallel Prefix Units (PPUs) calculate the multiplications using a radix-2 modified Booth encoding structure. Adders are used to assess the intermediate convolution result by adding two "WordLines" contributions. Each partial sum is added in the final adder tree to get the partial convolution result. Lastly, a multiplexer selects between σ and Bias, based on the evaluation step, to compute the final result by means of an accumulation adder. Other layers like pooling, cross-channel normalization, and zero-padding are implemented in a Near-Memory unit.

Results and comparisons In contrast to the previous case study, the findings of this one are examined in light of other state-of-the-art implementations. The performance estimation method begins with a circuit synthesis using Synopsys Design Compiler and a functional simulation using QuestaSim, which writes to a SAIF file all the information on the switching activity of each network node. This technique permits a more accurate calculation than assuming the worst-case maximum activity. The technology used is 45nm Nangate CMOS. The synthesis also provides the critical path delay, which is 3.55ns, so the maximum frequency is 281MHz. Memories are synthesized as registers in the design, therefore the power and area produced indicate an overestimation of a real-world scenario with a more accurate memory model. With performance results coming from the synthesis, a comparison with state-of-the-art AlexNet implementations is offered. WINNER can process a frame in a relatively small time (0.75ms), which is a good measure to determine the computation efficiency of the architecture, but alone is not enough.

In fact, to measure performance, a cost function Energy/FPS is considered: it represents the architecture's ability to reduce energy consumption while being as fast as possible. Thus it should be as small as possible. Table 2.6 highlights WINNER's performance compared to state-of-the-art AlexNet neural network implementations. Some implementations use a fixed point representation [119, 120, 19, 121], while others utilize a floating point representation [120]. Also investigated are binary implementations [19, 14, 122], and others are based on emerging technologies [19]. As shown by the data in Table 2.6, WINNER is the third-fastest implementation, surpassed only by binary implementations that sacrifice precision for speed. Fig. 2.4 depicts the relative energy efficiency of each architecture, where the function f on the y-axis is used to obtain a more clear graphical comparison by transforming the values of Energy/FPS into percentages relative to the XNOR-POP [14], which has the highest efficiency according to Table 2.6. WINNER has the fourth highest value: two binary implementations are better, but at the expense of accuracy, while the only non-binary design with a higher value is the Chain-NN [121], which is much slower. As can be observed, WINNER has around 70% of the relative energy efficiency of the XNOR-POP while having an accuracy of 84.7%. Overall, WINNER is excellent for any applications that need a fast rate of data capture while preserving a high level of energy efficiency. In more detail, the architectures considered have the following key features: the SOT MRAM [19] consists of a 1024×256 size memory array simulated with 45nm NCSU CMOS technology in combination with NVSim for nonvolatile memory modeling. XNOR-Pop [14] is a DRAM-based Near-Memory Wide-IO2 implementation with a memory size baseline of 1 GB. The memory part is simulated with Cacti, while the logic die is implemented in Cadence Virtuoso with 32nm PTM technology. YodaNN, Eyeriss, and Chain NN [122, 119, 121], on the other hand, are classic CNN accelerators, i.e., not implemented with a BvNC solution. Therefore, they are included in the comparison to assess the impact of standalone and application-specific architectures, unlike CPUs and GPUs. The YodaNN architecture consists of a shift register bank for filters (1024 7×7 filters), a 10.5 kB image memory, a 2.3 kB image bank, 32 multiply-and-accumulate units, and another 32 accumulation units. Synthesis and estimations are performed on UMC 65nm technology. Eyeriss, on the other hand, is composed of an off-chip DRAM and a core of $14 \times 12 = 168$ processing elements, implemented with TSMC 65nm technology. Inside each processing element, there are scratch-pad memories for the input image, the filter, and the partial accumulation, together with multiplexers, adders, I/O FIFOs, and optimized hardware like gating logic and pipelined multiplier. Lastly, Chain NN is synthesized with TSMC 28nm technology with 576 pipelined processing elements, each containing a multiply-and-accumulate unit and registers.

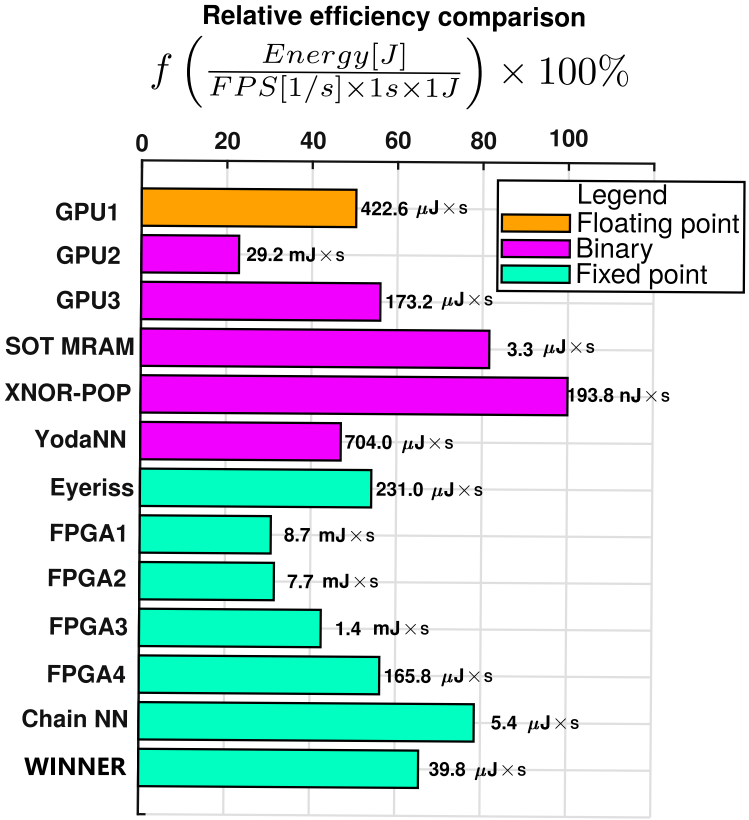


Fig. 2.4 Comparison of relative levels of efficiency. The numbers have been rescaled in the plot to be shown in percentage form. Above each bar is an indication of the value in absolute terms. In comparison to the XNOR-POP, for instance, the efficiency of the WINNER algorithm is 70% [14].

Table 2.6 State-of-the-art performance comparison with AlexNet model. Process time is rescaled to batch size equal to 1. The data of this table are partially taken from [19].

Architecture[a]	Process time (ms)	FPS (1/s)	Energy (mJ/frame)	Impl.[b]
Intel XEON E5-2637 [120]	195.00	5.0	25400.00	*
GPU1 [120]	1.30	769.0	325.00	*
GPU2 [19]	90.00	11.1	324.00	**
GPU3 [19]	0.73	1369.9	237.25	**
SOT MRAM [19]	10.70	93.5	0.31	**
XNOR-POP [14]	0.29	3390.0	0.66	**
YodaNN [122]	2000.00	0.5	0.35	**
Eyeriss [119]	28.825	34.7	8.01	***
FPGA1 [120]	21.61	46.3	402.00	***
FPGA2 [120]	20.10	50.0	384.00	***
FPGA3 [120]	2.56	391.0	77.00	***
FPGA4 [19]	5.94	168.4	27.92	***
Chain NN [121]	3.07	325.4	1.74	***
Our Work	0.75	1326.5	52.77	***

[a] GPU1 = GTX Titan X, GPU2 = NVIDIA Jetson TK1, GPU3 = NVIDIA Tesla K40, FPGA1 = Virtex-7 VX485T, FPGA2 = Stratix-V GSD8, FPGA3 = Virtex-7 VC709, FPGA4 = Xilinx Zynq-7000

[b] Impl. implementation type. *, ** and *** indicate floating point, binary approximation and fixed point architectures, respectively.

A Binary Neural Network LiM case study [13]

This study uses a binarized NN. Binary Neural Networks (BNNs) approximations like BinaryConnect (BC)[81], Binary-Weight Network (BWN), and XNOR-Net [7] reduce computational complexity by binarizing weights-inputs precision, as already discussed. Since a single XNOR gate can perform binary multiplication, the XNOR-Net approximation is best for LiM solutions and has a high accuracy rate compared to the floating-point model. In order to investigate the features of a LiM implementation in greater depth, two architectures were designed: an Out-Of-Memory (OOM) architecture that adheres to a standard approach (i.e., a computational core receiving data from external memory) and a derived LiM novel alternative, with the intention of comparing the levels of performance achieved by each of these configurations. By using the XNOR-Net as a case study, it is possible to deduce that the most important component of the BNN is the computation of the XNOR products paired with pop-counting to identify the outcome of the binary convolution. According to Equation 1.9, the OOM and LiM architectures' cores are composed of XNOR gates and pop-counters. Based on the kernel size of the considered network, the total number of XNORs must be equal to kernel size, as each performs a multiplication. A worst-case analysis must be done to correctly design a neural network circuit since hardware versatility relies on kernel size. In the LiM design, each LiM Cell contains an XNOR gate with binary inputs/weights sent straight to the XNOR inputs (Fig. 2.2 (c)). Each row in LiM comprises all the input items necessary for a single convolutional window calculation, assuming a bitwidth dimension of $W_x \times W_y$ bits, where W_x and W_y are the X-Y sizes of the kernels, respectively. The number of rows must be at least equal to the needed total number of convolutional windows D_{out}, which is also the dimension of the OFMAP. D_{out} may be determined when the kernel, IFMAP size (D_in), and stride are considered.

$$D_{out} = \frac{D_{in} - W_x}{stride} + 1 \tag{2.2}$$

Regarding the pop-counting calculation, processing a large number of inputs in parallel can be rather complex in hardware. Therefore, the outputs of the XNOR gates are multiplexed, and only one XNOR result is processed every clock cycle, as shown in Fig. 2.2 (c). Since most CNNs may accept multiple IFMAPs in input, each convolutional window must be independently calculated and afterward added to provide the OFMAP. This may be achieved by instantiating multiple XNOR-Popcounting architectures and, specifically, adding the contributions together with a Near-Memory processing unit. The design of a neural network must account for some scheduling when a large number of neurons must be processed since each neuron is associated with a memory column with additional logical gates inside the cell. A high number of neurons occurs in fully connected layers. For instance, in

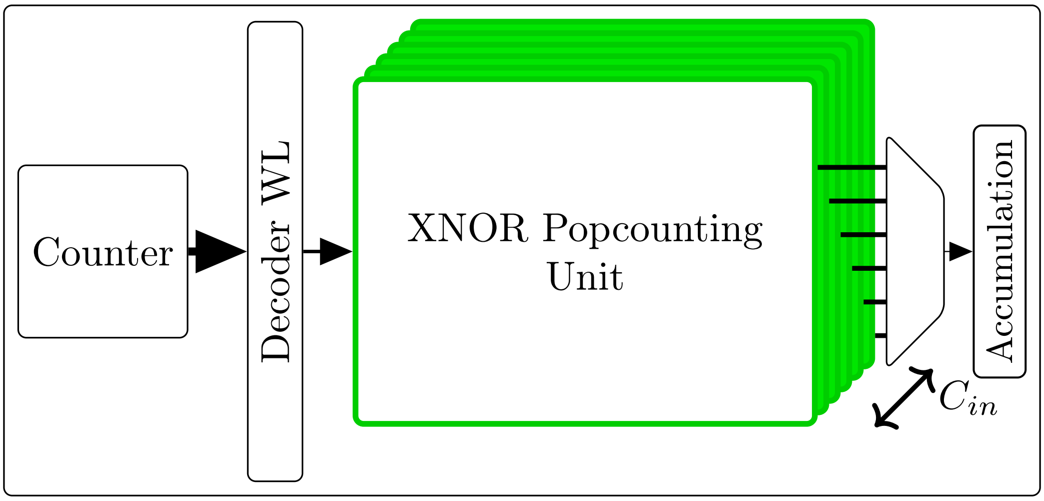


Fig. 2.5 Multiple input channels neural network design.

LeNet5 [6], the first fully connected layer contains 120 output neurons, requiring 120 columns, hence it is possible to process all neurons simultaneously. However, for more complex algorithms such as AlexNet [75], which has a maximum number of output neurons equal to 4096, having 4096 columns is unacceptable. Therefore, to implement layers with a large number of neurons, the algorithm is serialized, allowing reuse of existing undersized hardware.

Two LiM arrays capable of conducting XNOR bitwise and pop-counting operations are created. The structure of the XNOR LiM array is shown in Fig. 2.2 (c), which depics a basic 2×2 convolution example. Four pixels of IFMAP stored in the LiM must be convolved with the kernel provided to the XNORs' inputs, applying the bitwise multiplication between the binary input (X) and the corresponding weight (w):

$$\text{Incoming bit}_0 = \text{pop-count}(\overline{X_0 \oplus w_0}, \overline{X_1 \oplus w_1}, \overline{X_4 \oplus w_3}, \overline{X_5 \oplus w_3}) \qquad (2.3)$$

Regarding pop-counting technique, it is possible to simplify the pop-count equation as follows to lower the complexity of the LiM Cell:

$$\text{pop-count} = \#1s - \#0s = 2 \times \#1s - \text{length(word)} \qquad (2.4)$$

Where length(word) represents the size of the bitstring entering the pop-counter. A OneCounter is simply composed of half adders (HA), thus there will be one HA for each memory cell in the pop-counting section, as shown in Fig. 2.6.

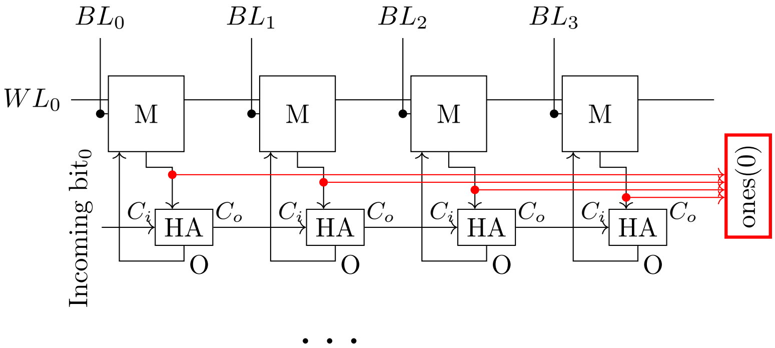


Fig. 2.6 Ones counter integrated in-memory.

Comparison OOM-LiM Two NN models were selected as case studies for both OOM and LiM implementations. These models were built, trained, and verified using the Keras framework [123] and a Matlab script. Then, both designs were synthesized using Synopsys Design Compiler and Place&Routed with Cadence Innovus, using 45nm CMOS technology, yielding results for power, area, Critical Path Delay (CPD), execution time, and energy consumption. In Table 2.7, energy and delay of LiM and OOM solutions are evaluated. For both NN implementations, LiM uses less energy and is quicker, although having a larger power value. Hence from an energy perspective, LiM design is more efficient for that specific implementation. Then, the discrepancies between our LiM model, based on flip-flops memory elements, and a more realistic LiM model were next analyzed. In [20], a CAM-based XNOR-Pop implementation is presented, which is a very comparable XNOR-Net implementation. Since the authors constructed a customized memory array using 65nm CMOS technology, the LiM architecture proposed was re-synthesized using 65nm CMOS technology @ 1.0V, attempting to apply the same metrics as [20] in order to determine how a true and accurate memory model differ from the aquired findings. Authors implemented the second convolutional layer of the LeNet5 NN model. To do this, they used five arrays of 30 × 10 size. To have a fair comparison with [20], the same contraints were applied to our LiM design: just the XNOR-Popcounting component, was synthesized with a dimension of 30 × 10 for the XNOR part. To provide the energy estimation, the second convolutional layer of the LeNet5 CNN is mapped, yielding the execution time which is multiplied by the worst-case power provided by Synopsys. In Table 2.8, the energy ratio relative to the CAM reference is around 4.22, while the ratio for the Bank Area is almost 4.50. Implementing a LiM solution at the transistor-level that is highly optimized and customized for a specific purpose improves performance in comparison to a flip-flop-based solution. However, creating

Table 2.7 Post place&route estimation for two neural network models.

Type	Area (mm^2)	Power (mW)	CPD (ns)	Execution time (ms)	Energy (µJ)
LiM - NN1	1.70	328.3	4.11	0.21	68.9
OOM - NN1	1.07	142.3	4.14	0.92	130.91
LiM - NN2	0.1033	13.06	4.22	0.132	1.72
OOM - NN2	0.086	10.68	4.32	1.62	17.30

Table 2.8 CAM-based XNOR-Pop [20] and our LiM architectures performance parameters comparison.

Design	Technology	Bank size	# of Banks	Bank Area (µm^2)	Energy (nJ)
[20]	65 nm	30 × 10	5	2456.6	~ 9
LiM	65 nm	30 × 10	5	12090.6	~ 38

a LiM from the realization of the transistor-level architecture involves much work and simulations at the SPICE level. As a conclusion of this study, LiM and OOM designs were created in order to compare them and highlight the primary benefits of an In-Memory implementation. Due to a greater degree of parallelization of the algorithm, the LiM architecture achieves exceptional results in terms of energy dissipation. Neural networks, and more specifically BNNs, are excellent applications for LiM benchmarking due to the fact that their computational requirements are well-suited for an In-Memory implementation.

2.1.2 General Purpose approach

Algorithm Profiling

To create a LiM architecture geared toward implementing General Purpose applications, an examination of numerous different algorithms is being carried out to find out which kind of basic operations are most frequently used. A look was given to the SPLASH-2 benchmark suite [124], which consists of a collection of complicated and parallel algorithms normally used to assess the performance of CPU-centric systems. In this investigation, each algorithm was profiled by evaluating the kind and frequency of instructions needed. The objective was to create an architectural model consisting of elements with both logic and memory capabilities so that the most frequently used operations could be executed and the results written directly inside the memory, thereby decreasing the number of memory accesses and increasing execution efficiency. Following are the steps used to profile each algorithm:

1. *Definition of the Instruction Set Architecture (ISA).* Since a RISC-V system is being investigated, the benchmarks for a RISC-V Instruction Set Architecture need to be cross-compiled. LiM architectures do not enable floating-point computations, so the riscv-gnu-toolchain from [125] was set with the base integer, multiplication/division, and atomic extensions but not with the floating-point one.

2. *Execute the benchmarks and monitor the algorithm's execution.* The Gem5 Simulator was used in system-call emulation mode for these specific applications. With the instructions tracing functionality enabled, Gem5 executes SPLASH-2 benchmarks. In this manner, the simulator outputs a deconstructed

version of each algorithm, detailing the exact instructions performed by the core. These data are stored in a file titled `program.out`.

3. *Estimate the instructions occurrences.* A Python script analyzed the `program.out` file and counted the number of instructions for each algorithm. In Table 2.9, all of the instruction counts contributions of each benchmark in percentage are shown. For example, the store double (**sd**) instruction's value was calculated as the sum of the number of **sd** instructions for each benchmark (or test) divided by the total number of instructions for each benchmark (which is $\sim$ 64M).

Table 2.9 The frequency of occurrence of each instruction for SPLASH-2 benchmarks. **barnes**, **fmm**, **ocean-contiguous-partitions**, **ocean-non-contiguous-partitions**, **radiosity**, **water-nsquared**, and **water-spatial** are the algorithms examined.

Instructions	**Instruction occurence (%)**
sub	0.8
blt	0.9
srl	1.2
subw	1.3
jalr	1.4
mul	1.7
bge	1.9
add	2.7
bne	2.7
and	3.0
jal	3.2
or	4.5
ld	7.0
beq	7.0
sd	7.3
andi	7.6
slli	8.7
srli	12.0
addi	21.0

According to the findings in Table 2.9, the most often used instructions are arithmetic (**addi**, **add**, **mul**, **subw**, and **sub**, etc.) and logical (**srli**, **slli**, **andi**, **or**, etc.). Based on these data, a plausible subset of hardware blocks can be included in a LiM architectural model. As a result, a considerable portion of the CPU operations may be transferred directly within the LiM co-processor. Furthermore, Table 2.9

demonstrates a considerable contribution of memory operations (**sd**, **ld**), indicating von Neumann Bottleneck's severe influence on conventional systems.

Hybrid-SIMD: a LiM architecture enabling Single Instruction Multiple Data operations [126, 127]

The LiM array shown in Fig. 2.7, is the central component of the Hybrid-SIMD architecture. It calculates and saves data, minimizing data flow between the CPU and memory. The LiM array is composed of LiM Cells (shown in Fig. 2.7), Intra Row Logic (IRL) blocks, and a memory interface. The structure is hierarchically organized: multiple LiM Cells create memory rows, which, when paired with IRLs and I/O buffers, form a smart row. Multiple smart and standard rows are stacked to build the LiM array. A standard row memorizes data without alteration, hence it can be categorized as a traditional memory. Standard rows play a crucial part in LiM computing by storing the local data utilized by smart rows. As seen in Fig. 2.7, the LiM array is constructed with interleaved smart and standard rows: this arrangement facilitates communication between two adjacent smart rows, which share a standard row in between.

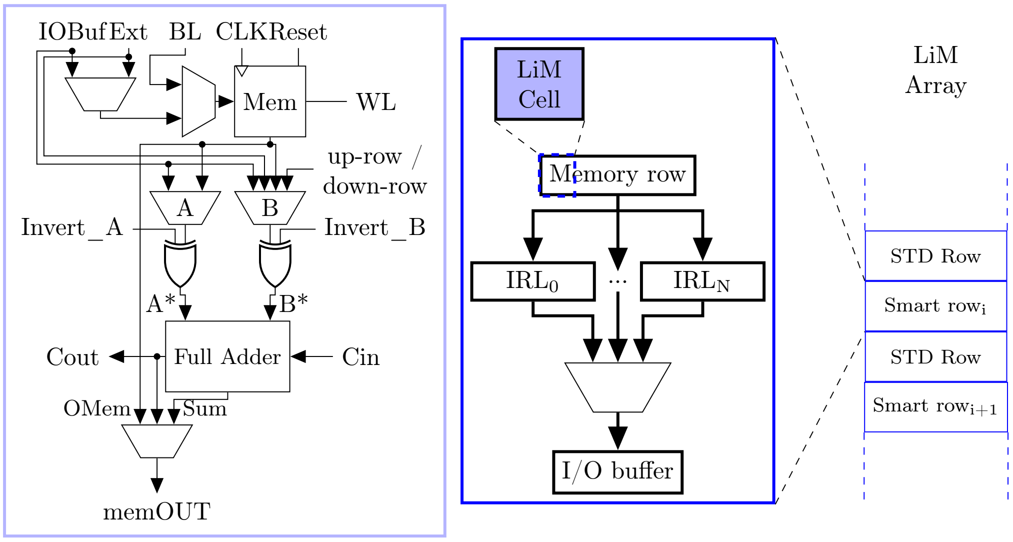


Fig. 2.7 Hybrid-SIMD LiM array and Cell structures.

LiM Cells, IRLs and the memory interface LiM Cells are the array's finer-grained building elements. The structure is shown in the figure Fig. 2.7: the storage

Table 2.10 Available operations performed by the LiM Cell's full adder.

Operations	Cin value	Output pin
A* OR B*	1	Cout
A* AND B*	0	Cout
$A^* \oplus B^*$	0	Sum
$\overline{A^* \oplus B^*}$	1	Sum
A* + B*	Cout(i-1)	Sum/Cout

element (shown in the diagram as Mem) is a flip-flop that accepts three different input values: bitline (BL) that comes from the CPU, Input/Output Buffer (IOBuf) and external (Ext) that come from the array itself. IOBuf is linked to the Input/Output buffer of the same smart row, which holds temporary results. Ext is coupled to the memory interface, enabling retrieval of data from any memory location of the array. Each LiM Cell has two XOR gates, a full adder, and several multiplexers, allowing to perform the operations listed in Table 2.10. The output pin defines which of the full adder's outputs between Cout and Sum corresponds to the target operation. Other logical and arithmetic functions (NAND, NOR, subtraction, etc.) can be achieved with relative ease using Invert_A and Invert_B. The arithmetic memory row, seen in Fig. 2.7, is created by instantiating Nbit cells, where Nbit is the data parallelism defined during the design process. The IRL conducts computations that cannot be performed by an arithmetic memory row (different operations from arithmetic addition/subtraction and logical functions). The IRLs are selected based on the algorithm's needed operations but also considering the Algorithm Profiling methodology explained in section 2.1.2. One may see that IRLs, I/O buffers, and memory rows must be built to be interconnected; hence, a common interface becomes a requirement in design. To improve the connections between LiM array blocks, a memory interface is used to manage the data flow between smart and standard sections. Fundamentally, the ability to access the whole memory address space is required to get data from any potential LiM array position and associate it with the Ext operand.

Control logic Control logic is an explicitly addressed microprogrammed machine comprising a micro-Control Unit (uCU) and a nano-Control Unit (nCU). The uCU receives high-level assembler-like instructions for each procedural step, which are then converted into low-level signals by the nCU to manage the LiM array. This arrangement simplifies the implementation and debugging of algorithms on Hybrid-SIMD by limiting design complexity to selecting architectural metrics (data parallelism, LiM array size, etc.) and writing high-level instructions. In particular, the micro-programmed machine relies on a micro-ROM (uROM), in which the instructions are written, together with the address of the next instruction: more details about this structure are given in subsection 3.1.1. An instruction is composed of different fields: an operational code (identifying the operation to perform), the source operands, the destination where results are written, an address (when the source operand comes from a particular location in the array), and a function (logical or arithmetical). To lower the design's dynamic power consumption, the LiM array was separated into blocks that may be turned off when they are not doing computations. Each block has N smart rows, where N is determined by the user during the design process. If, for example, N equals 1, it denotes the finest design granularity level since a single row can complete the calculation.

Implemented algorithms & Hybrid-SIMD derived structure Different applications are used as benchmarks to evaluate Hybrid-SIMD capabilities. Starting with the application, the smart row structure is created, and the algorithm is manually derived and written inside the uROM.

- *Discrete Fourier Transform.* The algorithm consists of performing the following computation:

$$X_k = \sum_{i=0}^{N-1} x_i \times \left[cos\left(\frac{2\pi ik}{N}\right) - jsin\left(\frac{2\pi ik}{N}\right)\right] \tag{2.5}$$

  Where N is the overall sample count, and k represents the element being evaluated. The structure of the smart row has a Look-Up Table (LUT) to store cosine-sine values, a multiplier, and an arithmetic memory row to calculate the total. Regarding the division, including the divider in the smart row structure reduces performance since it is a fairly complicated computing element. For this reason, N is chosen so that $N = 2p : \forall p \in \mathbb{N}$, to implement the division as a simple right shifting. However, this poses a limiting operational restriction for the Hybrid-SIMD. The procedure begins by calculating all $\left(\frac{2\pi ik}{N}\right)$ terms and storing the results in output buffers. There are two multiplications and a right shift carried out. The first multiplication is $2\pi \times i$. These operands are correspondingly stored in ADDRESS_2PI and the arithmetic memory rows.

The second multiplication includes the previous results (stored in the output buffers) and the term k (stored in the LiM array's ADDRESS_K). In this instance as well, the results of the calculation are kept in the output buffers, which are then utilized by the arithmetic right shifter block to calculate the divisions. The LiM array smart section is divided in half to yield the real and imaginary components. The first section is allocated for the real terms of the DFT and computes the cosine of $\left(\frac{2\pi Ik}{N}\right)$, while the second one computes the sine of the same value for the imaginary part. The splitting action may be managed by using the uCU's Block signal. At this time, the output buffers of the first half include cosine terms, whereas the output buffers of the second half contain sine terms. In real and imaginary cases, the samples x_i are multiplied, with the results stored in the up-rows. The contents of the output buffers, which hold the multiplication results, are now placed into the arithmetic memory rows.

Hybrid-SIMD previously conducted the procedures necessary to compute each contribution to the total in Equation 2.5. To determine the final X_k outcome, it is necessary to sum all contributions. To do this, the up-row/down-row mechanism is utilized: the down-row of the i-th smart row becomes the up-row of the i+1-th smart row, allowing data flow between two neighboring smart rows. However, the sum computed via the up-row/down-row technique loses the needed data of the up-rows x_i for further iterations. To prevent data loss, a load operation is done, saving the contents of the up-rows in the input buffers of the I/O buffer IRL. Then, a store operation is executed so that the contents of the output buffers are moved into the down-rows. The first sum iteration is computed: the up-row/down-row method adds rows and up-rows. As a result, the output buffers contain the sum between the i-th and (i-1)-th element. According to Equation 2.5, by simply iterating N-1 times this routine, the final sum of Equation 2.5 can be achieved.

- *K-Means.* K-means is the second proposed benchmark [128]. K-means seeks to identify Nc specified groupings. The distances between each dataset point and Nc centroids are calculated, and the cluster member with the shortest distance is determined. The following is the pseudo-code of the algorithm:

```
d(:,0) = MAX_N;
for j in Nc centroids {Xc(j), Yc(j)}:
    for i in 0 to #nodes:
        d(i,j+1) = (X(i) - Xc(j))^2 + (Y(i) - Yc(j))^2;
        if d(i,j+1) < d(i,j):
            clusterID(X(i),Y(i)) = j;
        else:
            d(i,j+1) = d(i,j);
        end
    end
end
```

Listing 5 K-Means pseudo-code.

The algorithm description is used to derive the smart row structure, which comprises an arithmetic memory row, a multiplier (for power-2 calculation), and a conditional block. This last consists of a comparator, which returns the smallest value among its two inputs. Therefore it computes *output = (A<B)? A:B*. One may see that, for each point in the dataset, a set of distances from Nc centroids must be calculated, and if each point is recorded for each smart row, there are not enough storage elements to hold all the necessary local data. To overcome this issue, the TemporaryStorage IRL is implemented. It is a register that functions as a storage element, storing all the needed data inside each smart row: if there are Nc centroids, then Nc TemporaryStorages are required to store Nc distances.

- *Matrix Vector Multiplication.* The following calculation is done in Matrix Vector Multiplication (MVM) algorithm:

$$\overline{Z} = \overline{\overline{X}} \times \overline{Y} \text{ where } \overline{\overline{X}} \in \mathbb{R}^{N \times M}, \overline{Y} \in \mathbb{R}^{M \times 1}, \overline{Z} \in \mathbb{R}^{N \times 1} \tag{2.6}$$

The smart rows from Equation 2.6 only include an arithmetic memory row, a multiplier, and I/O buffer IRLs. The up-rows and arithmetic memory rows are initial storage locations for $\overline{Y}$ and $\overline{\overline{X}}$. Next, the algorithm begins storing all $\overline{Y}$ values inside the input buffers in order to release the up-rows/down-rows. The punctual multiplication between $\overline{Y}$ and $\overline{\overline{X}}$ is then done, memorized in the output buffers, and relocated within the rows/down-rows. Using the up-row/down-row technique, the multiplications are propagated down the LiM array to calculate the partial sums.

- *K-NN.* K-Nearest Neighbor is a basic classification and regression approach that seeks to identify the k points nearest to a reference node. $D_i = |x_s - x_i| + |y_s - y_i|$ computes the distance between the considered point and the reference node, where (x_i, y_i) is the considered point, and (x_s, y_s) is the reference node. This method can be efficiently executed using Hybrid-SIMD by using an absolute value IRL (ABS).

- *Bitmap indexing.* Bitmap indexing (BMP) is a strategy for quick database searches, according to [129]. It involves altering the data representation in order to search for information inside the database using simple bitwise operations. Each bit represents a field in bit arrays (also known as bitmaps), which may be true or false. If the record corresponds to a certain field, the corresponding bit is set to true. The smart row structure includes the OneCounter IRL, which implements the samples count.

- *Mean & variance.* This procedure, abbreviated as VAR, calculates the mean and variance for an N-point set X. These points are stored in the arithmetic memory rows, and Hybrid-SIMD implements the following code:

```
sum1, sum2, sum3 = 0;
for i in 0 to N: sum1 += X(i);
mean = sum1/N;
for i in 0 to N:
    sum3 += (X(i) - mean);
    sum2 += (X(i) - mean)*(X(i) - mean);
end
variance = (sum2-sum3*sum3/N)/N;
```

Listing 6 Mean-variance pseudo-algorithm.

There are two for loops accumulating the same data, necessitating the employment of the up-row/down-row mechanism and Hybrid-SIMD in sequential computing mode. This algorithm highlights the limitations of Hybrid-SIMD, which results in being inefficient in sequential jobs. In this instance, the smart row includes an arithmetic memory row, a multiplier, an arithmetic right shifter, and two TemporaryStorages.

The final structure of the Hybrid-SIMD's IRLs is shown in Fig. 2.8. The implemented algorithms also yield the sizes for LiM arrays: for instance, if MVM has 256 elements,

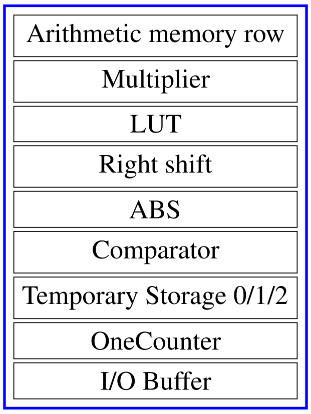


Fig. 2.8 Smart row structure implementing all the proposed benchmarks.

to complete all the concurrent calculations, the total number of smart rows should be 256. Moreover, the number of rows has been fixed to 1024, with NSmartRow = 256, NBit = 32, and nBlocks = 4 in each test. Finally, the memory size is calculated by multiplying NRow by Nbit, and it is equivalent to 4 kB.

Results and comparisons with von Neumann Post Place&Route results are shown in Table 2.11. Due to the fact that it analyzes both the speed and power consumption of the Hybrid-SIMD, comparing the findings in terms of total energy might be advantageous. Energy is equal to the product of the execution time of the algorithm and the total power of the circuit. As expected, VAR gets the greatest energy value for the inability to parallelize the two for loops. In contrast, the BMP scenario yields the lowest energy since its algorithm is simpler than that of the other instances.

Table 2.11 Place&Route results of the Hybrid-SIMD using CMOS 45nm technology [21].

Test	**Place&Route: backannotation**					**Clock (ns)**
	Power (mW)	**Area (mm^2)**	**Critical path (ns)**	**#clock cycles**	**Energy (nJ)**	
DFT	212.90	2.15	6.79	1033	2639.11	12.00
K-Means	93.85			554	623.91	
K-NN	84.79			522	531.12	
MVM	114.20			546	748.24	
BMP	86.12			13	13.44	
VAR	181.40			1799	3916.06	

Hybrid-SIMD is also investigated in a CPU-Memory context: the previously described algorithms are analyzed and implemented on a RISC-V processor, which is emulated using the Gem5 simulator [97]. The RISC-V von Neumann architecture

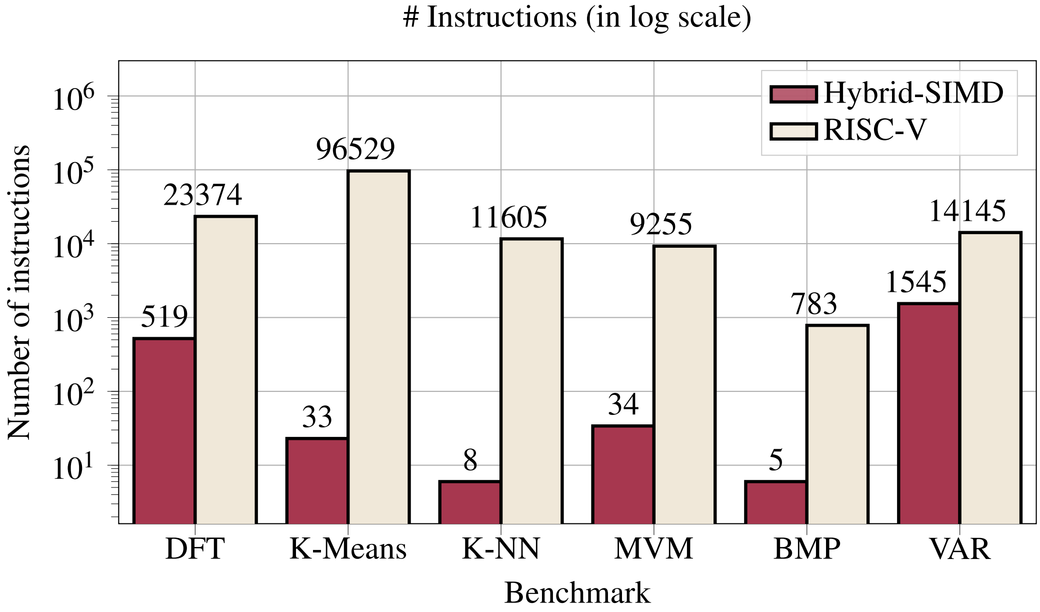


Fig. 2.9 Number of instructions required for RISC-V/Hybrid-SIMD architectures

consists of two cache levels, DRAM and an In-Order single-core CPU. Level-1 is separated into 4kB data and instruction caches. Level-2 cache is 256kB in capacity and is shared between data and instructions. Lastly, the DRAM is 512 MB DDR3 @ 1600MHz. The CPU-Memory system performance is compared with the Hybrid-SIMD architecture by analyzing this basic setup. Since memory is slower than the CPU, time and energy are lost waiting for data. Beyond von Neumann designs attempt to decrease these quantities, which often lowers the performance of conventional systems. Different figures of merit are considered:

- *Number of required instructions.* The amount of instructions needed by algorithms in RISC-V and Hybrid-SIMD architectures is the first comparison offered, which is shown in Fig. 2.9. C is used to implement apps on RISC-V. The source files are built and statically linked using the `-static` option, and then a trace-based simulation is performed to generate the instruction list of the algorithm. The total number of instructions needed in Hybrid-SIMD case is far less than that of the CPU. This is because Hybrid-SIMD is capable of SIMD calculations, making it significantly quicker and efficient than a CPU-based implementation.

- *Number of memory accesses.* Gem5 is used to give execution statistics for RISC-V algorithms. These values can be obtained in the output file `stats.txt`, which includes the total number of memory accesses of the program. In the Hybrid-SIMD case, memory access calculations consider the size of the algorithm's data: for example, K-Means uses 256 coordinates, Nc=3 centroids, and Nc masks to determine clusterIDs. The total number of memory accesses for K-Means is thus 521 $(256 \times 2 + 3 \times 2 + 3)$. For RISC-V, the results are shown in Table 2.12, where L1DCache, L1ICache, and L2SCache, respectively, represent Level-1 Data/Instruction caches and Level-2 Shared cache. The amount of memory accesses is one of the most influential contributors to the von Neumann bottleneck. In every scenario, Hybrid-SIMD is more efficient than RISC-V, minimizing memory accesses and presenting itself as a suitable coprocessor in a conventional system.

- *Classical memory hierarchy consumption vs Hybrid-SIMD.* Table 2.12 and Fig. 2.10 report the amount of energy of the memories and the number of memory accesses for each benchmark, respectively.

Table 2.12 Energy comparison between caches energy accesses and Hybrid-SIMD.

Test	**Total energy (nJ)**		
	RISC-V	**Hybrid-SIMD**	**Improvement (%)**
	Caches: accesses	Entire architecture	
DFT	2826.42	2704.83	4.30
K-Means	11077.19	628.09	94.33
K-NN	1434.12	532.14	62.89
MVM	1210.60	752.54	37.84
BMP	105.18	14.07	86.62
VAR	1997.46	4111.46	-105.83

Cacti [91] is used to estimate the caches' energy: Cacti is a tool created by HP Laboratories that can accurately simulate the performance of caches and memories. L1D/ICache, L2SCache, and uROM are modeled with Uniform Cache Access (UCA), and one read/one write port using CMOS 45nm technology. Using the energy/access values, the energy consumption of caches for the analyzed algorithms is evaluated and compared to that of Hybrid-SIMD. This is a worst-case scenario for Hybrid-SIMD since the total energy consumption of the design is examined (memory operations and computation). In contrast, RISC-V calculations are made in the best-case scenario, i.e., only the cache

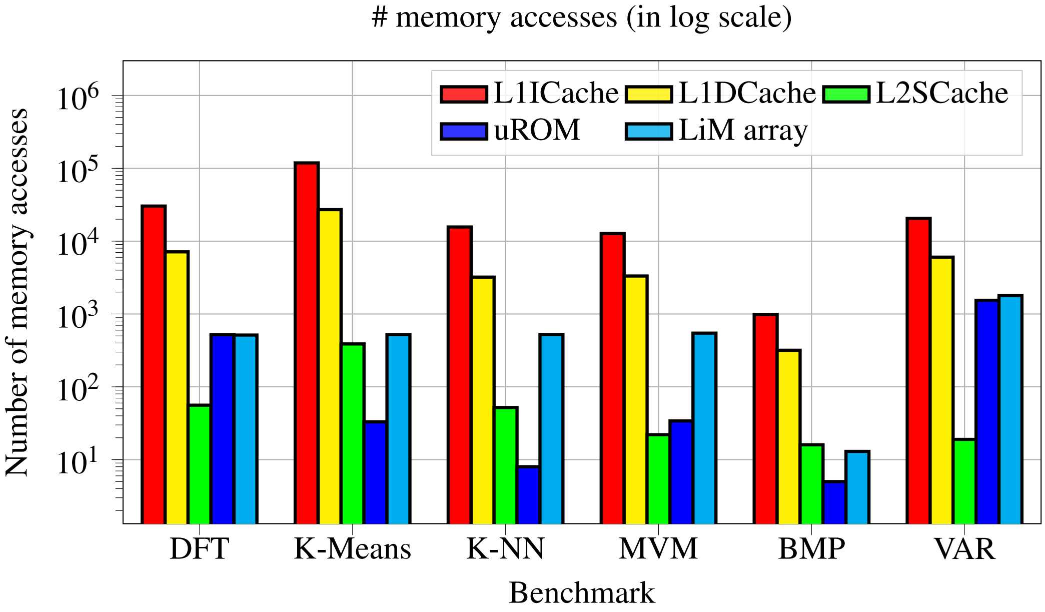


Fig. 2.10 Number of memory accesses for RISC-V and Hybrid-SIMD systems.

energies are examined, and the core consumption is ignored. In Table 2.12, the instruction cache contributes the most energy to the RISC-V architecture, as it is accessed often in all benchmarks. At first glance, the contribution of the Hybrid-SIMD architecture (provided by the total of uROM and LiM array energy consumptions) to the suggested benchmarks looks to be more energy efficient. VAR achieves the minimal figure of $\sim$ -100% energy improvement because, as described before, the Hybrid-SIMD architecture is very inefficient at implementing sequential algorithms, resulting in a deterioration of energy and execution time. As predicted, DFT achieves the second-worst value since it is slower than the others, although Hybrid-SIMD is still more energy-efficient. Other benchmarks provide very promising performance outcomes, with an energy improvement of at least 94% compared to RISC-V memory hierarchy. Moreover, compared to caches, Hybrid-SIMD memory accesses are dramatically decreased since the LiM array can simultaneously store and process data, reducing data traffic. Utilizing Hybrid-SIMD as a coprocessor enables the CPU to outsource massively parallel tasks to Hybrid-SIMD, hence enhancing performance.

MeMPA: Memory Mapped Programmable Architecture [15, 16]

Architecture In Fig. 2.11(a) the system environment hosting the MeMPA is shown. Since MeMPA functions as a memory-mapped co-processor, the CPU requires two

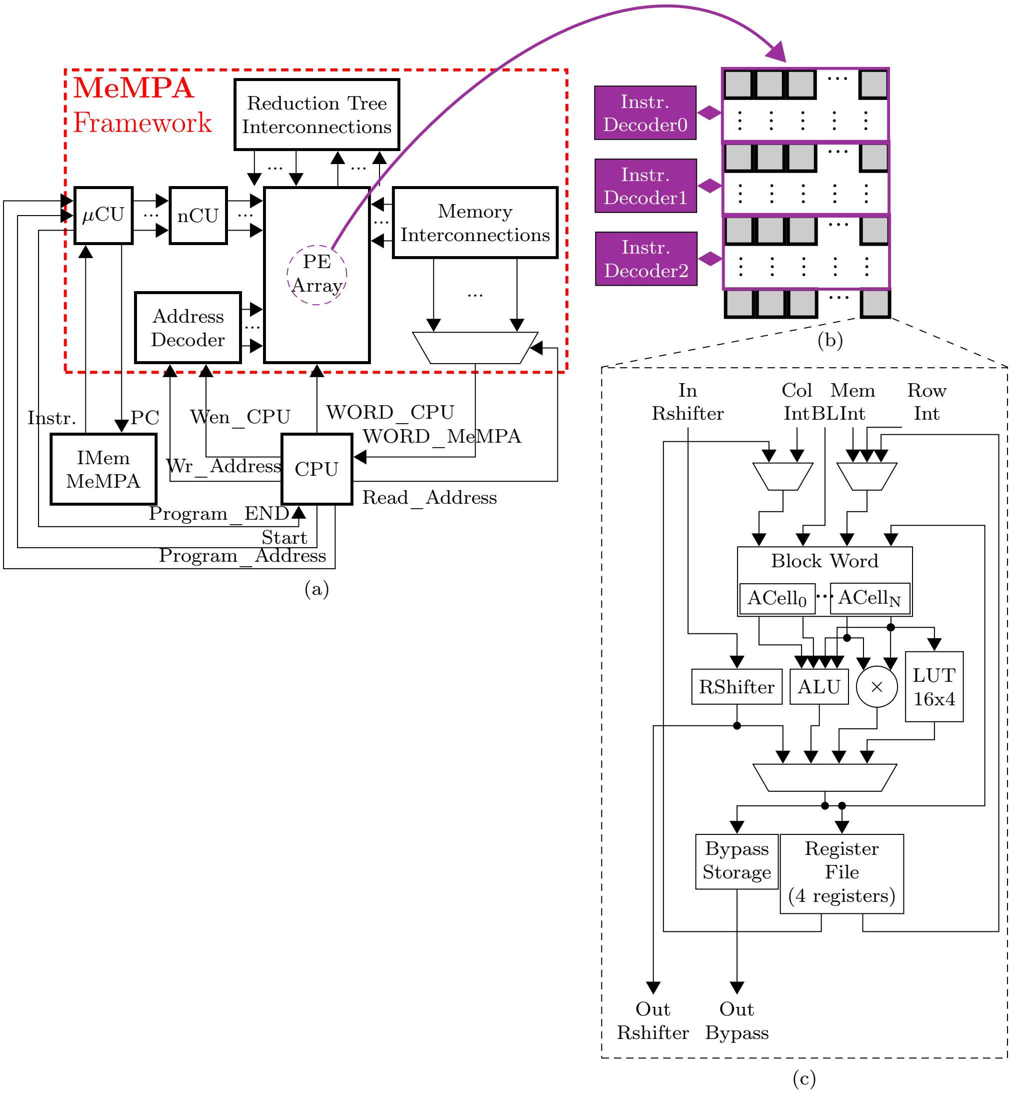


Fig. 2.11 MeMPA system. [15, 16] (a) MeMPA top-level view. (b) Processing Matrix structure, with Standard Blocks (only memory) and Smart Blocks (memory and computation), and M-SIMD implementation. (c) Smart Block architecture. (d) Structure of the arithmetic cell composing the Block Word.

sets of signals at its interface in order to interact correctly with it: one for exchanging data and another for initiating MeMPA to execute the data-intensive portions of the code loaded into the MeMPA Instruction Memory. In particular, the former set utilizes the standard double port data memory protocol (1 asynchronous reading port and 1 synchronous writing port). The internal structure of MeMPA consists of three macro sections: the control section, the datapath, which is comprised of a matrix of fully interconnected PEs (i.e., the Processing Matrix) and is where data processing occurs within MeMPA, and the section responsible for data exchange with the CPU. This last element links the first set of CPU external pins to the Processing Matrix using an address decoder and multiplexer to identify the PE where data must be written or read. The control portion works exactly as the Hybrid-SIMD case, in fact MeMPA's control unit is a microprogrammed machine as well. The datapath is represented by the Processing Matrix, which is the MeMPA entity responsible for data storage and processing. The Processing Matrix is composed of 256 PEs with memory capabilities, referred to as Smart Blocks, arranged in a grid of 16 columns and 16 rows. In addition, 80 Standard Blocks, which are standard registers that provide MeMPA with additional storage capacity, are inserted underneath the Smart Blocks. Due to the nCU structure, which consists of three instruction decoders, the Processing Matrix may concurrently execute up to three distinct instructions, each on a separate piece of data. Specifically, each instruction decoder (ID) receives as input a separate sub-portion of the MeMPA instruction and transforms it into control signals that drive a particular set of Smart Blocks rows. The mesh-like architecture of the Processing Matrix is used to construct a battleship-like enabling mechanism.

Routing Network Memory Interconnections and Reduction Tree Interconnections are the two types of interconnections in the routing network. All of them are implemented by a well-organized collection of multiplexers. Each Row Interconnection permits the transmission of data between Smart Blocks in the same row, while each Column Interconnection, which extends to Standard Blocks as well, enables Smart Blocks to receive data from any block in the same column. In contrast, Memory Interconnections span the whole Processing Matrix, allowing any Smart Block to obtain data from any block. However, the two kinds of interconnections are not distinguished by the collection of blocks they connect but rather by the manner in which they transfer data among the various blocks. Each Row or Column Interconnection enables all connected Smart Blocks to take different data simultaneously, even if they are controlled by the same instruction, whereas Memory Interconnections are used when multiple Smart Blocks controlled by the same instruction must select the same data from a random block of the Processing Matrix.

Smart Block Each Smart Block (refer to Fig. 2.11 (c)) includes all the storage components and computational blocks required to perform the most frequent operations detected by the Algorithm Profiling technique. Inside a Smart Block, there are a Right Shifter (RShifter) to perform division-by-two, an ALU to perform most common arithmetic-logic operations, a multiplier, a Register File to hold temporary values, a Bypass Storage to provide the input data for the Reduction Tree Interconnections, a few multiplexers to select the data, and a programmable Look-Up Table (LUT) to implement specific functions. The Block Word consists of 32 1-bit arithmetic cells, as seen in Fig. 2.11 (c), which are the finer-grained entities of the MeMPA Processing Matrix and are equipped with memory and processing capabilities. The arithmetic cells deal with ALU, Multiplier, and LUT in order to provide the right operand or execute certain preliminary bitwise operations. For a generic operation, the Smart Block may operate on data stored within the Smart Block itself, within the Block Word or Register File, or coming from a block on the same row, a block on the same column, or an arbitrary block within the Processing Matrix. To enable MeMPA to perform a finite number of divisions by powers of 2, the RShifter has a separated input to allow the cascade connections of all the RShifters belonging to the Smart Blocks on the same row. Therefore, the Processing Matrix may perform up to 16 divisions in parallel on data coming from Memory or Reduction Tree Interconnections.

Benchmark mapping For a straightforward evaluation of MeMPA's performance, the same benchmarks used for the Hybrid-SIMD evaluation [127] are used. In addition, the absence of a proper compiler for MeMPA prevented the implementation of complicated algorithms, such as SPLASH-2 benchmarks used for the profiling operation, for which manual mapping would have been very difficult. Table 2.13 summarizes the algorithms in terms of the quantity of processed data, the number of clock cycles required for algorithm execution, and the associated post Place&Route power consumption. These results are achieved using fixed dimensions of 16×16 Smart Blocks and 5×16 Standard Blocks with an M-SIMD degree of 3. However, performance is directly proportional to MeMPA scalability, which means that increasing the total number of Smart Blocks, Standard Blocks, their parallelism,

Table 2.13 Data initialization cycles, parameters, execution cycles and post-Place&Route back-annotated power of each algorithm

Benchmark		**Data Initialization** # clock cycles	**Algorithm Execution** # clock cycles	**Parameter**	**Power** (mW) @4ns
K-NN	with N samples	$2 \times N + 2$	7	$N = 160$	72.95
K-means	with K centroids and N samples	$2 \times N + 3 \times K + 2$	$15 \times K - 11$	$N = 160$ $K = 3$	74.48
MVM	$\overline{Z} = \overline{\overline{X}} \times \overline{\overline{Y}}, \overline{\overline{X}} \in \mathbb{R}^{u \times v}, \overline{Y} \in \mathbb{R}^{v \times 1}, \overline{Z} \in \mathbb{R}^{u \times 1}$	$v \times (u+1)$	$\lceil u/3 \rceil + log_2 v + 1$	$u = v = 16$	62.64
$\mu \& \sigma^2$	with N samples	N	$3 \times log_2 N + 13$	$N = 256$	65.77
DFT	with N samples	$2 \times N + 1$	$log_2 N + 39$	$N = 128$	94.44

or the degree of M-SIMD enhances processing power. In contrast, surpassing the MeMPA size diminishes crucial metrics for benefits such as area, power, time, and energy. The mapping of any algorithm on MeMPA consists of two major phases: the Processing Matrix initialization phase, during which the CPU feeds all data to be processed into the Processing Matrix, and the algorithm execution phase, during which the algorithm is executed. Regarding the MVM mentioned in Table 2.13, each matrix element is stored in a unique Block Word of the Smart Blocks, whilst all vector components are put into the first row of Standard Blocks. Then, the data from the Block Words is moved into the first location of the Register Files, to avoid losing the initial data when the MeMPA saves the results into the Block Words to make them visible to the CPU. After then, the 256 products between each matrix element and the vector are executed, in particular by considering 3 rows at the same time. The results are stored in the corresponding Bypass Storages. Once all the products are ready in the Bypass Storages of the Processing Matrix, all sums are executed in four instructions. To do this, the reduction tree of the Row Interconnections is fully used, reducing the number of instructions required to compute 16 simultaneous additions from 15 to 4. After the conclusion of the last instruction, all the final values are accessible in the Block Words column of the first Processing Matrix column.

Performance comparisons Firstly, comparisons of MeMPA and Hybrid-SIMD's energy and execution time are offered. Due to the fact that Hybrid-SIMD and MeMPA have different sizes and memory space, the energy and execution time are normalized by the number of samples evaluated for each case (# Samples) for a more accurate comparison. Secondly, and most crucially, MeMPA is implemented in a traditional CPU-memory architecture, and the performance of the algorithms in two unique instances is examined. In the first, labeled **CPU-Mem**, the structure of the von Neumann architecture is preserved, thus algorithms are run exclusively by the CPU

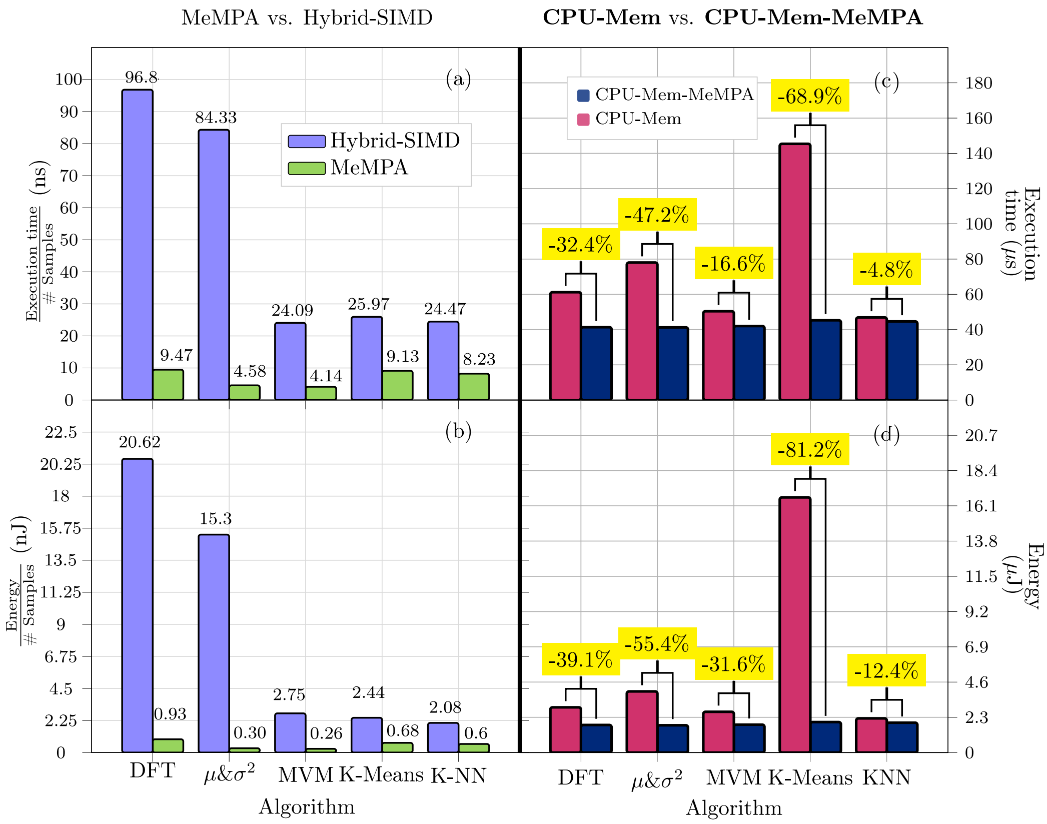


Fig. 2.12 MeMPA vs Hybrid-SIMD: (a) reports the Execution time/# Samples, while (b) evaluates the Energy/# Samples for both structures. CPU-Mem-MeMPA: Comparison of execution time (c) and energy (d) between CPU-Mem and CPU-Mem-MeMPA solutions.

and evaluated in a traditional context. The second proposal, **CPU-Mem-MeMPA**, examines the incorporation of MeMPA into the von Neumann architecture.

In Fig. 2.12 (a-b), the execution time and energy comparisons between MeMPA and Hybrid-SIMD are shown. With clock periods of 12ns and 4ns, respectively, the values for Hybrid-SIMD and MeMPA are determined. MeMPA has a smaller addressable area than Hybrid-SIMD, thus a lower number of data can be handled. MeMPA's reduced memory size is balanced by its high degree of programmability and capacity to conduct several complicated tasks simultaneously. With restricted flexibility, the Hybrid-SIMD architectural model can implement a narrower range of programs, with worse performance in sequential algorithms. As shown in Fig. 2.12 (a-b), MeMPA surpasses Hybrid-SIMD for all suggested execution time and energy benchmarks. In addition, MeMPA has a shorter critical path (3.88ns vs. 6.79ns) and uses less power (worst case of MeMPA: 102.18mW vs. 212.18mW for the DFT method).

Following a similar methodology of the Hybrid-SIMD case, a comparison in terms of energy and execution time between CPU-Mem and CPU-Mem-MeMPA frameworks is presented. A published RTL model is utilized to predict the RISC-V core's performance properly, Pulpino [130], an In-Order single-core RISC-V processor. Following the same process as MeMPA, the core is synthesized and Place&Routed. At the conclusion of the Place&Route phase, the algorithms are simulated, and the annotated switching activity is reused by Cadence Innovus for power estimation. The results are shown in Fig. 2.12 (c-d), with a clock period set to 6 ns, representing the worst critical path value between MeMPA and the RISC-V core. In terms of execution speed and energy consumption, the CPU-Mem-MeMPA framework surpasses the traditional CPU-Mem in all benchmarks. Thanks to the M-SIMD computing mode of MeMPA and the dense interconnections network, algorithms can be easily accelerated, resulting in improved performance, particularly for parallel algorithms such as K-Means and algorithms that can heavily exploit the Reduction Tree computing mechanism. It is important to note that the addition of MeMPA also minimizes memory accesses and energy consumption, as seen in Table 2.14 since once data are loaded into MeMPA, the calculation is conducted directly inside the Processing Matrix. Once the process is complete, data may be read from MeMPA. MeMPA reduces energy, execution time, and memory accesses up to 81.2%, 68.9%, and 83.6%, respectively, for the K-Means method.

Table 2.14 Comparison of the number of L1 and L2 cache memory accesses for CPU-Mem and CPU-Mem-MeMPA.

Algorithm	**Memory Accesses (L1&L2)**		**Reduction (%)**
	CPU-Mem	**CPU-Mem-MeMPA**	
KNN	19799	16702	15.6
K-Means	103362	16946	83.6
MVM	24153	15479	35.9
μ**&**σ^2	36606	15090	58.8
DFT	26599	15133	43.1

2.2 Conclusions

With these previous works, the goal is to prove the effectiveness of the LiM approach applied to a standard von Neumann system. In both Application Specific and

General Purpose approaches, LiM demonstrates the ability to reduce the effects of the von Neumann bottleneck, both reducing the memory accesses and the energy of the system. Application Specific implementations are more power efficient and faster than General Purpose Hybrid-SIMD and MeMPA, however they are able to implement a very small set of algorithms and operations. On the other hand, Hybrid-SIMD, and especially MeMPA, focus on a heavy parallelization, on different levels, of the algorithm execution. Moreover, the core parts of Hybrid-SIMD and MeMPA processing were designed to provide as much programming generality as possible by considering a wide range of algorithms from the SPLASH-2 benchmark suite and profiling the most used instructions. Due to their fully interconnected structures of processing elements integrating computing and storage capabilities, the insertion of Hybrid-SIMD and MeMPA inside a classical CPU-Memory context was confirmed to successfully bring overwhelming reductions in energy and execution time for the proposed benchmarks compared with the classical von Neumann solution. In the next chapters, the core part of this thesis will be explained, presenting DExIMA, a tool able to assist the designer in the LiM design flow. DExIMA, as discussed later, relies on an architectural model replicating the ones proposed before, starting from the finer-grained LiM Cell block towards the top-level LiM architecture. In contrast, MeMPA architectural model, in which the finer-grained element is N-bit block, will be implemented as a future work.

Part II

DExIMA tool for LiM design exploration

Chapter 3

Overview of DExIMA software

Summary

*The acronym DExIMA stands for **D**esign **Ex**plorer for **I**n-**M**emory **A**rchitectures, and it is intended as a tool able to assist the user during the LiM architecture design phases. Starting from the definition of the blocks composing the architecture, DExIMA is able to perform simulations of the circuit and estimations of the performance achieved. DExIMA is composed of a Graphical User Interface (GUI) incorporating a Schematic Capture (DExIMA-CAD), that guides the designer through the design flow of the BvNC devices, and by a Backend (DExIMA-Backend), which is an ad-hoc performance estimator implemented in C++. In Fig. 3.1, the high-level scheme of DExIMA is reported. The design flow starts with the architectural design of the memory array by defining the internal structure of the cells, moving then to the definition of the memory array characteristics. The LiM array is designed by means of the Schematic Capture tool provided by DExIMA-CAD, or it can be generated directly by translating an input algorithm into a LiM architecture using Octantis tool [131]. The interface with Octantis, however, is not covered in this thesis, because it is still under development in another doctoral project. DExIMA recalls the BvNC structure proposed by [127, 13, 63, 132], especially the Hybrid-SIMD [127], but it can also reproduce different architectural and structural models by changing the arrangements of the LiM blocks with high flexibility and modularity. DExIMA performs estimations on a given library of technologies that essentially contains the parameters of the devices (such as the channel length, the width of the transistors,*

etc.) and a set of LiM templates that are required to define boundaries in the design exploration. In the following sections, all these aspects are discussed in deep.

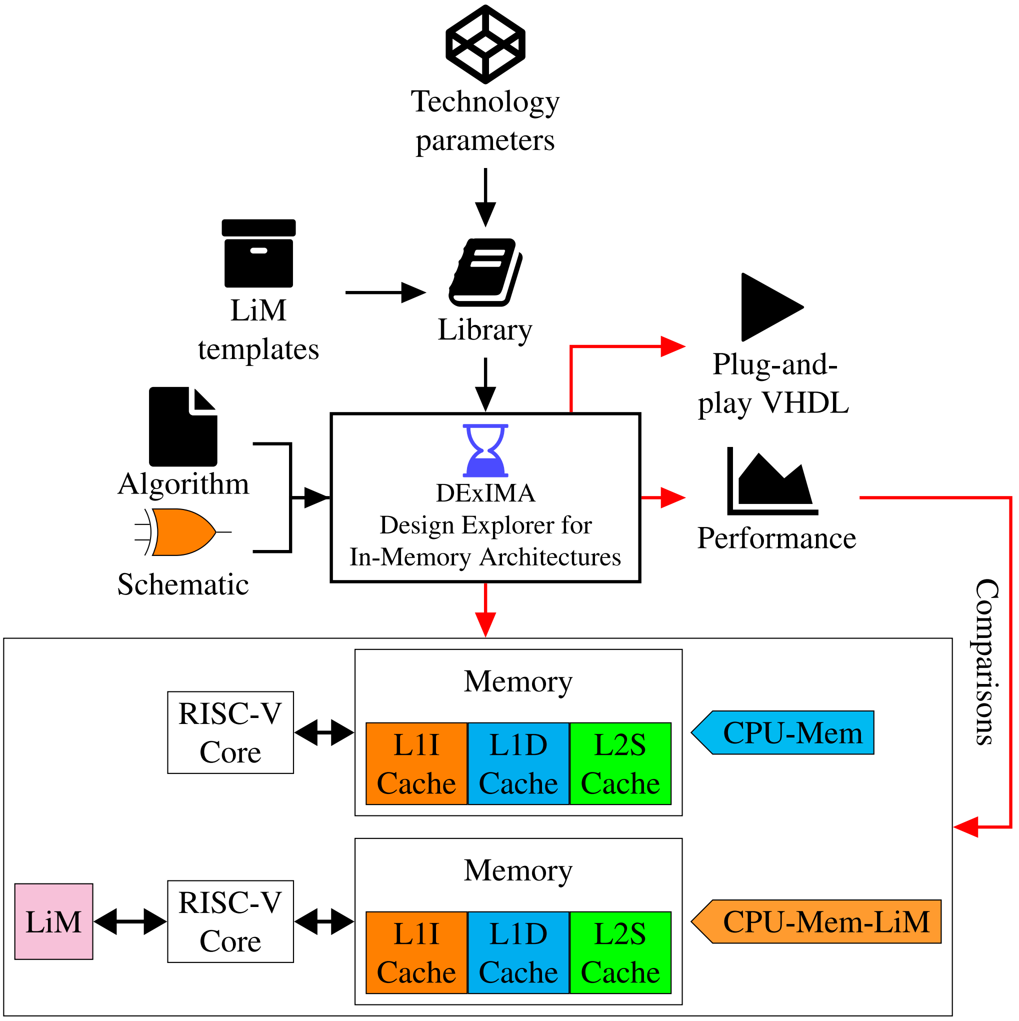


Fig. 3.1 Overview of the DExIMA software

3.1 DExIMA architectural reference structure

In Fig. 3.2, the reference architectural model of the LiM array used by DExIMA is defined. The finer-grained block is the LiM Cell, composed of a 1-bit memory element and optionally very simple logical circuits, usually made of a few logic gates. The Intra Row Logic (IRL) instead, is a more sophisticated computational part that is interposed between two consecutive rows and can contain blocks such as adders,

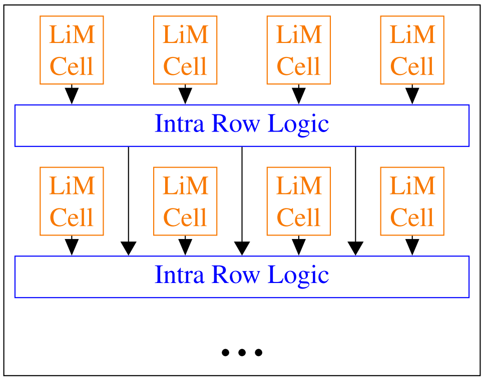


Fig. 3.2 Reference architectural model for the LiM Memory Array.

multipliers, shifters, registers etc., if required by the implementation. During the design phase, the user can choose the kind of LiM Cell and IRL he/she wants, also depending on the application or algorithm implemented. This structure is adopted in works like [127, 13, 63, 132]. For instance, in [13], an XNOR-Net Binary Neural Network was implemented where the convolution operation is approximated into an XNOR bitwise and pop counting operations, so the LiM Cell was equipped with an XNOR gate and a half adder to accomplish the convolution computation. Similarly, in [63], the memory array is equipped with one full adder and a configurable logic block for each LiM Cell, allowing to perform parallel convolutions. In [132], a similar structure is proposed, this time implementing a different algorithm named Bitmap Indexing, which consists of highly parallelizable bitwise logical operations aiming to perform search operations and classify data. In previous examples, the IRL part was absent because the implemented applications were very simple and specific, requiring simple hardware. As already said before, more complex structures like Hybrid-SIMD [127] implements sophisticated LiM Cells made of full adders, configurable logic blocks, and multiplexers to perform a wide range of 1-bit operations together with complex IRL circuits, aiming at covering the widest range of arithmetical and logical operations possible. With DExIMA, the idea is to achieve maximum flexibility by replicating all these architectural models. However, the reference structure can be modified to accommodate other LiM paradigms or solutions proposed in the literature.

3.1.1 Control part and design templates

One of the main aspects of LiM architecture is the definition of a controlling circuit that should be able to drive the LiM correctly. As already stated, DExIMA gives the user high flexibility in the design stage, meaning that the LiM Cells and IRL architectures can be defined arbitrarily, according to the application. However, this high flexibility increases the complexity of the Control Unit because the degrees of freedom tend to be unpredictable, requiring extremely complicated hardware and designing skills. To solve this problem and to better define the LiM design methodology while maintaining high generality, DExIMA provides a series of templates that the user can choose at the beginning of the design flow. These templates are linked to the number of input-output pins of the LiM Cell and IRL. In Fig. 3.3, the base versions of the LiM Cell and IRL templates are shown. Starting from the input pins, their functions are the following:

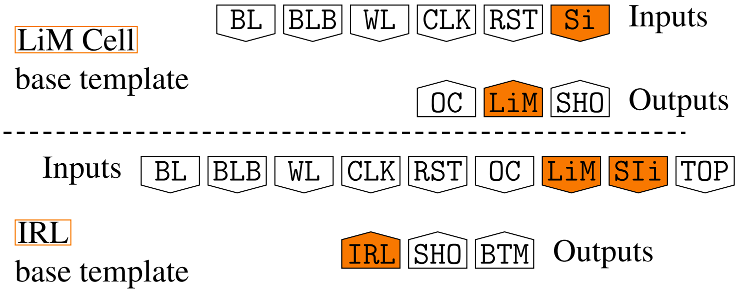


Fig. 3.3 Base templates of the LiM Cell and IRL.

- Bitline (BL) and bitline bar (BLB) are the signal that brings data from the CPU, written inside the LiM Cells.
- Wordline (WL) enables the memory row according to the input address.
- Clock (CLK) and Reset (RST) drive synchronous circuits such as memory elements, flip-flops, or registers.
- LiM Cell selector(s) (Si), Intra Row selector(s) (SIi) are signals driven by the Control Unit, according to the desired function. For instance, these signals can be connected to the selector input of a multiplexer to select the appropriate value or to enable a synchronous block to memorize a temporary variable. They can also be used as data signals.

- TOP signal is a special signal that connects a row to the previous one, as shown in Fig. 3.2. In particular, TOP signal of the i-th row is connected to BTM signal of the (i-1)-th row.

The output pins have the following meanings:

- Output Cell (OC) is normally connected to the output of the memory element of the LiM Cell, and it is useful to read the data stored inside the memory array.
- 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 the rows. A tristate buffer is usually employed to handle this output to avoid conflicts.

OC, LiM, SHO signals are used as outputs in the LiM Cell, but these signals are also connected to the IRL part as IRL can reuse them to perform other computations. As depicted in Fig. 3.3, some signals are highlighted in orange, meaning that the other templates for the LiM structure rely on these elements. More complicated templates can have more selector inputs (S0,S1,S2,S3,...) or more LiM/IRL outputs (LiM0,LiM1,LiM2,...,IRL0,IRL1,...). By defining the template, the Control Unit can be designed according to the available selectors since these inputs are used to program the behavior of the cells or IRL parts. The user can directly define the template by means of the template selector tool provided by DExIMA. The Control Unit architecture is based on a microprogrammed machine, which is the most flexible structure, following a similar approach to the Hybrid-SIMD [127]. A microprogrammed 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 LiM array are defined, and in particular, each LiM instruction defines the values of the selectors. In Fig. 3.4, the high-level scheme of the microprogrammed Control Unit is shown. It relies on the explicit addressing mode, where the next address is written inside one of the uRAM fields, allowing to perform operations in sequence: in the future, this part can be improved to support subroutines and loops. The uRAM fields are:

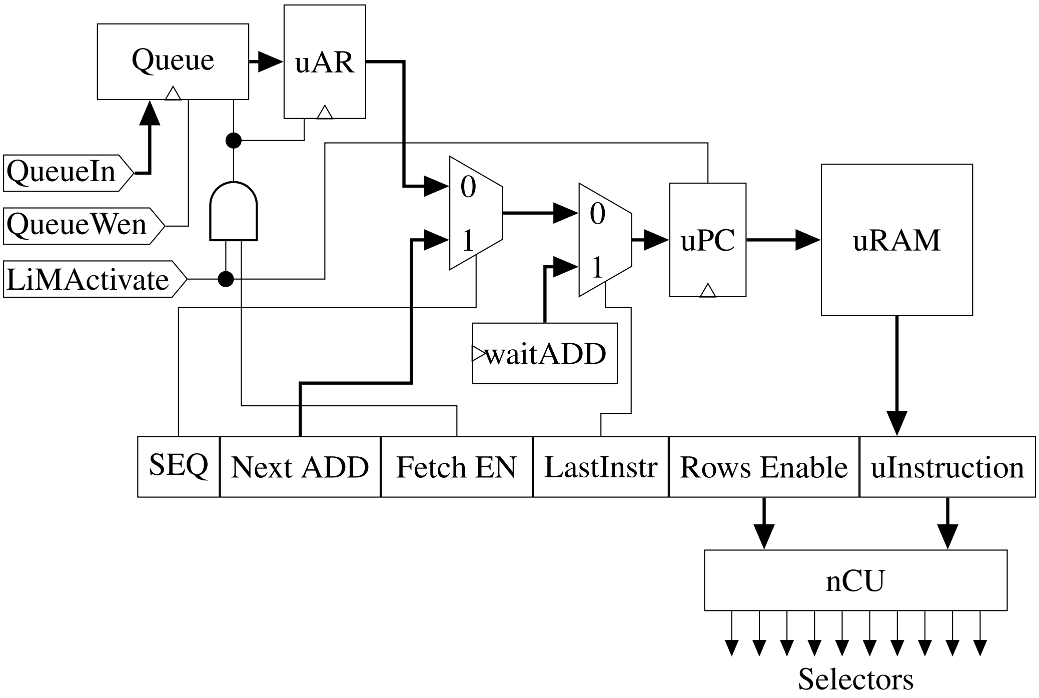


Fig. 3.4 Microprogrammed Control Unit implemented in DExIMA to support general and flexible driving.

- **SEQ** is a single-bit field indicating the next address to choose between the micro address register (uAR), where it is written the starting address parsed by the Queue, or the Next ADD field of the uRAM.

- **Next ADD** specifies the next address of the uRAM, containing the next instruction.

- **Fetch EN** is the fetch enable signal, asserted at the beginning of the program to sample the starting address from the Queue.

- **LastInstr** is the last instruction flag that selects the wait address (waitADD) and stalls the Control Unit.

- **Rows Enable** is a signal having parallelism equal to the number of rows, selecting which row(s) is(are) enabled in that particular instruction. This signal is useful for the operations that need a specific row activation pattern, as discussed in chapter 4.

- **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 the nano-Control Unit (nCU) that performs an AND bitwise between the value of the selectors and the enable of each row, generating the **Selectors** signal.

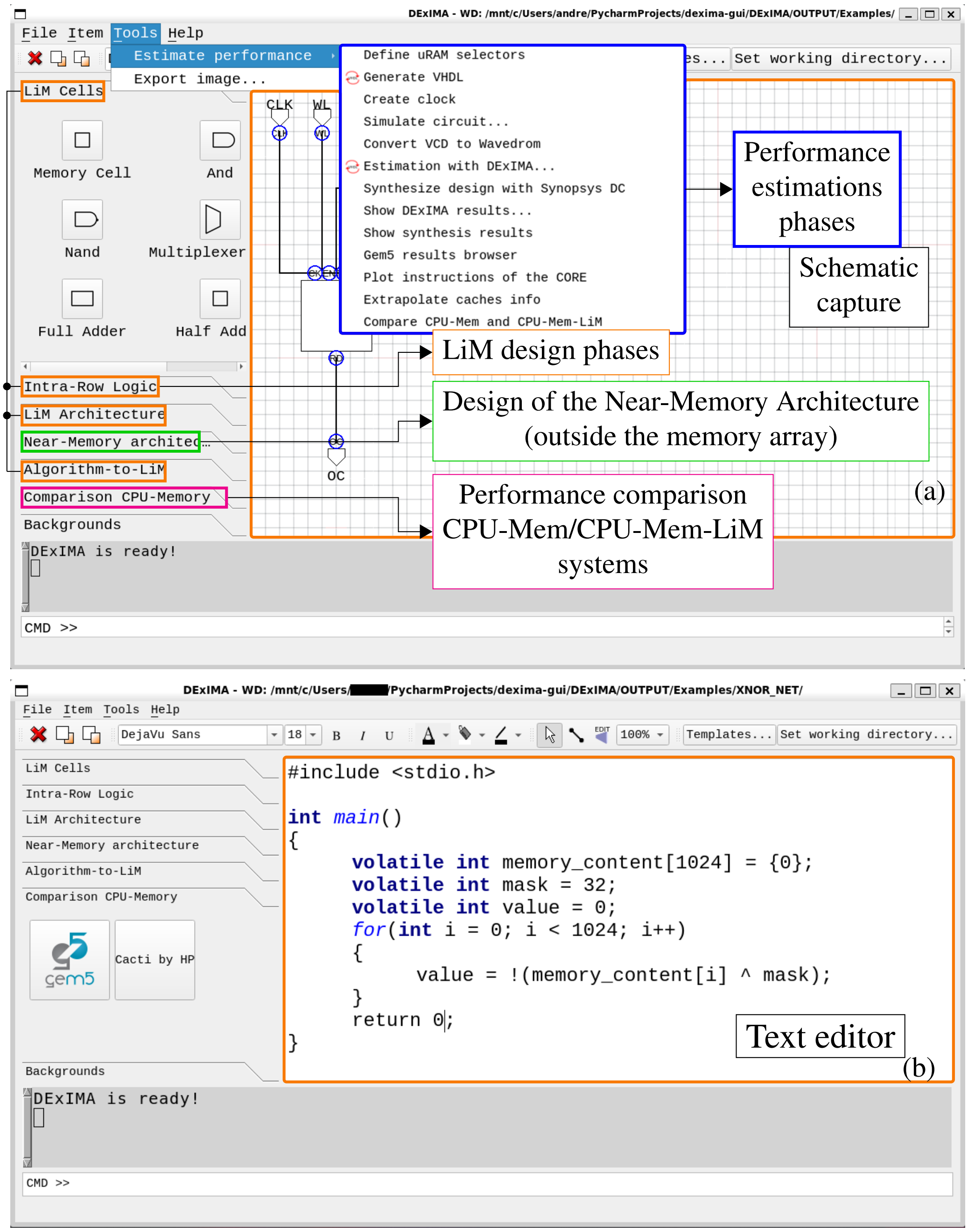
3.2 DExIMA-GUI: Graphical User Interface

The main Graphical User Interface of DExIMA software is shown in Fig. 3.5, that is composed of several tabs:

- *LiM Cells, Intra Row Logic, LiM Architecture, and Near-Memory architecture.* In these environments, the designer creates the LiM Cells, the Intra Row Logic blocks, the top-entity architecture, and, if required, a Near-Memory architecture. These tabs are structured with the component selector on the left and Schematic Capture on the right, where the blocks can be placed and connected.
- *Algorithm-to-LiM and Comparison CPU-Memory.* The other two tabs, instead, have a text editor widget in which the user can write the C code required for the automatic generation of the LiM architecture and the algorithm to implement inside the CPU-Memory standard system to be compared with, respectively.

DExIMA provides the user with several options that automatize the simulation, performance estimation and validation of the designed architecture.

- *Define uRAM selectors.* This tool assists the designer in the definition of the controlling part of the LiM architecture. In particular, as discussed later, it allows writing the control signals inside the uRAM memory that drives the selectors defined by the LiM templates and the row-enable patterns.
- *Generate VHDL.* Once the architecture has been defined in the schematic editor, a synthesizable VHDL code of the architecture can be directly generated by the tool.
- *Simulate circuit.* The generated RTL code can be automatically simulated by employing Mentor QuestaSim software, which is directly called by DExIMA, which generates the scripts and sets the environment variables properly.



DExIMA is ready!

```
#include <stdio.h>

int main()
{
      volatile int memory_content[1024] = {0};
      volatile int mask = 32;
      volatile int value = 0;
      for(int i = 0; i < 1024; i++)
      {
            value = !(memory_content[i] ^ mask);
      }
      return 0;
}
```

Fig. 3.5 Main window of the DExIMA tool. (a) Schematic capture. (b) Comparison CPU-Memory and the text editor widget.

- *Estimation with DExIMA & synthesis with Synopsys Design Compiler.* After the simulation, the architecture can be estimated with DExIMA-Backend, providing performance results on the chosen technology or synthesized with Synopsys Design Compiler.

- *Show DExIMA and Synthesis results.* Performance results can be visualized both in plot or tabular formats.

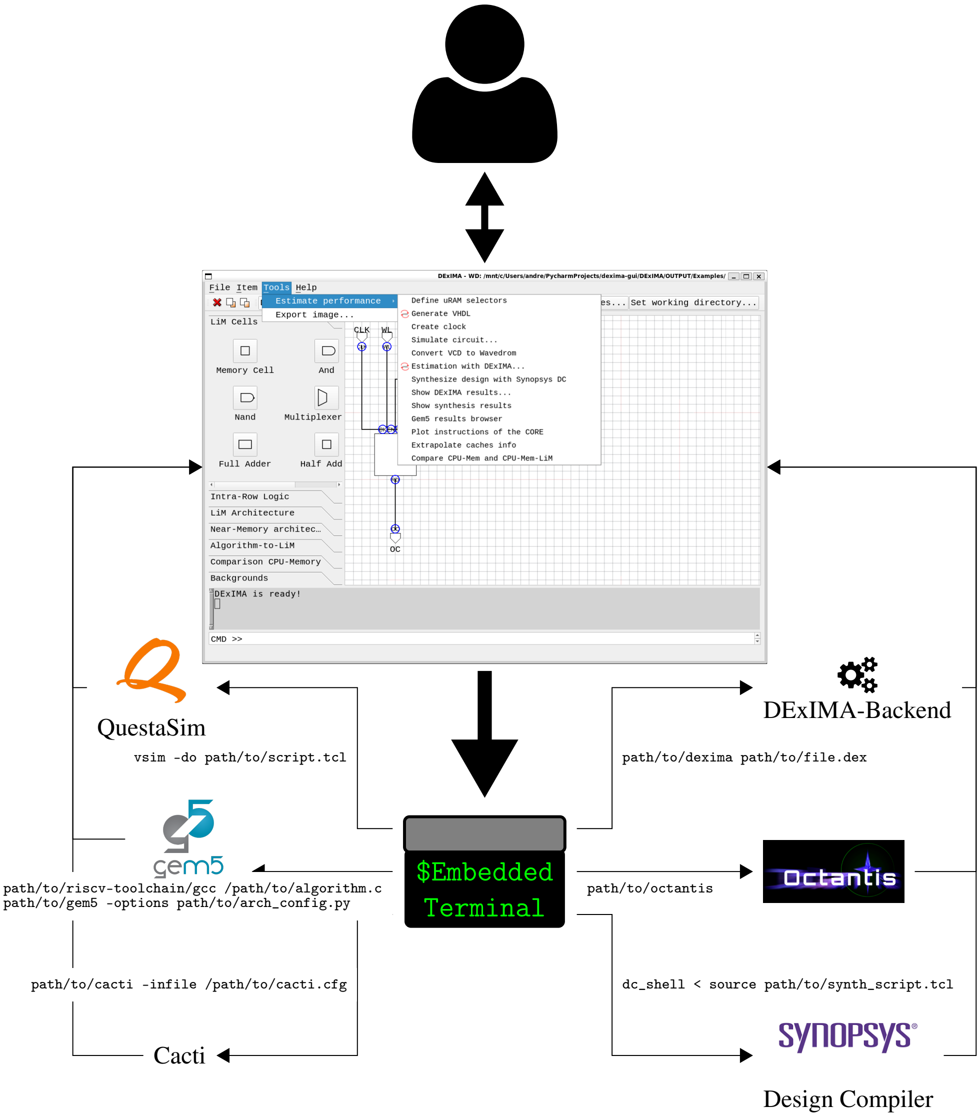
- *Gem5 results browser and plot instructions of the CORE.* These options are used in the "Comparison CPU-Memory". The first one allows visualizing the results of the Gem5 simulation, saved in HDF5 format, while the second one evaluates the number of instructions executed by the CPU core in two cases: the classical von Neumann architecture and the Beyond von Neumann architecture with the designed LiM.

- *Extrapolate caches info.* In the "Comparison CPU-Memory" phase, the user can extrapolate the memory statistics, such as the number of memory accesses, misses, hits, etc., for both CPU-Mem and CPU-Mem-LiM systems.

- *Compare CPU-Mem and CPU-Mem-LiM.* Finally, at the end of the LiM design process, the two systems are compared in terms of performance to clearly indicate the LiM impact in a standard context.

The user can select two design modes: the LiM, where the structure of the cells, the IRL, and the top-level entity can be defined, and the Near-Memory, where the user can design the surrounding logic to the memory array. The LiM mode is structured in a hierarchical way, starting from the finer-grained blocks (the cells) towards the top-entity architecture, so the user has to select the LiM Cells tab, implement the architecture of the cell, selecting the appropriate blocks and connecting them by means of the Schematic Capture, and then move to the IRL and the top-entity with the same modality. At the end of the design phase, the architecture reproduces the structure presented in Fig. 3.2, where the LiM Cells are directly connected to the IRL, and the final array is built.

In order to implement all the previously mentioned operations, as shown in Fig. 3.6, the user can interface with the DExIMA GUI which, through an embedded terminal, communicates with the various external tools required. The GUI then retrieves the different data produced by the external tools to, for example, show performance results, compare the results etc.



```
vsim -do path/to/script.tcl
path/to/dexima path/to/file.dex
path/to/riscv-toolchain/gcc /path/to/algorithm.c
path/to/gem5 -options path/to/arch_config.py
path/to/octantis
path/to/cacti -infile /path/to/cacti.cfg
dc_shell < source path/to/synth_script.tcl
```

Fig. 3.6 High-level scheme of DExIMA software interface.

3.3 LiM design phases

In this part, the design flow of the LiM is presented. Referring to Fig. 3.7, the flow starts with the schematic entry and goes through the LiM array definition, the VHDL and DExIMA-Backend codes generation, the simulation, the back-annotation and finally performance estimation by means of DExIMA-Backend.

① *Schematic capture*. This phase represents the crucial part of the LiM design. As anticipated previously, the LiM Cells, the IRL, and the top-entity LiM array are defined in this stage. The user can select among different logical blocks the ones required to implement the function required. In particular, the LiM Cell is always made of a memory element and, if required, some simple surrounding logic.

② *LiM array definition.* Once LiM Cell and IRL are defined, the user can select the `LiM Architecture` tab and instantiate the memory array. When instantiated, the LiM memory is an empty object: the tool allows to define the structure of the LiM by asking the user to provide the number of rows/columns, the cell type for each coordinate of the matrix, and the IRL circuit for each row. This selection is accomplished by means of two tables, where the content of each location can be changed according to the cell/IRL selected. After the definition of the LiM Memory, the program automatically instantiates the microprogrammed Control Unit needed to control the circuit, called Memory Interface.

③ *RTL code files generation.* In this step, DExIMA can generate a synthesizable VHDL description of the top-entity architecture of the LiM and the microprogrammed Control Unit.

④ *Automatic RTL simulation with UVM and Mentor QuestaSim.* The generated VHDL code can be simulated using QuestaSim. To do this, DExIMA generates a script that compiles the code and runs the simulation of a UVM testbench, which is fundamental to generalize the tests of the LiM architectures.

⑤ *Back-annotation and .vcd file generation.* At the end of the RTL simulation, two VCD files are generated, containing the values of the signals for each time instant. The first one is useful for the back-annotated power estimation with DExIMA-Backend and contains the waveforms of the LiM blocks (the LiM Cells and IRLs). The second one contains the signals of the top-level entity, which connects the microprogrammed Control Unit to the LiM array and is useful for the bus performance estimation (step ⑧).

⑥ *Architecture description: DExIMA-Backend input file.* After the simulation, DExIMA-Backend is used to estimate the performance. An intermediate and custom DExIMA-Backend input file with extension **.dex** is generated that contains not only the structural description of the LiM but also crucial information for performance estimations.

⑦ *Architecture performance estimation.* Once the DExIMA-Backend input file is ready, DExIMA-CAD calls DExIMA-Backend by means of an embedded terminal and passes the **.dex** file. DExIMA-Backend estimates the performance and prints out the results of the LiM inside a human-readable file with extension **.dof** (DExIMA Output File).

⑧ *Bus perfomance estimation.* Bus impact can be optionally evaluated in the performance estimation process. In this case, DExIMA-Backend automatically instantiates a netlist made of several drivers, a distributed network of capacitors, and resistances, aiming at emulating a real bus. Then, DExIMA runs the netlist simulation using the Ngspice tool.

⑨ *Show DExIMA results.* At this point, the control of the flow returns to DExIMA-CAD, and the user can visualize the results in bar plots or tabular format.

⑩ *Comparison CPU-Mem CPU-Mem-LiM.* At the end of the LiM design flow, the user can compare the two solutions to evaluate the impact of the LiM in a classical context.

3.3.1 Near-Memory design phase

DExIMA aims to assist the user not only in the LiM architecture definition, but also in the Near-Memory design phase, which is quite similar to the LiM one. In fact, in this part, with the same modality exploited in the LiM case, the designer can implement a standard circuit employing the Schematic Capture, generate the VHDL code, estimate its performance with DExIMA-Backend, and view the results. However, the Near-Memory phase does not include an automatic testbench, so the performance results are estimated without the back annotation process, emulating a worst-case scenario.

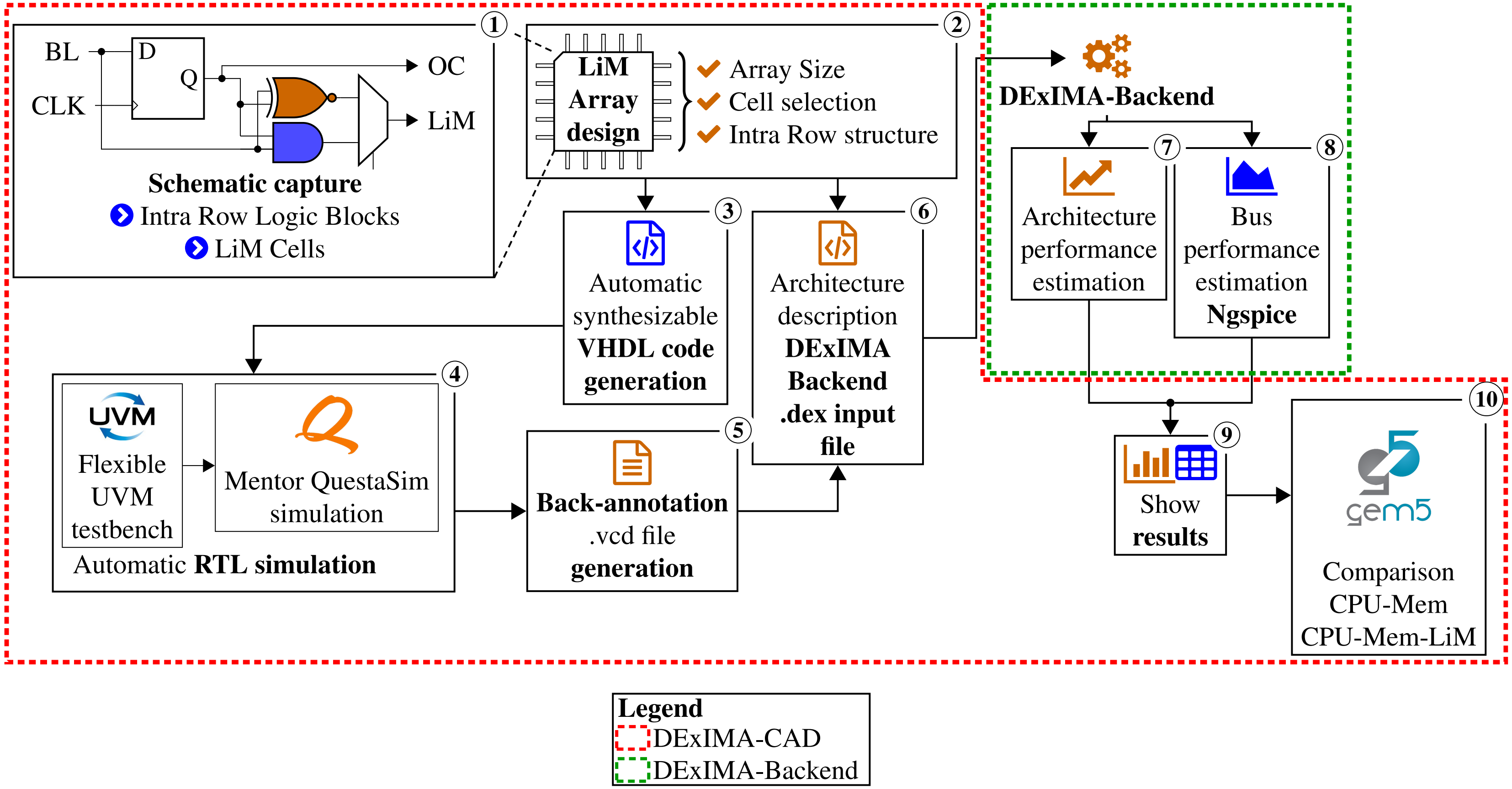

1
BL
CLK
D
Q
OC
LiM
Schematic capture
Intra Row Logic Blocks
LiM Cells
2
LiM Array design
Array Size
Cell selection
Intra Row structure
3
Automatic synthesizable VHDL code generation
6
Architecture description DExIMA Backend .dex input file
4
UVM
Flexible UVM testbench
Mentor QuestaSim simulation
Automatic RTL simulation
5
Back-annotation .vcd file generation
DExIMA-Backend
7
Architecture performance estimation
8
Bus performance estimation Ngspice
9
Show results
10
gem5
Comparison CPU-Mem CPU-Mem-LiM
Legend
DExIMA-CAD
DExIMA-Backend

Fig. 3.7 DExIMA high-level structure

3.4 Conclusions

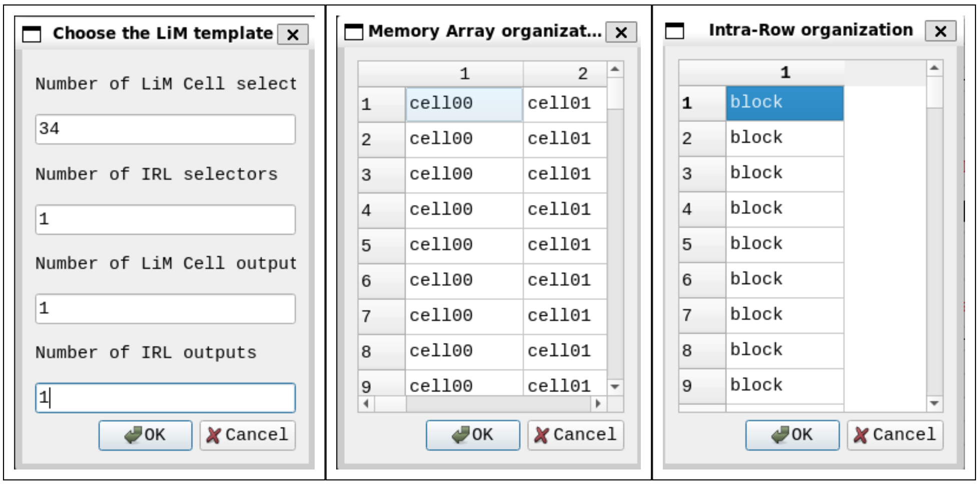
In this chapter, an initial overview of the LiM array architectural model and control logic used in DExIMA is provided. The architectural model was derived from past work and can be modified or expanded to implement other LiM models in the literature as well. The control unit is a microprogrammed machine that, together with the use of templates, is able to provide flexibility and programmability. Finally, an overview of the DExIMA tool is discussed, with an initial mention of the functionality and design steps.

Chapter 4

LiM design flow with DExIMA

Summary

This chapter presents the LiM design flow with DExIMA. The steps start from the definition of the LiM array with the schematic mode and go towards the definition of the LiM algorithm, the simulation, performance estimations, and the comparisons with a standard CPU-Mem system. An example is proposed to exhaustively explain the entire flow, which consists of a very simple architecture capable of performing three basic logic operations inside the cell (XNOR, NAND, NOT) between the cell's content and external data and additions for each line.



Number of LiM Cell select
34
Number of IRL selectors
1
Number of LiM Cell output
1
Number of IRL outputs
1

	1	2
1	cell00	cell01
2	cell00	cell01
3	cell00	cell01
4	cell00	cell01
5	cell00	cell01
6	cell00	cell01
7	cell00	cell01
8	cell00	cell01
9	cell00	cell01

	1
1	block
2	block
3	block
4	block
5	block
6	block
7	block
8	block
9	block

(a) (b) (c)

Fig. 4.1 (a) LiM template example. (b) LiM array cell pattern. (c) IRL organization.

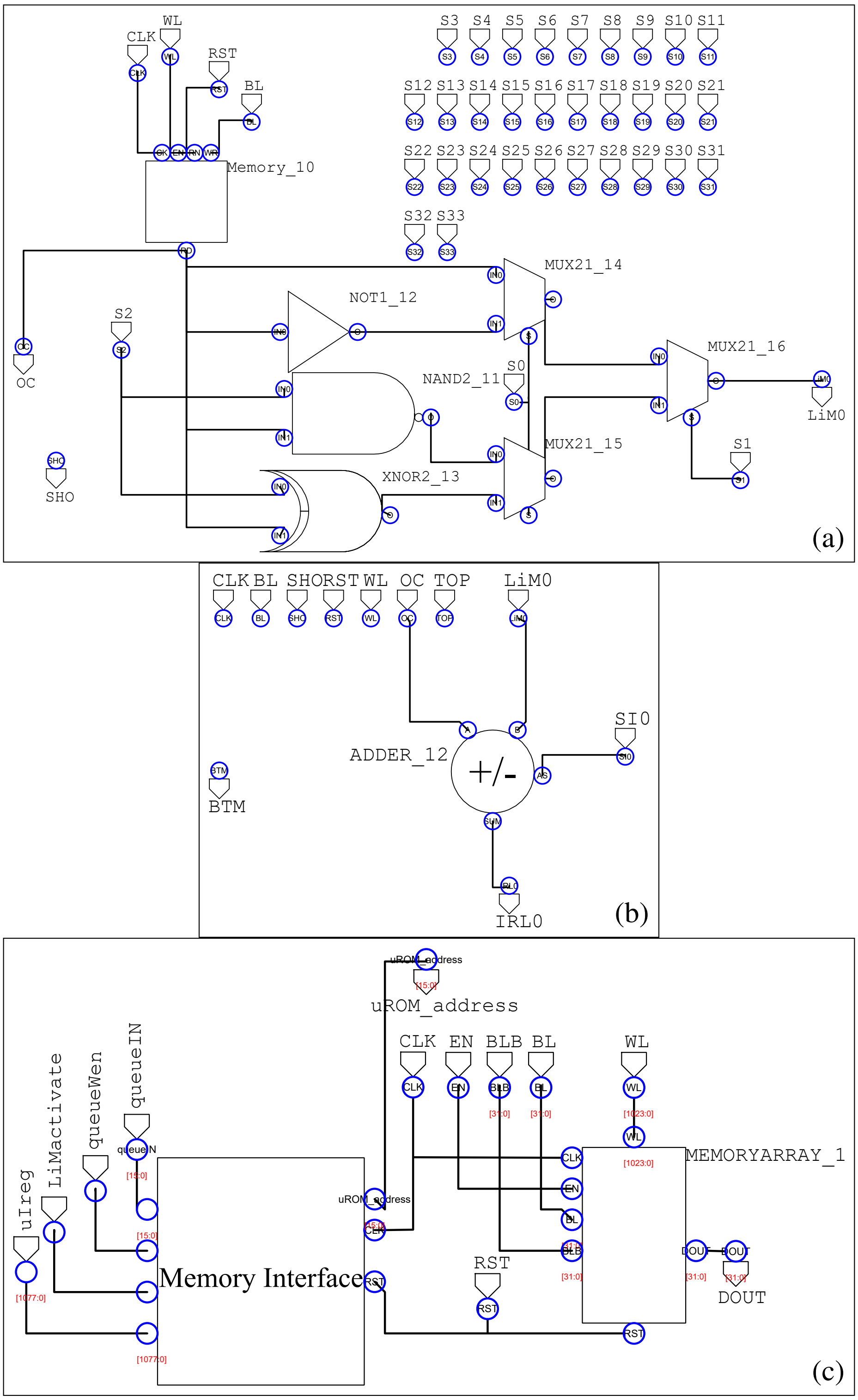


Fig. 4.2 (a) LiM Cell and (b) IRL block examples. (c) Top-level entity scheme.

4.1 Definition of the LiM template

The first step is to define the number of pins required inside the LiM and IRL blocks. These pins are used as control inputs and data signals in this particular example, but they can also be used as control-only signals. Since the LiM Cell computes one of three possible functions, a 4-1 multiplexer is required, and it should be able to select between the memory output, the XNOR, NAND, and NOT functions. The number of selectors for the LiM Cell equals 2 (for piloting the 4-1 multiplexer) plus 32 other pins dedicated for each bit of external data. Moreover, one LiM output is required to deliver the LiM Cell function to the IRL blocks. The IRL blocks require one selector to pilot the sum/subtract pin of the adder and one output. The LiM template is set by clicking on the `Templates...` button, which shows the dialog in Fig. 4.1 (a).

4.2 Definition of the LiM Cell

The LiM Cell is designed starting from the specifications, and it should be capable of performing XNOR, NAND, and NOT operations, so the cell must contain these gates. The architecture of the 0-th bit LiM Cell is shown in Fig. 4.2 (a).

4.3 Definition of the IRL

The IRL block is simply an adder. DExIMA provides an adder capable of performing both additions and subtractions, so the $\overline{\text{sum}}$/subtract pin ("AS" in the scheme of Fig. 4.2 (b)) is connected to the IRL selector SI0, that is fixed to 0. LiM Cell and IRL are saved into two files with the names `cellxx.lim` and `block.irl`, respectively, where `xx` is the bit index of the data inside the LiM array.

4.4 LiM array definition

By switching to the "LiM architecture" tab, it is possible to design the top-level entity containing the LiM array and the controlling part. Once instantiated, the user can click on `Edit` button , select the LiM and choose the array sizes, cell and IRL patterns. In this example, a 32-bit with 128 rows LiM memory is chosen, and each LiM Cell and IRL block are chosen as the ones shown in Fig. 4.2 (a-b) by using the dialogs shown in Fig. 4.1 (b-c). After the definition of the LiM array structure, the program automatically instantiates the Memory Interface block that acts as the Control Unit of the LiM architecture: the user has to manually connect all the pins, reaching the final structure shown in Fig. 4.2 (c).

4.5 Definition of the uRAM content

At this point, the algorithm can be defined by writing the values of the selectors of the chosen LiM template inside the uRAM. DExIMA provides a tool that allows defining the values of the selectors for each instruction, depending on the operations required. Considering, for example, the following pseudo-code:

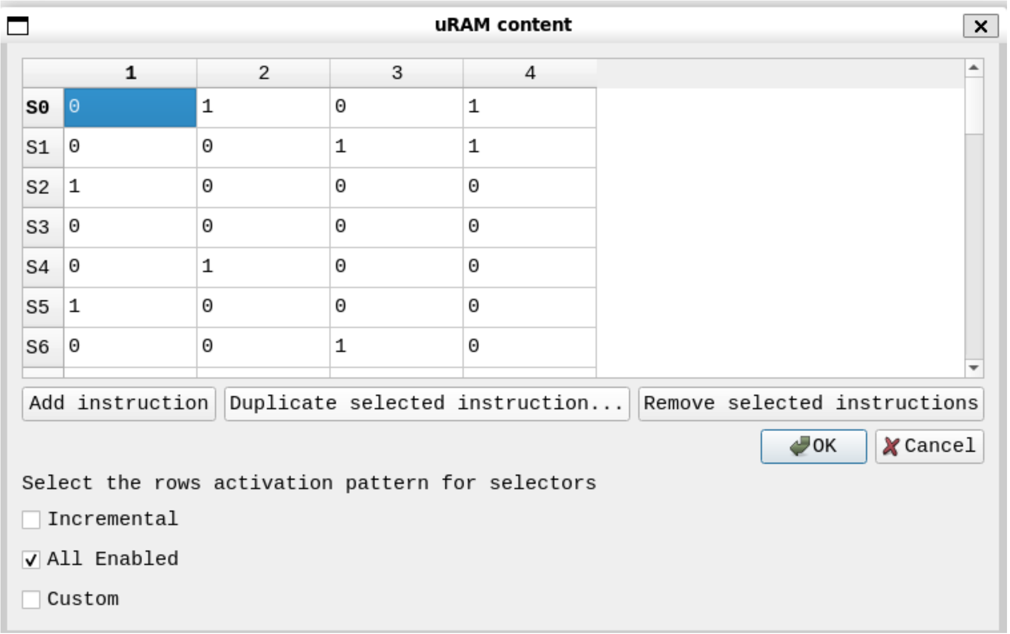
```
for(int i = 0; i < 128; i++)
RES[i] = MEM[i] + MEM[i];
for(int i = 0; i < 128; i++)
RES[i] = MEM[i] + NOT(MEM[i]);
EXT_DATA = 68624;
for(int i = 0; i < 128; i++)
RES[i] = MEM[i] + !(MEM[i] & EXT_DATA);
EXT_DATA = 2101376;
for(int i = 0; i < 128; i++)
RES[i] = MEM[i] + !(MEM[i]^EXT_DATA);
```

Listing 7 Pseudo-code of an algorithm implementable by the LiM architecture example.

The LiM array can speed up each iteration cycle since it is able to implement in parallel each instruction proposed in the algorithm. By means of the `Estimate performance -> Define uRAM selectors` tool, 4 LiM instructions are implemented, as shown in Fig. 4.3: each column refers to a specific instruction in which the selector values are assigned by writing them inside the table. Moreover, the pattern of the activation of the rows can be selected with the checkboxes: `Incremental` mode means that the rows are enabled incrementally, starting from the 0-th row towards the N-1-th row, where N is the number of rows inside the memory; `All Enabled`, as the name suggests, means that all rows are enabled at the same time, so the operations are performed in parallel on the entire array; `Custom` allows to specify a custom and irregular activation pattern of the rows (e.g., rows 0, 2, 5, 103, etc.). The uRAM content is saved inside a **.csv** file called `uRAM.csv`, in which the rows activation pattern and the selector values are reported for each instruction.

```
All Enabled,All Enabled,All Enabled,All Enabled
0,1,0,1
0,0,1,1
1,0,0,0
...
```

Listing 8 Example of a `uRAM.csv` file content.


uRAM content
Add instruction
Duplicate selected instruction...
Remove selected instructions
OK
Cancel

	1	2	3	4
S0	0	1	0	1
S1	0	0	1	1
S2	1	0	0	0
S3	0	0	0	0
S4	0	1	0	0
S5	1	0	0	0
S6	0	0	1	0

Select the rows activation pattern for selectors

☐ Incremental

☑ All Enabled

☐ Custom

Fig. 4.3 Selector values defined for the LiM architecture example using the `Define uRAM selectors` tool.

4.6 Clock and default toggle rate definitions

The architecture is ready to be simulated. Before starting the simulation, by clicking on `Estimate performance -> Create clock`, the clock period and the default toggle rate of the pins are requested. The clock period is useful for simulating and estimating the performance of the architecture, while the default toggle rate is required only for worst-case performance estimation without simulation and back-annotation processes.

4.7 Simulation

The simulation starts by clicking on `Estimate performance ->` Simulate circuit... : DExIMA generates the simulation script and starts QuestaSim that runs the UVM-based testbench of the architecture, as discussed in deep in chapter 6. The precharging part is executed at the beginning of the program: after the assertion of the reset signal, the UVM testbench starts filling the LiM array, piloting the internal signals of each LiM Cell according to the schematic and timing diagrams shown in Fig. 4.4 (a-c). Subsequently, the computing part starts, and the selectors are directly driven by the control unit, according to the LiM instructions reported in Fig. 4.3. In Fig. 4.4 (d), the timing diagram of the LiM Cell internal signals at (row, col) = (0,0) are considered as reference: at the 10-th time instant, the algorithm starts, and S2 is asserted to 1, while S0 and S1 are both equal to 0, so the signal `MUX21_14/IN0` and `MUX21_16/IN0` are both equal to `Memory_10/RD`, so the final operation on the `IRL0` signal is `MEM[i]+MEM[i]`. Next, S2 and S0 switch to 0 and 1, respectively, implementing the operation `MEM[i]+NOT(MEM[i])`, in fact `MUX21_14` selects `NOT1_12`'s output and `MUX21_16/IN0` is equal to `NOT(Memory_10/RD)`. Similarly, in the 14-th time instant, S0 and S1 change to 0 and 1, respectively, implementing `MEM[i]+!(MEM[i]&EXT_DATA)`, where the `EXT_DATA` is provided with the S2-S33 selectors. Finally, by switching S0 to 1, the last operation `MEM[i]+!(MEM[i]^EXT_DATA)` takes place at the 16-th time instant.

4.8 Performance estimation

At this point, the user can estimate the performance of the circuit by clicking on Estimation with DExIMA... , which generates the input **.dex** file, runs DExIMA-Backend, and performs the estimations. DExIMA-Backend input file and its working principle are explained in deep in chapter 7. In detail, the operations performed in this step are:

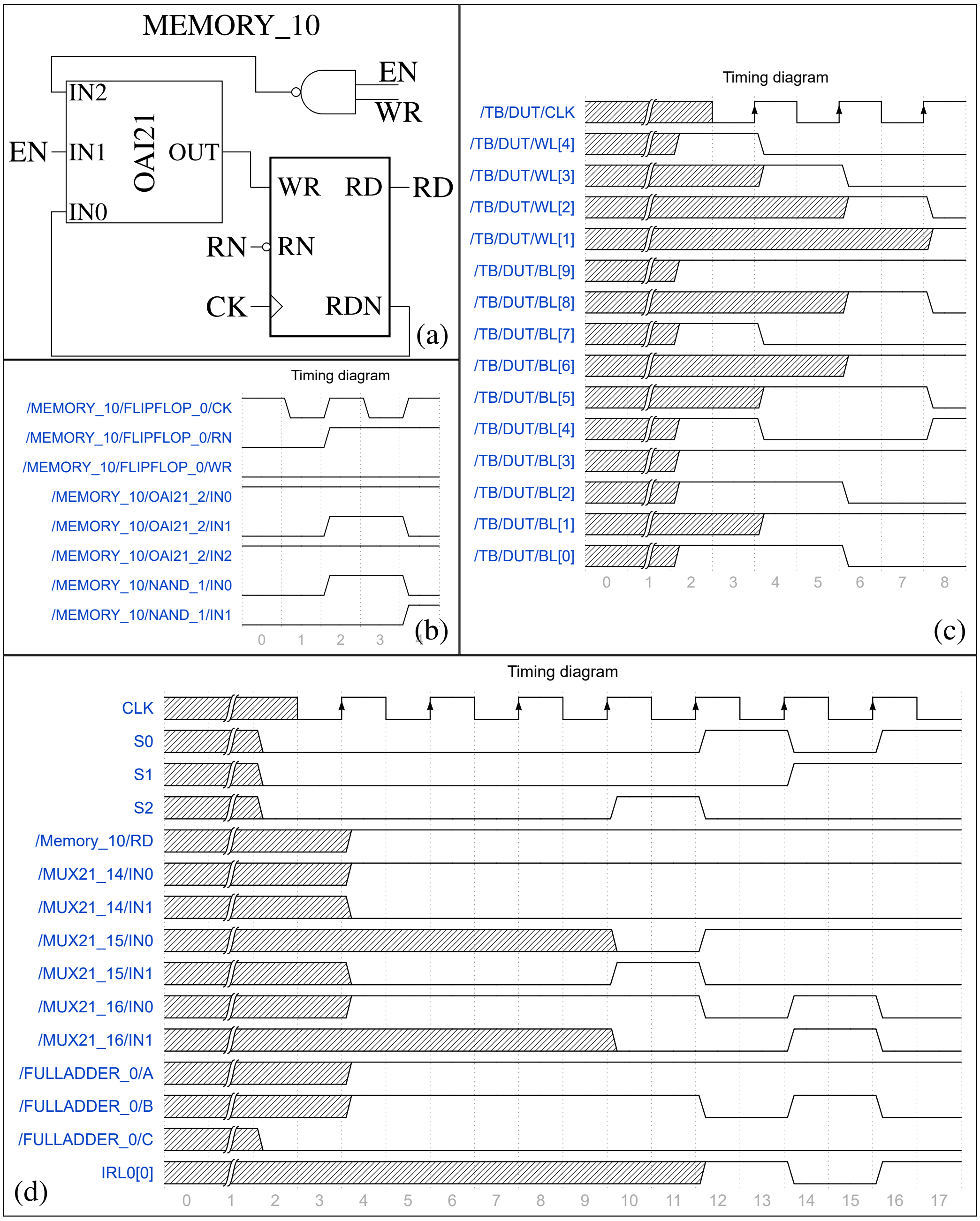
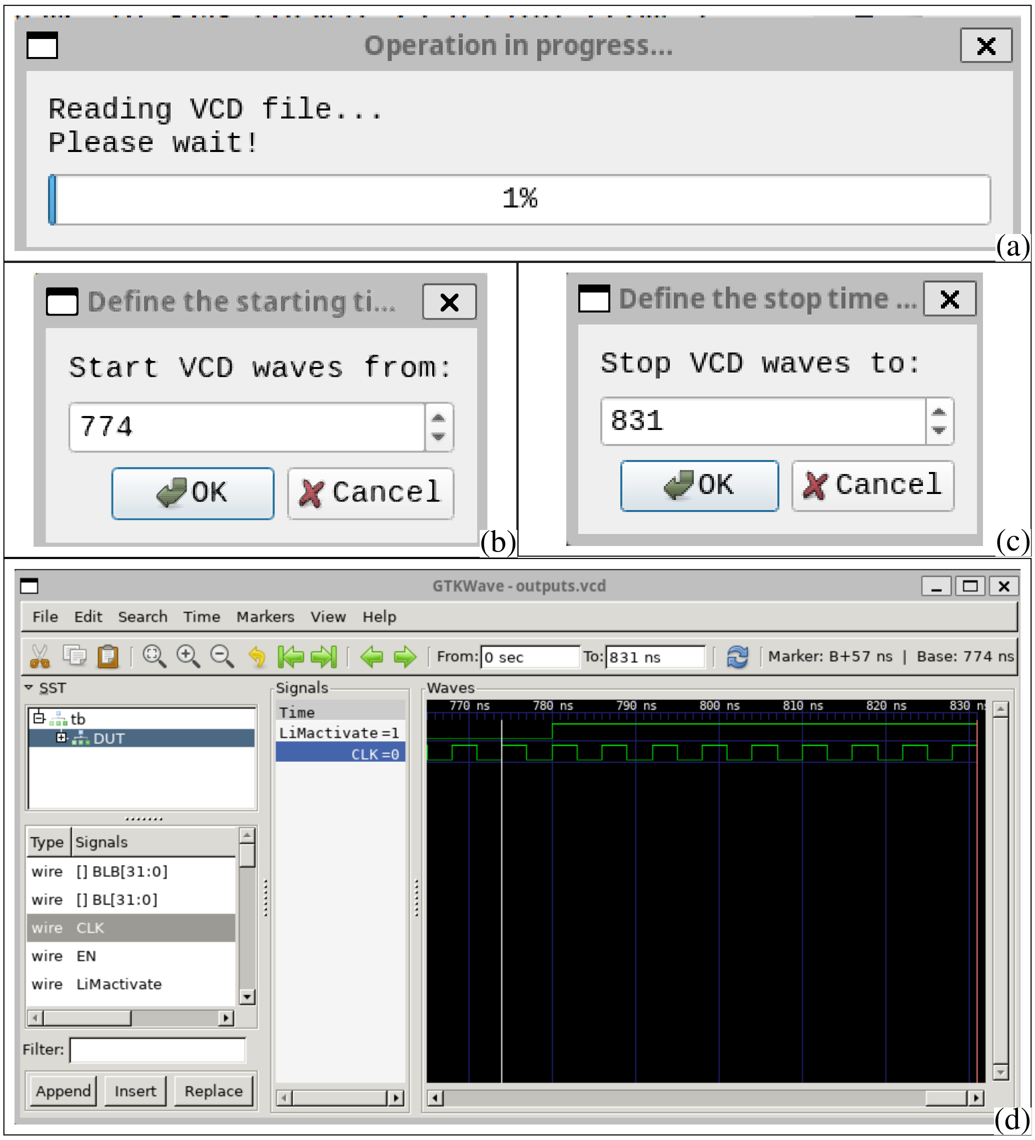


Fig. 4.4 (a) Schematic of the Flip-Flop-based memory cell. (b) Timing diagram example obtained from the output **.vcd** file: LiMCELL_0_0 data precharging during the initial phase. (c) Top-level testbench waveforms during the data precharge phase. (d) Example of the computational phase.

- Architecture translation from the schematic view and description inside the DExIMA-Backend input file.

- Parsing of the VCD `outputs.vcd` and `tb.vcd` files. This process can be very long if the architecture is particularly complex: for this reason, a dialog equipped with a progress bar (shown in Fig. 4.5 (a)) communicates to the user the percentage of completion of the operation, indicating that the program is busy. Moreover, the user has the possibility to select part of the algorithm



Operation in progress...

Reading VCD file...
Please wait!

1%

Define the starting ti...

Start VCD waves from: 774

Define the stop time ...

Stop VCD waves to: 831

GTKWave - outputs.vcd

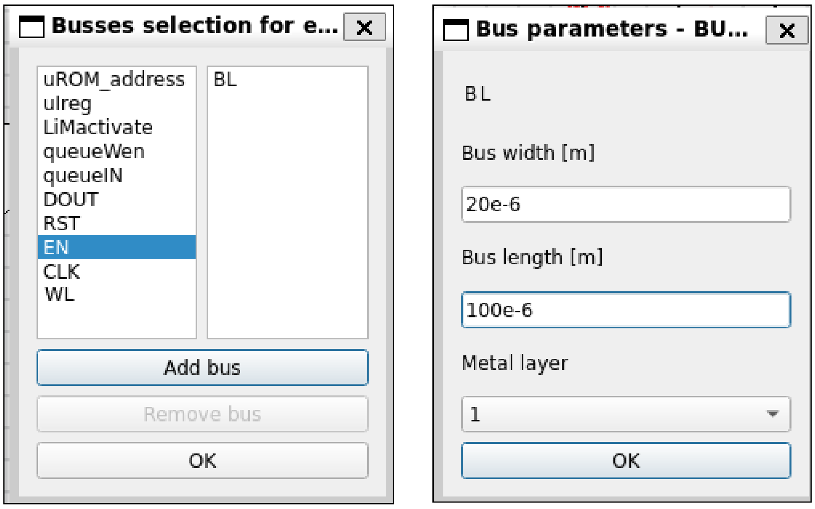
From: 0 sec To: 831 ns | Marker: B+57 ns | Base: 774 ns

Type	Signals
wire	[] BLB[31:0]
wire	[] BL[31:0]
wire	CLK
wire	EN
wire	LiMactivate

Fig. 4.5 (a) VCD parsing phase: progress bar indicating the percentage of completion of the operation. (b-c) Start-stop time instants selection dialogs. (d) Start and stop times selection for VCD parsing with GTKWave.

execution by means of the dialogs in Fig. 4.5 (b-c), where start and stop time instants can be specified. These are expressed with the same unit as the clock period and allow cutting only the portion of waveforms specified between the start and stop times. This feature is particularly useful to evaluate the impact of the individual contributions performed by the algorithm (i.e., memory precharging, computational part, etc.) and to reduce the computational time of DExIMA-Backend. The start and stop times can be defined with the help of waveform viewers like GTKWave [133], as shown in Fig. 4.5 (d). Particularly useful is the `LiMActivate` signal, which indicates when the LiM is used in computing mode.

- Selection of the bus lines. Once the waveforms of the signals are extrapolated from the VCD files, the user can select the bus(es) to estimate. The estimation is performed by means of Ngspice [94] tool, which enables the evaluation of the impact of the selected bus lines in the design, both internally and externally the LiM array, as explained in deep in subsection 7.3.9. The selection of the bus(es) and their physical characteristics are defined by the dialogs shown in Fig. 4.6.



BL

Bus width [m]: 20e-6

Bus length [m]: 100e-6

Metal layer: 1

Fig. 4.6 Bus(es) selection and parameters definition.

After these steps, DExIMA-Backend starts evaluating the architecture, showing on the console the output reported in Listing 9.

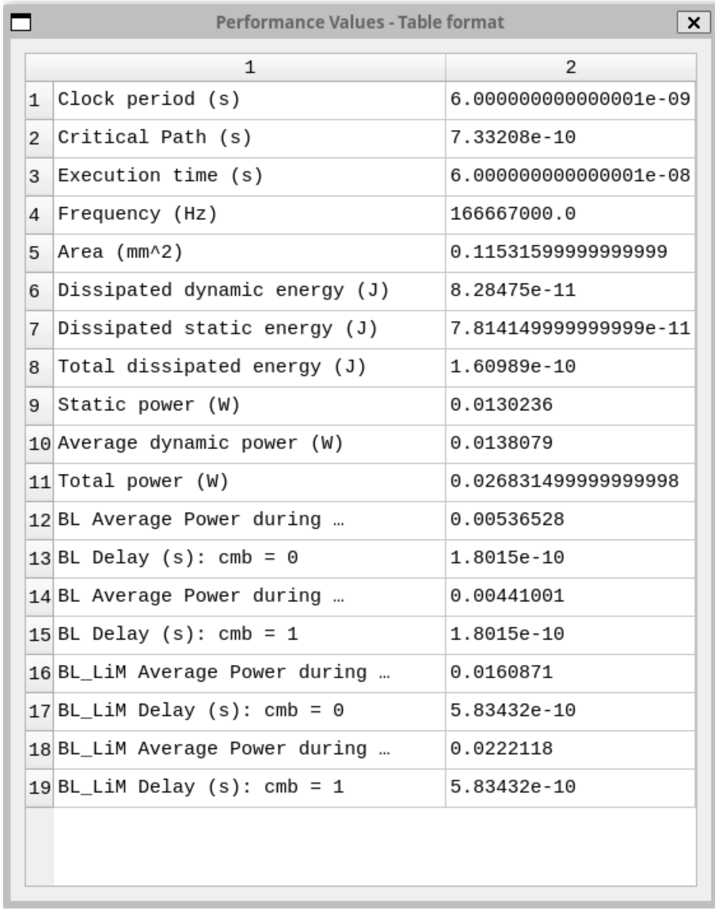
```
#############################################
#             DExIMA-Backend                #
#Design Explorer for In Memory Architecture#
#      Developed by VLSI Lab @Polito        #
```

```
############################################
Compiling --> input.dex
Parsing constants...complete
Loading Technology...complete
Parsing init...complete
Parsing MEMORYARRAY_1...complete
Parsing map...complete
Parsing instructions...complete
Parsing code...complete
Simulating BL - 32 bits [W = 1e-05; L = 0.01; Metal = 1]

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************
Parsing bus...complete
Printing log file...complete

Compilation time: 1.1421 min
Compilation finished successfully!

Simulating --> input.dex
Wiring the memories...complete
Encoding the Architecture...complete
Computing gates performance...
Computing performance of the design's modules...

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************
Computing performance of the LiM cells...

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************
Simulating internal LiM Bus(es)
BL - 32 bits [W = 1e-05; L = 0.000336112; Metal = 1]

0%   10   20   30   40   50   60   70   80   90   100%
|----|----|----|----|----|----|----|----|----|----|
***************************************************
complete
Computing area and static power...complete
Computing Instructions performance...complete
Computing Algorithm performance...complete
Printing output file...complete

Simulation time: 1.31147 min
Simulation finished successfully!
```

Listing 9 Output of DExIMA-Backend shown in the embedded console.

4.9 Visualization of the results

After the estimation performed by DExIMA-Backend, the user can visualize the results both in plot and tabular formats by clicking on `Show DExIMA results...` , as shown in Fig. 4.7 and Fig. 4.8. In this example, the BL bus is evaluated: its



	1	2
1	Clock period (s)	6.000000000000001e-09
2	Critical Path (s)	7.33208e-10
3	Execution time (s)	6.000000000000001e-08
4	Frequency (Hz)	166667000.0
5	Area (mm^2)	0.11531599999999999
6	Dissipated dynamic energy (J)	8.28475e-11
7	Dissipated static energy (J)	7.814149999999999e-11
8	Total dissipated energy (J)	1.60989e-10
9	Static power (W)	0.0130236
10	Average dynamic power (W)	0.0138079
11	Total power (W)	0.026831499999999998
12	BL Average Power during …	0.00536528
13	BL Delay (s): cmb = 0	1.8015e-10
14	BL Average Power during …	0.00441001
15	BL Delay (s): cmb = 1	1.8015e-10
16	BL_LiM Average Power during …	0.0160871
17	BL_LiM Delay (s): cmb = 0	5.83432e-10
18	BL_LiM Average Power during …	0.0222118
19	BL_LiM Delay (s): cmb = 1	5.83432e-10

Fig. 4.7 Tabular format results after DExIMA-Backend estimation.

contribution is split in "BL", which refers to the bus located outside the LiM array

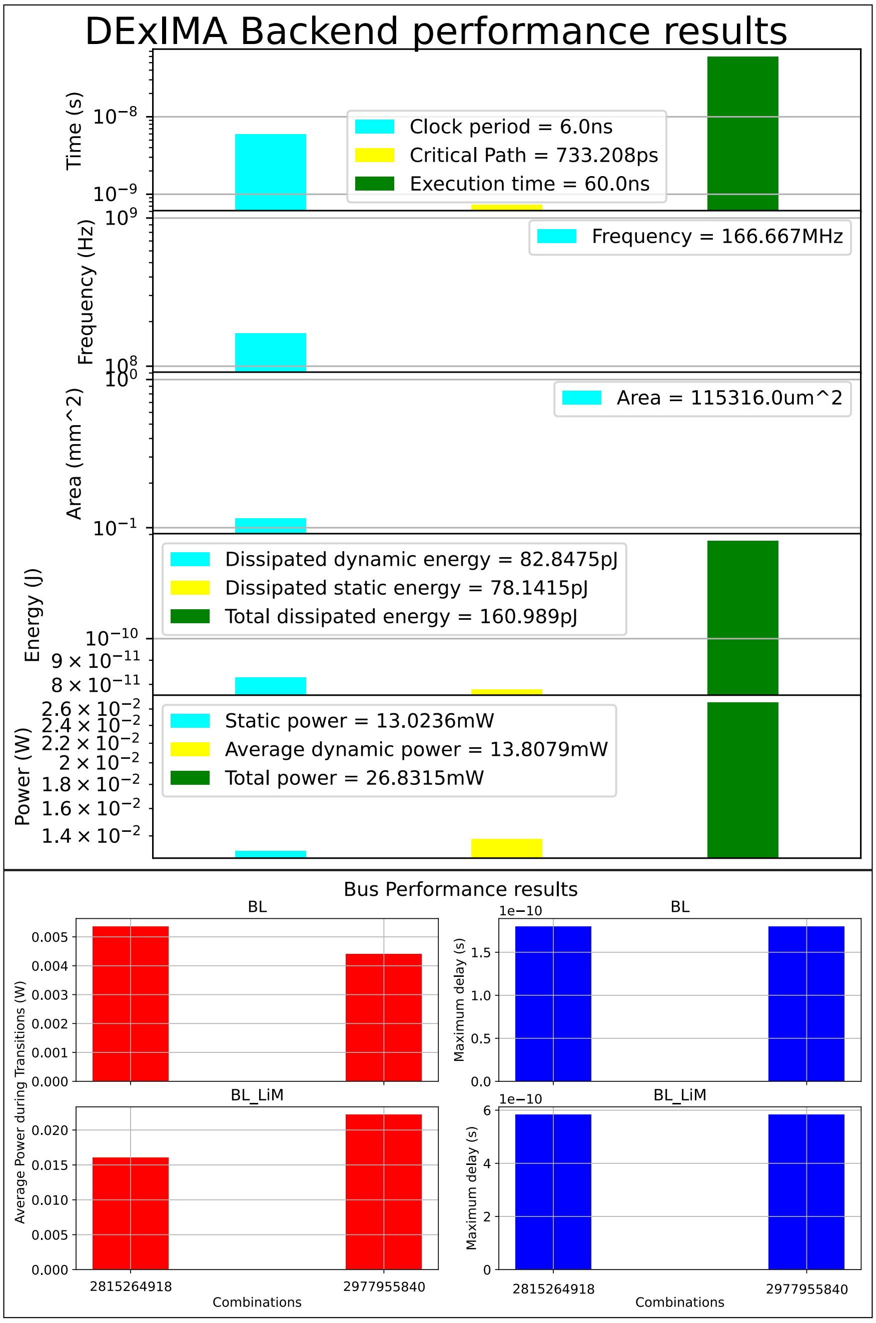


Fig. 4.8 Plot format results after DExIMA-Backend estimation.

(shown in Fig. 4.2 (c)) and "BL_LiM", i.e., the internal BL bus connected to each LiM Cell and IRL. Each number on the abscissas represents a combination of bits expressed in unsigned format.

4.10 Comparison CPU-Mem and CPU-Mem-LiM.

The user can evaluate the impact of the LiM solution on a classical von Neumann structure with the following steps:

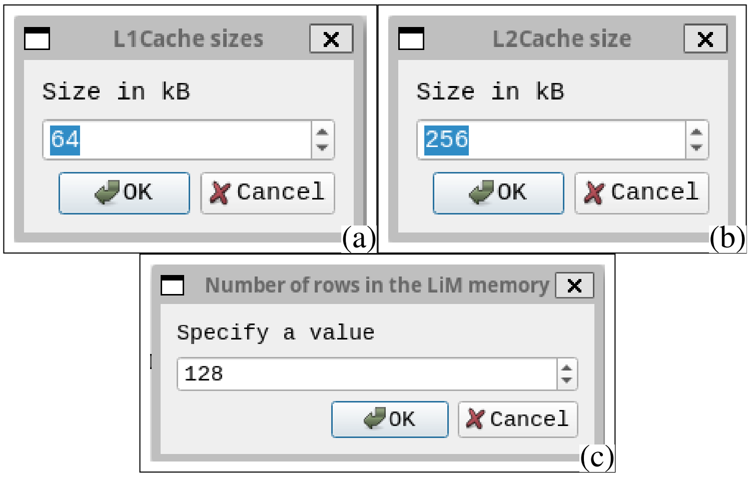
- Click on the "Comparison CPU-Memory" tab in the main window (Fig. 3.5 (a)), which switches the main widget from the schematic editor to a text editor.
- In the text editor, the user can write the C code that implements the reference algorithm on a classical von Neumann architecture, so without considering the LiM accelerator. In this example, the C algorithm is the following one:

```c
#define N 68
int main()
{
    volatile int MEM[128] = {N};
    volatile int EXT_DATA = 0;
    volatile int RES[128];
    for(int i = 0; i < 128; i++) RES[i] = MEM[i] + MEM[i];
    for(int i = 0; i < 128; i++) RES[i] = MEM[i] + !(MEM[i]);
    EXT_DATA = 68624;
    for(int i = 0; i < 128; i++) RES[i] = MEM[i] + !(MEM[i] & EXT_DATA);
    EXT_DATA = 2101376;
    for(int i = 0; i < 128; i++) RES[i] = MEM[i] + !(MEM[i]^EXT_DATA);
    return 0;
}
```

Listing 10 Algorithm example: classical CPU-Memory implementation.

All variables are declared as `volatile` to avoid optimizations.

- Run Gem5 [97] software to simulate the algorithm on the CPU-Memory architecture. By clicking on "Run Gem5", DExIMA asks the user to provide useful parameters like the size of the caches of the reference system (Fig. 4.9 (a-b)) and the number of LiM memory rows (Fig. 4.9 (c)). The program needs this last parameter to generate the equivalent C code of a system equipped with the LiM solution that emulates the data transfer from the main memory to the LiM architecture. The code is reported in Listing 11.



L1Cache sizes

Size in kB

64

L2Cache size

Size in kB

256

Number of rows in the LiM memory

Specify a value

128

Fig. 4.9 Steps required for CPU-Mem and CPU-Mem-LiM comparison. (a) L1 I/D cache sizes. (b) L2 shared cache size. (c) Number of LiM memory rows.

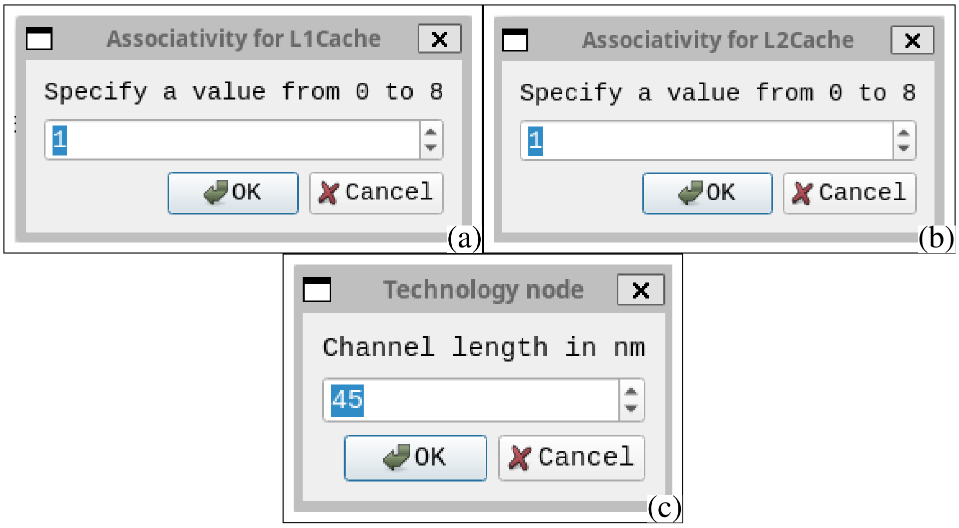
```
#include <stdio.h>
int main(){
volatile int memory_content[128] = {0};
volatile int data;
for(int i = 0; i < 128; i++){
    data = memory_content[i];
}
return 0;
```

Listing 11 Automatically generated LiM algorithm emulating the data transfer from the main memory inside the LiM.

Both codes are simulated with Gem5, which directly provides their statistics (executed instructions, number of memory accesses, etc.).

- Run Cacti by HP [91] to estimate the energies of the caches. By clicking on "Cacti by HP" (Fig. 3.5 (b)), DExIMA asks the user with the dialogs shown in Fig. 4.10 to specify the technology node and the cache associativities to estimate the memory performance properly. From now on, everything is ready for the CPU-Mem and CPU-Mem-LiM comparison.

The first comparison is the number of instructions executed in the two solutions. By clicking on Plot instructions of the CORE , two bar charts are shown, reporting the estimation of the occurrences of each instruction executed inside the processor, as shown in Fig. 4.11. These plots show that, in general, the LiM implementation requires fewer instructions (especially memory operations). However, it is impor-



Specify a value from 0 to 8

1

Specify a value from 0 to 8

1

Channel length in nm

45

Fig. 4.10 Steps required for the performance of the caches. (a) L1 I/D cache associativity. (b) L2 shared cache associativity. (c) Technology node.

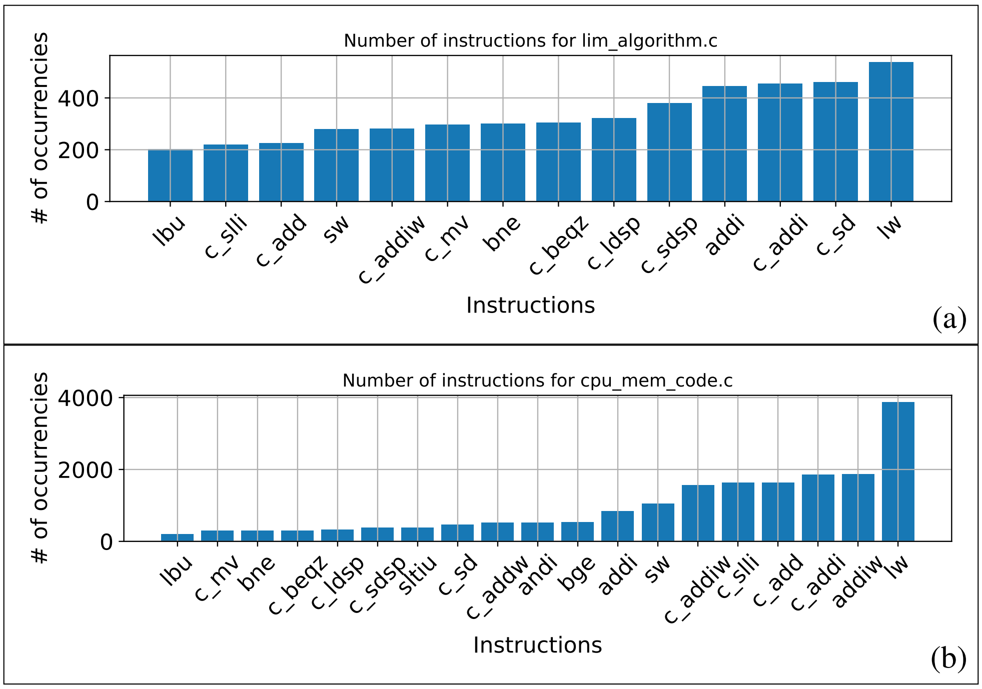


Fig. 4.11 Estimation of the number of instructions for both CPU-Mem-LiM (a) and CPU-Mem (b) codes.

tant to consider that the computational part performed by the LiM still needs to be considered in this estimation. The next step consists of extrapolating the memory statistics, particularly the number of memory accesses. Again, Gem5 directly provides these data by clicking on `Plot instructions of the CORE` and the most relevant ones are plotted by DExIMA, as shown in Fig. 4.12. The final step consists

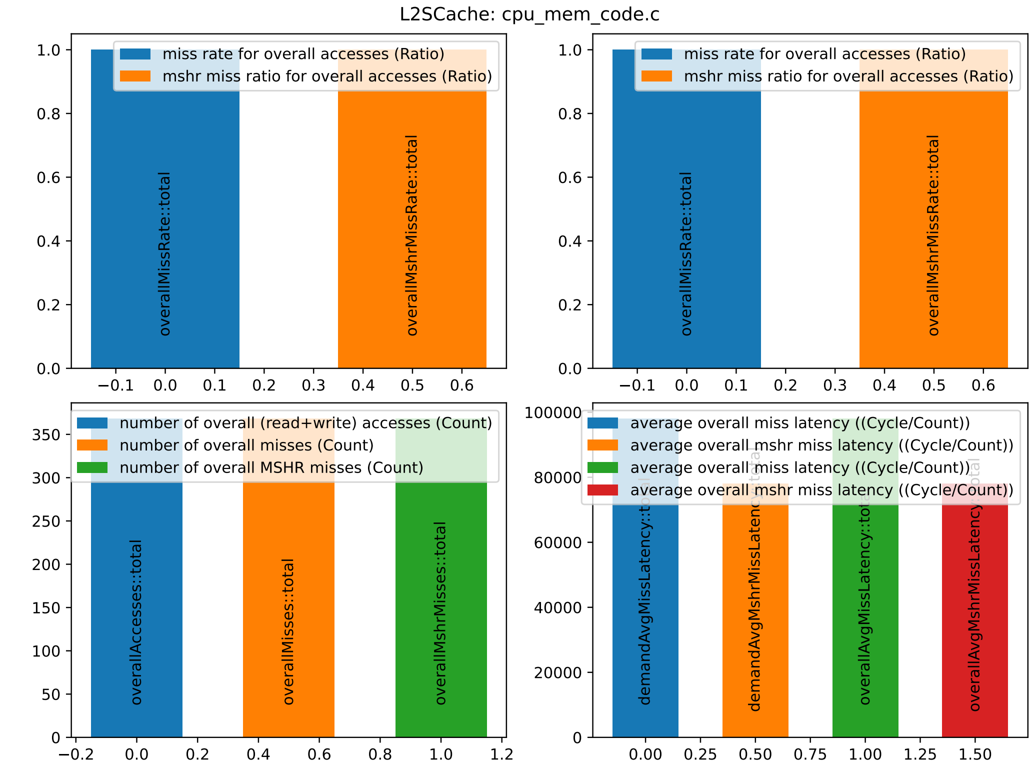


Fig. 4.12 L2SCache: memory statistics.

of effectively comparing the CPU-Mem and CPU-Mem-LiM systems. By clicking on `Compare CPU-Mem and CPU-Mem-LiM` , DExIMA considers different figures of merit, as shown in Fig. 4.13: CPU execution time, cache accesses, caches energy, LiM power-delay product and LiM execution time.

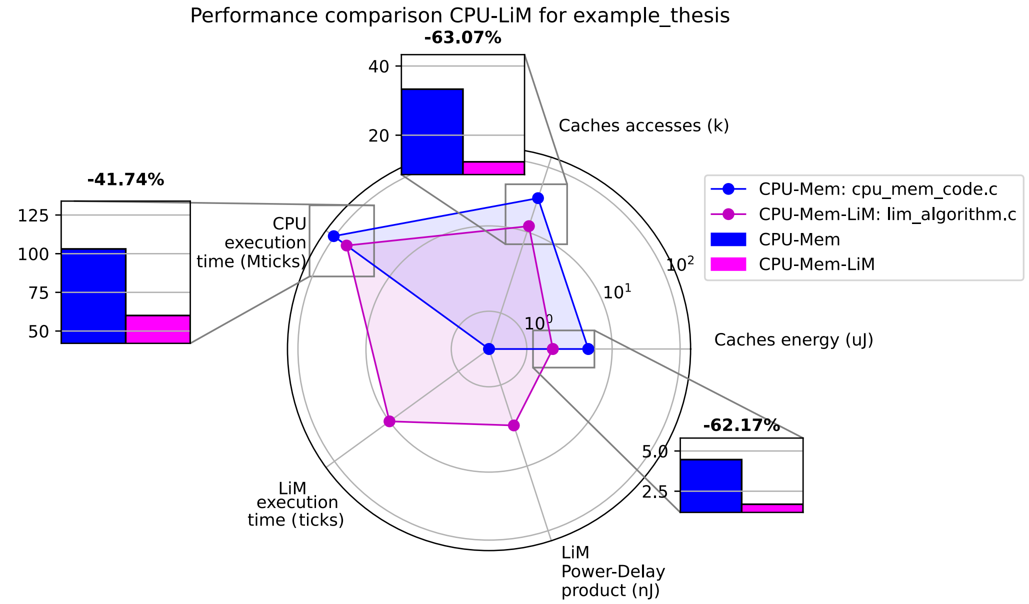


Fig. 4.13 Final comparison between CPU-Mem and CPU-Mem-LiM systems.

They are plotted in a logarithmic radar chart to emphasize the differences.

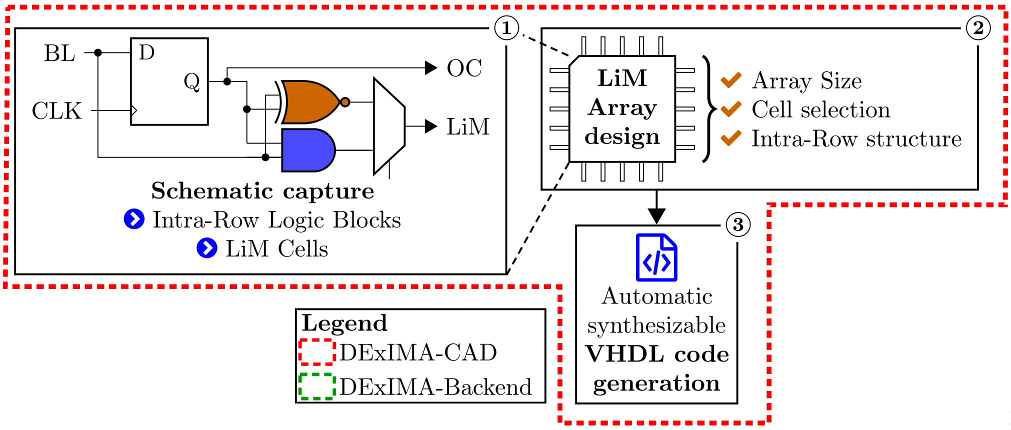
4.11 Conclusions

DExIMA is intended as a tool aiming at helping the designer with the Beyond von Neumann approach, focusing on the Logic-in-Memory paradigm. The tool is a full-fledged CAD realized in Python and C++, structured in a hierarchical way, starting from the definition of the LiM Cell, the IRL, and the top-level entity. All the other steps are automatized by DExIMA, from the definition of the control flow, the RTL simulation, the performance estimation with an ad-hoc DExIMA-Backend, and the comparisons with a standard RISC-V-based CPU-Memory system.

Chapter 5

Front-end code description: DExIMA-GUI

Summary



;

This chapter reports the code structure of the DExIMA front-end. In the writing of this thesis, the terms front-end and DExIMA-GUI are used interchangeably. Main codes are reported for each project's folder, specifying their functionalities. The Graphical User Interface of DExIMA is coded with Python using PyQt5 framework [134], based on the popular Qt framework for C++, specialized for creating user interfaces. Inside the DExIMA project, the files are organized as shown in Fig. 5.1. The main code is located in the `CODE` *directory, apart from the* `DExIMABackend` *folder that contains the C++ code for the DExIMA-Backend estimator part. The*

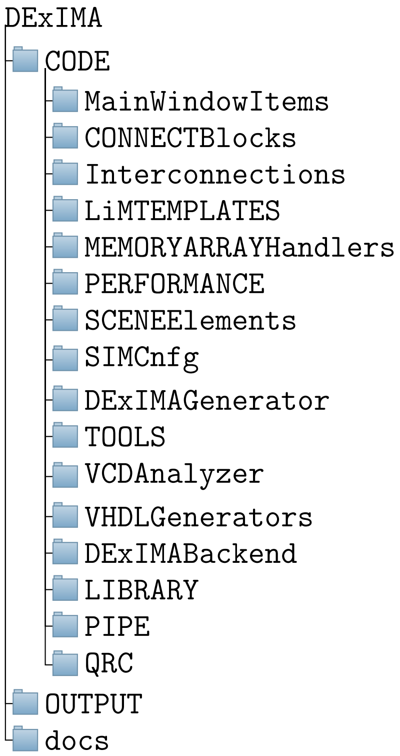


Fig. 5.1 Folders organizations of the DExIMA project.

Python code implementing the front-end of DExIMA software is distributed among different subdirectories, each referring to a specific functionality.

5.1 MainWindowItems

The Python codes implementing the main window graphical interface and functionalities are located inside `MainWindowItems` folder, which content is shown in Fig. 5.2. The main file is called `MainWindow.py`, which creates the main window interface of DExIMA and implements all front-end functions. `MainWindow.py` embeds all Python files of Fig. 5.2, which functionalities are explained in the following.

5.1.1 Graphical elements

- `Console.py` embeds a Linux terminal inside DExIMA main window. The Linux terminal is fundamental to execute external programs like QuestaSim for the simulation, Synopsys Design Compiler for the synthesis, and DExIMA-

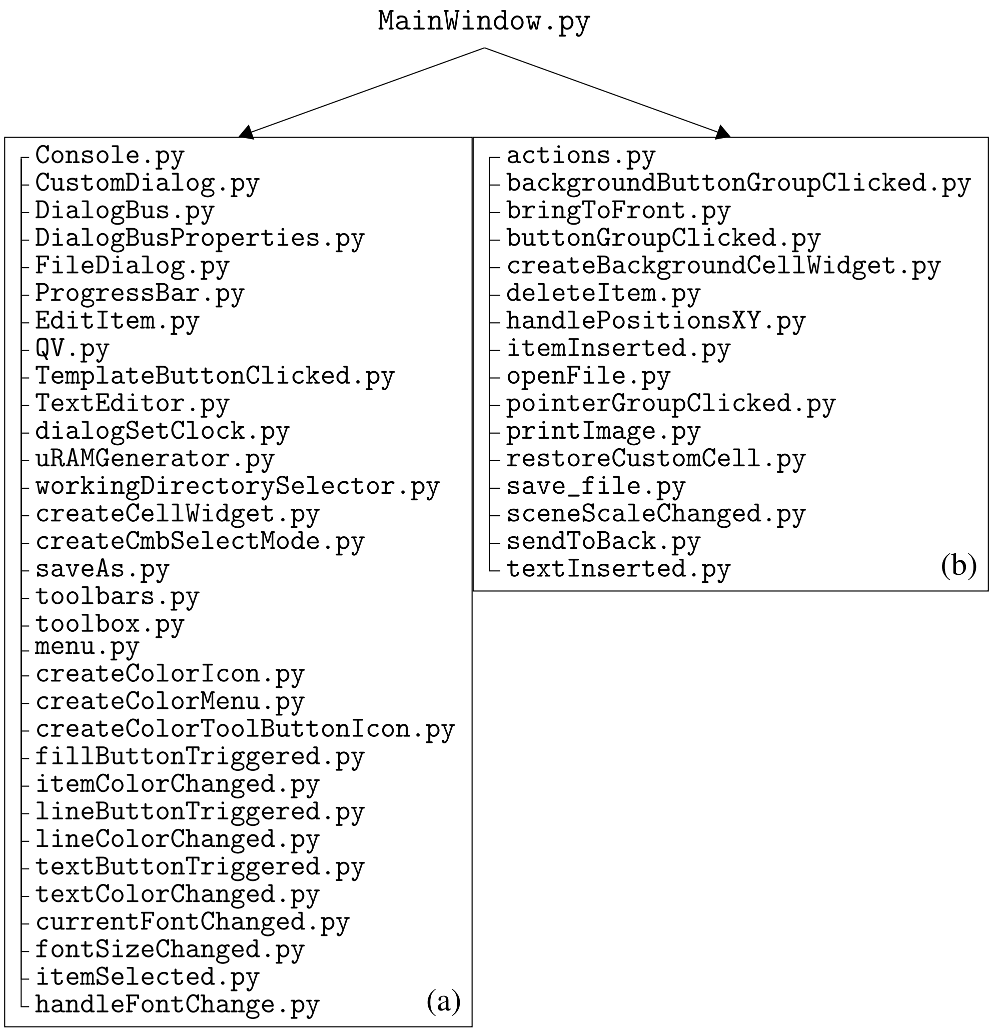


Fig. 5.2 Contents of the MainWindowItems folder. (a) Dialogs, progress bars, and graphical widgets and elements. (b) Python files performing specific routines or functions related to the graphical part.

Backend for the performance estimation and to show contextual messages informing the user on the steps performed and their completion. `Console.py` comprises a terminal, particularly a **rxvt-unicode** terminal, and a `QTextEdit` widget used to send user commands (the terminal itself is in read-only mode). The code implementing the embedded terminal is the following:

```python
class EmbTerminal(QtWidgets.QWidget):
    def __init__(self, parent=None):
        super(EmbTerminal, self).__init__(parent)
        self.process = QtCore.QProcess(self)
        self.terminal = QtWidgets.QWidget(self)
        #...
        self.process.start('urxvt -embed ' + str(int(self.winId())) +
                           ' -fn \"xft:Bitstream Vera Sans Mono:pixelsize=20\"' +
                           ' -bg lightgray -e sh -c \"./CODE/PIPE/terminal_routine.sh\"')

```

Listing 12 Snippet of code for the embedded **rxvt-unicode** terminal.

The EmbTerminal class, when instantiated, creates a QProcess, which starts **rxvt-unicode** as `urxvt -embed $WINDOW_ID ... -e sh -c "./CODE/PIPE/terminal_routine.sh"`, where the $WINDOW_ID is the identification number of the QWidget. The **-e** flag specifies a command to run when **urxvt** is launched: the program is an **sh** script called terminal_routine.sh, located in /CODE/PIPE/ folder, which content is shown in Listing 13.

```
1  #!/bin/bash
2  SCRIPT_DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" &> /dev/null && pwd )"
3  export SCRIPT_DIR
4  pipe=$SCRIPT_DIR/testpipe
5  if [[ ! -f $pipe ]]; then
6      mkfifo $pipe > .tmp
7  fi
8  echo "DExIMA is ready!"
9  while true
10 do
11     if read line <$pipe; then
12         eval $line
13     fi
14     sleep 0.1
15 done
```

Listing 13 Script terminal_routine.sh, implementing the embedded terminal functionalities.

In lines 2-3, the script writes inside $SCRIPT_DIR variable the path where terminal_routine.sh script is located since the **urxvt** terminal is run from the main DExIMA folder. Then a pipe file called testpipe is declared, located in the same directory of the `terminal_routine.sh`. In lines 5-15, the code creates a named pipe (or FIFO) with the name stored in the variable `$pipe`, then enters an infinite loop. The FIFO substitutes the `stdin`, so the commands are directly taken from it. The content of the `testpipe` file is the following:

```
. $SCRIPT_DIR/terminal_script.sh
```

Listing 14 Content of the `testpipe` file.

As shown in Listing 14, `testpipe` only contains one command that essentially executes another script located in the same `$SCRIPT_DIR` called `terminal_script.sh`. At the beginning of the program, `terminal_script.sh` is empty, but during the execution of DExIMA, the commands to run external programs are written there. These are directly interpreted by the `testpipe` file, which is read by `terminal_routine.sh`. The **if** statement of `terminal_routine.sh`, shown at the line 5 in Listing 13, checks whether the file `pipe` exists, and if it does not, the `mkfifo` command creates the named pipe. The > .tmp redirects any output from the `mkfifo` command to a file named .tmp, but this condition is never verified since the file `testpipe` is always present. Inside the loop from line 9, the `read` command reads a line of input from the named pipe and stores it in the variable `$line`. The `eval` executes the contents of the `$line` variable as a command. The loop also includes a `sleep` command to prevent the script from consuming too much CPU time since the script will run indefinitely, waiting for commands from the `terminal_script.sh`. The message "DExIMA is ready!" is printed once at the beginning of the script to indicate that the tool is ready to operate.

```
def simulate(self):
    #...#
    pipe_file = open("./CODE/PIPE/terminal_script.sh", "w+")
    #...#
    pipe_file.write("clear\n")
    pipe_file.write("echo \"-----------STARTING SIMULATION-----------\"\n")
    pipe_file.write("cd ./OUTPUT/Sim\n")
    pipe_file.write(". ../../CODE/SIMCnfg/setvsim\n")
    pipe_file.write("make all\n")
    pipe_file.write("cd ../../\n")
    pipe_file.write("echo \"-----------SIMULATION
     ↪  ENDS-----------\"\n")
    pipe_file.write("> $SCRIPT_DIR/terminal_script.sh")
    pipe_file.close()
    #...#
```

Listing 15 Example of the embedded terminal mechanism. A snippet of code of the `simulate(self)` method from `MainWindow.py` file for QuestaSim simulation of the LiM circuit.

An example of the terminal functionality is shown in Listing 15, representing the method `simulate(self)` of the `MainWindow.py` file, which is in charge of simulating the LiM circuit by calling QuestaSim. All commands are simply written inside the `terminal_script.sh` file and executed by `terminal_routine.sh` with the mechanism explained before.

- `CustomDialog.py` creates a custom dialog to instantiate new external pins in the design. The user has to choose the label name and the direction of the pin ("input" or "output"). The dialog is shown in Fig. 5.3 (a).

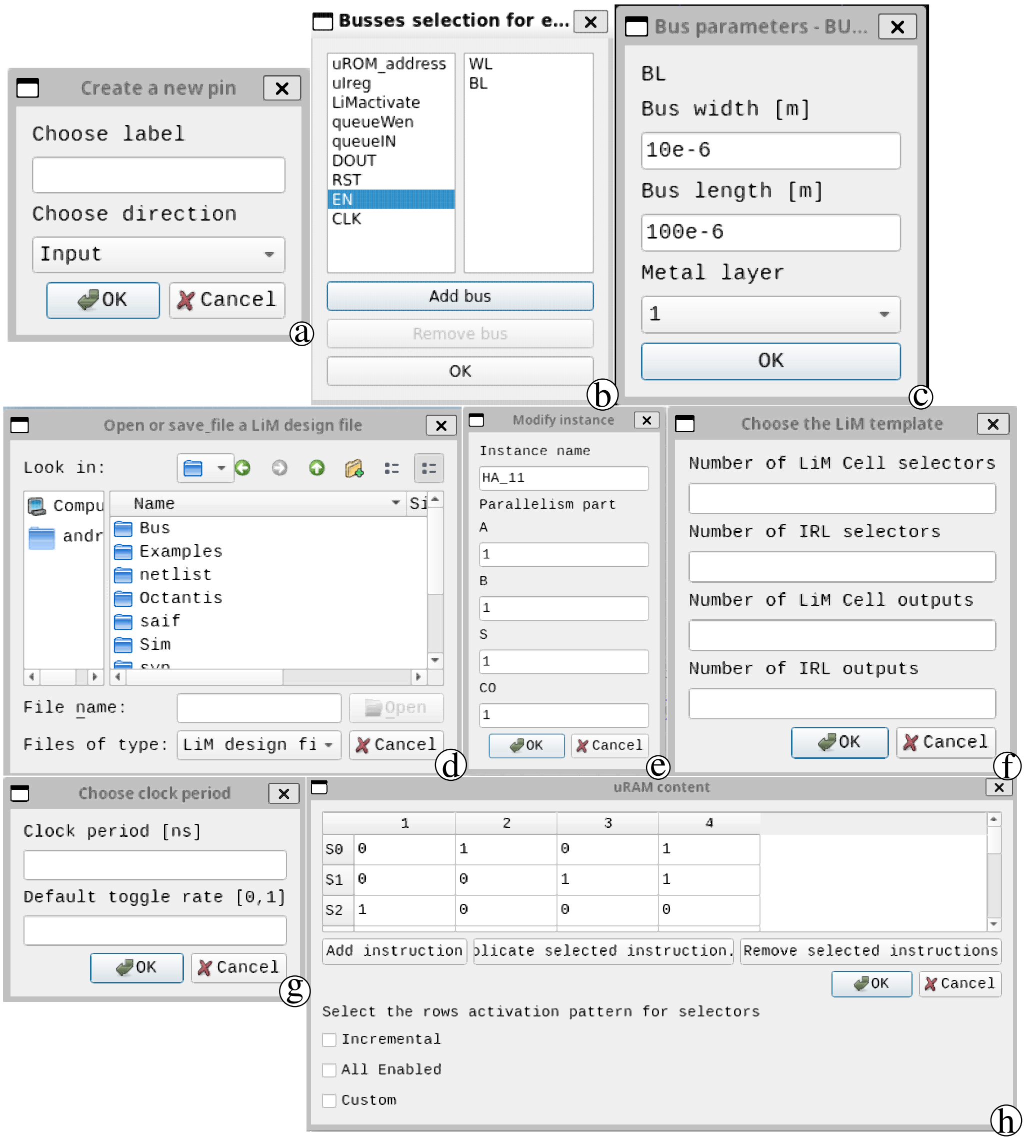
- `DialogBus.py` creates a `QDialog` that allows choosing which bus must be included in the bus analysis performed by DExIMA-Backend (Fig. 5.3 (b)).

- `DialogBusProperties.py` creates a `QDialog` in which the user specifies, for each bus chosen for the analysis, its characteristics, i.e., the width, the length, and the metal layer (Fig. 5.3 (c)).

- `FileDialog.py` handles the `save_file/ open` routines for the designs. File extensions are **.lim**, **.irl** or **.c** depending on the design scope: the first for the LiM Cells, top-level and Near Memory architecture, the second for the Intra Row Logic and the last is the C code for the Octantis and Comparison CPU-Memory sections (Fig. 5.3 (d)).

- `ProgressBar.py` creates the Progress Bar widget employed for long-time tasks (e.g., VCD file parsing). This widget helps indicate to the user that the program is not stuck but is still computing.

- `EditItem.py` implements a derivative class of `QDialog` that describes a dialog window to edit the characteristics of an instance (item). In particular, it can edit the parallelism, the item name, etc. This dialog shows up when



Create a new pin

Choose label

Choose direction: Input

Busses selection for e...

uROM_address
ulreg
LiMactivate
queueWen
queueIN
DOUT
RST
EN
CLK

WL
BL

Bus parameters - BU...

BL

Bus width [m]: 10e-6

Bus length [m]: 100e-6

Metal layer: 1

Open or save file a LiM design file

Look in:

Name

Bus
Examples
netlist
Octantis
saif
Sim

File name:

Files of type: LiM design fi

Modify instance

Instance name: HA_11

Parallelism part

A: 1

B: 1

S: 1

CO: 1

Choose the LiM template

Number of LiM Cell selectors

Number of IRL selectors

Number of LiM Cell outputs

Number of IRL outputs

Choose clock period

Clock period [ns]

Default toggle rate [0,1]

uRAM content

	1	2	3	4
S0	0	1	0	1
S1	0	0	1	1
S2	1	0	0	0

Select the rows activation pattern for selectors

□ Incremental

□ All Enabled

□ Custom

Fig. 5.3 Overview of the implemented dialogs. From left to right and top to bottom: `CustomDialog`, `DialogBus`, `DialogBusProperties`, `FileDialog`, `EditItem`, `TemplateConfigurator`, `dialogSetClock` and `uRAMGenerator`.

the scene is in EditItem mode. When a MEMORYARRAY instance is edited, extra entries are inserted in the dialog, asking the user to modify the number of rows/columns and the memory type (Fig. 5.3 (e)).

- `QV.py` contains a derivative class of `QGraphicsView` that handles the `QGraphicsScene` and defines its characteristics: the size and the scrollbar policies. Moreover, this class contains the routine to define the grid background and the snap-to-grid behavior in positioning the objects on the scene.

- `TemplateButtonClicked.py` calls the `TemplateConfigurator.py` (Fig. 5.3 (f)), located in `LiMTemplates` folder, that contains the description of a `QDialog` for the LiM template configuration. The user can specify the number of selector pins and the outputs, as discussed in subsection 3.1.1.

- `TextEditor.py` contains the description of the main text editor widget shown in Fig. 3.5 (b), where the user can write the C code for the Comparison CPU-Memory and Octantis parts.

- `dialogSetClock.py` implements the `QDialog` that asks for the clock period and the default toggle rate of the circuit (Fig. 5.3 (g)).

- `uRAMGenerator.py` implements the `QDialog` for the definition of the uRAM content (Fig. 5.3 (h)). This part is discussed in deep in chapter 4.

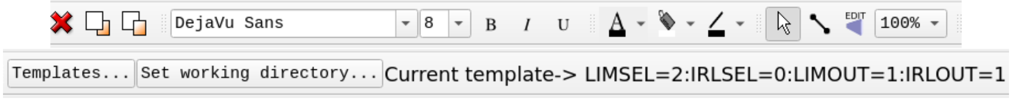
- `workingDirectorySelector.py` relies on `FileDialog` class and allows the user to choose the output working directory, where all design files and results are stored.

- `createCellWidget.py` handles the creation of an instance when it is chosen from the main window toolbox, i.e., the left panel containing the components shown in Fig. 3.5 (a).

- `createCmbSelectMode.py` creates the `Templates...` button, belonging to the main window toolbar, shown in Fig. 3.5 (a).

- `saveAs.py` calls the `FileDialog` class in Save mode, allowing to save the work in the directory specified in the `FileDialog` dialog.

- `toolbars.py` creates the main toolbar belonging to the main window of DExIMA, shown in Fig. 5.4. The functions implemented in the toolbar are, starting from the left: delete a selected item, bring to front, bring to back, font name (for the C code), font size, bold, italic, underline, text color, object color, line color, pointer mode (to move objects), wire mode (to draw the connections), EditItem mode, zoom selector, templates selector button, set working directory and lastly a label indicating the template configuration.



Current template-> LIMSEL=2:IRLSEL=0:LIMOUT=1:IRLOUT=1

Fig. 5.4 Main toolbar of DExIMA software.

- `toolbox.py` creates the main toolbox of DExIMA, containing the sections for each design phase, shown on the left side of DExIMA main window in Fig. 3.5 (a). The sections are LiM Cells, Intra Row Logic, LiM Architecture, Near-Memory Architecture, Algorithm-to-LiM, Comparison CPU-Memory. It is also possible to change the background pattern of the schematic editor using the Backgrounds section.

- `menu.py` creates the top-most menu of DExIMA, which entries are shown in Fig. 5.5.

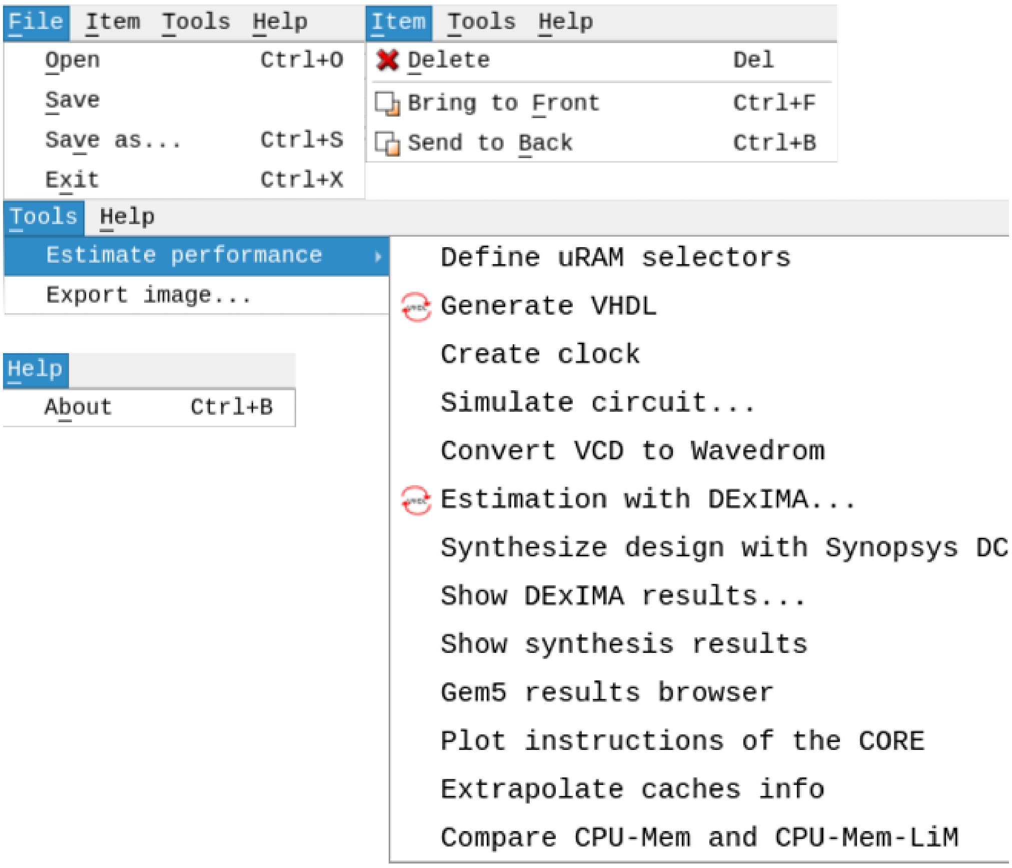


Fig. 5.5 Top-most menu entries of DExIMA.

- `createColorIcon.py`, `createColorMeny.py`, `createColorToolButtonIcon.py`, `fillButtonTriggered.py` create the color icon to color the blocks, the menu of colors, the color tool button icon and colors the selected item if the color button is pressed, respectively. The routine implemented in `itemColorChanged.py` handles the color changes of the items. Using colors is helpful for very congested schematics, and this feature can be used both for blocks and also wires with `lineButtonTriggered.py`, that sets the color of a line and `lineColorChanged.py` that updates the line icon when its color is changed. Moreover, also the color of the text can be changed by the user, and this is accomplished with `textButtonTriggered.py`, which sets the color of the text in the scene and `textColorChanged.py`, that changes the icon of the text color, based on the selected color.

- `currentFontChanged.py`, `fontSizeChanged.py`, `itemSelected.py`⌋ , `handleFontChange.py` handle the font type and size selections. `Ite`⌋ `mSelected` method allows modifying the current font of a text line. The `handleFontChange` method reads the current font from the font combobox and the font size from the font size combobox. Then the font will be set through `scene.setFont()` method.

5.1.2 Functions and routines

Specific functions and routines are associated with the graphical items described before. In particular, in the `MainWindowItems` folder, these functions are essentially related to graphical actions (adding, removing, modifying items, opening files, etc.). The Python files implementing these functions are reported in Fig. 5.2 (b), and they are briefly described in the following:

- `actions.py` contains the main actions that the user can perform in the main window. These operations include moving an item in the foreground or background in a diagram, deleting an item, changing text's font to bold or italic, generating VHDL code, creating a clock, simulating a circuit, converting a VCD file to Wavedrom format, synthesizing a design using Synopsys Design Compiler, and showing/plotting the results. Additionally, there are actions for showing program information and accessing and saving files. Each action is specified as a `QAction` object, which includes a name, an optional icon, a parent widget, and numerous optional parameters, including a shortcut key and a status tip that appears in the status bar when the mouse hovers over it. An action is declared in the `actions.py` file, and it is instantiated as in the example shown in Listing 16.

```
""" File actions.py """
#....
self.createClockAction = QAction("Create clock", self,
 ↪  triggered=self.createClock)
#....
""" File menu.py """
self.estimatePerf = self.toolsMenu.addMenu('Estimate performance')
#....
self.estimatePerf.addAction(self.createClockAction)
#....

```

Listing 16 `QAction` mechanism. The `QAction` is created in the `actions.py` file and instantiated in the top-most menu of DExIMA.

- `backgroundButtonGroupClicked.py` handles the background of the `QGraphicsScene`.

- `buttonGroupClicked.py` handles the components insertion of DExIMA. This file implements the component's selector, implemented as button groups of the LiM Cells, Intra Row Logic, LiM Architecture, and Near-Memory architecture sections, shown on the left side of Fig. 3.5 (a).

- `createBackgroundCellWidget.py` creates the toolbuttons group that handle the scene background.

- `deleteItem.py` deletes an item from the scene. If an item is deleted, also its connections are deleted. As discussed in the next parts, this can be done by removing the item from the scene and employing `IdentifyConnections.disconnectAll`. If the selected item is a wire, then only the wire will be deleted by using `IdentifyConnections.disconnect` and `IF.scene.removeItem` to remove the wire from the scene. Also, the deleted item will be purged from the component list.

- `handlePositionsXY.py` handles the position of the objects loaded from an external file, where explicit positioning coordinates on the scene are not specified. For instance, Octantis generates a **.lim** file containing positions that are all zero-ed.

- `itemInserted.py` handles the button behavior (move button, wire, and edit button) when an item is inserted in the scene. Wire and edit buttons are disabled, signaling the user that the software is in Insert Item mode.

- `openFile.py` creates an Open File dialog and runs the open file routine. It opens **.lim**, **.irl** and **.c** files and loads them into the scene or the text editor. When called, `openFile.py` calls `restoreCustomCell.py` and restores a previously designed LiM Cell or IRL block. When a file is opened, elements instantiated can be loaded in the scene or saved in a memory model. If an open (`File->Open`) action is executed, then the element will be loaded in the scene, otherwise stored in the memory characteristics.

- `pointerGroupClicked.py` sets the mode of the scene based on the checked button (move, wire, and EditItem).

- `printImage.py` prints a snapshot of the circuit, saving it in **.svg** format.

- `save_file.py` implements a routine that saves in a file what is shown on the scene in a textual format. The output can assume three possible formats: **.lim**, **.irl** and **.c** for the C code.

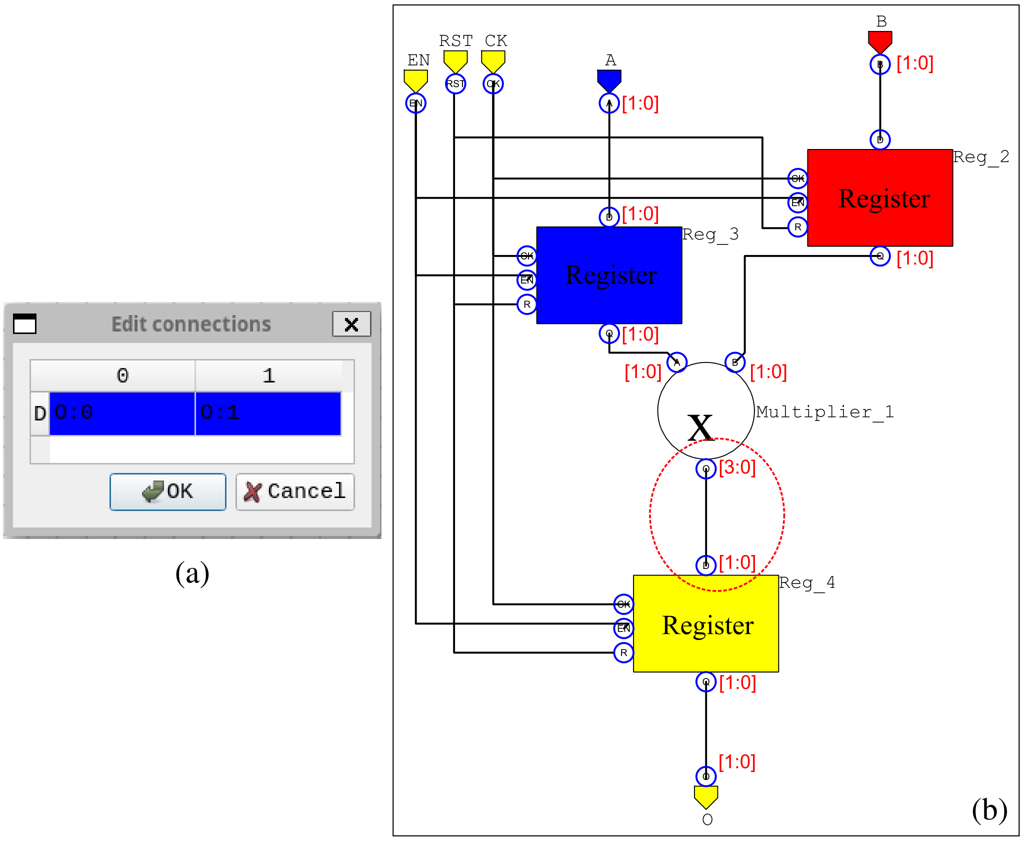
- `sceneScaleChanged.py` sets the zoom scale of the scene.

- `sendToBack.py` sends to the background an element by decreasing its z value.

- `textInserted.py` sets the scene mode to "insert text" and changes the text button to false.

5.2 CONNECTBlocks

The Python code inside this folder defines a connection between two elements (or ports). In particular, two Python scripts are located here: `ConnectDialog.py`, implementing a dialog that handles the case of a connection between different parallelisms and `IdentifyConnections.py`, i.e., the module that effectively links two ports. In Fig. 5.6 (a), the `ConnectDialog` is shown: it is composed of a `QTableWidget` with a single row and N columns, where N is the number of bits of the port with the lowest parallelism. Considering the example of Fig. 5.6 (b), the circuit needs a connection between the multiplier output, on four bits, and the register input,



	0	1
D	0:0	0:1

Fig. 5.6 (a) `ConnectDialog` window, where the user specifies, bit-per-bit, the connections between two ports having different parallelisms. (b) Example of a circuit having a connection between ports with different parallelisms.

with two bits. In this case, the user specifies by means of the `ConnectDialog` that the bit-0 and bit-1 of the D port of the `Reg_4` are connected to bit-0 and bit-1 of `Multiplier_1` output port.

5.3 Interconnections

Inside the **Interconnections** folder, there is only one Python script called `lee.py`, which is in charge of connecting items from a graphical point of view. During the connection phase, the user has to select the **wire** mode and manually draw a line between two ports. After that, the path is automatically derived by DExIMA by means of the Lee algorithm, which aims at finding the shortest path in a maze [135, 136]. The working principle is shown in Fig. 5.7.

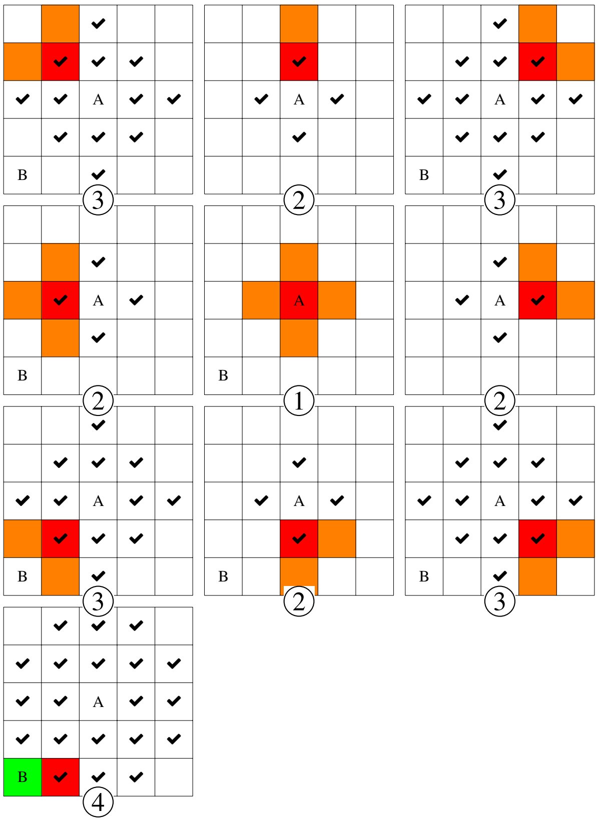


Fig. 5.7 Lee algorithm example applied to a 5x5 grid.

① The scene containing the graphics elements is represented as a matrix. In this way, the starting and ending points are associated with two matrix coordinates. If, for example, the starting point A is in the coordinates (2,2), the program starts by exploring the neighboring points on the left and right and up and down.

②-③ If the ending point is not found in the new points set, the algorithm takes each new point and explores its neighbors again, considering also the ones already explored.

④ At the end, the ending point B is found iterating the ②-③ routines, and the program finishes and returns the found path between A-B. Note that, with this algorithm, only one path corresponds to the shortest one.

A drawback of this algorithm is its complexity, that is $O(M \times N)$, where M and N are the numbers of rows and columns of the matrix, respectively, so it could be very slow for large matrices and many components to connect.

5.4 LiMTEMPLATES

Inside the LiMTEMPLATES folder, there are two Python scripts: `TemplateConfigurator.py`, shown in Fig. 5.3 (f), which implements the `QDialog` derivative class asking the user for the template characteristics, and `Templates.py` which sets the template properly. In particular, `Templates.py`, based on the number of selectors and output for the LiM and IRL parts, updates the list of inputs/outputs for the cells and IRL and then the program adds the new pins to the scene.

5.5 MEMORYARRAYHandlers

Modules, functions, and dialogs needed to define the memory component (i.e., the LiM array) are located in this folder. In particular, the Python scripts are the following:

- `MemoryDialog.py` implements the main dialog for the LiM item. In particular, by entering in EditItem mode and clicking on the MEMORYARRAY component, the program asks to provide the memory type and the number of rows and columns. Once defined, the program calls the class `MemoryCharacteristics` implemented in `MemoryCharacteristics.py`, which associates the parameters defined in the dialog with the MEMORYARRAY item.

- `LiMPattern.py` allows to choose the configuration of each memory cell. In particular, it implements a `QDialog` whose main widget is a `QTableWidget` having a size equal to the number of rows and columns defined for the LiM memory. Each table location represents a memory cell; the user can choose which circuit to integrate for each memory location.

- `IntraRowPattern.py` follows a similar approach as the `LiMPattern.py`, but this time for the IRL part. Differently from the `LiMPattern.py` case, the `QTableWidget` has only one column because the IRL part is placed on an entire memory row, while it has the same number of rows as the memory one.

5.6 PERFORMANCE

Inside the PERFORMANCE folder are the Python scripts that parse, plot, and show the results of simulations, synthesis, and DExIMA-Backend processes.

- `PerformancePlotDexima.py` handles the results stored in the DExIMA output file (**.dof**), where DExIMA-Backend writes all data and performance estimations. The class `PerformancePlotDexima` can plot the results or can show a dialog implementing a `QTableWidget`, where the results are written in tabular format. This last option is implemented in `PerformanceTable.py`.

- `InstructionsParser.py` refers to the CPU-Mem comparison part, which details will be given in section 8.1. In particular, this script parses the instruction list generated by Gem5 software for a given program and plots the occurrences on a bar chart.

- `HDF5Gem5Results.py` implements a function called `HDFGem5Results`, where the results of Gem5 simulation and written inside `stats.txt` default file is converted in HDF5 format [137], allowing to organize the data in a human-readable and hierarchical format. An example of the `stats.txt` file is reported in Listing 17: with HDF5, a specific data is obtained by simply navigating through the hierarchy, and each data (or attribute) contains both its value and its description. For example, by addressing `system.cpu.dcache.overallMisses::data`, the returned value will be 170 and its description "number of overall misses (Count)".

```
...
system.cpu.dcache.demandHits::cpu.data           2473
 ↪   # number of demand (read+write) hits (Count)
system.cpu.dcache.demandHits::total              2473
 ↪   # number of demand (read+write) hits (Count)
system.cpu.dcache.overallHits::cpu.data          2473
 ↪   # number of overall hits (Count)
system.cpu.dcache.overallHits::total              2473
 ↪   # number of overall hits (Count)
system.cpu.dcache.demandMisses::cpu.data          170
 ↪   # number of demand (read+write) misses (Count)
system.cpu.dcache.demandMisses::total             170
 ↪   # number of demand (read+write) misses (Count)
```

```
system.cpu.dcache.overallMisses::cpu.data           170          # number of overall misses (Count)
system.cpu.dcache.overallMisses::total             170          # number of overall misses (Count)
system.cpu.dcache.demandMissLatency::cpu.data      17400000     # number of demand (read+write)
...

```

Listing 17 Example of the Gem5 output `stats.txt` file.

- `PlotCachesStats.py` plots the caches memory results of the Gem5 simulation in bar chart (shown in Fig. 4.12). Some important figures of merit are considered: the number of write/read/overall accesses, hits, misses, hit ratio, and miss ratio. This part is discussed in detail in chapter 4.

- `PlotCPU_MemCPU_Mem_LiM_comparison.py` evaluates the differences between the CPU-Mem and CPU-Mem-LiM systems by considering fundamental figures of merit like the number of memory accesses, the energy of the memories, the CPU execution time, LiM energy and LiM execution time and showing a radar chart comparing these data (Fig. 4.13). Further details are provided in chapter 4.

- `ParseDCPerformance.py` parses the area, power, and timing performance results from Synopsys Design Compiler synthesis. In addition, data are inserted in a `QTableWidget` that improves readability.

5.7 SCENEElements

The schematic editor of DExIMA relies on a `QGraphicsScene` widget that enables to draw shapes (representing the different logical blocks available in DExIMA) and lines that can be used to connect blocks. In particular, the SCENEElements folder contains modules, functions, and graphical items needed to define the `QGraphicsScene` and the elements to insert into it. Elements include wires, logical blocks (DiagramItems), text, and ports.

- `DiagramItem.py` implements `DiagramItem` class that creates the components (by defining the shape and the characteristics) and handles them in the `QGraphicsScene`. Until now, the available components are NAND, OR, XOR, XNOR, TBUF, NOT, half adder and full adder gates, multiplexer 2-to-1, memory cells (flip-flop or SRAM), flip-flops, right-shifter, generic N-bit multiplexer 2-to-1, the ripple-carry adder, the array multiplier, registers, equality comparator, the memory interface, external I/O pins (a pin interfacing the external peripherals) and the memory array. The routines implemented in the `DiagramItem.py` are essentially the same for all components. Inside the constructor, the component shape and interfacing ports are defined.

```
"""DiagramItem.py constructor"""
#...
if self.diagramType == Nand:
    self.typeLogic = "NAND2"
    """Defines the typeLogic string"""
    self.inputs.append(PortItem('IN0', -70, -40, 1, self))
    """Adds a port as input in the inputs list and specifies
    ↪  position of the port."""
    self.inputs.append(PortItem('IN1', -70, 20, 1, self))
    self.outputs.append(PortItem('O', 110, -5, 1, self))
    """Adds a port as output in the outputs list and specifies
    ↪  position of the port."""
    path.moveTo(-50, -50)
    path.arcTo(-50, 0, 100, 100, 180, 0)
    path.arcTo(0, -50, 100, 100, -90, 0)
    #
    path.arcTo(0, -50, 100, 100, 270, 90)
    path.arcTo(100, 0, 10, 10, 180, 360)
    path.arcTo(0, -50, 100, 100, 0, 90)
    """Specifies the path"""
    if not (restored):
        """If the diagram item is not restored from a previous load
        ↪  design, then the name is given by
        self.typeLogic + idxLogic"""
        self.textItem = QGraphicsSimpleTextItem(self.typeLogic +
        ↪  str(IF.idxLogic), self)
    else:
        """Otherwise, the name is specified by the input variable
        ↪  name."""
        self.textItem = QGraphicsSimpleTextItem(name, self)
    self.myPolygon = path.toFillPolygon()
    self.textItem.setPos(self.myPolygon.boundingRect().topRight())

```

Listing 18 Example of the NAND gate shape definition inside the `DiagramItem.py` constructor.

As shown in Listing 18, the `diagramType` indicates the type of component to be instantiated. The user chooses it during the instantiation of the item inside the scene. For example, suppose it corresponds to a `Nand`. In that case, the code creates the ports (declared as `PortItem` objects), their directions (input, output), and their position. Then it creates the shape corresponding to the chosen gate. If the item is restored, the name and index in the component list should be read from the **.lim** or **.irl** file; otherwise, they are set by the program. In the end, the item is positioned on the scene.

- `DiagramScene.py` implements a derivative class of the `QGraphicsScene` and handles all functions of the scene like mouse events, working modalities (insert the component, EditItem, wire drawing, and insert text), load and restore of designs from file (**.lim** or **.irl**) and external pin automatic creation (based on the selected LiM template). The `def mousePressEvent(self, mouseEvent)` method, handles the mouse left click event on the `QGraphicsScene` and, based on the chosen working method, DExIMA can insert an item, open the dialog to modify a selected item, draw a wire or insert a text.

- `DiagramTextItem.py` implements a derivative class of a `QGraphicsTextItem`, a textual item that can be added to the scene.

- `PortItem.py` defines the object port of a component, in particular, the shape and the characteristics (parallelism and connections). The port is drawn with the method `def paint(self, painter, option, widget=None)`, that overloads the original `paint` method of `QGraphicsEllipseItem`. Here, the program draws the ellipse and inserts text inside the ellipse, which is the port's name. Moreover, if the port has parallelism greater than 1, [parallelism-1:0] text is written over the ellipse item. The connections are managed by the method `def connectTo(self, item)`, which connects two ports. For ports with parallelism greater than 1, a list is created with a size equal to the parallelism: each location corresponds to a specific bit index, and each entry includes a list of ports. Hence, it is a list of lists. When a port is connected, an example of a list is `[[[MULTIPLIER_0,3]]["NC"]["NC"]...]`: the first element indicates the port name and the pin number of the connected port. "NC" stands for "not connected", so the bit location will not be connected anywhere.

- `Wire.py` implements the class `Wire`, which graphically defines the connection between two ports. It is a derivative class of `QGraphicsPathItem`, which relies on the Lee algorithm, described in section 5.3, to define the shortest path between two points. This is accomplished by overriding the `def updatePosition(self)` method and calling the Lee method, which returns the lists of points defining the path, as shown in Listing 19.

```python
def updatePosition(self):
    """
    Updates the shape of the Wire depending on the position of the
 ↪  starting and ending elements.
    It is based on the Lee algorithm that returns a list of points
 ↪  belonging to the path.
    """
    self.path = QPainterPath()
    """Defines the path"""
    self.setBrush(QBrush(Qt.transparent))
    """Sets the brush as transparent"""
    A = self.mapFromItem(self.myStartItem, 0, 0)
    B = self.mapFromItem(self.myEndItem, 0, 0)
    """Gets A and B, starting and ending points respectively"""
    offsetX = QPointF()
    offsetX.setX(10)
    offsetX.setY(10)
    A = A + offsetX
    B = B + offsetX
    """Defines an offset of (10,10) to add to the points. In this
     ↪  way, the center of the PortItem is
    considered."""
    list_of_points = Lee(A.x(),A.y(),B.x(),B.y())
    """Lee returns a list of points (tuples). The path will
     ↪  connects all these points together from A to B."""
    self.path.moveTo(A.x(), A.y())
    for tuple in list_of_points:
        self.path.lineTo(tuple[0],tuple[1])
    self.path.lineTo(B.x(),B.y())
    self.setPath(self.path)
    """Sets the final path."""

```

Listing 19 Overridden `updatePosition` method for the class `QGraphicsPathItem`, including the Lee algorithm.

5.8 SIMCnfg

The simulation configuration (SIMCnfg) folder contains the scripts required to run the simulations and properly set the environment variables. Its contents are shown in Fig. 5.8:

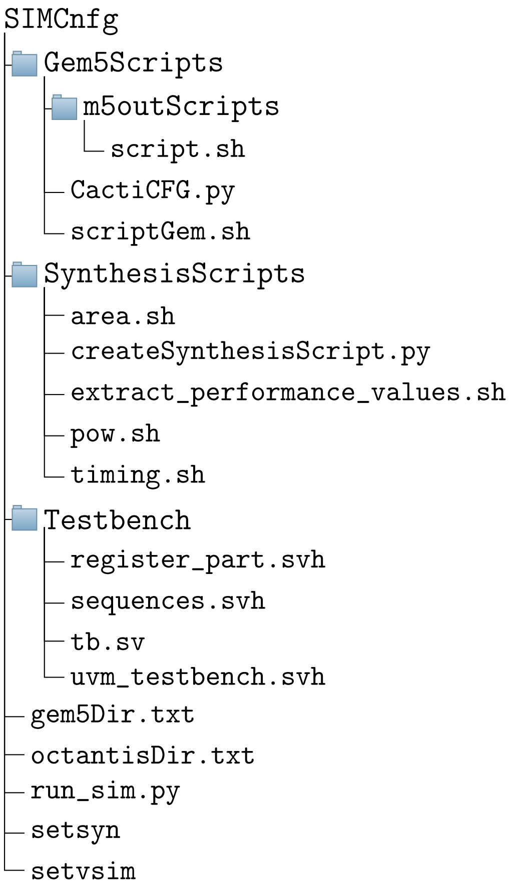


Fig. 5.8 Contents of SIMCnfg directory.

- Gem5Scripts contains all scripts needed by Gem5 to start the simulation of the algorithms on the Gem5 software. The main file is the scriptGem5.sh, which content is reported in Listing 20.

```
#!/bin/bash
script_directory=$(pwd)
#the gem5 directory
gem5_dir=$1
#the path of the riscv-toolchain
riscv_toolchain_path=$2
```

```
#the path of the destination directory, where the C code and the results are saved.
destination_directory=$3
#the cache sizes
l1cache=$4
l2cache=$5

cd $destination_directory
for f in $(ls | egrep '\.c$');
do
        cd $destination_directory
        algorithm=${f%%/}
        echo "--------------------ALGORITHM: '$algorithm'--------------------"
        $riscv_toolchain_path/riscv64-unknown-linux-gnu-gcc $f --static -o $gem5_dir/tests/test-progs/hello/riscv/linux/program
        cd $gem5_dir
        build/RISCV/gem5.opt --debug-flags=Exec,-ExecKernel,-ExecThread,-ExecEffAddr,-ExecResult --debug-file=trace.out configs/learning_gem5/part1/two_level.py --l1d_size=$l1cache --l1i_size=$l1cache
        cd m5out
        rm -rf program.out
        cp trace.out program.out
        $script_directory/m5outScripts/script.sh $gem5_dir $destination_directory $algorithm
        rm -rf trace.out program.out
        mkdir -p $destination_directory/Gem5Results
        cp stats.txt "$destination_directory/Gem5Results/stats_$algorithm.txt"
        rm -rf $destination_directory/Gem5Results/instructions_$algorithm.txt
        touch $destination_directory/Gem5Results/instructions_$algorithm.txt
done
```

Listing 20 Content of the `scriptGem5.sh` script to run simulations with Gem5.

The script sets the directories of Gem5 [97] and the `riscv-gnu-toolchain` [125] required for compiling the C codes into a RISC-V binary and the values of the L1 and L2 cache sizes: these data are passed as arguments to the script. Then, each file with extension **.c** is statically compiled with `riscv64-unknown-linux-gnu-gcc`, and the corresponding binary is saved inside the Gem5 directory containing the test programs. The simulation starts with `build/RISCV/gem5.opt` command, that runs a two-level caches system and exports the instruction execution trace on `trace.out` and system statistics on `stats.txt`. Next, the `trace.out` file is modified by the script `m5outScripts/script.sh`, which simplifies `trace.out` removing useless data and moves it inside the project destination directory. Finally, after moving `stats.txt` inside the destination directory as well, a new file called `instructions_algorithm.txt` is created, which purpose is explained in the following parts. Lastly, the `CactiCFG.py` file modifies the memory model to be simulated in Cacti by setting the size and the associativity.

- The scripts inside `SynthesisScripts` directory are needed to generate the synthesis script for Synopsys Design Compiler and to extract performance values from the reports. In particular, `createSynthesisScript.py` creates the **.tcl** script used for the synthesis and `extract_performance_values.sh` extracts the values of area, power, and timing from the Synopsys reports, by calling `area.sh`, `power.sh` and `timing.sh`, respectively.

- `Testbench` contains the SystemVerilog code implementing the UVM testbench for LiM architectures. This part is detailed in chapter 6.

- The files `gem5dir.txt` and `octantisDir.txt` contain the paths of the Gem5 and Octantis tools. DExIMA uses them to run the programs on the command line correctly.

- `run_sim.py` contains a script in Python that modifies the QuestaSim simulation script. Based on the design, it specifies which VHDL files must be considered for compilation.

- `setsyn` and `setvsim` are two scripts that set the environment variables to launch Synopsys Design Compiler and QuestaSim tools, respectively.

5.9 TOOLS, VCDAnalyzer and VHDLGenerators

<Tools like VCD (Value Change Dump) file parser, VHDL code generation, VCD to Wavedrom conversion and Gem5 results browser are located in the VCDAnalyzer, VHDLGenerators, and TOOLS directories, respectively. The VHDL generation part is relatively straight-forward, so only the VCD parsing and Wavedrom conversion are discussed in this part, while Gem5 results browser is illustrated in section 5.12.2.

5.9.1 VCD file format

The VCD file is a standard format used by EDA tools that allow to back-annotate the design by writing the changing of states of each signal inside the DUT for each time instant. An example of a VCD file format is reported in Listing 21.

```
$timescale
        1ns
$end

$scope module tb $end

$scope module DUT $end
$var wire 1 ! CLK $end
$var wire 1 " EN $end
$var wire 1 # BL [31] $end
$var wire 1 $ BL [30] $end
...
$enddefinitions $end
#0
$dumpvars
0#
0$
0%
...
$end
#3
0!
#6
1!
1'"
...
```

Listing 21 Example of a VCD file format.

In lines 1-3, the timescale is defined, which indicates the unit of measure of each time instant used in the following parts. Then, the definition of the signals starts, and each of them is defined within a `scope` environment; if more than one `scope` directives are specified without an `upscope`, it means that the signals refer to a more profound block in the hierarchy: for example, in lines 5-7, there are two `scope` declarations, so the signals refer to `/tb/DUT/` instance. Next, each signal is declared with the `var` keyword (lines 8-11), which specifies the type, the parallelism, the symbol ID, the signal name, and, optionally, the bit index in case of a multi-bit signal. After the definition of the signals, the section `dumpvars` starts, preceded by `#0`: in fact, this part declares the values of the signals at the very beginning of the simulation, so at the time instant 0. Finally, the format for declaring the signal values is in Fig. 5.9.

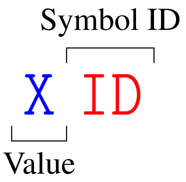


Fig. 5.9 Declaration format of the signal values in a VCD file.

The `dumpvars` section ends with a `end` keyword. After that, the annotation of the signal changes starts for each time instant greater than 0: a time instant is declared with `#N`, where `N` is a number (line 21). Note that not all signal states are declared for each time instant since VCD annotates only the changes between one instant and another.

5.9.2 Conversion to Wavedrom

For better visualization of the waveforms, DExIMA is equipped with a VCD to Wavedrom converter that employs Wavedrom, a tool capable of rendering waveforms in SVG format starting from a plain text [138]. The VCD file is parsed and converted in a Wavedrom-compatible JSON format that can be directly interpreted by Wavedrom, which outputs the corresponding timing diagram. An example of a Wavedrom-compatible JSON file is reported in Listing 22.

```
{signal: [
{name: '/TB/DUT/CLK', wave: 'x|.lHlHlHlHlHlHlHl'},
{name: '/TB/DUT/LiMactivate', wave: 'x|..1...............'},
{name: '/LIMCELL_0_0/MEMORY_9/NAND_1/IN1', wave: 'x|1...0.............'},
{name: '/LIMCELL_0_0/NAND2_10/IN1', wave: 'x|............1.0...'},
],
 head:{ text:'Timing diagram'},
 foot:{tock:0},
}
```

Listing 22 Example of a Wavedrom-compatible file

The code in Listing 22 specifies the waveform for four signals: `/TB/DUT/CLK`, `/TB/DUT/LiMactivate`, `/LIMCELL_0_0/MEMORY_9/NAND_1/IN1` and `/LIMCELL_0_0/NAND2_10/IN1`. The waveform for each signal is represented as a list of characters, where "x" represents "undefined", "." represents a no-change period, "1"/"H" and "0"/"l" active high and low, respectively. Additionally, there can be a data array, which provides the actual data values for each time step in the waveform. The head field specifies a label for the entire waveform diagram. When this Wavedrom file is rendered, it will display a waveform diagram, as shown in Fig. 5.10.

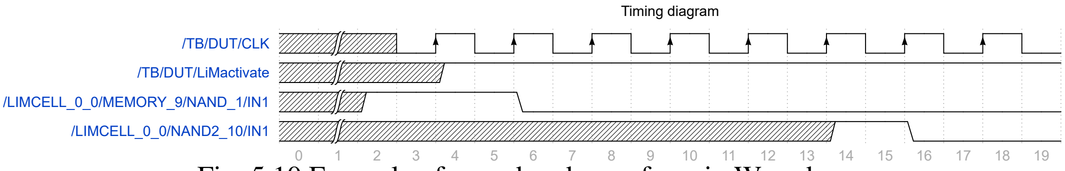


Fig. 5.10 Example of a rendered waveform in Wavedrom.

5.10 Available blocks: LIBRARY folder and SPICE description

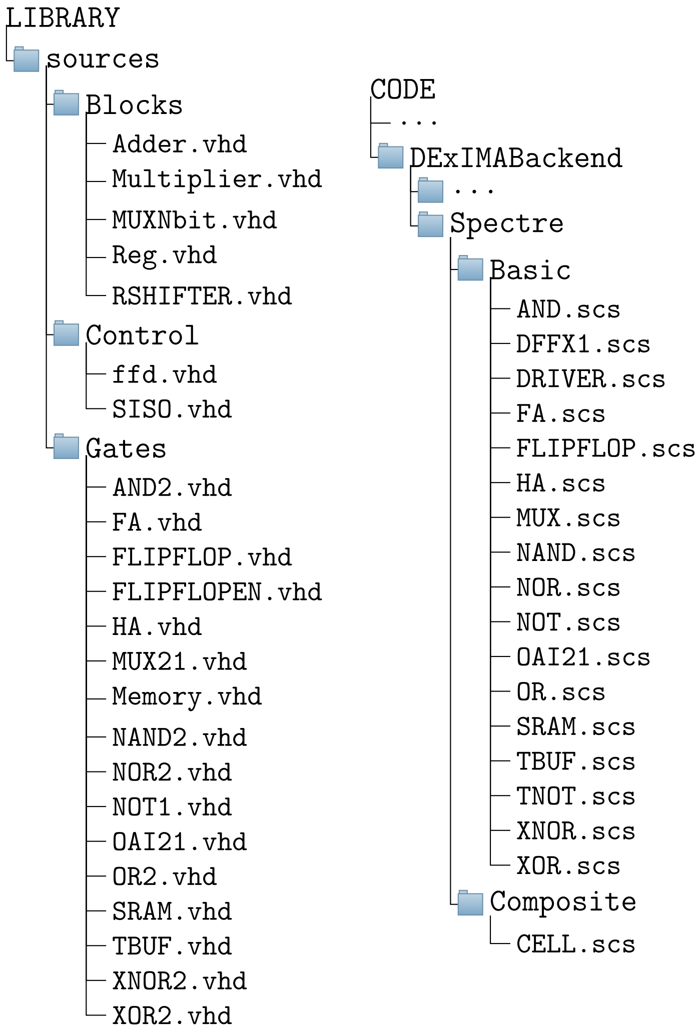
As already discussed in section 5.7, there are a limited number of components in DExIMA, and they are associated with the `DiagramItems` present in the program. The list of available components is reported in the following:

- Flip-flops and registers;
- Full adder, half adder and ripple-carry adder;

- AND, NAND, OR, XOR, XNOR, multiplexer 2-to-1, tristate buffer, multiplexer 2-to-1 with N bits, NOT, SRAM cells;
- Array multiplier;
- Equality comparator;
- Right shifter;
- Memory array and microprogrammed Memory Interface.

However, only declaring these components as `DiagramItems` is insufficient since further descriptions are required for the RTL simulation and the DExIMA-Backend estimation. In fact, for each DExIMA component, inside the LIBRARY folder, shown in Fig. 5.11, there is the corresponding VHDL file containing the functional description of the block. The LIBRARY folder is split in `Blocks` (multi-bit components), `Control` (needed for the microprogrammed machine), and `Gates` (standard cells). Moreover, as discussed in the DExIMA-Backend part (chapter 7), the standard cells also require a SPICE description at the transistor-level needed by the Backend to perform estimations and to build multi-bit hardware models. These SPICE files can be extrapolated with Cadence Virtuoso in Spectre format and saved inside the DExIMABackend/Spectre folder. Based on these concepts, if the user wants to insert a new component, the steps are the following:

1. Create the corresponding `DiagramItem` inside `DiagramItem.py`, by defining the shape and the characteristics of the new component.
2. Describe the component in VHDL to replicate its functional behavior. Save the new file inside the LIBRARY folder.
3. If the new component is a standard cell, extrapolate the corresponding SPICE description from Cadence Virtuoso or create it from scratch. On the other hand, if the component is a multi-bit block, it is necessary to write the description of the new block inside the DExIMA-Backend tool by updating the `Multibit Block class`. This part is discussed in subsection 7.2.4.



```
LIBRARY
└ sources
  ├ Blocks
  │ ├ Adder.vhd
  │ ├ Multiplier.vhd
  │ ├ MUXNbit.vhd
  │ ├ Reg.vhd
  │ └ RSHIFTER.vhd
  ├ Control
  │ ├ ffd.vhd
  │ └ SISO.vhd
  └ Gates
    ├ AND2.vhd
    ├ FA.vhd
    ├ FLIPFLOP.vhd
    ├ FLIPFLOPEN.vhd
    ├ HA.vhd
    ├ MUX21.vhd
    ├ Memory.vhd
    ├ NAND2.vhd
    ├ NOR2.vhd
    ├ NOT1.vhd
    ├ OAI21.vhd
    ├ OR2.vhd
    ├ SRAM.vhd
    ├ TBUF.vhd
    ├ XNOR2.vhd
    └ XOR2.vhd
```

(a)

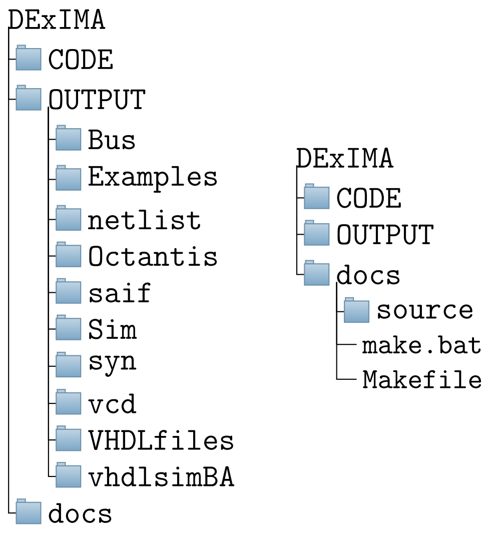
```
CODE
├ ...
└ DExIMABackend
  ├ ...
  └ Spectre
    ├ Basic
    │ ├ AND.scs
    │ ├ DFFX1.scs
    │ ├ DRIVER.scs
    │ ├ FA.scs
    │ ├ FLIPFLOP.scs
    │ ├ HA.scs
    │ ├ MUX.scs
    │ ├ NAND.scs
    │ ├ NOR.scs
    │ ├ NOT.scs
    │ ├ OAI21.scs
    │ ├ OR.scs
    │ ├ SRAM.scs
    │ ├ TBUF.scs
    │ ├ TNOT.scs
    │ ├ XNOR.scs
    │ └ XOR.scs
    └ Composite
      └ CELL.scs
```

(b)

Fig. 5.11 (a) Content of the LIBRARY folder. (b) Corresponding SPICE file descriptions of the standard cells for DExIMA-Backend.

5.11 OUTPUT and Documentation folders

By default, DExIMA stores all results, design files, RTL code, netlists, and simulation/synthesis scripts inside the `OUTPUT` directory. The `OUTPUT` folder is organized in subfolders. `Bus` folder contains the Spice netlists generated by DExIMA-Backend that are used for the bus performance simulation and estimation, which is explained in deep in subsection 7.3.9. `Examples` contains some sample LiM projects. `VHDLFiles`, `netlist`, `saif` and `syn` contain the RTL code description of the LiM architecture generated by DExIMA, the synthesized netlist (with the **.sdf** extracted



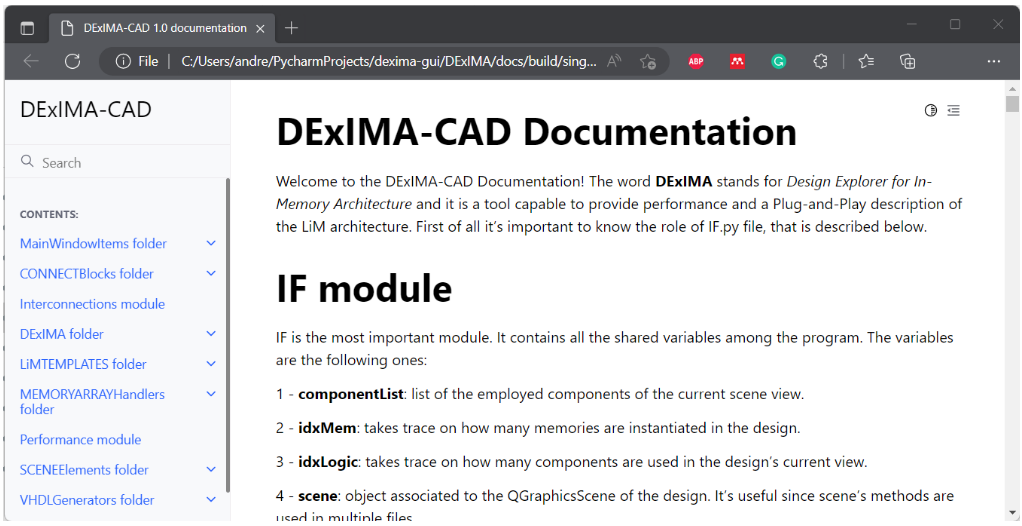
(a) (b)

Fig. 5.12 (a) OUTPUT folder organization. (b) Documentation folder organization.

delays and **.sdc** design constraints), the switching activity back-annotation file for Synopsys Design Compiler and the scripts to synthesize and estimate performance, respectively, while `Sim`, `vhdlsimBA` and `vcd` contain the scripts for the QuestaSim simulation of the functional VHDL description, the synthesized netlist and the VCD annotation file, respectively. The last folder is `Octantis`, where the input C code to be converted by Octantis is stored. When launched, Octantis automatically analyzes the **.c** files inside this directory and provides the description of the corresponding LiM architecture. The `docs` folder contains the files required to generate the documentation of the front-end part of DExIMA. The tool used is called `sphinx` [139], which relies on `reStructuredText` files and on Python Docstrings to generate both HTML and LaTeXdocuments using a `Makefile` recipe: for instance, by typing `make singlehtml`, a single HTML documentation file, shown in Fig. 5.13, is generated. The documentation is organized into sections corresponding to the organization of the folders of the front-end code, with a brief description of their contents and functionalities.

5.12 Project files

In a typical DExIMA project, several files are required to describe a LiM design, as shown in Fig. 5.14 (a-b). Project files are stored inside the `OUTPUT` directory



DExIMA-CAD Documentation

Welcome to the DExIMA-CAD Documentation! The word **DExIMA** stands for *Design Explorer for In-Memory Architecture* and it is a tool capable to provide performance and a Plug-and-Play description of the LiM architecture. First of all it's important to know the role of IF.py file, that is described below.

IF module

IF is the most important module. It contains all the shared variables among the program. The variables are the following ones:

1 - **componentList**: list of the employed components of the current scene view.

2 - **idxMem**: takes trace on how many memories are instantiated in the design.

3 - **idxLogic**: takes trace on how many components are used in the design's current view.

4 - **scene**: object associated to the QGraphicsScene of the design. It's useful since scene's methods are used in multiple files

;

Fig. 5.13 Screenshot of the HTML documentation file.

by default. Each of them refers to a very specific part of the design flow, starting from the architectural structure of the cells, the Intra Row Logic blocks, and the top-level entity, to the internal structure of the LiM, the uRAM content, the C codes of the algorithms implemented in the CPU-Mem-LiM and CPU-Mem systems and the performance results.

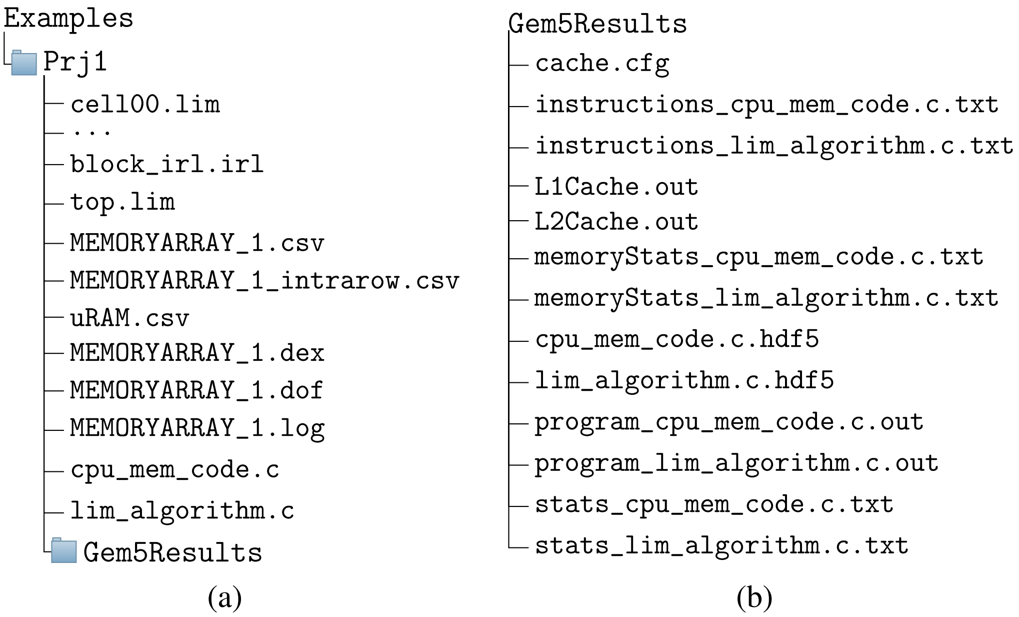


Fig. 5.14 Typical content of a DExIMA project directory. (a) Main folder. (b) Gem5 results directory.

5.12.1 Main folder

Schematic editor files: .lim and .irl

DExIMA can save the schematics realized in the Schematic Capture. They have two possible extensions: **.lim**, for the LiM Cells and the top-level LiM architecture, and **.irl** for the Intra Row Logic blocks. Both typologies have the same structure, in fact they differ only in the extensions type. An example is provided in the following Listing 23.

```
AND2_10 560.0 800.0 AND2 0
    IN0[1] : Cell_9.WR
    IN1[1] :
    O[1] :
MEMORYARRAY_1 560.0 800.0 MEMORYARRAY 3 LiM 10 10 1 1 0
...
...
Cell_9 180.0 700.0 Cell 1
    CK[1] :
    EN[1] :
    RN[1] :
    WR[1] : BL.BL
    RD[1] :
BL 300.0 480.0 Ext 5 Input
    BL[1] :
```

Listing 23 Example of a **.lim** file.

In the first line, for example, there are different fields:

- `AND2_10`: name of the instance;
- 560.0 and 800.0: X, Y coordinates of the block in the scene. These are useful to restore the items to the exact original position determined during the schematic realization;
- `AND2`: name of the item (the component);
- 0 is the index of the instantiated block.

Then, there is the pin section, where `IN[1],IN[1],O[1]:` indicate the pin names and their parallelism (inside the square brackets). The character ":" separates the pin name of the instance from the list of connected ports. The format of each element in the list is the following: `name_instance.name_port[bit]` if the connected port has parallelism greater than one. In the external pin case, another element is specified: the direction of the pin ("input" or "output"). Different is the case of the MEMORYARRAY component, which contains some additional fields:

1. Memory type: fifth element of the list.
2. Rows: how many rows are used in the memory. The sixth element of the list.
3. Columns: how many columns are used in the memory. The seventh element of the list.
4. Number of read/write/read and write ports in the memory: eighth, ninth, and tenth elements of the list, respectively.

Internal structure of the LiM: .csv files

Apart from `uRAM.csv` file that reports the content of the uRAM and the enabled rows for each instruction, CSV files are used to specify the internal structure of the LiM array. For each memory location, the cell type and IRL blocks for each memory row must be specified. In this example, files `MEMORYARRAY_1.csv` and `MEMORYARRAY_1_intrarow.csv` specifies the LiM Cell instance for each memory location and the IRL block mapped for each memory row, respectively. These files assume the same name as the memory array component instantiated inside the top-entity architecture. The LiM Cells file has the exact size of the LiM, while the IRL one has the same number of rows but with only one column: examples are provided in Listing 24.

```
-- FILE MEMORYARRAY_1.csv
cell00,cell01,cell02,cell03,...,cell31
cell00,cell01,cell02,cell03,...,cell31
cell00,cell01,cell02,cell03,...,cell31
...
cell00,cell01,cell02,cell03,...,cell31

-- FILE MEMORYARRAY_1_intrarow.csv
block_irl
block_irl
```

```
block_irl
...
block_irl
```

Listing 24 Example of `MEMORYARRAY_1.csv` and `MEMORYARRAY_1_intrarow.csv` files for the definition of the LiM Cells and IRL blocks.

DExIMA-Backend files: .dex, .dof and .log

Once the LiM design has been defined in the front-end, the input **.dex** file is automatically generated, and DExIMA-Backend uses it to estimate the performance. As discussed later in section 7.1, the **.dex** file contains the description of the LiM architecture, while **.dof** and **.log** contains the results of the estimation.

Algorithms for CPU-Mem and CPU-Mem-LiM comparison: .c files

At the end of the LiM design process, the user may want to compare CPU-Mem and CPU-Mem-LiM solutions to evaluate the impact of the LiM approach in a classical context. To do this, the user can realize the CPU-Mem version of the algorithm in C (in this example `cpu_mem_code.c`), and the program automatically creates the LiM version (called `lim_algorithm.c`). These files are both simulated by Gem5, which emulates a two-level cache RISC-V-based von Neumann architecture and provides meaningful data like the instruction execution and the statistics (e.g., number of memory accesses, number of executed instructions, etc.) that are used to compare the two solutions.

5.12.2 Gem5 output directory

Results of the comparison between CPU-Mem and CPU-Mem-LiM are saved inside `Gem5Results` directory, which contains data about the algorithm statistics, the list of executed instructions, and information about the cache memories.

Cache configuration and output files: cache.cfg, memoryStats and cache.out

To properly estimate the performance of the cache memories, Cacti [91] tool is used, which needs a configuration file in input telling information about the cache memory size, the associativity, the type, the technology used, etc. This file is called `cache.cfg`, and DExIMA modifies it according to the parameters defined by the user in the final stage of the LiM design. Once called, Cacti produces an output file called `cache.out` (`L1Cache.out` and `L2Cache.out` in the example), where the performance results of the considered cache are written, in particular, the amount of energy required for each write/read access. This parameter is useful to estimate the energy impact of the memories, given the total number of memory accesses of each algorithm, which is provided in the `memoryStats` files after the Gem5 simulation.

```
--------L2SCache--------
number of overall (read+write) accesses (Count): 368.0
--------L1DCache--------
number of overall (read+write) accesses (Count): 6749.0
--------L1ICache--------
number of overall (read+write) accesses (Count): 26183.0
```

Listing 25 Example of `memoryStats_cpu_mem_code.c.txt` file, containing the number of memory accesses for the CPU-Mem code.

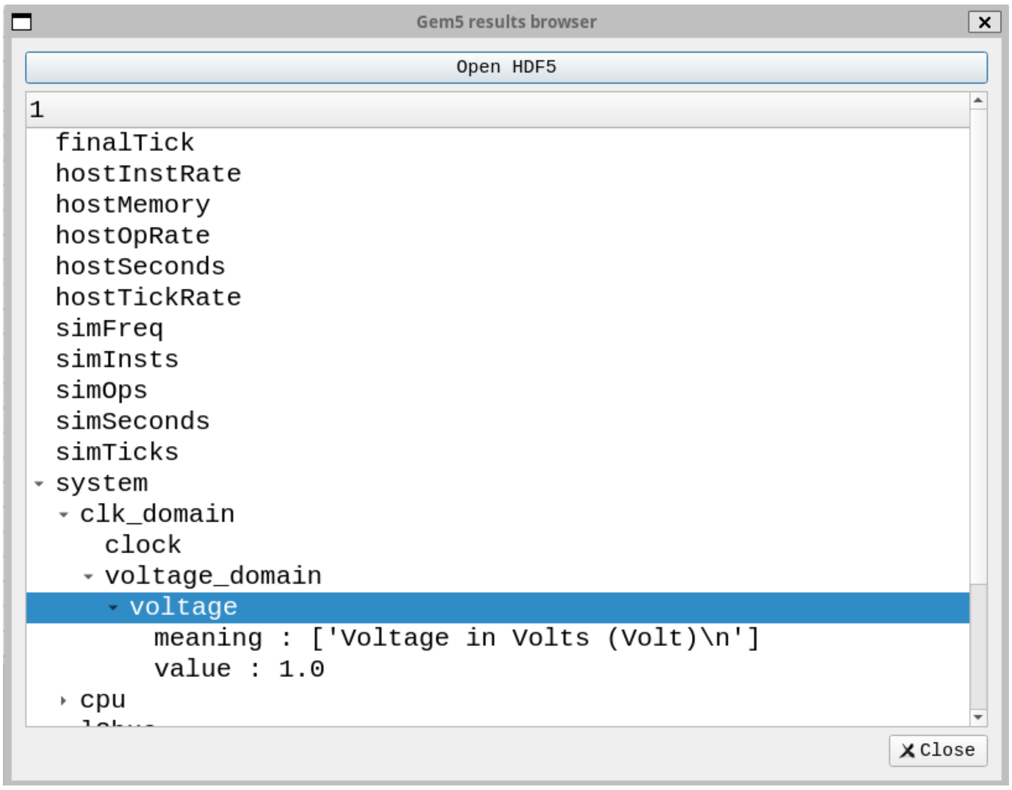
Gem5 output files: list of executed instructions, instructions count and statistics

When Gem5 runs an algorithm, it can provide statistics about the program execution and the list of instructions. These data are saved inside `stats_*.c.out` and `program_*.c.out` files, respectively, and provided for each algorithm: one for the CPU-Mem and the other for the CPU-Mem-LiM systems. Files `stats_*.c.txt` and `*.hdf5` are equivalent since the second one is the result of a conversion of the default `stats.txt` file provided by Gem5 into a more human-readable and organized format. The example shown in Fig. 5.15 represents the content of the HDF5 file opened with the embedded Gem5 results browser.The instruction list is read by DExIMA, which counts the number of occurrences for each opcode and saves them inside `instructions_*.c.txt` files. The formats of these files are reported in the example shown in Listing 26.

```
-- FILE program_cpu_mem_code.c.out
system.cpu: 0x104f0 @_start    : jal ra, 46                 : IntAlu :
system.cpu: 0x1051e @load_gp    : auipc gp, 96               : IntAlu :
system.cpu: 0x10522 @load_gp+4    : addi gp, gp, -1150         : IntAlu :
system.cpu: 0x10526 @load_gp+8    : c_jr ra                    : IntAlu :
```

```
...
-- FILE instructions_cpu_mem_code.c.txt
sh 1
c_srai 1
divu 2
c_fsd 2
mul 3
lhu 3
...
c_addi 1863
addiw 1865
lw 3868
```

Listing 26 Example of `program_cpu_mem_code.c.out`, reporting the list of executed instructions and `instructions_cpu_mem_code.c.txt`, reporting the number of occurrences.



```
1
  finalTick
  hostInstRate
  hostMemory
  hostOpRate
  hostSeconds
  hostTickRate
  simFreq
  simInsts
  simOps
  simSeconds
  simTicks
▾ system
  ▾ clk_domain
      clock
    ▾ voltage_domain
      ▾ voltage
          meaning : ['Voltage in Volts (Volt)\n']
          value : 1.0
  ▸ cpu
```

Fig. 5.15 Statistics extracted from Gem5 simulation.

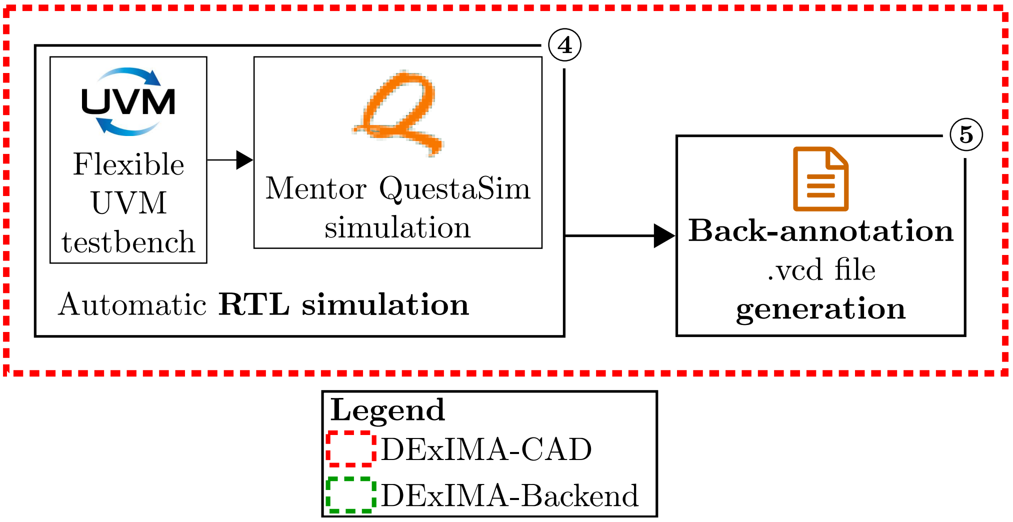
5.13 Conclusions

This chapter discusses the structure of the DExIMA front-end. The discussion starts with the structure of folders and subfolders containing Python codes, which are organized according to the function implemented and to facilitate the user in understanding the code.

Chapter 6

Automatic RTL simulation

Summary



This chapter explains in detail how the automatic simulation of LiM architectures is handled. In particular, a UVM testbench is employed, which allows generalization and great flexibility of the tests to be performed on the LiM. In order for the testbench to be executed, DExIMA generates the VHDL, modifies the QuestaSim simulation script, and launches the program directly from the Console. Each of these parts is discussed in more detail below.

6.1 Simulation script

To support the automatic RTL simulation of the LiM architecture, DExIMA implements some routines that generate the simulation script required for QuestaSim to run properly. In particular, the simulation script is reported in Listing 27.

```
vlib work
vcom -mixedsvvh -work work ../VHDLfiles/configpkg.vhd
vcom -reportprogress 300 -work work
 ↪  ../../CODE/LIBRARY/sources/Gates/AND2.vhd
vcom -reportprogress 300 -work work
 ↪  ../../CODE/LIBRARY/sources/Gates/FA.vhd
...
vlog +define+UVM_REG_DATA_WIDTH=512 $env(UVM_HOME)/uvm.sv
 ↪  +incdir+$env(UVM_HOME) +define+SYNTHESIS=0
 ↪  ../../CODE/SIMCnfg/Testbench/tb.sv
vsim -sv_lib $env(DPI_DIR)/uvm_dpi -t ns work.tb -voptargs=+acc
vcd files ../vcd/outputs.vcd
vcd files ../vcd/tb.vcd
vcd add -r -in -internal -file ../vcd/outputs.vcd /tb/DUT/*
vcd add -in -out -file ../vcd/tb.vcd /tb/DUT/*
do wave.do
run 10ms

```

Listing 27 Automatic generated simulation script of the LiM architecture

Firstly, all VHDL sources are compiled with `vcom`. In particular, the configuration package `configpkg.vhd` that contains information about the memory size, the parallelism, the size of the uRAM and the uRAM field, is compiled with the `-mixedsvvh` flag, that enables to share these constants with the SystemVerilog code used in the testbench. Then, the main testbench file `tb.sv` is compiled with `vlog`, with `UVM_REG_DATA_WIDTH` and `SYNTHESIS` values set to 512 and 0, respectively: the first modifies the default data width of the `uvm_reg` class to accommodate higher memory parallelisms, while the second one is a user-defined variable that specifies to the UVM testbench if the simulated RTL code is a synthesized netlist or not. The simulation starts with the command `vsim`, which uses `-sv_lib` and `-voptargs=+acc` flags, indicating the shared UVM library and the no-optimization option applied to the entire circuit. During the simulation, as already discussed, two VCD files are produced: `outputs.vcd` that report the waves of the internal nets of the Design Under Test and it is used in the back-annotation procedure in DExIMA-Backend; `tb.vcd` that contains only the waveforms of the top-level testbench, so it is used in the bus performance estimation phase.

6.2 Universal Verification Methodology testbench

To support an automatic simulation routine, a high degree of flexibility is required: in particular, the testbench environment should be capable of handling LiM structures with different parallelisms and complexities. To do this, the Universal Verification Methodology (UVM) is used [140]. The high-level scheme of a generic UVM testbench is shown in Fig. 6.1 (a): typically, it is organized hierarchically. The main

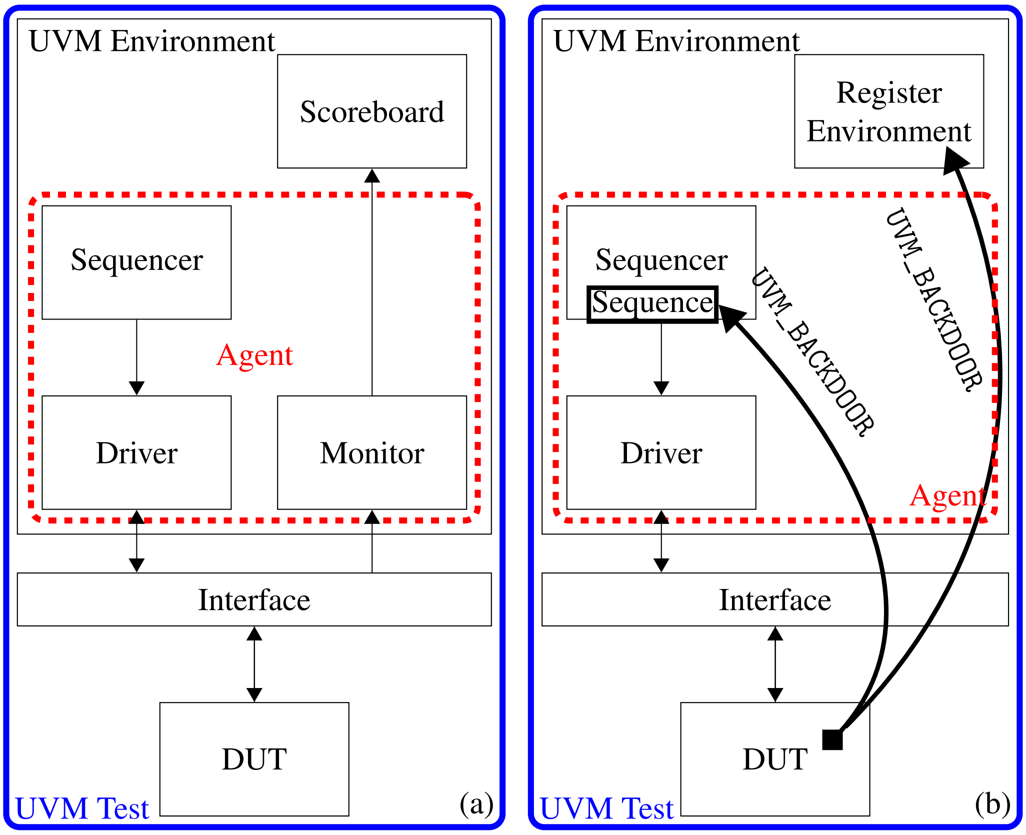


Fig. 6.1 (a) High-level scheme of a generic UVM testbench. (b) High-level scheme of the UVM testbench employed in the DExIMA project.

components are:

- Agent, containing the Sequencer (responsible for generating and scheduling the transactions that are sent to the DUT), the Driver (drives the stimuli into the DUT interfaces), and the Monitor (which observes the DUT interfaces and logs the transactions and responses).
- Environment that encapsulates the Agent and the Scoreboard. The Scoreboard compares the expected results with the actual results obtained from the DUT.
- Test that encapsulates the environment, the Device Under Test (DUT), and the Interface.

DExIMA incorporates a simplified version of a UVM testbench without the Scoreboard and Monitor components since the main goal of the testbench is only to control the phases of the LiM simulation (data precharging, algorithm execution), and not to verify the correctness of the results. However, it is possible to access the internal data of the design by using the `UVM_BACKDOOR` that is useful for verifying states of the DUT that cannot be accessed through the regular interfaces. The backdoor interface allows the testbench to bypass the regular stimuli and responses and directly access the DUT's internal registers and state. An additional component is the Register Environment, which is particularly useful in the LiM context since it allows to read, manipulate and perform memory-level tests of the registers or memories inside the DUT. Inside the Register Environment, the LiM is completely mapped by means of `UVM_BACKDOOR`, allowing access to any location. The high-level scheme of the UVM testbench employed in the DExIMA project is shown in Fig. 6.1 (b).

6.3 Interface

```
interface dut_if (input clk);
        logic EN;
        logic[columns-1:0] BL;
        logic[columns-1:0] BLB;
        logic[rows-1:0] WL;
        logic RST;
        logic[columns-1:0] DOUT;
        logic[size_uram_address-1:0] queueIN;
        logic queueWen;
        logic LiMactivate;
        logic[size_uram_instruction-1:0] uIreg;
        logic[size_uram_address-1:0] uRAM_address;
endinterface
```

Listing 28 UVM interface SystemVerilog code.

The code for the UVM Interface is shown in Listing 28. The Interface is named `dut_if`, and it is clocked by the clock signal `clk`. The `logic EN` signal enables the LiM array, both in reading/writing and operation functions; `logic[columns-1:0] BL` and `logic[columns-1:0] BLB` are the bitline and bitline bar buses, respectively, having parallelism equal to the number of columns inside the LiM memory; `logic[rows-1:0] WL` is the wordline bus, which enables each memory row; `logic RST` is the active-low asynchronous reset signal; `logic[columns-1] DOUT` is the output LiM bus, where the data can be read from memory; `logic[size_uram_address-1:0] queueIN`, `logic queueWen`, are signals related to the microprogrammed control unit to initialize and enable the queue with the uRAM starting address and finally, `logic[size_uram_instruction-1:0] uIreg`, and `logic[size_uram_address-1:0] uRAM_address` are the extracted micro-instruction and the uRAM address, respectively.

6.4 Sequence Item

A Sequence Item is a fundamental UVM object since it represents the set of stimuli used to drive the DUT. This object is passed to the DUT by means of the Sequencer, that schedules all the operations needed in the simulation. The Sequence Item used in the testbench is called `packet` and is the following:

```
class packet extends uvm_sequence_item;
        rand logic EN;
        rand logic[columns-1:0] BL,BLB;
        logic[columns-1:0] DOUT;
        logic queueWen;
        logic LiMactivate;
        rand logic read_write_n;
        rand logic[rows-1:0] WL;
        rand logic lim_ready;
        `uvm_object_utils_begin(packet)
                `uvm_field_int(LiMactivate, UVM_DEFAULT|UVM_BIN)
                `uvm_field_int(queueWen, UVM_DEFAULT|UVM_BIN)
                `uvm_field_int(read_write_n, UVM_DEFAULT|UVM_BIN)
                `uvm_field_int(EN, UVM_DEFAULT|UVM_BIN)
                `uvm_field_int(BL, UVM_DEFAULT|UVM_DEC)
                `uvm_field_int(BLB, UVM_DEFAULT|UVM_DEC)
                `uvm_field_int(DOUT, UVM_DEFAULT|UVM_DEC)
        `uvm_object_utils_end
        function new(string name = "packet");
                super.new (name);
        endfunction : new
endclass : packet
```

Listing 29 Code of `packet`, the `uvm_sequence_item` used in the UVM testbench.

It contains the same signals as the interface but with `read_write_n` and `lim_ready` additional signals that indicate if the LiM memory is read/written and if the LiM is ready to perform computations. The `` `uvm_field_int `` indicates that the signal is an integer value and, in this case, its representation is specified with `UVM_BIN` or `UVM_DEC` for binary and decimal, respectively.

6.5 Driver

According to the processed sequence and `packet` values, the Driver is in charge of driving the Interface signals. The code for the Driver object is reported in the following:

```
class Driver extends uvm_driver #(packet);
`uvm_component_utils(Driver)
packet pkt;
virtual dut_if vif;

function new(string name = "Driver", uvm_component parent);
    super.new(name, parent);
endfunction : new
// constructor for Driver class
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // attempt to get handle to virtual interface from configuration
↪   database
    if(!uvm_config_db#(virtual dut_if)::get(this, "*", "dut_if", vif))
        `uvm_error(get_type_name(), "Did not get dut_if handle")
    // if it fails, terminate and return error.
endfunction

virtual task run_phase (uvm_phase phase);
    logic [columns-1:0] data;
    forever begin
        // get next packet from sequence item port
        seq_item_port.get_next_item(pkt);
        if(!pkt.read_write_n)
            // check read_write_n signal. If it is '0', run
            // write task
            write(pkt.BL,pkt.WL);
        else begin
            //if read
            if(!pkt.lim_ready) begin
            //if the LiM is not ready to compute, then read.
                read(pkt.WL,data);
                pkt.DOUT = data;
            end
            else begin
            //the LiM is ready: start the algorithm.
                run_algorithm(pkt.BL);
            end
        end
        //mark packet as done
        seq_item_port.item_done();
    end
endtask : run_phase
virtual task write(input logic[rows-1:0] word, input logic[rows-1:0] WL);
    //set enable signal and queue write enable signal to 1
    vif.EN <= 1;
    vif.queueWen <= 1;
    //deactivate the LiM
    vif.LiMactivate <= 0;
    //set "1" as starting queue address
    vif.queueIN <= 1;
    //drive the bit-bar line and bitline as
    //equal to the packet word
```

```
53      vif.BLB <= ~(word);
54      vif.BL <= word;
55      //drive the wordline
56      vif.WL <= WL;
57  endtask : write
58  virtual task read(input logic[rows-1:0] WL,
59      output logic[columns-1:0] DOUT);
60      //enable the array
61      vif.EN <= 1;
62      //disable the queue
63      vif.queueWen <= 0;
64      //enable the LiM
65      vif.LiMactivate <= 1;
66      //set the queuein to 0
67      vif.queueIN <= 0;
68      //drive the wordline
69      vif.WL <= WL;
70      @(posedge vif.clk);
71      //wait one clock cycle to fetch data
72      DOUT = vif.DOUT;
73  endtask : read
74  virtual task run_algorithm(input logic[rows-1:0] BL);
75      vif.queueIN <= 0;
76      vif.queueWen <= 0;
77      vif.EN <= 0;
78      vif.WL <= 0;
79      vif.BL <= BL;
80      vif.BLB <= ~(BL);
81      vif.LiMactivate <= 1;
82  endtask : run_algorithm
83  endclass: Driver
```

Listing 30 Code of the Driver object.

First, the Driver is instantiated in line 2 and lines 6-16. Then, the virtual interface `dut_if` is extracted from the configuration database since it is needed by the `run_phase` to drive the signals to the DUT. During the `run_phase`, an important distinction is made based on the considered simulation phase. In the write memory phase, the `read_write_n` signal is set to 0 by the Sequencer, and the `write` task starts. In the read memory phase, `read_write_n` is set to 1 and `lim_ready` to 0, while in the computational phase, the `lim_ready` is set to 1.

6.6 The Register Environment

The main components of the Register Environment can be distinguished in:

- the uvm_reg, that models the register or memory element by specifying its characteristics.

```
class memory_row extends uvm_reg;
rand uvm_reg_field data;
`uvm_object_utils(memory_row)

function new (string name="memory_row");
        super.new(name, columns, build_coverage(UVM_NO_COVERAGE));
endfunction : new

virtual function void build();
        this.data =
        uvm_reg_field::type_id::create("data",,
        get_full_name());
        this.data.configure(this, columns, 0, "RW", 0,
        100, 1, 0, 0);
endfunction : build
endclass : memory_row
```

Listing 31 Snippet of code representing the memory_row register model.

In the definition of a register reported in Listing 31, some important parameters are passed in this.data.configure method: in order, the parallelism (equal to the number of columns of the LiM), the LSB position, the access policy (read-write in this case), the volatileness, the reset value, if it has a reset signal, if it can contain random values and if it can be individually accessed.

- uvm_reg_block: the uvm_reg_block is a key component of the UVM register model, which is used to model a set of registers in a design for verification. In particular, it creates a memory model as an array of uvm_reg elements that can be accessed in read-write mode. Similarly to the uvm_reg, the uvm_reg_block needs an HDL path, that is used to link the register model to the actual model inside the HDL code.

- uvm_reg_adapter: creates the interface between the UVM register model and the actual RTL register/memory described in the VHDL code.

```
class REG_ADAPTER extends uvm_reg_adapter;
`uvm_object_utils (REG_ADAPTER)

function new (string name = "REG_ADAPTER");
  super.new (name);
endfunction

virtual function uvm_sequence_item reg2bus (const ref uvm_reg_bus_op rw);
        packet pkt = packet::type_id::create ("pkt");
        pkt.WL = rw.addr;
        pkt.BL = rw.data;
  return pkt;
endfunction
virtual function void bus2reg (uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
  packet pkt;
  if (! $cast (pkt, bus_item)) begin
     `uvm_fatal (get_type_name(), "Failed to cast bus_item to pkt")
  end
rw.kind = UVM_READ;
rw.data = pkt.DOUT;
rw.addr = pkt.WL;
endfunction
endclass
```

Listing 32 Code of the `uvm_reg_adapter`.

By combining these elements, together with the `uvm_reg_predictor`, the Register Environment is built.

6.7 Sequences

The core of the UVM simulation for the LiM design relies on the defined sequences. In particular, five sequences are defined: reset, read/write memory, set LiM operation, and algorithm execution. Every simulation has the same scheduling of the sequences. Firstly, the system is reset, then the memory is written, and, at the end of the memory initialization, it is set in computing mode. Finally, after the execution of the algorithm, the memory can be read.

6.7.1 Reset sequence

During the reset sequence, UVM drives the active-low RST signal to 0, waits for a clock cycle and rises the RST to 1.

```
class reset_seq extends uvm_sequence;
`uvm_object_utils (reset_seq)
function new (string name="reset_seq");
  super.new (name);
endfunction
virtual dut_if    vif;
task body ();
  if (!uvm_config_db #(virtual dut_if)::get(null, "uvm_test_top.*", "dut_if", vif))
     `uvm_fatal ("VIF", "No vif")
  `uvm_info ("RESET", "Running reset ...", UVM_MEDIUM);
    vif.RST <= 0;
        vif.EN <= 0;
        vif.queueWen <= 0;
        vif.LiMactivate <= 0;
        vif.queueIN <= 0;
        vif.BL <= 0;
        vif.WL <= 0;
  @(posedge vif.clk) vif.RST <= 1;
endtask
endclass
```

Listing 33 UVM code of the reset_seq.

6.7.2 Write and read sequences

During the write sequence, the memory is written completely. As shown in the code in Listing 34, the for cycle at line 11 starts for the zeroth location until the last rows-1-th location. The signal read_write_n is set to 0, so in this way, the driver recognizes that the UVM is trying to write data inside the memory and delivers the values of the WL and BL as equal to the WL and BL values set inside the write sequence: the WL is one-hot encoded, while the BL is a random value in the range (0,1000).

```
class write_sequence extends uvm_sequence;
`uvm_object_utils(write_sequence)
virtual dut_if    vif;
function new (string name="write_sequence");
        super.new(name);
endfunction : new
virtual task body();
if (!uvm_config_db #(virtual dut_if)::get(null, "uvm_test_top.*",
↪   "dut_if", vif))
`uvm_fatal ("VIF", "No vif")
`uvm_info(get_type_name(), $sformatf("Write sequence starts."), UVM_LOW)
        for(int i = 0; i < rows; i ++) begin
                packet m_item = packet::type_id::create("m_item");
                logic[rows-1:0] tmp_address = 0;
                tmp_address[rows-1-i] = 1;
                start_item(m_item);
                m_item.randomize() with { read_write_n == 0; WL ==
↪   tmp_address; BL inside {[0:1000]}; };
                finish_item(m_item);
                @(posedge vif.clk);
        end
        `uvm_info(get_type_name(), $sformatf("Done generation of %0d
↪   items",rows),UVM_LOW)
endtask : body
endclass : write_sequence
```

Listing 34 UVM code of the `write_sequence`.

In the read sequence instead, the operation is similar to the write one, but this time the signals `read_write_n` and `lim_ready` are set to 1 and 0, respectively.

6.7.3 Set LiM operation sequence

This sequence simply sets the `read_write_n` and `lim_ready` signals to 1. In this way, the driver starts the algorithm execution task.

6.7.4 Algorithm execution sequence

The algorithm execution sequence is called during the LiM computational phase. In particular, the sequence reads the value coming from the uRAM. If an operation is found, the content of the BL, the first LiM Cell, LiM output, and IRL output are printed with an `` `uvm_info `` directive.

```
class vsequence extends uvm_sequence;
`uvm_object_utils(vsequence)
stdout   m_stdout;
uRAM_content m_uram;
OLIM_out m_OLIM;
OIRL_out m_OIRL;
int opcode;
uvm_status_e      status;
logic[size_instr-1:0] uram_instruction;
virtual dut_if    vif;
function new (string name = "vsequence");
    super.new(name);
endfunction : new
virtual task body();
    automatic logic[columns-1:0] output_cell;
    automatic int address;
    automatic logic[columns-1:0] output_lim,output_irl;
    if (!uvm_config_db #(virtual dut_if)::get(null, "uvm_test_top.*", "dut_if", vif))
     `uvm_fatal ("VIF", "No vif")
    if(!uvm_config_db#(stdout)::get(null, "uvm_test_top", "m_stdout", m_stdout))
    `uvm_fatal("STDOUT", "Could not get stdout.")
    if(!uvm_config_db#(OLIM_out)::get(null, "uvm_test_top", "m_OLIM", m_OLIM))
    `uvm_fatal("m_OLIM", "Could not get m_OLIM.")
    if(!uvm_config_db#(uRAM_content)::get(null, "uvm_test_top", "m_uram", m_uram))
    `uvm_fatal("uRAM", "Could not get uRAM.")
    if(!uvm_config_db#(OIRL_out)::get(null, "uvm_test_top", "m_OIRL", m_OIRL))
    `uvm_fatal("m_OIRL", "Could not get OIRL.")
    @(negedge vif.clk);
    m_uram.rom_row.read(status, uram_instruction, UVM_BACKDOOR);
    m_OLIM.OLIM[0].read(status, output_lim, UVM_BACKDOOR);
    m_stdout.OC[0].read(status, output_cell, UVM_BACKDOOR);
    m_OIRL.OIRL[0].read(status, output_irl, UVM_BACKDOOR);
    opcode = uram_instruction[size_opcode-1:0];
    if(!uvm_hdl_read("tb.my_uRAM.add",address))
```

```
        `uvm_fatal(get_type_name(), "Could not get uRAM address.")
    `uvm_info(get_type_name(), "Found operation!", UVM_LOW)
    `uvm_info(get_type_name(), $sformatf("\n
uram_value = %b\n
################## CELL PART ##################\n
            BL = %d\n
Output Cell[0] = %d\n
 Output LiM[0] = %d\n
###############################################",
uram_instruction,vif.BL,output_cell,output_lim), UVM_LOW)
    `uvm_info(get_type_name(), $sformatf("\n
uram_value = %b\n
------------------- IRL PART ------------------\n
            BL = %d\n
Output Cell[0] = %d\n
 Output LiM[0] = %d\n
 Output IRL[0] = %d\n
 ↪ -
-----------------------------------------------",
uram_instruction,vif.BL,output_cell,output_lim,output_irl), UVM_LOW)
endtask : body
endclass : vsequence
```

Listing 35 UVM code of the `vsequence`.

This sequence can be improved in the future by printing more values of the LiM array or by incorporating some verification routines on the data read by means of the `UVM_BACKDOOR` access.

6.8 The UVM main test

The `main_test` UVM test is in charge of defining the sequence schedule. As shown in Listing 36, in lines 8-25, the sequences explained before are created, but they start when the `phase.raise_objection(this);` is called: firstly, the write sequence (`wsequence`), then the set LiM operation (`setLim`) and then the algorithmic sequences. The algorithmic sequences start each clock cycle, and the `main_test` has to create a number of sequences equal to the number of instructions inserted in the uRAM. However, since some waiting statements are present in the algorithmic sequences to get the results at the correct time instant (e.g., `@(posedge` vif.clk)), they must run in parallel. This can be accomplished by means of the fork directive, where each vseq waits for k+1 clock cycles before starting. To better understand the code functionality, the index i=0 is chosen as a starting point: k is equal to 0. A new thread is created with fork, and the internal for cycle performs one iteration to wait for one clock positive edge. In the second outer loop iteration, i=1, k=1 so the vseq[1] has to wait for two clock positive edges and so on: parallel threads allow for a correct synchronization of the algorithm execution sequences.

```
class main_test extends base_test;
`uvm_component_utils (main_test)
function new (string name="main_test", uvm_component parent);
    super.new (name, parent);
endfunction
virtual dut_if vif;
virtual task main_phase(uvm_phase phase);
    write_sequence wsequence =
 ↪  write_sequence::type_id::create("wsequence");
    vsequence vseq[65536];
    set_lim_operation setLim;
    read_memory rsequence;

    int n_seq_initial = 1;
    int n_seq_inter = number_of_uram_instructions + 2;
    int index = 0;
    string str = "";
    for(int i=0; i < number_of_uram_instructions+2; i++) begin
        str = $sformatf("vseq_%0d",i);
        vseq[i] = vsequence::type_id::create(str);
    end
    rsequence = read_memory::type_id::create("rsequence");
    setLim = set_lim_operation::type_id::create("set_lim_operation");
    if(!uvm_config_db#(virtual dut_if)::get(this, "*", "dut_if", vif))
        `uvm_error(get_type_name(), "Did not get dut_if handle")
    phase.raise_objection(this);
    wsequence.start(m_env.m_agent.seqr);
    setLim.start(m_env.m_agent.seqr);
    for(int j = 0; j < n_seq_initial; j++) @(posedge vif.clk);
    for(int i=0; i < n_seq_inter; i++)
    begin
        automatic int k = i;
        fork
```

```
            begin
                for(int p = 0; p < k+1; p++)
                @(posedge vif.clk);
                vseq[k].start(m_env.m_agent.seqr);
            end
        join_none
    end
    @(posedge vif.clk);
    wait fork;
    phase.drop_objection(this);
endtask : main_phase
endclass
```

Listing 36 Code of the `main_test`.

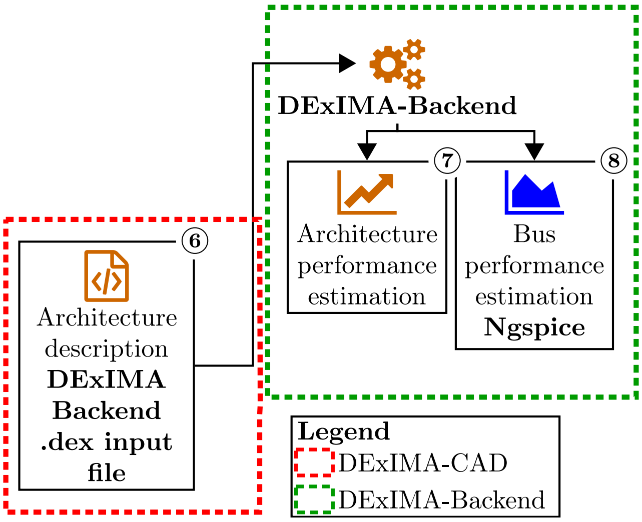
6.9 Conclusions

This chapter reports the simulation procedure embedded in DExIMA. The RTL simulation is carried out by relying on the commercial tool QuestaSim, combined with UVM, which allows greater flexibility in the realization of testbenches of LiM architectures, which involve test phases that are always composed of a data precharging part and the execution of the LiM algorithm. DExIMA takes care of generating the simulation script, which is then executed by QuestaSim.

Chapter 7

DExIMA-Backend

Summary


DExIMA-Backend
7
8
Architecture performance estimation
Bus performance estimation
Ngspice
6
Architecture description
DExIMA Backend .dex input file
Legend
DExIMA-CAD
DExIMA-Backend

To evaluate the LiM architecture in terms of area, power, and critical path delay, DExIMA is equipped with an ad-hoc estimator called DExIMA-Backend. This tool is realized in C++, and it is able to estimate the performance of CMOS-based designs, with the possibility to use different technology nodes and, in the future, other emerging technologies. The development of DExIMA-Backend began with two master theses [141, 142], in which an initial structure of the estimator was implemented, considering very simplified models. The first version, developed in [141], implemented all logic gate models with NANDs, providing very rough *performance estimations, unfortunately far from reality. In addition, the initial version had four redundant input files instead of one, describing the architecture, the LiM blocks to be implemented, the pseudo instructions and the code to execute. In the second version, on the other hand, developed in work [142], the DExIMA-Backend interface was greatly improved, reducing the number of input files to one (with the extension* ***.dex****), but still maintaining a description of the logic blocks based on NAND gates. Both these versions required a C++ class describing the structure of each logic port or hardware block, making the tool extremely complex and difficult to be expanded with other models. Both versions implemented power estimation based on the declaration of which blocks are active and involved in a given operation. This method is completely unrealistic when compared with classical estimators and synthesizers (e.g., Synopsys Design Compiler), since power estimation, especially dynamic power estimation, depends on the switching activity of the nodes. The hardware blocks are always active unless power-aware synthesis with UPF is performed. In this thesis work, DExIMA-Backend has been completely restructured by including features such as the transistor-level description of standard cells, power estimation with back-annotation, bus performance estimation, capacitive models more faithful to existing standards (e.g., BSIM4), and many other features. In Fig. 7.1, the complete flow adopted by the Backend is shown. Apart from the* ***.dex*** *file and transistor-level description of the standard cells in Spectre format, the actual version of DExIMA-Backend requires the technological parameters, 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 instances representing the internal architecture of the LiM array or the Near-Memory hardware. These class instances can be standard cells* (`STDCell class`) *such as OR, XNOR, NOT, AND, NAND, etc. that are the smaller 1-bit elements that can be instantiated; composite cells (*`CompositeGate class`*), that are 1-bit blocks containing more than one standard cell and multibit blocks (*`MultibitBlock class`*), that describe complex structures having parallelism greater than 1 bit (such as multipliers, adders, etc.). Basically,* `CompositeGate class` *and* `MultibitBlock class` *contain standard cells, so the core of the computations of the dynamic and static powers, the area and the critical path is located inside the* `STDCell class`*. This chapter explains in detail the structure of DExIMA-Backend and the computation models used in performance estimation of CMOS-based architectures.*

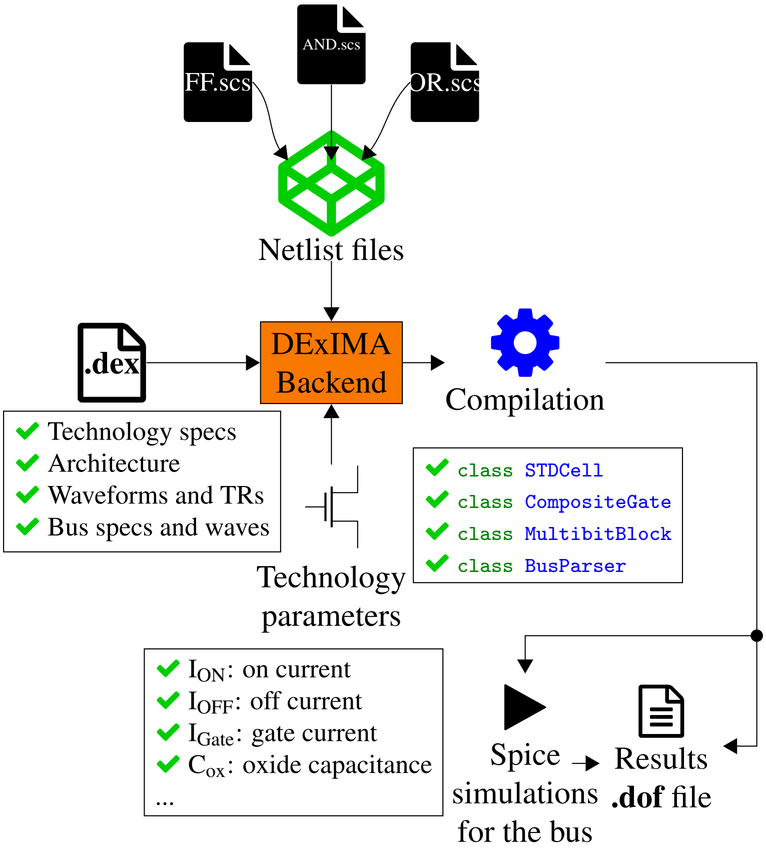


Fig. 7.1 DExIMA-Backend estimation steps

7.1 Main classes overview

In this section, the code structure of DExIMA-Backend is presented. The discussion regards the main classes and methods responsible for performing the estimations and creating the circuit models organized in a hierarchical format. To clarify the explanations better, the Unified Modeling Language (UML) is used, which provides diagrams and schemes to represent the DExIMA-Backend structure visually: UML is used as a starting point, focusing more on the main aspects of the program and computational models. Further details on the methods and attributes functionalities can be found in the documentation provided at this link. In Fig. 7.2 (a), the high-level scheme of the DExIMA-Backend tool is presented: the main class is called `Dexima` and it is mainly composed of the `Compiler class`, which compiles the input **.dex** file, creates the model of the architecture, checks for syntax errors, etc., and the `Simulator class` that is in charge of performing the estimations and providing the results. In the scheme, two other classes are aggregated to `Dexima class`,

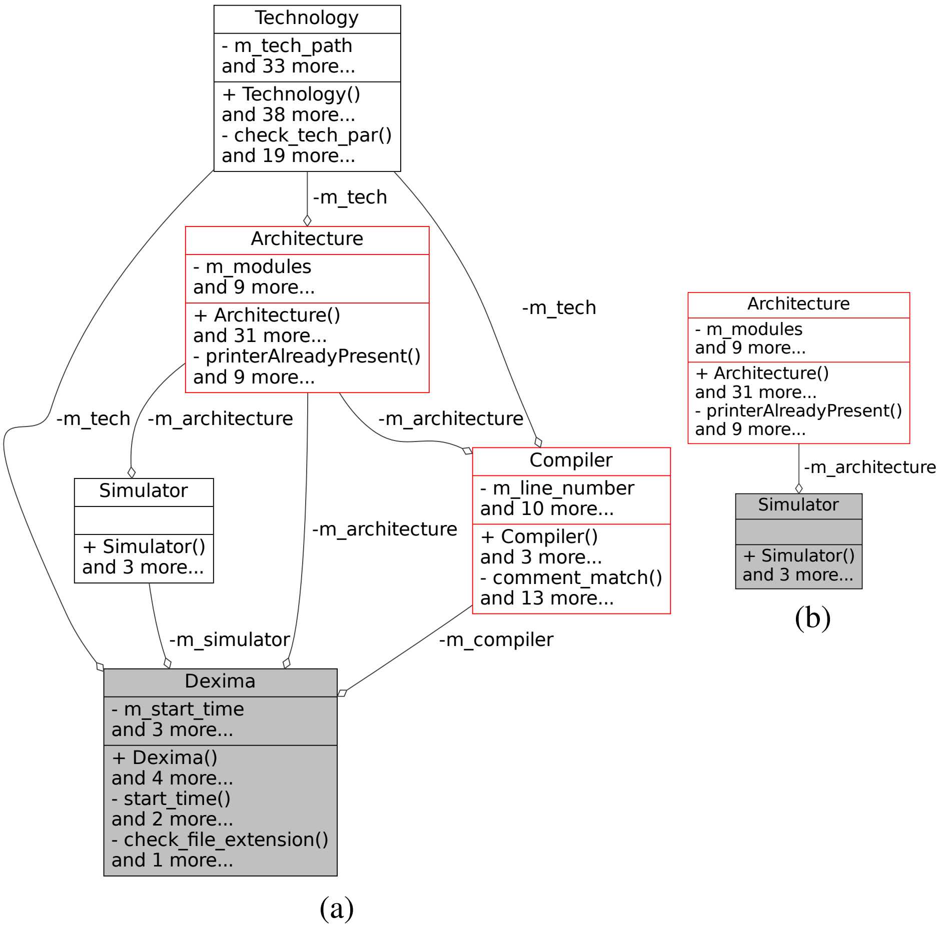


Fig. 7.2 (a) UML high-level scheme of the DExIMA-Backend tool: collaboration diagram of `Dexima class`. (b) Collaboration diagram of `Simulator class`.

that are `Architecture class`, containing a description of the entire architecture to estimate (i.e., which types of blocks/standard cells are present, how they are connected, etc.) and `Technology class`, that models the N-P MOS devices used to implement the standard cells. Focusing on `Dexima class` aggregations (denoted with the connection ──◇) the `Simulator class`, which UML collaboration diagram is reported in Fig. 7.2 (b), shows that it has only one aggregation, that is the `Architecture class`, on which the simulator performs the estimations.

7.2 DExIMA-Backend input file

Architecture and system specifications are provided into an input **.dex** file: an example is provided in the following Listing 37, that reproduce the structure of a LiM architecture shown in Fig. 7.3.

```
begin constants
        BUILT_IN CLOCK 6.0
        BUILT_IN VDD 1.1
        BUILT_IN PROB 0.5
end constants
begin init
        LIM MEMORYARRAY_1(1,2)
end init
begin MEMORYARRAY_1
    begin memdef
                ROWS 2
                COLUMNS 2
                TYPE FLIPFLOP
        end memdef
        begin logic
        end logic
        begin cells
                NAND NAND2_10() -> Cell(0,0)
                NAND NAND2_10() -> Cell(0,1)
                NAND NAND2_10() -> Cell(1,0)
                NAND NAND2_10() -> Cell(1,1)
        end cells
        begin map
                Memory(0,0).RD -> NAND2_10(0,0).IN0
                Memory(0,1).RD -> NAND2_10(0,1).IN0
                Memory(1,0).RD -> NAND2_10(1,0).IN0
                Memory(1,1).RD -> NAND2_10(1,1).IN0
        end map
end MEMORYARRAY_1
begin map
end map
begin instructions
        LIM_INSTRUCTION MEMORYARRAY_1 timing_paths
        LIM_INSTRUCTION MEMORYARRAY_1 algorithm
        begin timing_paths
```

```
                PIPELINE 2
                begin power
                end power
                begin path[0]
                        NAND2_10(0,0)
                end path[0]
                begin path[1]
                        Memory(0,0)
                        NAND2_10(0,0)
                end path[1]
                begin path[2]
                        Memory(0,0)
                end path[2]
        end timing_paths
        begin algorithm
                PIPELINE 0
                begin power
                /LiMcell_0_0/Memory_9/FLIPFLOP_0/RN/0.08{001...}
                /LiMcell_0_1/Memory_9/FLIPFLOP_0/RN/0.08{001...}
                /LiMcell_1_0/Memory_9/FLIPFLOP_0/RN/0.08{001...}
                ...
                end power
                begin path[0]
                end path[0]
        end algorithm
end instructions
begin bus
    BL[0](2);100e-9;100e-9;1;{0000001111000000000000000}
    /LiMcell_0_0/Memory_9/NAND_1/IN1/
    /LiMcell_1_0/Memory_9/NAND_1/IN1/
    BL[1](2);100e-9;100e-9;1;{0000111111000000000000000}
    /LiMcell_0_1/Memory_9/NAND_1/IN1/
    /LiMcell_1_1/Memory_9/NAND_1/IN1/
end bus
begin code
        algorithm 1
end code
```

Listing 37 Example of a **.dex** file for the architecture in Fig. 7.3.

The file is organized in the following sections:

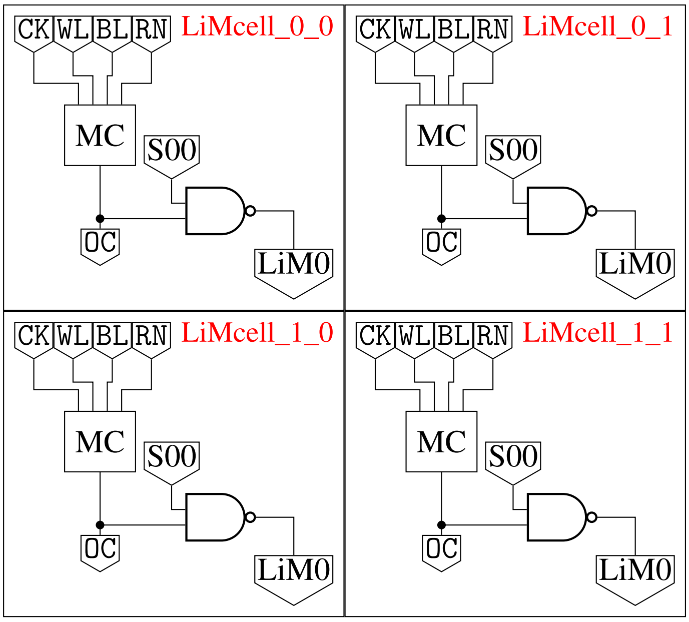


Fig. 7.3 Toy example of a LiM architecture and the associated DExIMA file in Listing 37.

- `begin constants` contains the values of the clock period, the supply voltage (VDD) and the default switching probability for the dynamic power estimation;
- `begin init` declares the blocks belonging to the top-level entity, so the LiM array or other Near-Memory circuits. A LiM array is instantiated by declaring its name, parallelism, and log_2 of the total number of rows, which are the bits required to address the memory. For example, a 2-bit and 2 rows LiM array named MEMORYARRAY_1 is written as `LIM MEMORYARRAY_1(1,2);`
- `begin MEMORYARRAY_1` defines the structure of the LiM array. This section is divided into four subsections that are `begin memdef`, declaring the memory size and the memory element type, `begin logic` and `begin cells`, where the IRL and LiM Cells sub-blocks are instantiated and `begin map`, where all the blocks are connected. For instance, in the architecture shown in Fig. 7.3, there is a NAND gate for each LiM Cell directly connected to the output of the memory cell. Through the `->` operator inside the `begin cells` section, DExIMA-Backend puts the specified logic block inside the cell, that is addressed with two indexes specified in round brackets. The connections between outputs and inputs are identified again with the `->` operator in the `begin map` section;

- `begin map`, similarly to the `MEMORYARRAY_1` case, describes the connections between the top-level blocks;

- `begin instructions` contains data related to the algorithm executed in the simulation. This section always contains two instructions defined as `LIM_INSTRUCTION`, called `timing_paths` and `algorithm` respectively. In `timing_paths`, DExIMA-CAD writes down all the paths inside the LiM array needed by DExIMA-Backend to perform the critical path computation: in the example proposed in Fig. 7.3, the total number of paths are 3, involving the memory cell, the NAND gate and both. In `algorithm` instead, DExIMA-CAD reports the waveforms and the toggle rates (TRs) parsed from the LiM array **.vcd** file for each time instant. These data are specified in a hierarchical way, meaning that each pin of each module of the LiM array is clearly defined. For instance, at line 53 of Listing 37, the toggle rate (i.e., the number between the last forward slash "/" and the curly bracket "{") and the waveform of signal RN (the bit sequence included inside the curly brackets) of the FLIPFLOP_0 cell inside the Memory_9 component of LiMcell_0_0 are unequivocally defined;

- `begin bus` reports the data needed to perform the bus consumption estimation. The estimation of the bus performance is made with **Ngspice** and requires parameters like the connection name, the width, the length, the metal layer type, and the wave transitions of the chosen connection. DExIMA-CAD handles this part and the fundamental parameters are asked to the user utilizing some graphical dialogs. The wave transitions are derived from the top-level **.vcd** file and written inside the **.dex** file following a similar approach to the instructions part. In lines 63 and 66 of Listing 37, the BL signal is selected for bus analysis, of which the bit index (indicated inside "[]"), the parallelism (indicated inside "()"), the width (first value after ";"), the length (second value after ";"), the metal layer (third value after ";") and the bit sequence for each time instant (last field, indicated inside "{}") are specified. Note that, for each signal chosen for the bus analysis, DExIMA-CAD also specifies the LiM internal blocks to which the considered signal is connected (lines 64-65 and 67-68): this is required for estimating the impact of the internal memory bus, as discussed in subsection 7.3.9;

- `begin code` specifies which instruction and how many times is executed.

Starting from this file, DExIMA-Backend is able to create a model of the architecture. To accomplish this purpose, the `Compiler class` and its collaborators take care of the **.dex** code parsing, code syntax checking, mapping, error handling, and compilation.

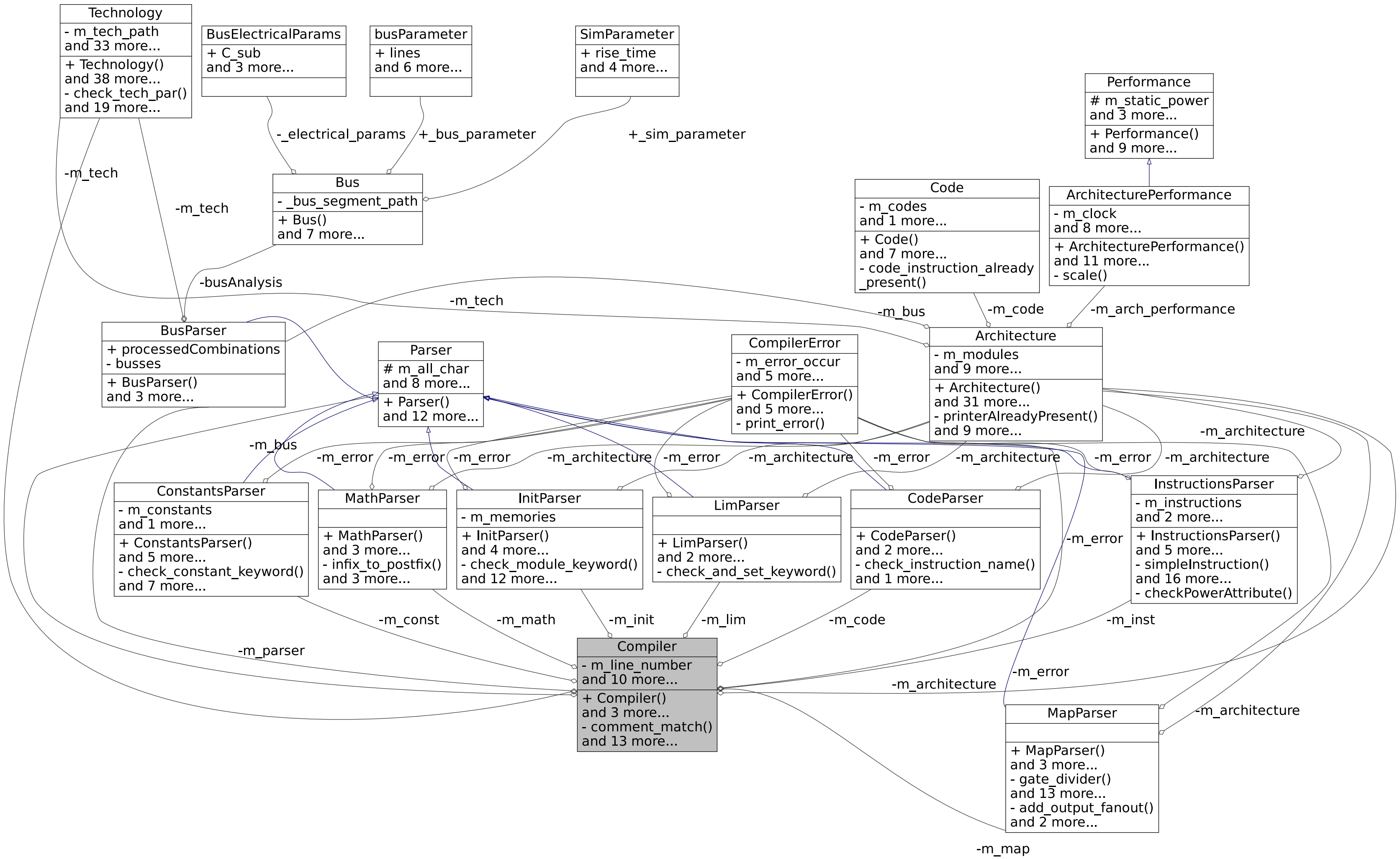


Fig. 7.4 Collaboration diagram of the `Compiler` class.

The aggregate classes are shown in Fig. 7.4, and their functions are the following:

- The `CodeParser class`, parses the code section of the **.dex** file containing information on which operation and how often it is executed.
- The `MathParser class` parses and computes the math operations written inside the **.dex** source file.
- The `InitParser class` parses the initialization section of the **.dex** file, indicating the hardware blocks instantiated inside the design, their parallelism, the LiM array sizes, etc.
- The `MapParser class` parses the map section of the **.dex** file that specifies how the blocks of the design are connected.
- The `InstructionsParser class` parses the instruction section of the **.dex** file. This section is essential since it provides the data paths for the critical path calculations and the switching activity information for each node if the back-annotation process is activated.
- The `ConstantsParser class` parses the constants section of the **.dex** file, containing some constant values that are used during the performance evaluation. These constants can be, for instance, the clock period, the default toggle rate, the supply voltage, etc.
- The `LimParser class` parses the LiM section of the **.dex** file. The LiM section describes, if present, the main characteristics of the LiM array, in particular the number of rows/columns, the logic blocks of the cell and IRL, and how they are connected.
- The `BusParser class` parses the bus section of the **.dex** file, where information about the parallelism, the waves, and the width and length of the bus interconnection are specified.

All these parser classes inherit from a superclass called `Parser class`. Eventual errors are handled by the `CompilerError class` that is in charge of identifying the error, telling the user its nature, and specifying the erroneous line of the **.dex** file and, possibly, suggesting a correction.

7.2.1 The Technology class

The Technology class has a fundamental role in the DExIMA-Backend tool since it models the technology used to implement the circuits. The UML class representation is reported in Fig. 7.5. Starting from a technological file that contains the useful parameters of the devices, the Technology class models the MOSFET and its intrinsic characteristics: for instance, the equivalent capacitances of the MOSFETs are modeled by means of `compute_Cjs`, `compute_Cjd`, `compute_Cdd`, `compute_Css`, `compute_Cgg`, `compute_CgsOvl`, `compute_CgdOvl` that refer to source/drain junction capacitances, drain-drain, source-source, gate-gate intrinsic self capacitances and gate-source, gate-drain overlap capacitances. These capacitances

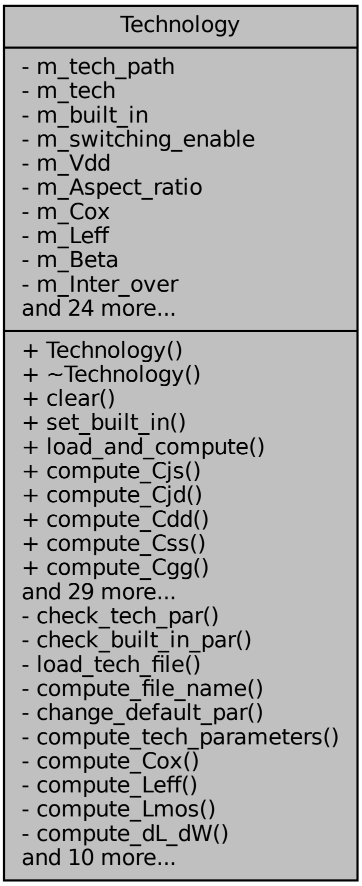


Fig. 7.5 UML class diagram of the Technology class.

are fundamental to estimating the dynamic power and delay of the standard cells, so precise estimations are needed: this part is discussed in detail in subsection 7.3.4. A technology source file content example is reported in Table 7.1 for the 45nm CMOS technology. These parameters are extracted by the BSIM4 model of the device and are used in the calculations explained from section 7.3.

Table 7.1 Parameter list of the 45nm CMOS technology.

Parameter	Value	Meaning
Year	2005	-
epsox	3.9	*Relative dielectric constant of the oxide.*
Lgate	5.00E-08	*Nominal channel length.*
Leff	2.25E-08	*Effective channel length.*
Xj	1.98E-08	*Junction depth of the S/D.*
Aspect_ratio	2.045	*P-N transistor aspect ratio.*
Vdd	1.1	*Supply voltage* (V).
m_Wn	9.00E-08	*Minimum transistor width* (m).
Cox	0.03029	*Oxide capacitance* (F/m^2).
Ion	1340	*On Current* (A/m^2).
Ioff	1.18E-01	*Off Current* (A/m^2).
Igate	1.28E-02	*Gate Current* (A/m^2)
CJ0N	0.0005	*NMOS: Zero bias source bottom junction capacitance per unit area* (F/m^2).
CJ0P	0.0005	*PMOS: Zero bias source bottom junction capacitance per unit area* (F/m^2).
CJSWN	5.00E-10	*NMOS: Source sidewall junction capacitance per unit periphery* (F/m).
CJSWP	5.00E-10	*PMOS: Source sidewall junction capacitance per unit periphery* (F/m).

CGD0	1.10E-10	*Gate-Drain overlap capacitance* (F/m)
CGS0	1.10E-10	*Gate-Source overlap capacitance* (F/m)
MJN	0.5	*NMOS: Source bottom junction capacitance grading coefficient.*
MJP	0.5	*PMOS: Source bottom junction capacitance grading coefficient.*
MSWN	0.33	*NMOS: Isolation-edge sidewall source junction capacitance grading coefficient.*
MSWP	0.33	*PMOS: Isolation-edge sidewall source junction capacitance grading coefficient.*
PBN	1	*NMOS: Source/Drain bottom junction built-in potential.*
PBP	1	*PMOS: Source/Drain bottom junction built-in potential.*
PBSWN	1	*NMOS: Isolation-edge sidewall source junction built-in potential.*
PBSWP	1	*PMOS: Isolation-edge sidewall source junction built-in potential.*
SDDiffusionLength	0	*Length of the Source/Drain diffusions* (m).
dlc	3.75E-09	*Delta L for capacitance model* (m).
Vth0	0.322	*Long channel threshold voltage at* $V_{bs} = 0$ (V).

cgdl	2.65E-10	*Overlap capacitance between gate and lightly-doped drain region* (F/m).
meto	0	*Metal overlap in fringing field.*
cgsl	2.65E-10	*Overlap capacitance between gate and lightly-doped source region* (F/m).
Ndep	3.40E+18	*Channel doping concentration at depletion edge for zero body bias* (m^{-3}).
k1	0.4	*First-order body-bias coefficient.*
k2	0	*Second-order body-bias coefficient.*
w0	2.50E-06	*Narrow width coefficient.*
k3b	0	*Body effect coefficient of K3.*
a0	1	*Non-uniform depletion width effect coefficient.*
ags	0	*Gate-bias dependence of Abulk.*
b0	0	*Bulk charge coefficient due to narrow width effect.*
b1	0	*Bulk charge coefficient due to narrow width effect.*
keta	0.04	*Body-bias coefficient for non-uniform depletion width effect.*
lpeb	0	*Lateral non-uniform doping effect on K1.*
clc	1.00E-07	*CLC and CLE consider the channel-length modulation.*
cle	6.00E-01	

VFBCV	-1	*Flatband voltage for capMod = 0* (V).
Tox	1.14E-09	*Oxide thickness* (m).
llc	0	*Coefficient of length dependence for CV channel length offset.*
lwc	0	*Coefficient of width dependence for CV channel length offset.*
lwlc	0	*Coefficient of length and width cross-term dependence for CV channel length offset.*
xl	-2.00E-08	*Channel length offset due to mask/ etch effect* (m).
lln	1	*Power of length dependence for length offset.*
lwn	1	*Power of width dependence for length offset.*
xw	0	*Channel width offset due to mask/etch effect* (m).
wlc	0	*Coefficient of length dependence for CV channel width offset.*
wwc	0	*Coefficient of width dependence for CV channel width offset.*
wwlc	0	*Coefficient of length and width cross-term dependence for CV channel width offset.*
dwc	5.00E-09	*Channel-width offset parameter for CV model* (m).
NF	1	*Number of fingers in the device.*

wln	0	*Power of length dependence of width offset.*
wwn	0	*Power of width dependence of width offset.*
cjswgs	3.00E-10	*Gate-side source junction capacitance per unit width* (F/m).
cjswgd	5.00E-10	*Gate-side drain junction capacitance per unit width* (F/m).
as	0	*Source area* (m^2).
ps	0	*Source periphery* (m).
ad	0	*Drain area* (m^2).
pd	0	*Drain periphery* (m).

At the beginning of the operations, inside the Technology class, the technology file is parsed and an internal model of the device is created, then the computations can start. Many classes use the technology object, each of which involves calculations of the performance, as discussed later. For instance, the STDCell class, CompositeGate class and MultibitBlock class are the ones responsible for the estimations of the standard cells, composite cells (i.e., cells composed of multiple standard cells) and multibit blocks (e.g., registers, adder, multipliers, etc.), respectively. These classes need a precise model of the technology devices, so they strictly depend on the Technology class.Other objects that have a strict relation with the Technology class, are the following:

- The FlipFlopArchitecture class builds the memory model, organized as an array of flip-flops or, alternatively, SRAM cells. Up to present, DExIMA is capable of estimating the performance of a memory array made of flip-flops and SRAMs: the capability to implement other arrays will be expanded in the future.

- The Architecture class builds the architecture model, comprehending the LiM and, possibly, other Near-Memory logics.

- The `LiM class`, as suggested by its name, build the model of the LiM array, linking and mapping the LiM cells, the IRL, etc.
- The `LiMPrinter class` and `Printer` are the corresponding printer classes of the LiM and a generic block (standard cell, composite gate, multibit block), respectively. The concept of the printer class will be explained later in subsection 7.2.2.
- The `Load class` represents the model of a capacitive load that can be inserted in the output of a generic block if needed.
- The `Dexima class` is the main file class of the DExIMA software that is in charge of handling all the evaluation phases (compilation, simulation, CPU usage time measurements, etc.), as discussed previously.

7.2.2 Concept of the Printer class

The `Printer class` is intended as a superclass from which multiple classes inherit its properties. In particular, it is the superclass of the subclasses that implements a component or a circuit, so the `STDCell class`, `CompositeGate class`, `MultibitBlock class` and `LiMPrinter class`. The `Printer class` is extremely advantageous because through the inheritance concept, its methods can be declared as `virtual`, and they can be overridden by the subclasses, in particular the ones responsible for computing the performance values. In this way, there is no need to differentiate between the subclasses in the performance estimation phases. The inheritance diagram of the `Printer class` is shown in Fig. 7.6: the inheritance is indicated with the connection ——▷. The `Printer class` defines the name of the component model by means of `m_printer_type` variable (e.g., NAND2_X1) and the interface of the component itself. Five vectors of strings specify the interface of the block:

- `m_port_names` contains the names of the ports used by the component in the map section;
- `m_port_type` specifies the ports directions (input or output);
- `m_port_parallelism` defines the parallelism of the ports;

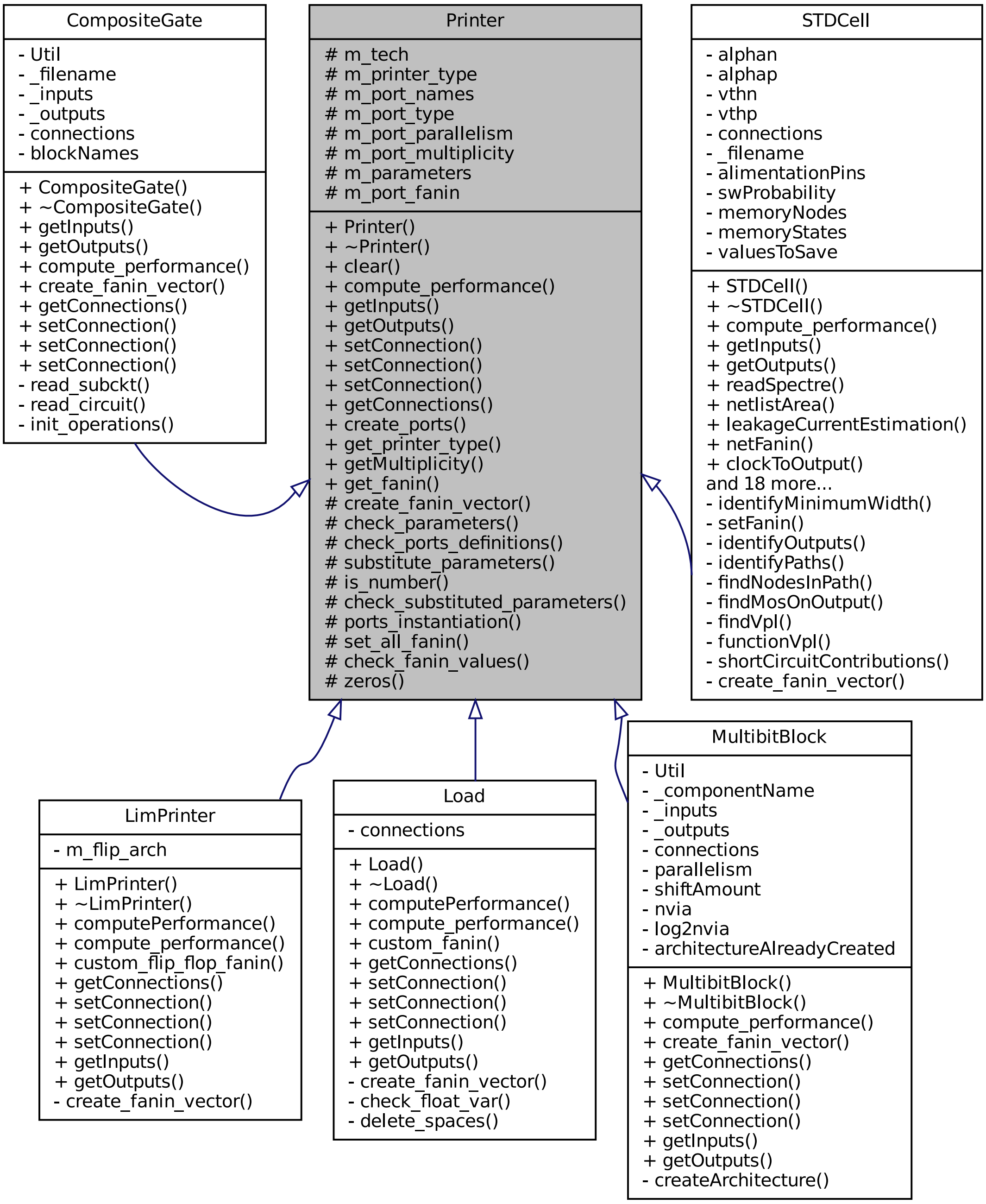


Fig. 7.6 Inheritance graph of the `Printer` `class`.

- `m_port_multiplicity` is used to specify how many ports are instantiated;
- `m_parameters` defines the parameters needed by the block at compile time, e.g., considering the multiplier block, the instance requires at least the number of bits of input and output ports;
- `m_port_fanin` specifies the fanin value of each input port.

These vectors are initialized inside the constructor of a block that can be a `STDCell`, `CompositeGate` or `MultibitBlock` and the position of the parameters in the vectors is related to the index of the `m_port_names` vector. An example of a ripple-carry adder is shown in Listing 38.

```
m_printer_type = _componentName;
if (_componentName == RCA_KEYWORD)
{
m_parameters = {"PARALLELISM"};
_inputs = {"A", "B", "AS"};
_outputs = {"SUM", "CO"};
m_port_parallelism = {"PARALLELISM", "PARALLELISM", "1", "PARALLELISM",
 ↪  "1"};
m_port_multiplicity = {"1", "1", "1", "1", "1"};
}
for (const auto &x: _inputs)
{
m_port_names.push_back(x);
m_port_type.emplace_back("input");
}
for (const auto &x: _outputs)
{
m_port_names.push_back(x);
m_port_type.emplace_back("output");
}
```

Listing 38 Ripple-carry adder definition of the initialization vectors.

The ripple-carry adder has only one parameter called PARALLELISM, which specifies the parallelism of the "A", "B" and "SUM" ports, as indicated in the `m_port_parallelism` variable. PARALLELISM is defined in round brackets near the component name in the Init section of the input **.dex** file, e.g. `RCA Adder(16)`. The multiplicity parameter specifies the number of ports at the specific index: if, for instance, the multiplicity for port "A" in the RCA example is equal to 2, DExIMA automatically creates two ports "A" named "A0" and "A1". The multiplicity is particularly useful in blocks like N-via multiplexers. The fanin vector `m_port_fanin` is defined in the component instantiation since it depends on the technology and the model.

7.2.3 The STDCell class

The `STDCell class` is the main class of DExIMA-Backend, which contains and describes all the calculations needed to estimate the performance of a standard cell at the architectural-level. Starting from a netlist description at the transistor-level, which is fundamental for mapping the connections between the MOSFETs and defining their sizes (widths and lengths), the class can build an internal model of the standard cell. In Fig. 7.7, the methods and attributes of the `STDCell class` are shown. The attributes are in the first part delimited by a horizontal line, while the methods are located in the remaining part. The high-level scheme of the related modules of the `STDCell class` is shown in Fig. 7.8. As highlighted by this scheme,

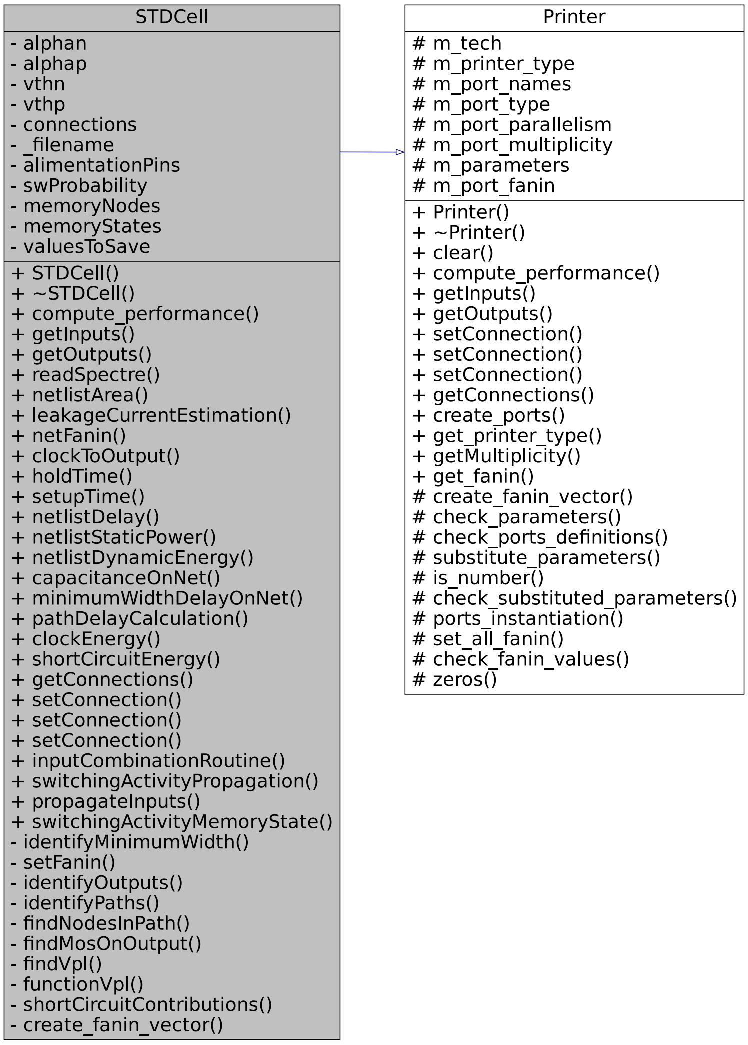


Fig. 7.7 UML diagram representation of the `STDCell class`.

the STDCell class has an important aggregation to the STDCellAttributes class. This class, instantiated as the private attribute called valuesToSave, contains important parameters for the standard cell, such as the netlist description, the inputs/outputs lists, the output nodes, the minimum width of the transistors, the value of the static power, short circuit energy and area, and finally the chosen path containing the lists of internal nets involved in the critical path. Similarly, the struct MOS is a composition of the STDCellAttributes class and uniquely indentifies the connections of a MOS object (by means of Gate, Drain, Source, Bulk attributes), the name of the MOS, the size (width and length) and the MOS type (NMOS or PMOS). A reference to the technology object, together with a string, are

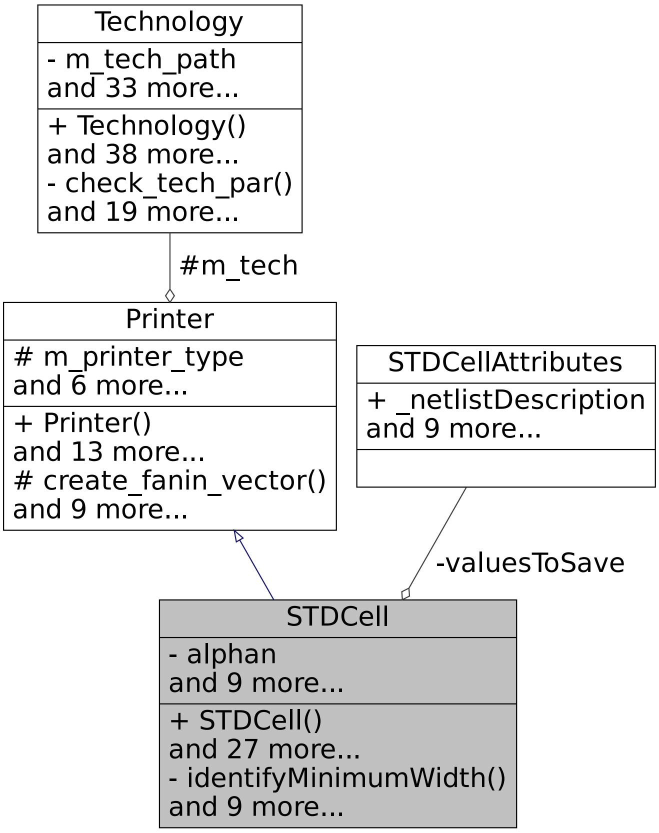


Fig. 7.8 Related class elements of the STDCell class.

passed to the constructor STDCell(Technology &tech, const std::string &filename). The technology object contains all the parameters and functions for the technological part needed by STDCell class to perform evaluations. The const std::string &filename instead refers to the file name containing the netlist description at transistor-level of the selected standard cell. Inside the constructor class, several operations are made:

1. *Definition of the input/output pins of the cell.* Inside the `valuesToSave` variable of type `STDCellAttributes`, the vectors containing the input and output names are initialized.

2. *Netlist parsing.* The netlist is parsed calling the method `void readSpectre()` and saved inside the `std::vector<MOS> _netlistDescription` variable, stored inside the `valuesToSave` object. Moreover, each MOS is also mapped by connecting its terminals to the internal nets of the standard cells, replicating the input Spectre file.

3. *Identification of the intermediate outputs inside the netlist.* An intermediate output is defined as the common node between a PMOS and an NMOS transistor. Intermediate outputs are fundamental, particularly for the critical path, dynamic power, and switching activity calculations. An example of the intermediate output concept and the critical path calculation is shown in Fig. 7.26: the standard cell is represented as a graph, which nodes are the inputs, intermediate outputs, and outputs of the standard cell itself, and the critical path is estimated as the longest path between one input and output.

4. *Identification of the minimum transistor width.* If not provided, the minimum width is identified by parsing the initial netlist and finding the transistor having the minimum width.

5. *Memorize the analyzed standard cell.* To speed up the execution time of DExIMA-Backend, the parsed standard cell and its parameters described inside the `STDCellAttributes` object are saved inside a shared map named `parsedBlocks`. If a similar standard cell is instantiated again, the values are fetched directly from this map, avoiding useless and time-consuming calculations.

6. *Definition of the Printer attributes.* Some common parameters are set in the `Printer` base class, such as the ports' name, their parallelisms, and the direction (input/output).

Routine to compute the performance

ModulePerformance *STDCell::compute_performance(Module &module, const float &sPower) is the most important method that is in charge of evaluating the performance of the standard cell. It returns a ModulePerformance* object that essentially contains all the information about the standard cell regarding power, area, delay, etc. The C++ code for compute_performance method is reported in Listing 39, while the call graph is shown in Fig. 7.10.

```
ModulePerformance *STDCell::compute_performance
(Module &module, const float &sPower)
{
        //computes the performance
        vector<string> par;
        vector<float> fanout;
        //Base temp variables
        ModulePerformance *performance = nullptr;
        float dynamicEnergy{0};
        float delay{0};
        float contamination{0};
        float clockOutput{0};
        float setup{0};
        float hold{0};
        for (auto &output: valuesToSave->_outputs)
        {
                //push back the fanout values
                fanout.push_back(module.get_fanout(output, 0));
        }
        //*** Computation of the performance parameters ***
        //if the static power is not already present
        //in the map of parsed blocks
        if
        (sPower == -1.0 && valuesToSave->staticPower == -1.0)
        {
                valuesToSave->staticPower =
                netlistStaticPower();
        }
        else if
        (sPower != -1.0 && valuesToSave->staticPower == -1.0)
        {
                valuesToSave->staticPower = sPower;
        }
        switchingActivityPropagation(module.getPowerAttributes());
        dynamicEnergy = netlistDynamicEnergy(fanout);
        //if the area is not already present
        //inside the map of parsed blocks
        if (valuesToSave->area == -1.0)
        {
                valuesToSave->area = netlistArea();
        }
        contamination = netlistDelay(fanout);
        //Create the object and return it
        performance = new
        ModulePerformance(valuesToSave->staticPower,
        dynamicEnergy, valuesToSave->area, contamination);
        //Insert the contamination delay
        performance->insert_timing_attribute
        (CONTAMINATION, contamination);
        if
        (contains({FLIPFLOP_KEYWORD, FLIPFLOPC2MOS_KEYWORD,
        FLIPFLOP_KEYWORD, FLIPFLOPCELL_KEYWORD},
        m_printer_type))
        {
                clockOutput = clockToOutput(fanout);
                setup = setupTime(fanout);
                hold = holdTime(fanout);
                performance->insert_timing_attribute
                (SETUP, setup);
                performance->insert_timing_attribute
                (HOLD, hold);
                performance->insert_timing_attribute
                (CK_TO_OUTPUT, clockOutput);
        }
        return performance;
}

```

Listing 39 Code of the compute_performance method.

Inside this method, several calculations are made:

1. *Estimation of the standard cell fanout.* To correctly calculate the performance, it is necessary to know if the standard cell is connected to other blocks and the value of the output load. Therefore, the fanout of the cell is computed

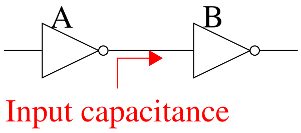


Fig. 7.9 Fanout evaluation example: two inverters.

considering the connections to other cells, and this information is stored inside the `Module class`, which is passed as a reference to the `compute_performance` method. Considering, for instance, two NOT gates connected as in Fig. 7.9, the fanout of inverter A is equal to the equivalent input capacitance of inverter B. The capacitances are estimated by DExIMA-Backend using the model discussed in subsection 7.3.4. The corresponding code for the fanout calculation is reported in Listing 39, lines 15-20.

2. *Calculation of the static power.* At this point, the performance estimation begins, and the first figure of merit is the static power. In lines 23-33 of Listing 39, static power can be both computed by means of `netlistStaticPower()` method or set by a variable named `sPower`: this last variable is passed to the `compute_performance` method, which will be different from -1 when the static power of that particular block was already evaluated since it always assumes the same value. The concept of `sPower` variable reduces computational efforts, improving performance. The method `netlistStaticPower()` evaluates the leakage of the cell, following the approach explained in subsection 7.3.3. In the call tree graph shown in Fig. 7.10, `netlistStaticPower` estimates the leakage current, that considers the contributions of the I_{gate} and I_{off}, provided by the Technology object.

3. *Propagation of the switching activity.* At line 34 of Listing 39, the switching activities are propagated inside the netlist by means of the `switchingActivityPropagation()` method. It takes in input a `power_attribute` object that contains information about the pin, the toggle rate, the combination of bits associated with that pin (obtained after the simulation), and the hierarchy of the signal in the project. The switching activities, indicated with α_i where i indicates a net, are fundamental for estimating the dynamic energy and power. The switching activity α_i of each node is calculated considering the back-annotation process, and in particular, 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 net state is different from 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. Otherwise, when a worst-case estimation is performed without the simulation process, a constant toggle rate is set for each net specified inside the input **.dex** file. In the scheme in Fig. 7.10, the method `switchingActivityPropagation` calls the `switchingActivityMemoryState` and `Technology::get_probability` methods. The first one calculates the switching activities for the cells composed of synchronous memory elements, i.e., a flip-flop or a latch. In this particular case, the switching activities are propagated considering the previous values stored in the memory node of the netlist; in all the other cases, the `switchingActivityMemoryState` method propagates the switching activities normally by means of the `propagateInputs` method. All the switching activities of each net are written inside a private map. More details about the switching activity calculations are provided in subsection 7.3.5.

4. *Estimation of the netlist dynamic energy.* The dynamic energy is composed of two contributions: the first comes from the charge/discharge of the internal net capacitances, while the second considers the short circuit current. The equation of the dynamic energy due to the capacitance charge/discharge is given by:

$$E_{dyn} = \frac{V_{dd}^2}{2} \sum_{\forall \text{nets}_i} (C_{load_i} \times \alpha_i) = P_{dyn} \times T_{ck} \tag{7.1}$$

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 in `STDCell class`. Instead, the short circuit energy considers the current between PMOS and NMOS during switching of the output: this contribution is discussed in detail in subsection 7.3.2. At line 35 of Listing 39, DExIMA-Backend considers each net of the standard cell and evaluates both parameters. The fanout vector is passed to the `netlistDynamicEnergy()` to consider potential connections to other blocks in output. In Fig. 7.10, the entire call tree of the `netlistDynamicEnergy()` method is shown. The short circuit energy estimation evaluates the fanin of the block, the capacitances of each net, and the internal functions for the short circuit computational model (see subsection 7.3.2). Similarly, the dynamic energy of the charge/discharge capacitances considers again the fanin and the switching activity values, already written inside the map. More details about estimating the dynamic energy and power are provided in subsection 7.3.6.

5. *Estimation of the standard cell area.* If the area was not already been evaluated for the considered standard cell (meaning that the `valuesToSave->area` is different from -1), it is evaluated inside the `netlistArea()` method. The computational model is explained in subsection 7.3.7.

6. *Delay calculation.* In lines 42-64, different delays are evaluated. Based on the standard cell type (combinational or sequential), the values of the contamination, clock to output, setup, and hold delays are computed with methods `netlistDelay`, `clockToOutput`, `setupTime` and `holdTime`, respectively. In Fig. 7.10, all the standard cells firstly identify all the paths inside the block (with `identifyPaths`) and evaluate their delays (with `pathDelayCalculation`). The path identifications rely on the graph object that finds all the paths between two distinct nodes. Instead, the delay is computed considering the minimum current on the node and uses the on-current parameter: further details on the delay calculation model are discussed in subsection 7.3.8.

Each performance parameter is passed to the `performance` object, as written in line 48 of Listing 39.

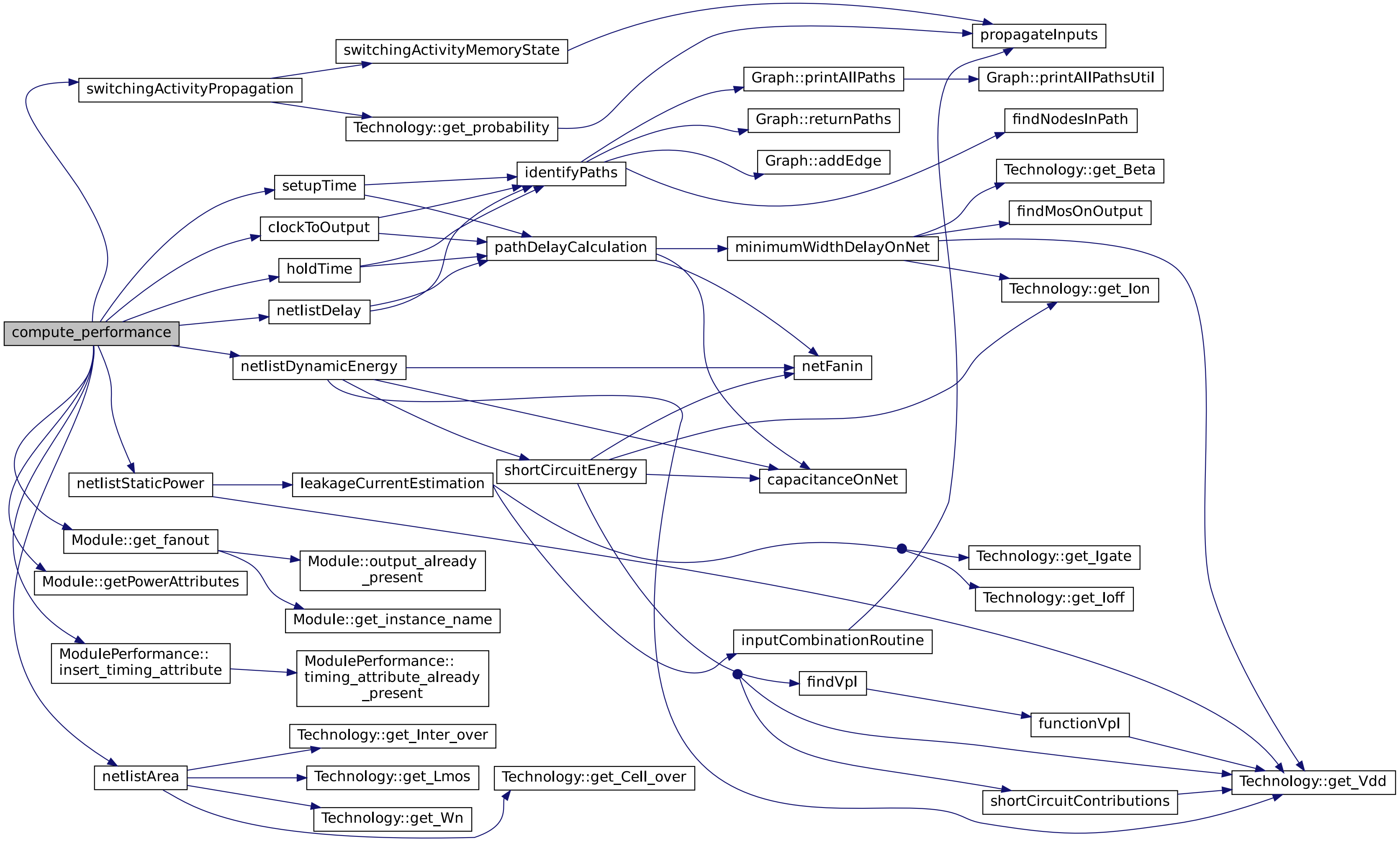


Fig. 7.10 Call tree of the `compute_performance` method.

7.2.4 The CompositeGate and MultibitBlock classes

Only some basic gates can be represented as a simple standard cell. However, there are cases in which a gate is composed of multiple standard cells, so a proper class is needed to describe these objects. The `CompositeGate class` comes in handy: its collaboration diagram is shown in Fig. 7.11. The `CompositeGate class` has a similar behavior to the `STDCell class` since it starts by parsing a Spectre netlist. However, this time, the netlist contains multiple standard cells described at transistor-level. For instance, a Spectre netlist of a flip-flop with enable composite gate will be composed of a DFF_X1 with an OAI21_X1 and a NAND2_X1 cells, as reported in Listing 40.

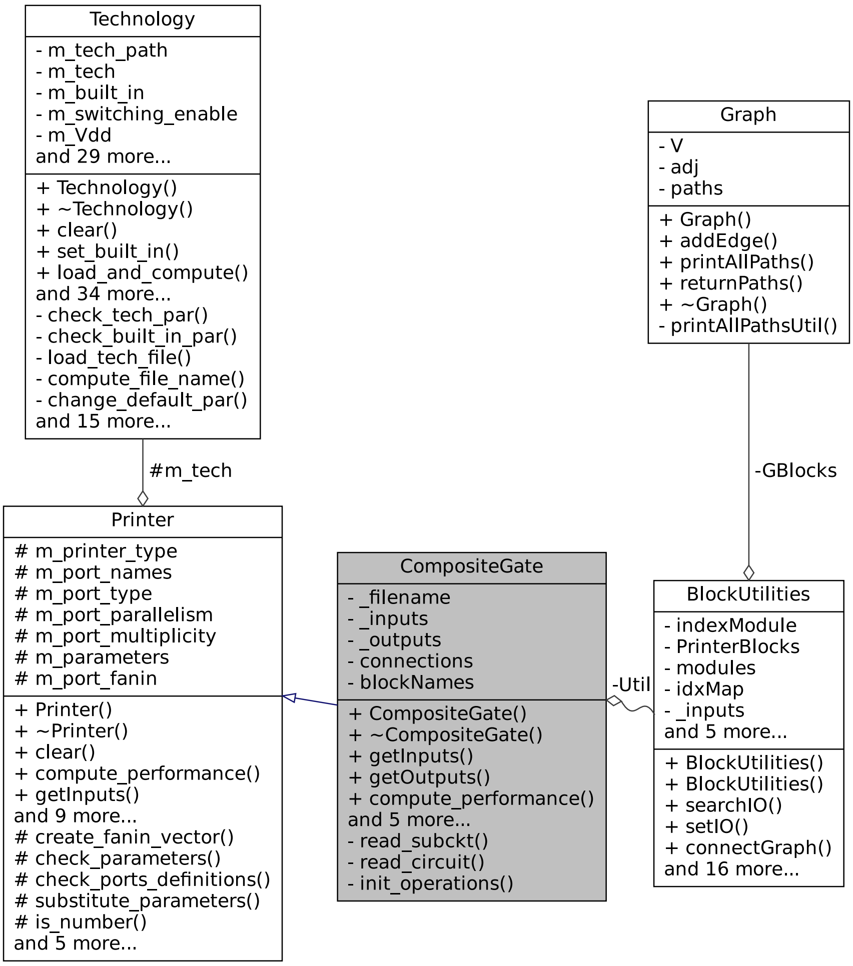


Fig. 7.11 Collaboration diagram for the `CompositeGate class`.

```
// Library name: LiM
// Cell name: CELL
// View name: schematic
I3 (net1 CK RN RD net3) FLIPFLOP
I5 (EN WR net2) NAND
I4 (net3 EN net2 net1) OAI21

```

Listing 40 Extract of the flip-flop with enable composite gate Spectre netlist.

Inside the constructor of the `CompositeGate class`, the Spectre netlist will be parsed, and inputs, outputs, and parallelism of the ports will be defined exactly as the `STDCell class`. As shown in Fig. 7.11, the `CompositeGate class` has a link with the `BlockUtilities class`, a fundamental object that provides the methods and tools for the performance estimations of more complex blocks than the standard cells, so the composite gates and the multibit blocks. Similarly to the `STDCell class`, also in `CompositeGate class` and `MultibitBlock class` there is `ModulePerformance *compute_performance` method. However, this time the `BlockUtilities class` is called, which properly redirects the calculations to the `STDCell class`. The collaboration diagram of the `BlockUtilities class` is shown in Fig. 7.12. The usage of the `BlockUtilities class` is summarized in the following.

1. *Creation of the subcomponents.* During the architecture creation inside the `CompositeGate`, `MultibitBlock` and `FlipFlopArchitecture` classes, the subcomponents are instantiated by calling the `void createComponent` function, that instantiates a `STDCell` or `CompositeGate` object. For instance, in a ripple-carry adder, the instantiated objects will be full adders that belong to the standard cell category, while in a register with enable, multiple `CompositeGates` of flip-flop with enable cells are required. The calling tree of the `void createComponent` method is shown in Fig. 7.13 (a).

2. *Representation of the CompositeGate/MultibitBlock as a directed graph.* `BlockUtilities class`'s method `void connectGraph()` is called by `CompositeGate class`, `MultibitBlock class` and `FlipFlopArchitecture class` at the moment of the composite block creation. The calling tree of the `void connectGraph` function is shown in Fig. 7.13 (b). In the graph representation, the nodes are the subblocks or the input/output pins, while the

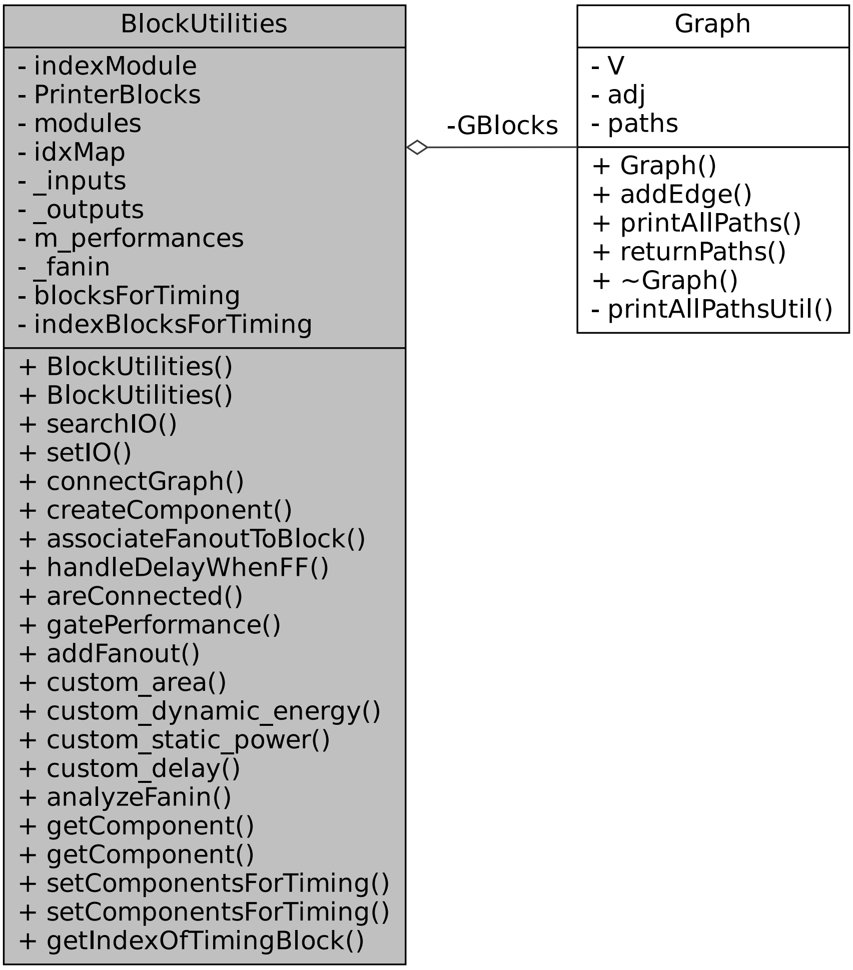


Fig. 7.12 Collaboration graph of the `BlockUtilities` `class`.

edges are the connections between the subblocks composing the `CompositeGate` or the `MultibitBlock`. This representation is useful to estimate the critical path of the block since the delay calculation simply becomes a Depth First Traversal algorithm, finding all the possible paths between two nodes (an input and an output). This part will be deeply explained in subsection 7.3.8.

3. *Associate the fanout to the block.* When the `compute_performance` is called in a `CompositeGate`, `MultibitBlock` or `FlipFlopArchitecture`, the fanout is computed and associated to the top-level block for dynamic power and delay calculations.

4. *Computation of the subblocks performance.* The `gatePerformance` method allows estimating the performance of the subblocks. It takes in input a `std::vector <power_attribute> &pa` that contains the information of the

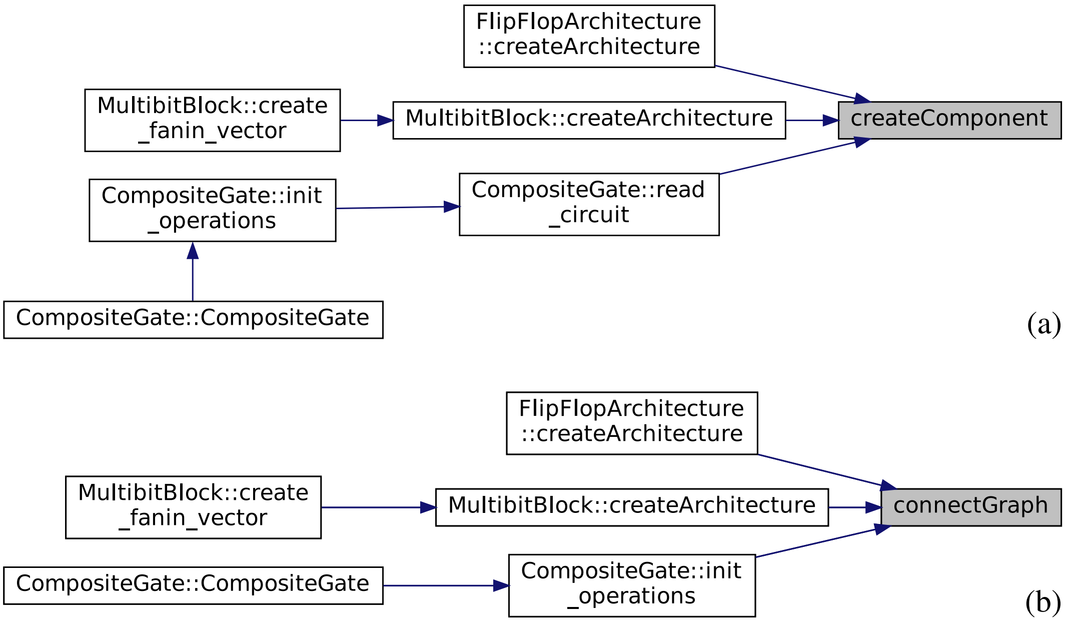

FlipFlopArchitecture ::createArchitecture
MultibitBlock::create _fanin_vector
MultibitBlock::createArchitecture
createComponent
CompositeGate::init _operations
CompositeGate::read _circuit
CompositeGate::CompositeGate
(a)
FlipFlopArchitecture ::createArchitecture
MultibitBlock::create _fanin_vector
MultibitBlock::createArchitecture
connectGraph
CompositeGate::CompositeGate
CompositeGate::init _operations
(b)

Fig. 7.13 (a) Call graph of the `void BlockUtilities::createComponent` function. (b) Call graph of the `void BlockUtilities::connectGraph` function.

switching activities and toggle rates for each pin, extracted from the power section of the input **.dex** file. In this method, a for cycle considers all the subcomponents composing the complex block and calls the `ModulePerformance* compute_performance` method, explained in deep in section 7.2.3, saving the results inside a map of `ModulePerformance*` objects.

The `MultibitBlock class` is based on a similar behavior of the `CompositeGate class`. However, instead of using a procedure of netlist parsing, the architecture is created inside by means of the `void MultibitBlock::createArchitecture` method. The reason behind this choice is that, differently from the `CompositeGate` objects, `MultibitBlock` objects have structures that strictly depend on the parallelism, so a generic procedure must be taken into account. For instance, in the snippet of code for the ripple-carry adder model generation, shown in Listing 41, the subblocks are created in lines 9-10 and connected in lines 15-28. Finally, the components that contribute to the timing calculations are set with `Util.setComponentsForTiming()`. This last function, if called without arguments, sets all the components instantiated as belonging to the critical path calculations; but `setComponentsForTiming` can also be used with arguments that specify a subset of components: this is extremely helpful in regular and complex structures with many subblocks since by selecting a smaller subset, it is possible to reduce the delay calculation execution time.

```
//create the vector of internal blocks connections
std::vector<std::vector<std::string>> internalBlocksConnections;
if (_componentName == RCA_KEYWORD)
{
//if the component is an RCA
for (int i = 0; i < parallelism; i++)
{
//create the full adders
Util.createComponent("FullAdder_" + to_string(i), new STDCell(m_tech,
 ↪  FULL_ADDER_KEYWORD));
Util.createComponent("EXOR_" + to_string(i), new STDCell(m_tech,
 ↪  XOR_KEYWORD));
}
for (int i = 0; i < parallelism - 1; i++)
{
//for each component, set the connections
Util.getComponent("EXOR_" + to_string(i))->setConnection({_inputs[1] + "["
 ↪  + to_string(i) + "]", _inputs[2], "out_xor_" + to_string(i)});
if (i == 0)
{
Util.getComponent("FullAdder_" + to_string(i))->setConnection({_inputs[0]
 ↪  + "[" + to_string(i) + "]","out_xor_" + to_string(i),"net_Cin_" +
 ↪  to_string(i),_outputs[0] + "[" + to_string(i) + "]",_inputs[2]});
}
else
{
Util.getComponent("FullAdder_" + to_string(i))->setConnection({_inputs[0]
 ↪  + "[" + to_string(i) + "]","out_xor_" + to_string(i),"net_Cin_"
 ↪  +to_string(i),_outputs[0] + "[" +to_string(i) + "]","net_Cin_" +
 ↪  to_string(i + 1)});
}

}
Util.getComponent("EXOR_" + to_string(parallelism -
 ↪  1))->setConnection({_inputs[1] + "[" + to_string(parallelism - 1) +
 ↪  "]", _inputs[2], "out_xor_" + to_string(parallelism - 1)});
Util.getComponent("FullAdder_" + to_string(parallelism - 1))->setConnection({_inputs[0] + "[" + to_string(parallelism - 1) + "]","out_xor_" + to_string(parallelism-1),"net_Cin_" + to_string(parallelism - 1),_outputs[0] + "[" + to_string(parallelism - 1) + "]","CO"});
//set the components for timing calculations
Util.setComponentsForTiming();
}
```

Listing 41 Snippet of code of the `createArchitecture` method for a ripple-carry adder model.

7.2.5 The Module class

Each element of the architecture is described as a module, meaning that inside DExIMA, each component (`STDCell`, `CompositeGate`, `MultibitBlock`, `FlipFlopArchitecture`) is associated to a `Module` object. The `Module` object is created in the parsing phase or the `void BlockUtilities::createComponent` method and contains useful information external from the component model for the estimation of the performance. The word "external" refers to the data that is not linked to the component structure but on how the component is connected to others and how the component is seen from the top-level view. The `Module class` contains the value of the fanout for each output pin, the fanin, the parallelism of each port, the instance name etc. The UML view of the `Module class` is shown in Fig. 7.14.

7.2.6 The Lim and the Architecture classes

Memory parameters and specifications are stored and managed by the `Lim class`. These are the LiM name and type (e.g., SRAM, FLIPFLOP), the memory dimensions, the read/write address parallelism, the number of rows and columns, and the modules to be instantiated inside the LiM array. As shown in Fig. 7.15, the `Lim` object contains two maps, one for the modules inside the LiM cells, called `memory_array` and the other for the IRL modules (`m_modules`). The first one is a 4D map that is addressed by row, column indexes, and the module name, while the second one is a simple map addressed by the IRL module name: these maps are essential since they contain the reference to each module that will be addressed in the performance

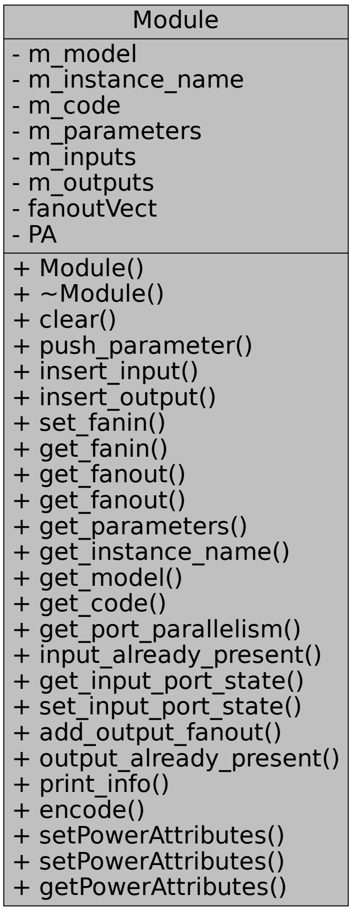


Fig. 7.14 `Module class` collaboration diagram.

estimation phase. Similarly to the `Lim class`, the `Architecture class` contains all the modules and components of the circuit, including the LiM memories: the collaboration graph of the `Architecture class` is shown in Fig. 7.16. As can be seen from the figure, there is also a `BusParser` object within the `Architecture class`. This is because the `Architecture class` contains the detailed description of the LiM's internal components, which is necessary in estimating the contribution of the internal bus to the LiM, as will be explained later in section 7.3.9.

7.2.7 The Performance class and its inherited classes

The `Performance class` contains the performance data regarding power, delay, energy, and area. `Performance class` is a superclass of other three classes, `Arc`⌋

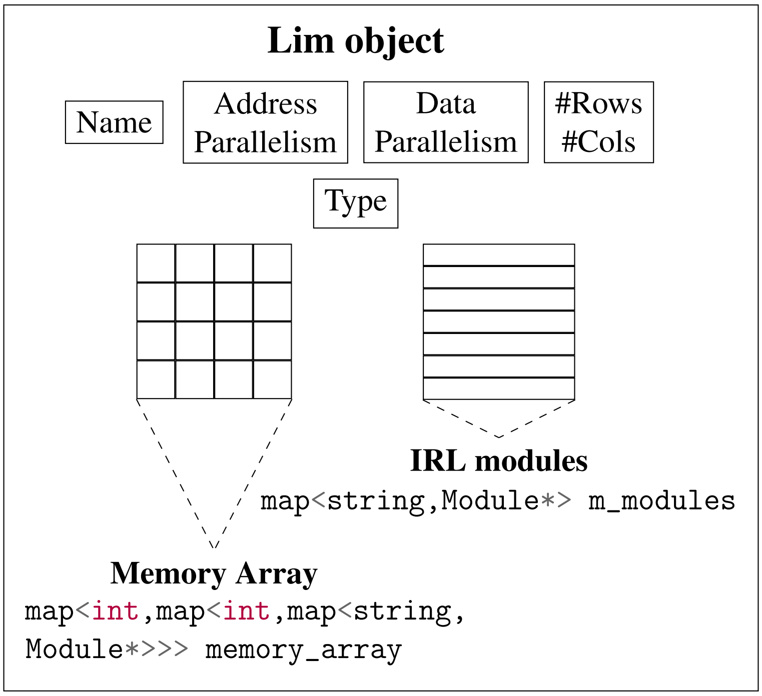


Fig. 7.15 Lim object: high-level scheme

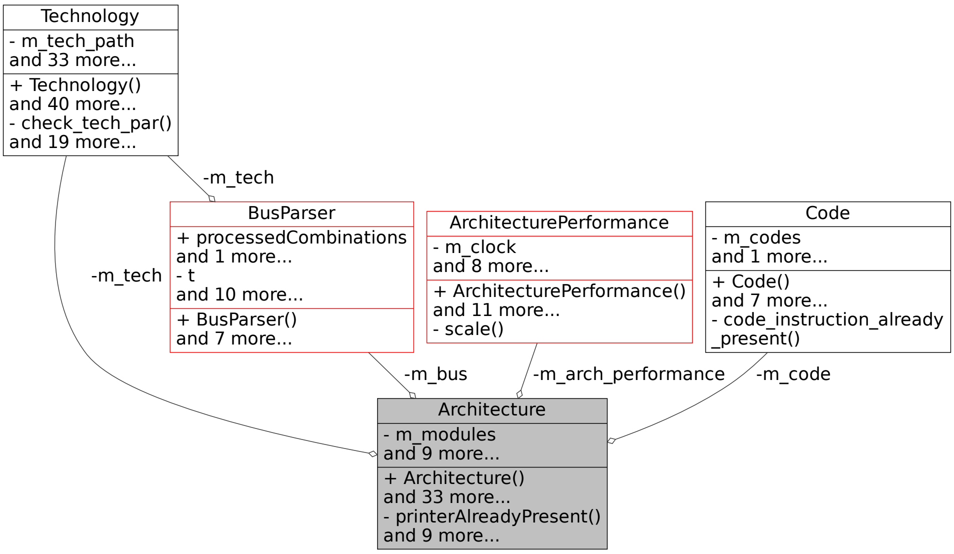


Fig. 7.16 Collaboration graph of the `Architecture` `class`.

`hitecturePerformance` `class`, `InstructionPerformance` `class` and `ModulePerformance` `class` as shown in Fig. 7.17.

The `ArchitecturePerformance` `class` contains the performance of the entire architecture, as suggested by its methods; the `InstructionPerformance` `class`

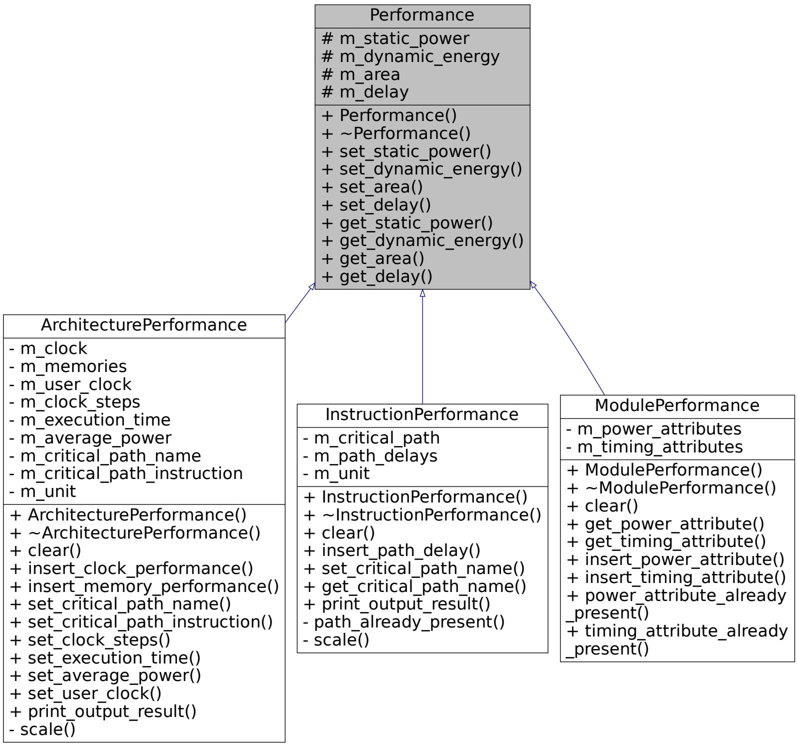


Fig. 7.17 Performance class inheritance graph.

contains the performance of a particular instruction. An instruction is defined in the begin instructions part of the **.dex** input file, as already explained in section 7.1; the ModulePerformance class instead, as suggested by its name, contains the performance results of a single module.

7.3 Computational model

This section explains the model employed within DExIMA-Backend for computing the performance of CMOS-based circuits.

7.3.1 Model for parallel and series transistors

From the Cadence Virtuoso reference manual, the calculations on parallel MOS devices for Layout-Versus-Schematic (LVS), parallel transistors having the same signal on the gate can be collapsed into a single equivalent transistor having a width and length equal to:

$$\begin{cases} A & = \sum_{i=0}^{N-1} W_i \times L_i \\ C & = \sum_{i=0}^{N-1} \dfrac{W_i}{L_i} \\ W_{eq} & = \sqrt{A \times C} \\ L_{eq} & = \sqrt{\dfrac{A}{C}} \end{cases} \tag{7.2}$$

For instance two parallel transistors with equal length $L_1 = L_2 = L$, that is a condition that is almost always true in digital circuits, the corresponding width is $W_{eq} = W_1 + W_2$. Similarly, two series transistors having the same input signal on the gate can be reduced to an equivalent transistor, which dimensions are defined as:

$$\left(\frac{W}{L}\right)_{eq} = \frac{\prod_{i=0}^{N-1}\left(\dfrac{W_i}{L_i}\right)}{\sum_{i=0}^{N-1}\left(\dfrac{W_i}{L_i}\right)} \tag{7.3}$$

Considering the same example with $L_1 = L_2 = L$, the width of the equivalent transistor is equal to $W_{eq} = \frac{W_1 W_2}{W_1 + W_2}$ and its length is $L_{eq} = L$.

7.3.2 Short circuit current

Estimating the short circuit current is a challenging task. Normally, it is estimated by means of simulations and measurements of the peak current between VDD and GND during a transition of the output. However, DExIMA-Backend cannot simulate the transient of the circuit and replicate this situation, so an approximation must be taken into account. The short circuit current estimation is exhaustively discussed in works [143–146], that implement models with different complexities and accuracy. In DExIMA-Backend, a trade-off between complexity and accuracy must be considered, with the clear goal of avoiding models that necessarily require parameters obtainable only with simulations. The chosen model is presented in [143], based on the α-power law, i.e., including carrier velocity and saturation effects which are predominant in short-channel devices. This calculation estimates the short circuit power as the short circuit transient in a NOT gate. This means that the other gates are reduced to an equivalent inverter, following the principles explained in subsection 7.3.1. However, the transistor reduction constitutes a clear approximation of the problem since the gates of the MOS transistors are rarely connected to the same input signal. The estimation of the short circuit current necessarily requires dynamic simulations of the gates, which are not possible with DExIMA-Backend, so the results will be affected by errors. However, the chosen approximation represents a good trade-off between complexity and accuracy since it gives an indication of the short-circuit component. The steps to estimate the short circuit current are:

1. *Find the equivalent netlist made of only inverters.* This operation is performed considering the intermediate outputs of the standard cell (see subsection 7.2.3) and by obtaining the equivalent PMOS and NMOS transistors that have in common the intermediate output net. An example of an AND2_X1 gate reduction is shown in Fig. 7.18, where $\mathrm{WP_{eq1}} = \mathrm{WP_1} + \mathrm{WP_2}$, $\mathrm{WN_{eq1}} = \frac{\mathrm{WN_1WN_2}}{\mathrm{WN_1+WN_2}}$, $\mathrm{WN_{eq2}} = \mathrm{WN_3}$ and $\mathrm{WP_{eq2}} = \mathrm{WP_3}$.

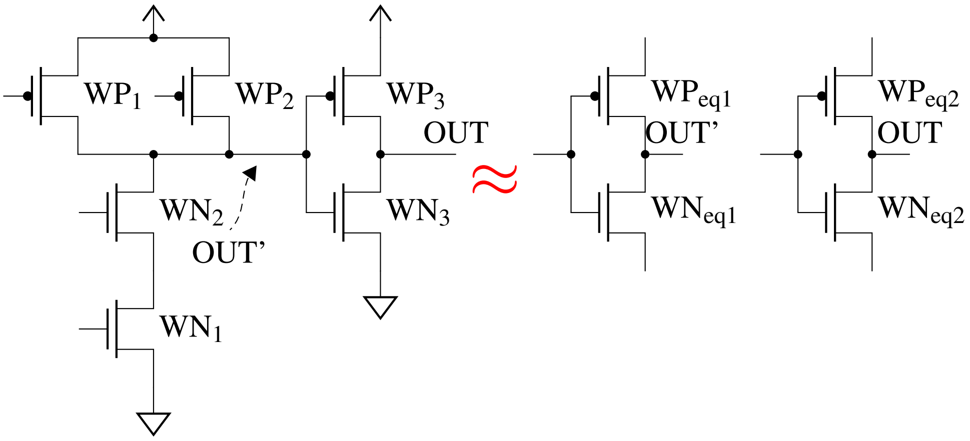


Fig. 7.18 AND2_X1 gate reduction example for the short circuit power computation.

2. *For each inverter in the equivalent netlist, perform the computation of the short circuit power.*

3. *Sum all the inverter short circuit power contributions to obtain the final value.*

Short circuit current of an inverter

The short circuit current is estimated considering the equations presented in [143]. The starting point is the definition of the drain current I_D, that is equal to:

$$I_D = \begin{cases} 0, & V_{gs} \leq V_{th} \\ k_1 \left(V_{gs} - V_{th}\right)^{\alpha/2} V_{ds}, & V_{ds} < V'_{do} \\ k_s \left(V_{gs} - V_{th}\right)^{\alpha}, & V_{ds} \geq V'_{do} \end{cases} \tag{7.4}$$

Where α is the velocity saturation index and

$$k_l = \frac{I_{do}}{V_{do}\left(V_{dd} - V_{th}\right)^{\alpha/2}} \tag{7.5}$$

$$k_s = \frac{I_{do}}{\left(V_{dd} - V_{th}\right)^{\alpha}} \tag{7.6}$$

$$V'_{do} = V_{do}\left(\frac{V_{gs} - V_{th}}{V_{dd} - V_{th}}\right)^{\alpha/2} \tag{7.7}$$

$$I_{do} = I_D(V_{gs} = V_{dd}) \tag{7.8}$$

$$V_{do} = V_{dsat}(V_{gs} = V_{dd}) \tag{7.9}$$

The input voltage is modeled as a ramp, both in rising and falling directions as reported in Equation 7.10.

$$V_{in} = \begin{cases} s_r t, & \text{rising input} \\ V_{dd} - s_f t, & \text{falling input} \end{cases} \tag{7.10}$$

The terms s_r and s_f are the input slopes, expressed as (V/s). Considering first the input rising case, the equation of the inverter output is written as:

$$C_L \frac{dV_o}{dt} = -I_n \tag{7.11}$$

So the equation for the output $V_o(t)$ is written as:

$$V_o(t) = V_{dd} - \frac{k_{s_n}}{C_L s_r} \frac{\left(s_r t - V_{th_n}\right)^{\alpha_n + 1}}{\alpha_n + 1}, \qquad \text{for } \frac{V_{dd} + V_{th_p}}{s_r} > t > \frac{V_{th_n}}{s_r} \tag{7.12}$$

Considering $V_o(0) = V_{dd}$. Most of the time, the NMOS transistor is in saturation region during the short circuit, so the Equation 7.12 represents a valid estimate. During a falling transition, the PMOS operates in linear region in the time interval defined by $n = \frac{V_{th_n}}{s_r}$ and $t_{pl} = \frac{V_{pl}}{s_r}$, while it is in saturation in the time interval between t_{pl} and $1 + p = \frac{V_{dd}+V_{th_p}}{s_r}$. The V_{pl} is the voltage in which the PMOS switches from linear to saturation regions, and it is defined as:

$$\frac{k_{s_n}\left(V_{pl} - V_{th_n}\right)^{\alpha_n+1}}{V_{do,p}C_L s_r\left(\alpha_n+1\right)} = \left(\frac{V_{pl} - V_{th_n}}{V_{dd} - V_{th_n}}\right)^{\alpha_p/2} \tag{7.13}$$

The power during the falling transition is obtained as:

$$\begin{aligned} P_{scf} = V_{dd}\int i_p(t)dt =& V_{dd}\left[\int_n^{t_{pl}} k_{l_p}\left(V_{dd} + V_{th_p} - s_r t\right)^{\alpha_p/2}\left(V_{dd} - V_o\right)dt\right] + \\ &+ V_{dd}\left[\int_{t_{pl}}^{1+p} k_{s_p}\left(V_{dd} + V_{th_p} - s_r t\right)^{\alpha_p} dt\right] \end{aligned} \tag{7.14}$$

Solving the integrals, the final equation for the falling power becomes:

$$\begin{aligned} P_{scf} =& \frac{k_{s_n}k_{l_p}V_{dd}\left(s_r t_{pl} - V_{th_n}\right)^{(\alpha_n+\alpha_p/2+2)}\alpha_n\Gamma\left(\alpha_p/2+1\right)\Gamma(\alpha_n)}{s_r^2 C_L\left(\alpha_n+\alpha_p/2+2\right)\left(\alpha_n+\alpha_p/2+1\right)\left(\alpha_n+\alpha_p/2\right)\Gamma\left(\alpha_n+\alpha_p/2\right)} + \\ &+ \frac{V_{dd}k_{s_p}\left(V_{dd} + V_{th_p} - s_r t_{pl}\right)^{(\alpha_p+1)}}{s_r(\alpha_p+1)} \end{aligned} \tag{7.15}$$

Where $\Gamma = (n-1)!$ is the gamma function. Following a similar approach, also the rising power is defined:

$$\begin{aligned} P_{scr} =& \frac{k_{l_n}k_{s_p}V_{dd}\left(s_f t_{nl} + V_{th_p}\right)^{(\alpha_p+\alpha_n/2+2)}\alpha_p\Gamma\left(\alpha_n/2+1\right)\Gamma(\alpha_p)}{s_f^2 C_L\left(\alpha_p+\alpha_n/2+2\right)\left(\alpha_p+\alpha_n/2+1\right)\left(\alpha_p+\alpha_n/2\right)\Gamma\left(\alpha_p+\alpha_n/2\right)} + \\ &+ \frac{V_{dd}k_{s_n}\left(V_{dd} - V_{th_n} - s_f t_{nl}\right)^{(\alpha_n+1)}}{s_f(\alpha_n+1)} \end{aligned} \tag{7.16}$$

From the Equation 7.15 and Equation 7.16, the total short circuit power can be obtained as:

$$P_{sc} = (P_{scr} + P_{scf})f \tag{7.17}$$

Where f is the frequency. The tricky part of the short circuit power modeling is obtaining the value of V_{pl}, expressed in Equation 7.13, which is a non-linear function.

In DExIMA-Backend, this value is obtained with an iterative procedure based on the false-position Ridders' method [147, 148]. The operative steps are the following:

1. Define two initial points x_0 and x_2 on two different sides of the root.
2. Evaluate the midpoint $x_1 = (x_0 + x_2)/2$.
3. Find the exponential function such that $h(x) = f(x)e^{ax}$ satisfies $h(x_1) = (h(x_0) + h(x_2))/2$. The value of a can be found with the following equation:

$$a = \frac{ln\left(\frac{f(x_1) - sign[f(x_0)]\sqrt{f(x_1)^2 - f(x_0)f(x_2)}}{f(x_2)}\right)}{(x_1 - x_0)} \tag{7.18}$$

4. Apply the false position method to find a value x_3 between x_0 and x_2. The equation of x_3 is the following:

$$x_3 = x_1 + (x_1 - x_0)\frac{sign[f(x_0)]f(x_1)}{\sqrt{f(x_1)^2 - f(x_0)f(x_2)}} \tag{7.19}$$

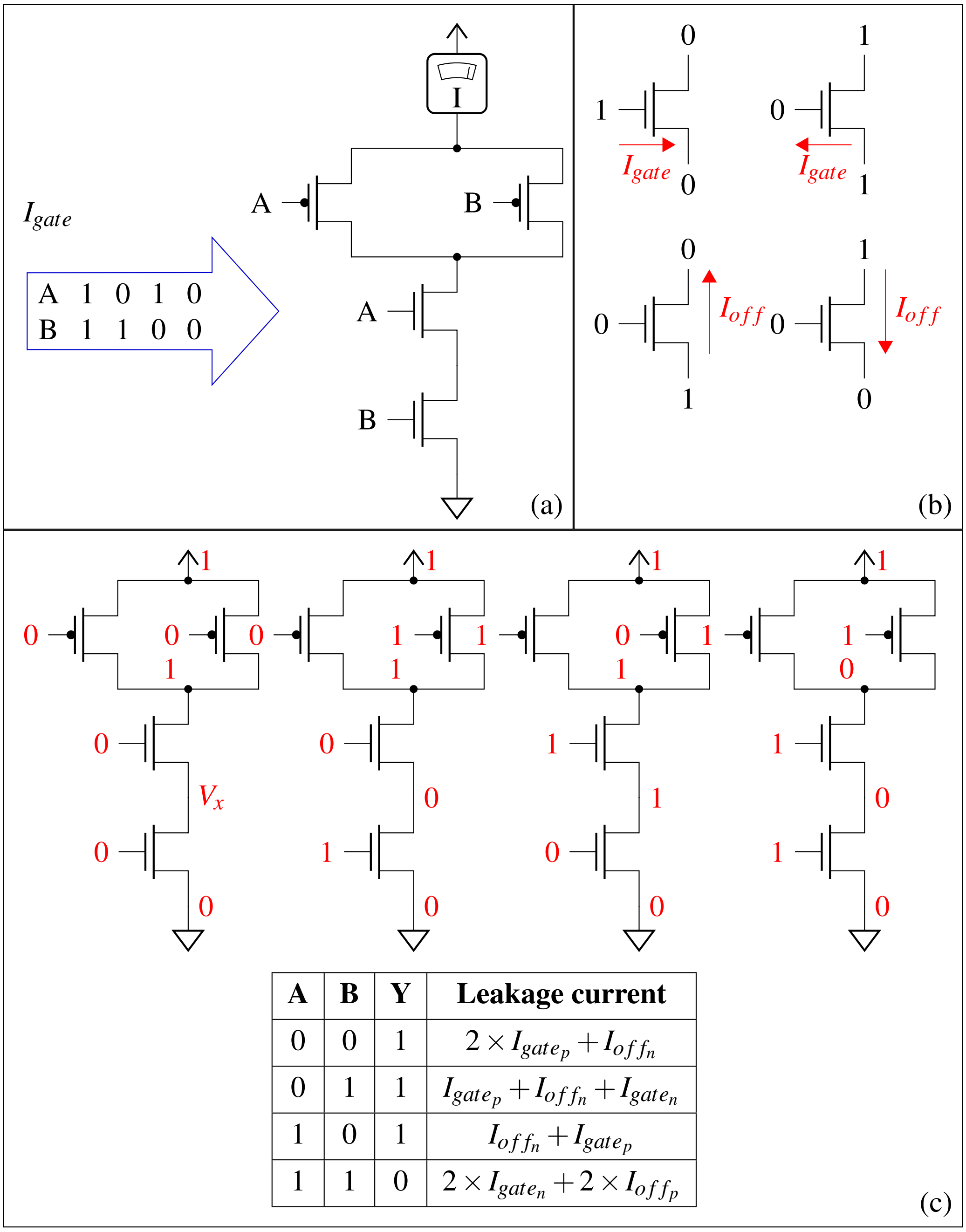
The C++ code that implements the Ridders' method belongs to the `STDCell` `class` and it is the following:

```
float STDCell::findVpl(float cl, float start, float stop, float idon)
{
    //final guess
    float y1 = functionVpl(stop, cl, idon);
    //initial guess
    float y0 = functionVpl(start, cl, idon);
    //max number of iterations
    int maxIter = 100;
    float y2, y;
    float vpl0, vpl1, vpl;
    vpl0 = start;
    vpl1 = stop;
    //definition of the tollerances
    float xtol = 1e-12;
    float ytol = 1e-12;
    for (int i = 0; i < maxIter; i++)
    {
        //definition of the middle point
        float vpl2 = (vpl0 + vpl1) / 2;
        //evaluation in the middle point
        y2 = functionVpl(vpl2, cl, idon);
        vpl = vpl2 + (vpl2 - vpl0) * sign(y0 - y1) * y2 / sqrt(y2 * y2 - y0 * y1);
        if (min(abs(vpl - vpl0), abs(vpl - vpl1)) < xtol) break;
        y = functionVpl(vpl, cl, idon);
        if (abs(y) < ytol) break;
        if (sign(y2) != sign(y))
        {
            vpl0 = vpl2;
            y0 = y2;
            vpl1 = vpl;
            y1 = y;
        }
        else if (sign(y1) != sign(y))
        {
            vpl0 = vpl;
            y0 = y;
        }
        else
        {
            vpl1 = vpl;
            y1 = y;
        }
    }
    return vpl;
}
```

Listing 42 Ridders' method implementation for V_{pl} computation.

7.3.3 Modeling the static power

Static or leakage power can be modeled by measuring the static current flowing from the supply voltage inside the standard cell for all possible input combinations. The high-level static current measurement scheme is depicted in Fig. 7.19 (a). This measurement is automatically done in characterization tools like Cadence Liberate. At the end of the procedure, the final value of the leakage power is obtained by performing the sum of the quiescent current and the gate current of high inputs over all combinations. DExIMA-Backend uses a different methodology based on static



A	B	Y	Leakage current
0	0	1	$2 \times I_{gate_p} + I_{off_n}$
0	1	1	$I_{gate_p} + I_{off_n} + I_{gate_n}$
1	0	1	$I_{off_n} + I_{gate_p}$
1	1	0	$2 \times I_{gate_n} + 2 \times I_{off_p}$

Fig. 7.19 (a) Measurement of the static power for all input combinations. (b) Leakage currents model used in DExIMA-Backend for the NMOS transistor.

estimations since real measurements of the currents are not possible. The idea is to start from the values of the I_{off} and I_{gate} currents, which can be easily obtained from the technological model, and to evaluate each input combination: the state of each transistor defines its contribution to the static power, that can be the off current or the gate current. The combinations in which the off or the gate currents contribute to the static power are shown in Fig. 7.19 (b) for an NMOS transistor, which are the same for the PMOS apart from the off current cases, in which the gate input is 1 instead of 0. I_{gate} and I_{off} of the reference 45nm CMOS technology are measured from Cadence Virtuoso by simply measuring the current flow in the cases shown in Fig. 7.19 (b). The results obtained are expressed as current per unit width since the current scales linearly with the width of the transistor. The values obtained are reported in Table 7.2.

Table 7.2 Off and gate current measured with Cadence Virtuoso for a NMOS transistor of the 45nm CMOS technology.

Parameter	**Value**	**Meaning**
Ioff	1.18E-01	*Off Current* (A/m^2)
Igate	1.28E-02	*Gate Current* (A/m^2)

An example of the calculation of the leakage current for a NAND gate is depicted in Fig. 7.19 (c): DExIMA-Backend considers all the possible input combinations and evaluates each contribution that depends on the states of the transistors and obtains the final current as the mean value, following the Equation 7.20.

$$I_{leak} = \frac{\sum_{j=\text{combination}} I_j}{\#\text{combinations}} \tag{7.20}$$

And finally the equation of the leakage power is:

$$P_{leak} = I_{leak} \times V_{DD} \tag{7.21}$$

However, DExIMA-Backend does not consider voltage values different from GND or VDD because it does not perform analog simulations but static estimations. Therefore, in the first combination case, the value V_x is treated as a logic-0, and the equation of the leakage current becomes:

$$2 \times I_{gate_p} + I_{off_n} \approx 2 \times I_{gate_p} \tag{7.22}$$

This approximation implies an error in estimating the leakage power.

7.3.4 MOS capacitance model

Estimating the MOS capacitances is very important to get accurate results since crucial parameters like dynamic power and delay strongly depend on these values.

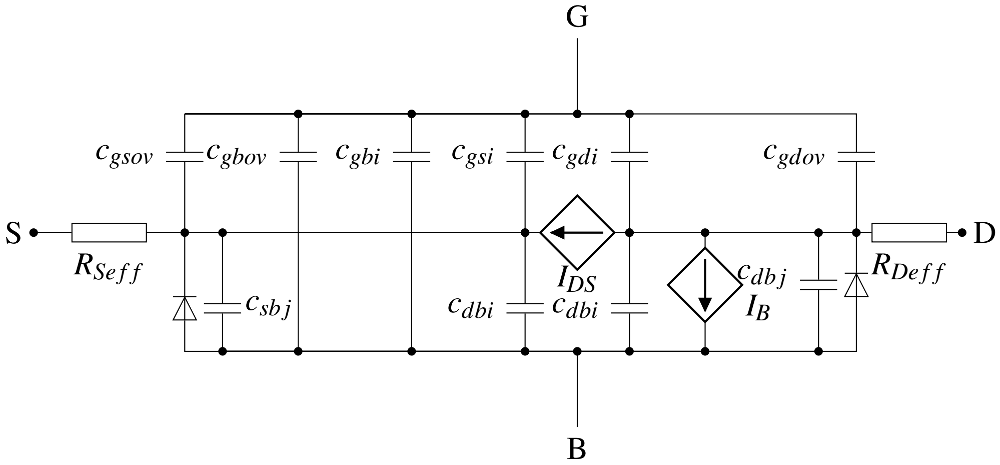


Fig. 7.20 MOS capacitance model used in Cadence Spectre [17].

In Fig. 7.20, the MOS Capacitance model is shown, which is the one used by Cadence Spectre [17] to perform simulations. Several contributions are considered in the model, such as junction capacitances (labeled by "j"), overlap capacitances (labeled by "ov"), and intrinsic capacitances (labeled by "i"): the accuracy of the MOS capacitances depends on how well these contributions are estimated, so a starting point should be clearly defined. For this purpose, the Cadence Virtuoso tool is employed together with the Physical Design Kit (PDK) of the 45nm CMOS technology from North State Carolina University (NCSU) [149]. The PDK contains the Spectre models of NMOS and PMOS transistors, characterized with BSIM4 [150]. BSIM4 is a very complex model that describes the behavior of the transistor devices in a very accurate way, also considering complex effects like the short-channel effects (SCEs), drain-induced barrier lowering (DIBL), and so on. Starting from the MOS model, the standard cell is designed and simulated, and if the results coming from the simulation are good, the standard cell is ready to be used. By performing a process called **characterization**, the standard cells are modeled in terms of power, area, and delay with different output loads and input transition times.

However, the BSIM4 Capacitance model is very complicated and strongly depends on the applied bias and the currents/voltages for each time instant, requiring a SPICE-level simulation unsuitable in DExIMA-Backend. For this reason, the starting BSIM4 model was simplified to replicate DExIMA-Backend static computations. According to the BSIM4 manual [150], this operation can be accomplished by changing `capMod` parameter: in particular, by setting `capMod = 0`, the capacitances are modeled with a simple and piece-wise approximation, with respect to the default `capMod = 2`, where a single-equation, bias-dependend and charge-thickness models are used. The self capacitances can be expressed as a sum of intrinsic and extrinsic contributions, as written in Equation 7.23.

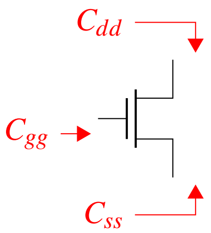


Fig. 7.21 Self capacitances of the MOS device

$$\begin{cases} C_{gg} & = C_{gg_i} + C_{gb_{ov}} + C_{gs_{ov}} + C_{gd_{ov}} \\ C_{ss} & = C_{ss_i} + C_{gs_{ov}} + C_{bs} \\ C_{dd} & = C_{dd_i} + C_{gd_{ov}} + C_{bd} \end{cases} \tag{7.23}$$

In the following parts, each capacitance contribution is discussed in detail.

Estimation of the intrinsic capacitances

The intrinsic capacitances, especially the equivalent gate-gate, source-source, and drain-drain (C_{gg_i}, C_{ss_i}, C_{dd_i}) self capacitances, are the most important contributions in the MOS capacitance model. These self capacitances are shown in Fig. 7.21, constituting the equivalent capacitance value present on each MOS terminal. With these, it is possible to estimate the intrinsic capacitive load of each transistor that contributes to the final dynamic power and timing computations. However, estimating their values requires knowledge of the MOS state for each instant, so some approximations should be made. According to BSIM4 and Spectre manuals [150, 17], by setting `capMod = 0`, `cvchargemod=0` and 50-50 charge partitioning in the channel (`xpart=0.5`), the intrinsic charges equations, which are extensively explained in

[150], become the following:

$$\text{Subthreshold}\begin{cases} Q_g = \frac{C_o k_1^2}{2}\left(\sqrt{1-\frac{4VFBCV+4V_{bs}-4V_{gs}}{{k_1}^2}}-1\right) \\ Q_d = 0 \qquad Q_b = 0 \qquad Q_s = 0 \end{cases} \tag{7.24}$$

$$\text{Triode}\begin{cases} Q_g = -C_o\left(VFBCV+\frac{V_{ds}}{2}-V_{gs}+\phi_s\right)+ \\ -C_o \times \left(\frac{A_{bulk}{V_{ds}}^2}{12V_{th}-12V_{gs}+6A_{bulk}V_{ds}}\right) \\ Q_b = -C_o\left(V_{th}-VFBCV-\phi_s+\frac{V_{ds}\left(A_{bulk}-1\right)}{2}\right)+ \\ -C_o\left(\frac{A_{bulk}{V_{ds}}^2\left(A_{bulk}-1\right)}{12V_{th}-12V_{gs}+6A_{bulk}V_{ds}}\right) \\ Q_d = +0.5C_o\left(V_{th}-V_{gs}+\phi_s+\frac{A_{bulk}V_{ds}}{2}\right)+ \\ +0.5C_o\left(\frac{{A_{bulk}}^2{V_{ds}}^2}{12V_{th}-12V_{gs}+6A_{bulk}V_{ds}}\right) \\ Q_s = Q_d \end{cases} \tag{7.25}$$

$$\text{Saturation}\begin{cases} Q_g = -C_o\left(VFBCV-V_{gs}+\phi_s+\frac{V_{gs}-V_{th}}{3A_{bulk}}\right) \\ Q_b = -C_o\left(VFBCV-V_{th}+\phi_s-\frac{\left(V_{gs}-V_{th}\right)\left(A_{bulk}-1\right)}{3A_{bulk}}\right) \\ Q_d = -\frac{C_o\left(V_{gs}-V_{th}\right)}{3} \\ Q_s = Q_d \end{cases} \tag{7.26}$$

The term `cvchargemod`, if set to 0, disables the new V_{gsteff} calculation and corresponds to the long-channel charge model, with constant mobility and no velocity saturation. In the equations, k_1 is the first-order body bias coefficient; $VFBCV$ is the flat-band voltage for `capMod=0`, which is equal to -1 V; 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, which equation is shown in Equation 7.29, and C_o is the gate capacitance expressed as $C_{ox} \times W_{eff} \times L_{eff}$. The parameters needed in the A_{bulk} equation are derived from the BSIM4 description of the NMOS/PMOS devices, and they are reported in Table 7.3.

$$A_{bulk0} = \left\{ 1 + F_{doping} \left[\begin{array}{l} \dfrac{a_0 L_{eff}}{L_{eff} + 2\sqrt{X_j X_{dep}}} \times \\ \times \left(1 - a_{gs} V_{gst} \left(\dfrac{L_{eff}}{L_{eff} + 2\sqrt{X_j X_{dep}}} \right)^2 \right) + \\ + \dfrac{b_0}{W_{eff} + b_1} \end{array} \right] \right\} \frac{1}{1 + ketaV_{bs}} \tag{7.27}$$

$$F_{doping} = \frac{\sqrt{1 + LPEB/L_{eff}} k_1}{2\sqrt{\phi_s - V_{bs}}} + k_2 - k3B \times \frac{T_{oxe}}{W_{eff} + W_0} \phi_s \tag{7.28}$$

$$A_{bulk} = A_{bulk0} \left(1 + \left(\frac{CLC}{L_{eff}} \right)^{CLE} \right) \tag{7.29}$$

A clarification needs to be made: it can be seen that the intrinsic capacitance equation of Q_b given here does not correspond exactly to the equation in the BSIM4 manual [150], in fact the $-\phi_s$ term for the triode zone should be $+\phi_s$. This equation reported here is in fact related to the BSIM3.3 model [17] instead of BSIM4 [150]. Several simulations were conducted in Matlab, directly comparing the resulting capacitance equations with the SPICE measurements, and it was derived that the values most faithful to the measurements are those of BSIM3.3, as shown in Fig. 7.22.

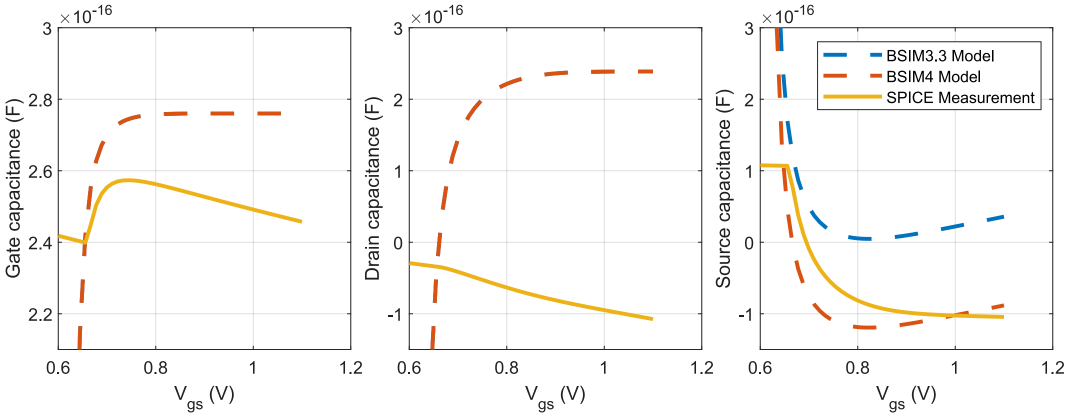


Fig. 7.22 BSIM3.3, BSIM4 capacitance models comparisons with SPICE measurements for the triode region.

Table 7.3 Parameters required for A_{bulk} computation. Values are extracted from the NMOS model file.

Parameter	**Meaning**	**Value**
a_0	*Non-uniform depletion width effect coefficient.*	1
a_{gs}	*Gate-bias dependence of Abulk.*	0
X_j	*Source/drain junction depth.*	19.8 nm
X_{dep}	*Depletion depth.*	$\sqrt{2\varepsilon_s \frac{\phi_s - V_{bs}}{qN_a}}$
$keta$	*Body-bias coefficient for non-uniform depletion width effect.*	0.04
b_0	*Bulk charge coefficient due to narrow width effect.*	0
b_1		0
$LPEB$	*Lateral non-uniform doping effect on K1.*	0
k_1	*First-order body-bias coefficient.*	0.4
k_2	*Second-order body bias coefficient.*	0
$k3B$	*Body effect coefficient of K3.*	0
T_{oxe}	*Effective oxide thickness.*	$T_{oxe} = T_{ox} = 1.14nm$
W_0	*Narrow width coefficient.*	2.5×10^{-6}
CLC	*CLC and CLE consider the effect of channel-length modulation.*	100 nm
CLE		600×10^{-3}

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, as reported in Equation 7.30.

$$C_{(g,s,d,b),g_i} = \frac{\partial Q_{(g,s,d,b)}}{\partial V_{gs}} \tag{7.30a}$$

$$C_{(g,s,d,b),d_i} = \frac{\partial Q_{(g,s,d,b)}}{\partial V_{ds}} - \frac{\partial Q_{(g,s,d,b)}}{\partial V_{gs}} \frac{\partial V_{th}}{\partial V_{ds}} \tag{7.30b}$$

$$C_{(g,s,d,b),b_i} = \frac{\partial Q_{(g,s,d,b)}}{\partial V_{bs}} - \frac{\partial Q_{(g,s,d,b)}}{\partial V_{gs}} \frac{\partial V_{th}}{\partial V_{bs}} \tag{7.30c}$$

Starting from equations 7.30, the following formulas are derived for C_{gg_i}, C_{ss_i} and C_{dd_i}, proposed in BSIM4 model [150, 17].

$$C_{gg_i} = \frac{\partial Q_g}{\partial V_{gs}} \tag{7.31}$$

$$C_{dd_i} = \frac{\partial Q_d}{\partial V_{ds}} - \frac{\partial Q_d}{\partial V_{gs}} \frac{\partial V_{th}}{\partial V_{ds}} \tag{7.32}$$

$$C_{ss_i} = -C_{sg} - C_{sd} - C_{sb} = -\frac{\partial Q_s}{\partial V_{gs}} - \left(\frac{\partial Q_s}{\partial V_{ds}} - \frac{\partial Q_s}{\partial V_{gs}} \frac{\partial V_{th}}{\partial V_{ds}} \right) - \left(\frac{\partial Q_b}{\partial V_{bs}} - \frac{\partial Q_b}{\partial V_{gs}} \frac{\partial V_{th}}{\partial V_{bs}} \right) \tag{7.33}$$

The equation of C_{ss_i} is derived with the following formula, with j=s:

$$\sum_j C_{jk} = \sum_k C_{jk} = 0 \tag{7.34}$$

DExIMA-Backend evaluates C_{gg_i}, C_{dd_i} and C_{ss_i} equations in each MOS region, considering $V_{gs} = 0V, V_{ds} = V_{dd}$ in subthreshold, $V_{gs} = V_{dd}, V_{ds} = 0V$ in triode and $V_{bs} = 0$ in all regions. In saturation, once the partial derivatives are computed, there are no dependencies on the V_{gs} or V_{ds}. The final resulting equations are the following:

$$\text{Subthreshold}\begin{cases} C_{gg_i} = \dfrac{C_o}{\sqrt{1-\dfrac{4VFBCV+4V_{bs}-4V_{gs}}{{k_1}^2}}} \\ C_{ss_i} = 0 \\ C_{dd_i} = 0 \end{cases} \tag{7.35}$$

$$\text{Triode}\begin{cases} C_{gg_i} = -C_o\left(\dfrac{12A_{bulk}{V_{ds}}^2}{(12V_{th}-12V_{gs}+6A_{bulk}V_{ds})^2}-1\right) \\ C_{dd_i} = \dfrac{C_o}{2}\left(\dfrac{A_{bulk}}{2}+\dfrac{2{A_{bulk}}^2V_{ds}}{12V_{th}-12V_{gs}+6A_{bulk}V_{ds}}\right)+ \\ \quad +\dfrac{C_o}{2}\left(-\dfrac{6{A_{bulk}}^3{V_{ds}}^2}{(12V_{th}-12V_{gs}+6A_{bulk}V_{ds})^2}\right) \\ C_{ss_i} = -\dfrac{C_o\left(V_{th}-V_{gs}+\phi_s+\dfrac{A_{bulk}V_{ds}}{2}+\dfrac{{A_{bulk}}^2{V_{ds}}^2}{12V_{th}-12V_{gs}+6A_{bulk}V_{ds}}\right)}{2}+ \\ \quad -\dfrac{C_o\left(\dfrac{12{A_{bulk}}^2{V_{ds}}^2}{(12V_{th}-12V_{gs}+6A_{bulk}V_{ds})^2}-1\right)}{2}+ \\ \quad -\dfrac{C_o\left(\dfrac{A_{bulk}}{2}+\dfrac{2{A_{bulk}}^2V_{ds}}{12V_{th}-12V_{gs}+6A_{bulk}V_{ds}}-\dfrac{6{A_{bulk}}^3{V_{ds}}^2}{(12V_{th}-12V_{gs}+6A_{bulk}V_{ds})^2}\right)}{2}+ \\ \quad -\dfrac{C_ok_1\left(\dfrac{12{A_{bulk}}^2{V_{ds}}^2}{(12V_{th}-12V_{gs}+6A_{bulk}V_{ds})^2}-1\right)}{4\sqrt{\phi_s-Vbs}} \end{cases} \tag{7.36}$$

$$\text{Saturation}\begin{cases} C_{gg_i} = -C_o\left(\dfrac{1}{3A_{bulk}}-1\right) \\ C_{dd_i} = 0 \\ C_{ss_i} = \dfrac{C_o}{3}+\dfrac{C_ok_1}{3\sqrt{\phi_s-V_{bs}}} \end{cases} \tag{7.37}$$

Validation of the intrinsic capacitances The intrinsic capacitances equations' validation step is performed using Cadence Virtuoso and Matlab tools together. In Virtuoso, an inverter gate is used as a reference model, in which input voltage is

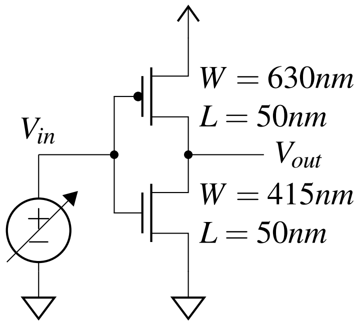


Fig. 7.23 Schematic of the inverter used in Cadence Virtuoso for intrinsic capacitance validation.

swept linearly between 0 and V_{dd} to evaluate each transistor phase. Matlab is used to model the equations derived for each intrinsic capacitance and to compare the results obtained with Virtuoso with the computed ones. By selecting the "Save DC Operating Point" option, Virtuoso outputs the results of each input voltage V_{in}, obtained with a DC simulation. These results includes the values of C_{ggb0}, C_{ddb0}, C_{bsb0}, C_{dsb0} and C_{gsb0}, corresponding to the intrinsic capacitances without the extrinsic contributions. As shown in Fig. 7.24, the computed values of the intrinsic capacitances differ from the measured ones in a range between -6.8% and 24.3%. These values represent the relative percentage error and are computed as:

$$\text{Relative error}\,(\%) = \frac{(C_{gg_i}, C_{dd_i}, C_{ss_i})_{\text{Model}} - (C_{gg_i}, C_{dd_i}, C_{ss_i})_{\text{Ref}}}{(C_{gg_i}, C_{dd_i}, C_{ss_i})_{\text{Ref}}} \times 100\% \qquad (7.38)$$

The error range is acceptable, considering that the adopted capacitance model represents a simplification of the real one.

Estimation of the extrinsic capacitances

The intrinsic capacitances alone are not sufficient to model the MOS correctly. As shown in Fig. 7.21, other contributions, called extrinsic capacitances, are present and given by the overlap and junction capacitances.

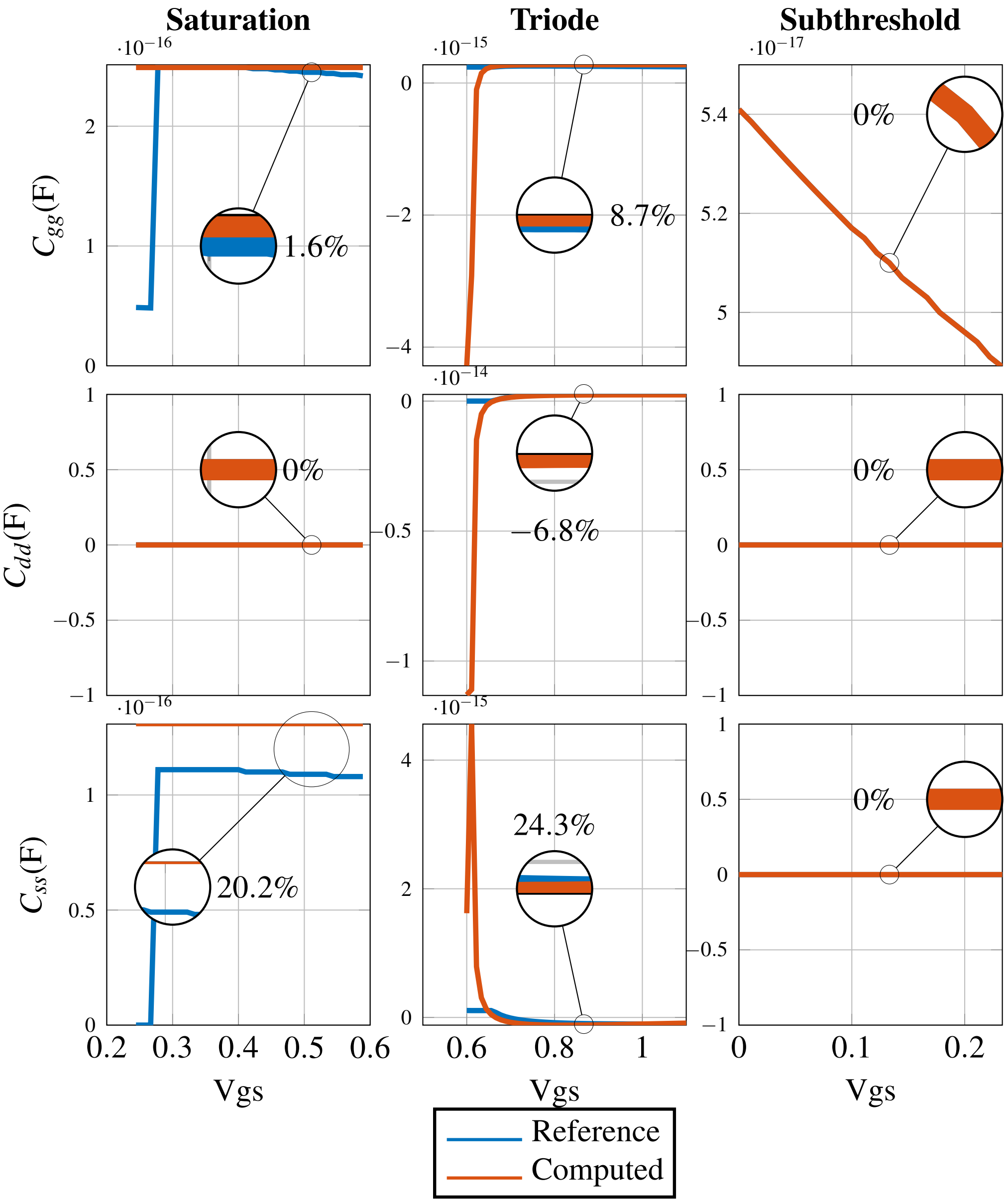


Fig. 7.24 Comparison between reference and computed C_{gg}, C_{ss} and C_{dd} intrinsic capacitances.

Overlap capacitances According to BSIM4 manual [150], for `capMod = 0` the overlap capacitances can be modeled in the following way:

$$\begin{cases} C_{gs,gd_{ov}} = C_{gd0,gs0}, & \text{when } C_{gd0,gs0} \text{ are given} \\ C_{gs,gd_{ov}} = (dlc + meto) \times C_{ox} - C_{gsl,gdl}, & \text{when } dlc \text{ is given and } dlc > \frac{C_{gsl,gdl}}{C_{ox}} \\ C_{gs,gd_{ov}} = 0.6 C_{ox} \times X_j, & \text{otherwise.} \end{cases} \tag{7.39}$$

$$\begin{cases} C_{gb_{ov}} = C_{gb0}, & \text{when } C_{gb0} \text{ is given} \\ C_{gb_{ov}} = 2 \times dwc \times C_{ox}, & \text{otherwise.} \end{cases} \tag{7.40}$$

The parameters, their meanings and values are reported in Table 7.4. Since the values of $C_{gs0,gd0,gb0}$ are defined in the model, the equations for the overlap capacitances become the following:

Table 7.4 Parameters used in the overlap capacitances calculations for the NMOS transistor.

Parameter	Meaning	Value
C_{gs0}	*Gate-source overlap capacitance per unit length.*	$1.1 \times 10^{-10} F/m$
C_{gd0}	*Gate-drain overlap capacitance per unit length.*	$1.1 \times 10^{-10} F/m$
C_{gb0}	*Gate-bulk overlap capacitance per unit length.*	$2.56 \times 10^{-11} F/m$
dlc	*Delta L for capacitance model.*	$3.75nm$
$meto$	*Metal overlap in fringing field.*	0
C_{ox}	*Oxide Capacitance per unit area.*	ε_{ox}/T_{ox}
C_{gsl}	*Overlap capacitance between gate and lightly-doped source region.*	$2.653 \times 10^{-10} F/m$
C_{gdl}	*Overlap capacitance between gate and lightly-doped drain region.*	$2.653 \times 10^{-10} F/m$
X_j	*Source/drain junction depth.*	$19.8nm$
dwc	*Delta W for capacitance model.*	$5nm$

$$C_{gs,gd_{ov}} = C_{gs0,gd0} \times W_{eff} \tag{7.41}$$

$$C_{gb_{ov}} = C_{gb0} \times L_{eff} \tag{7.42}$$

Moreover, the fringing effects must be considered in the overlap capacitance equations. If it is not explicitly written inside the technology file, the fringing capacitance is computed as:

$$CF = \frac{2\varepsilon_{ox}\varepsilon_0}{\pi} \times ln\left(1 + \frac{4 \times 10^{-7}\ \mathrm{m}}{T_{ox}}\right) \tag{7.43}$$

Finally, the equations for the overlap capacitances become:

$$C_{gs,gd_{ov}} = C_{gs0,gd0} \times W_{eff} + CF \times W_{eff} \tag{7.44}$$

$$C_{gb_{ov}} = C_{gb0} \times L_{eff} \tag{7.45}$$

These values are added in all MOS regions since, for `capMod = 0`, they are bias-independent.

Junction capacitances Similarly to the overlap capacitances, the junction capacitances equations are reported inside the BSIM4 manual [150] and rewritten here for reference.

$$C_{bs} = A_{seff}C_{jbs} + P_{seff}C_{jbssw} + W_{eff} \times NF \times C_{jbsswg} \tag{7.46}$$

$$C_{bd} = A_{deff}C_{jbd} + P_{deff}C_{jbdsw} + W_{eff} \times NF \times C_{jbdswg} \tag{7.47}$$

The terms $A_{seff,deff}$ are the source/drain effective area, and they are equal to AS,AD given in the technology file; $C_{jbs,jbd}$ are the zero bias source/drain bottom junction capacitance per unit area; $P_{seff,deff}$ are the source/drain effective perimeter, equal to PS,PD given in the technology file; $C_{jbssw,jbdsw}$ are the source/drain sidewall junction capacitance per unit periphery; NF is the number of fingers in the device, and lastly $C_{jbsswg,jbdswg}$ are the gate-side source/drain junction capacitance per unit width. In the MOS technology model, these parameters assume the following values: $AS = AD = 0$ and $PS = PD = 0$, meaning that the drain/source area and perimeter are not considered since these are defined in the layout view of cell; $C_{jbs,jbd} = 0.0005\,F/m^2$, $C_{jbssw,jbdsw} = 5 \times 10^{-10}\,F/m$, $NF = 1$, $C_{jbsswg} = 3 \times 10^{-10}\,F/m$ and $C_{jbdswg} = 5 \times 10^{-10}\,F/m$.

7.3.5 Switching activity propagation model

The switching activities play a fundamental role in the dynamic power estimation since they provide a clear indication on the circuit functionality. Considering the example of the And-Or-Invert 2-1 gate (AOI21) shown in Fig. 7.25, the switching activity propagation process works in the following way:

1. *Parsing of the input/output states.* As already discussed in subsection 7.2.3, inside the **.dex** file, there are waveforms for each input/output pin. These data are parsed by DExIMA-Backend and used inside each standard cell to derive the internal net values for each time instant.

2. *Evaluation of the states of the transistors.* The values of the inputs define the state of each transistor. Considering the example of the AOI21 cell in Fig. 7.25,

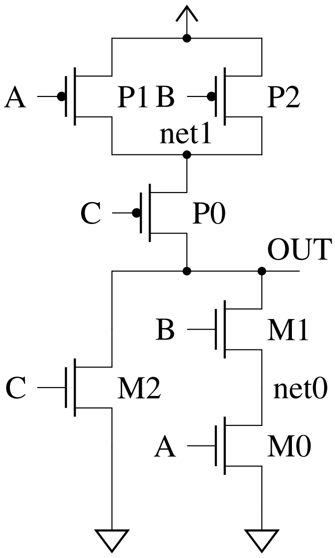


Fig. 7.25 Schematic of the And-Or-Invert 2-1 gate.

if the input combination is ABC = "010", the M1, P1, and P0 are in saturation, while the others are in cut-off.

3. *Internal nets values derivation.* For each input combination, the states of the transistors are determined. In the cut-off state, transistors act as an open switch while, in saturation, as a closed switch. So, if M1, P1, and P0 are in saturation, the values of net1 and net0 are equal to 1, as well as the OUT value.

4. *Memorize the values of the nets for each time instant.* The values of the nets are stored for each time instant since they will be fundamental for deriving the switching activity of the internal nodes.

5. *Evaluation of the switching activity of the internal nodes.* Once the internal node values' evaluation procedure finishes, each net's stored waveforms are scanned to get the associated switching activity. The pseudo-code for the switching activity computation is shown in Listing 43, in which the number of toggles is incremented if an actual commutation of the signal in the i+1 position with respect to the i position is present. Finally, the number of toggles is divided by the total combination length, obtaining the toggle rate of the signal, and the switching activity is evaluated by dividing the toggle rate by the clock period.

```
int toggles = 0;
std::string combination = "100101010";
unsigned int combinationLength = combination.size();
for(unsigned int i = 0; i < combinationLength-1; i++)
{
    if(combination[i+1] != combination[i]) toggles += 1;
}
float toggle_rate = toggles/combinationLength;
```

Listing 43 Pseudo-code for the toggle rate/switching activity computation.

Note that the internal nets evaluation process should start by evaluating the states of the transistors that are close to the supply pins. Otherwise, this procedure applied to more complex cells could not correctly identify the values of the nets.

7.3.6 Dynamic energy and power models

Once the switching activities and the capacitances of the internal nets are determined, the dynamic energy and power can be estimated. The formula to compute the dynamic energy contribution is reported in Equation 7.48, where E_{sc} is the short-circuit energy and TR_{net_i} is the toggle rate of the net_i.

$$E_{dyn} = E_{sc} + \frac{1}{2}V_{dd}^2 \sum_{\forall net_i} C_{net_i} \times TR_{net_i} \tag{7.48}$$

The dynamic power is obtained by multiplying the dynamic energy by the clock frequency.

7.3.7 Area model

The occupation of a single transistor is obtained by adding the channel length and the source/drain lengths and 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 technology's minimum channel length, which is written inside the technology parameters file. The technology file may also define the minimum width, or it can be obtained by multiplying the minimum channel length by the aspect ratio. However, in the netlists of the standard cells, the widths of the transistors can change due to the logical effort method. Therefore, in the calculations, DExIMA-Backend considers the ratio between the actual transistor width and the minimum width of the technology.

7.3.8 Delay model

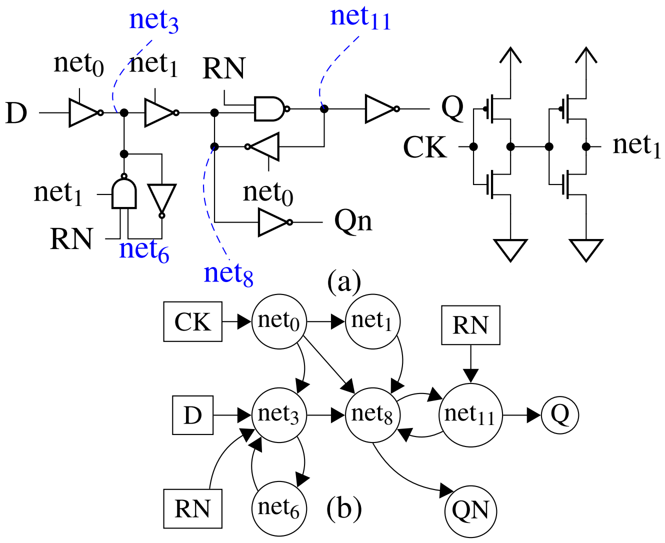


Fig. 7.26 (a) Schematic of a flip-flop with the equivalent logic gates. (b) Equivalent graph of the flip-flop.

An equivalent graph of the standard cell is created to estimate the delay. For example, a flip-flop cell represented in Fig. 7.26 is loaded by DExIMA-Backend, which interprets the schematic as an oriented graph, where the nodes represent the intermediate outputs (or logic gates), and the edges correspond to the connections between the gates. 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 7.49.

$$\text{Delay}_{\text{path}_i} = \sum_{\forall \text{net} \in \text{path}_i} \frac{V_{dd} \times C_{load_{net}}}{I_{on_{min}}} \tag{7.49}$$

The capacitances of each net are estimated following the same approach presented in subsection 7.3.4, while the $I_{on_{min}}$ is the minimum on current on the net, that is obtained considering the transistor having the minimum width connected to that net. Finally, the critical path search is performed by applying the Depth First Traversal method [151] on the equivalent graph, which is reported in Listing 44.

```cpp
void Graph::printAllPathsUtil(int u, int d, bool visited[], int path[], int& path_index)
{
    //mark the node as visited
    visited[u] = true;
    //add the node to the path
    path[path_index] = u;
    path_index ++;
    //if the initial node is equal to the destination
    if(u == d)
    {
        //save the path
        std::vector<int> tmpVector;
        for(int i = 0; i < path_index; i++) tmpVector.push_back(path[i]);
        paths.push_back(tmpVector);
    }
    else
    {
        //re-iterate the procedure to find all the paths.
        list<int>::iterator i;
        //adj contains the edges of the graph
        for(i = adj[u].begin(); i != adj[u].end(); i++)
            if(!visited[*i]) printAllPathsUtil(*i, d, visited, path, path_index);
    }
    path_index --;
    visited[u] = false;
}
```

Listing 44 C++ code implementing the Depth First Traversal algorithm to find all the delay paths. *The code was repurposed to DExIMA-Backend from the source [151].*

7.3.9 Bus model

DExIMA-Backend can also provide estimations of the bus involved in the LiM design. In particular, two contributions are evaluated: the external bus, i.e., the bus that delivers or receives data to/from the LiM, and the internal bus, i.e., the bus employed inside the memory connecting the LiM cells/IRL logic. For example, by selecting the bus "BL", DExIMA-Backend evaluates the impact of the bitlines considering, for the external contribution, the resistances/capacitances per unit length of the wires and for the internal contribution, both the resistances/capacitances per unit length and the capacitance loads of each cell/IRL, as shown in Fig. 7.27. From

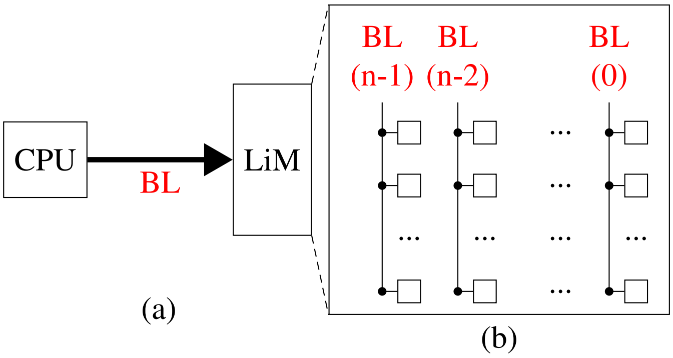


Fig. 7.27 Bus estimation. External (a) and internal (b) bus contributions.

these analyses, DExIMA-Backend directly provides the values of the 50%-50% delay and the average power consumption during the transitions. According to Fig. 7.28, the value of the average power during the transitions is calculated by considering the average value of the current leaving the power supply generator during the transitions and multiplied by the voltage Vdd.

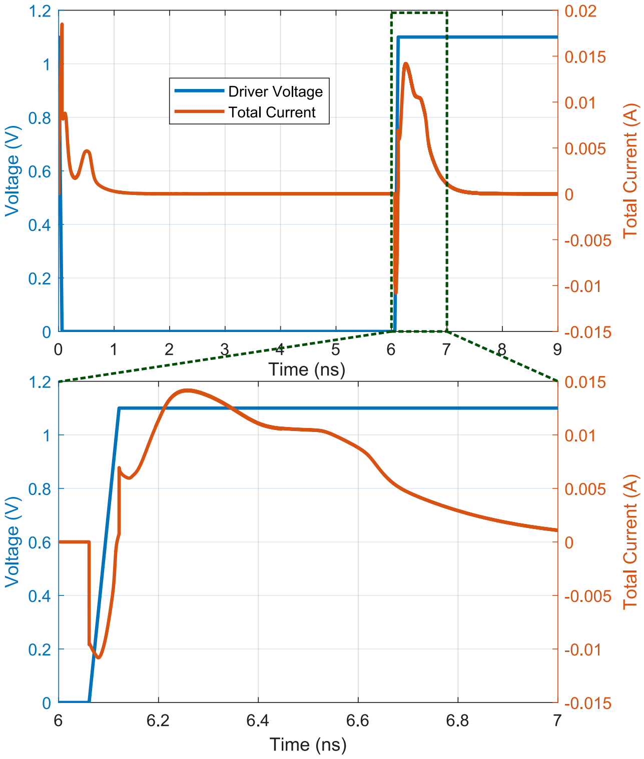


Fig. 7.28 Simulation window in which the average transition power of a bus is computed.

External bus model

The model generated by DExIMA is a distributed RC model, meaning that the resistances and capacitances per unit length are required for the calculation. The equivalent network of the external bus is shown in Fig. 7.29: the model, based on the work exposed in [152, 153, 141], also considers the cross-talk capacitances between two adjacent lines and the number of lines is associated to the bus parallelism. Regarding the parasitic inductances, they must be considered until the following condition on l_{wire} is verified:

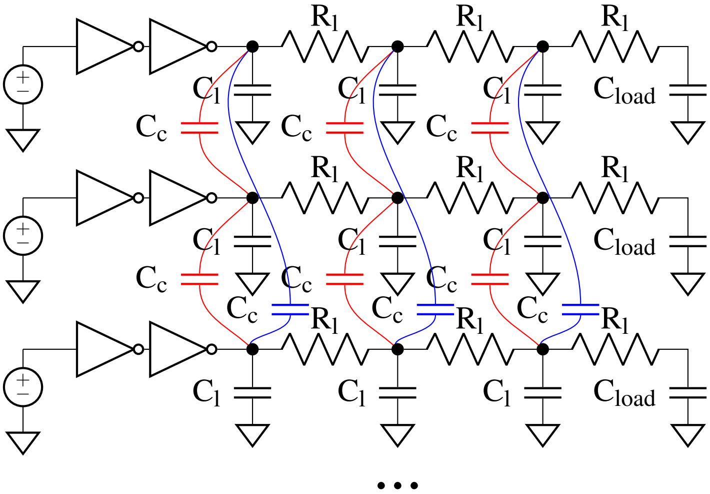


Fig. 7.29 Equivalent netlist of an external bus, used for the bus performance estimation. The number of lines is associated to the number of bits of the bus.

$$\frac{t_r}{2\sqrt{LC}} < l_{wire} < \frac{2}{R}\sqrt{\frac{L}{C}} \tag{7.50}$$

Where t_r is the rise time of the input signal and L, C, R are the inductance, capacitance, and resistance per unit length. To neglect the inductance, the condition on the t_r becomes as follows:

$$t_r > 4 \times \frac{L}{R} \tag{7.51}$$

The bus model implemented in DExIMA-Backend considers this condition true, so the inductance is always neglected in this phase. Moreover, to compute the capacitances and resistances per unit length, three parameters are required: resistance per unit square R_Q, wire underside capacitance C_Q and wire edge capacitance C_e.

The resistance and capacitance per unit length can be obtained as follows:

$$R_l = R_Q \frac{l_{wire}}{W} \tag{7.52}$$

$$C_l = l_{wire}(C_Q W + 2C_e) \tag{7.53}$$

While the cross-talk capacitances can be estimated as:

$$C_c = \varepsilon_0 \varepsilon_r \frac{T l_{wire}}{D} \tag{7.54}$$

Where l_{wire} is the length, W is the width, T is the thickness of the wire, and D is the distance between two wires. The bus is split into sections of length $\frac{l_{wire}}{N}$, where N is the number of sections, and at the end of each section, the cross-talk capacitances are connected to the other wires. In general, as discussed in [152], a number of sections equal to 3 is sufficient to reach a good trade-off between accuracy (almost 5% of error) and complexity. The netlist is simulated with Ngspice, and the results are directly written inside the **.dof** file for each input combination.

Internal bus model

Following a similar methodology as the external bus estimation, the internal bus impact is evaluated considering the resistive/capacitive contributions of the wire and the equivalent capacitances connected to the selected bus. Considering the Fig. 7.27 (b), the BL bus is connected to each cell of the LiM array: DExIMA-Backend iterates over the entire array, searching for each device connected to the BL bus and adding its contribution. The equivalent model for the internal bus (Fig. 7.30) is different from the external one (Fig. 7.29). The internal estimation does not consider the cross-talk capacitances since the bus lines inside the memory are distant. The number of sections (i.e., the number of RC nets) equals N, where the definition of N depends on the selected bus. In general, N is derived by DExIMA-Backend during the estimation procedure and is equal to the number of equivalent capacitances connected to that line. In the BL example, the number of sections will equal the number of rows inside the memory since each bit of the BL is connected to the LiM Cells in the same column.

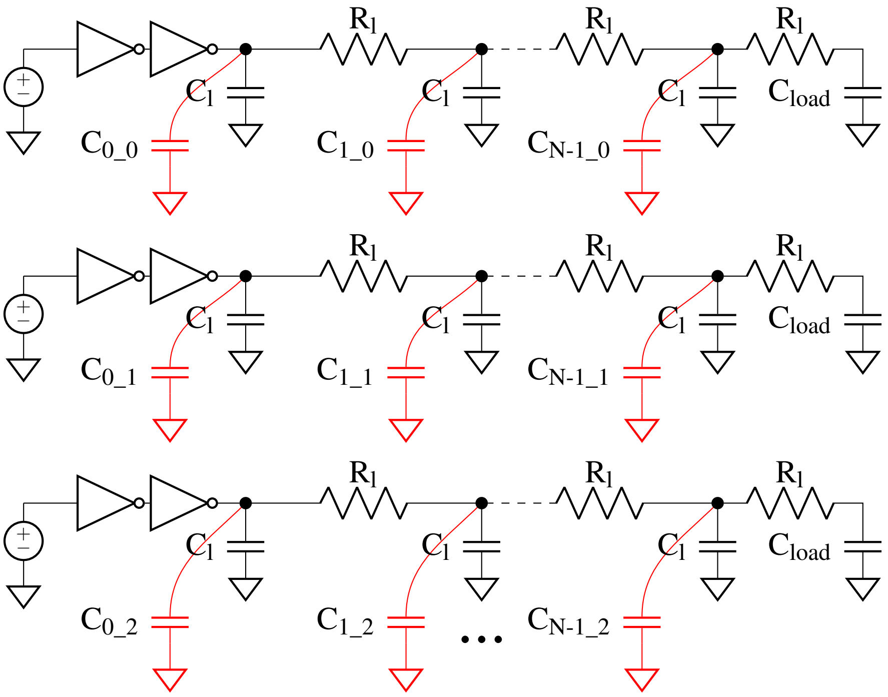


Fig. 7.30 LiM internal equivalent model for the bus estimation.

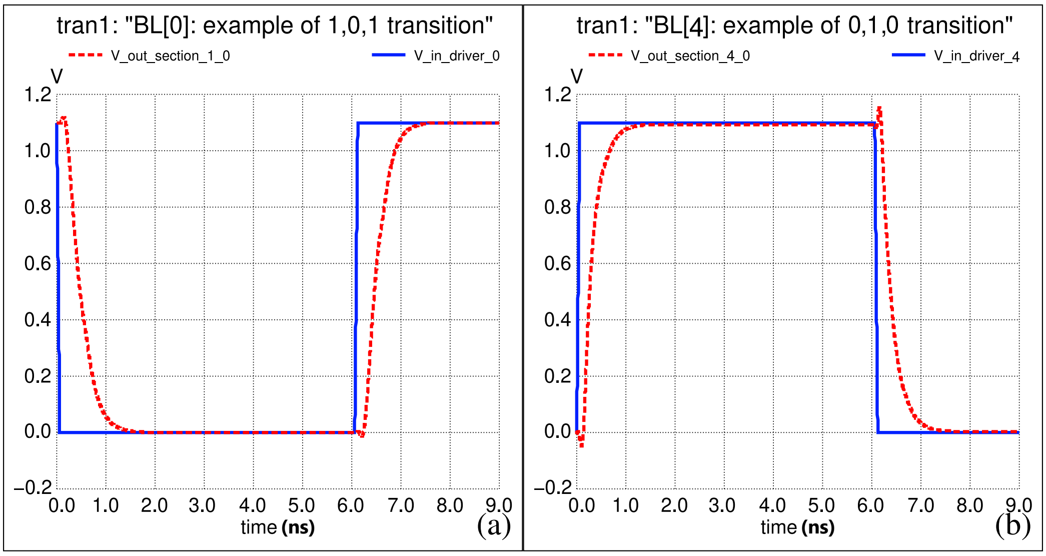


Fig. 7.31 Ngspice-based bus estimation waveforms. (a) Transition example for the bit-0 of the BL bus. (b) Transition example for the bit-1 of the BL bus.

The BusParser class

The `BusParser class` is in charge of parsing and simulating the bus lines. The collaboration graph of the `BusParser class` is shown in Fig. 7.32. Essentially, `BusParser` has a series of `Bus` objects that contains the characteristics of the selected

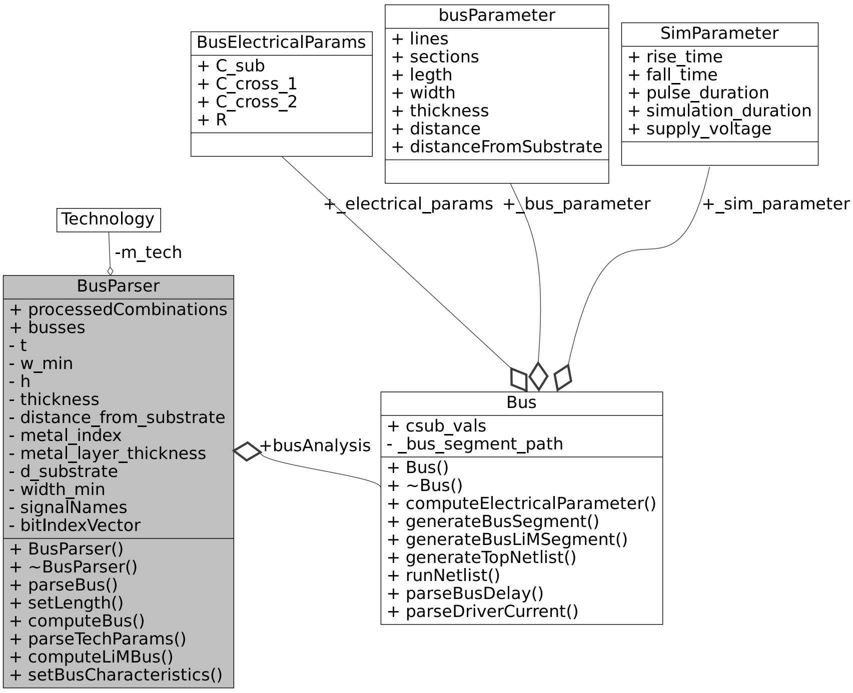


Fig. 7.32 Collaboration graph of the BusParser class.

bus(es), like the length, the width, the parallelism, etc. The main steps are discussed in the following parts.

Parsing the Bus data

To properly analyze the bus impact, inside the input **.dex** file are written all data required for the estimation. A snippet of bus-related code inside the **.dex** file is reported in Listing 45.

```
BL[0](2);100e-6;10e-5;1;{1111000000000000000}
        /LiMcell_0_0/Memory_9/NAND_1/IN1/
        /LiMcell_1_0/Memory_9/NAND_1/IN1/
```

Listing 45 Snippet of code containing bus data for the estimation.

Inside `BusParser`, the method `void BusParser::parseBus(const string &line)`:

1. *Looks for the character '[' in the input string.* If it finds one, it extracts everything before it as the signal's name and stores it in the variable. If it does

not find one, it means that the line parsed refers to the internal LiM hierarchy of the bus, so it is parsed and stored inside the data structure of the signal.

2. *Next, the code uses a regular expression to search for a pattern matching a number followed by a closing parenthesis ')' in the input string.* If it finds a match, it extracts the number as a string, converts it to an integer using `stoi`, and stores it in the variable parallelism.

3. *The code then uses another regular expression to search for a pattern matching a number between square brackets '[' and ']' in the input string.* If it finds a match, it extracts the number as a string, converts it to an integer using `stoi`, and stores it in the variable bit index.

4. *The code uses another regular expression to search for a pattern matching a string between curly braces '{' and '}' in the input string.* If it finds a match, it extracts the string and stores it as the waveform of the parsed signal.

5. *The code then checks if a signal with the same name as signalName has already been stored in the buses map.* If it has, it represents another bit of the already stored signal (e.g., BL1). The signal stored in the map is updated with the new waveform, passing the bit index and the wave as arguments. If it has not, the code extracts and saves the width, length, and metal layer number of the selected bus.

Simulation of the Bus

After the bus parameter parsing, the `BusParser class` implements methods called `void BusParser::computeBus()` and `void BusParser::computeLiMBus(map<string, busValuesStruct> &pinCapacitances)`, that are in charge of generating the output Ngspice-compatible netlists and running Ngspice to start the simulations: the first one, simulates the external bus, while the second one the internal. The codes of `void BusParser::computeBus()` and `void BusParser::computeLiMBus(map<string, busValuesStruct> &pinCapacitances)` begin a loop over the elements of the buses map and sets the bus characteristics parsed from the `void BusParser::parseBus(const string &line)` method. Each segment of the bus model is generated with functions `generateBusSegment(const string filePath)` and `generateBusLiMSegment(const string file

`Path, const string csubName, std::vector<float>& capacitances)` realizing a netlist containing a set of interconnected capacitors and resistors for the external and internal bus models, respectively. If the number of lines are greater than 1 and the analyzed bus is external, it writes a series of lines defining the cross-capacitance between pairs of input nodes with even indices. Moreover, if the number of lines exceeds 2, it writes a series of lines defining cross-capacitors between pairs of input nodes with odd indices. After the generation of the bus segments, the waves stored for each signal are iterated: each bit of the wave vector represents the value of the signal in a time instant defined by half the clock period. Within the loop performed for each half-clock step, a local variable called `combination` is initialized to the half-clock step-th element of the waves vector member of bus data structure: the code checks if the combination has been processed before, and, if not, the code does the following:

- *Calls the function* `generateTopNetlist`, *passing in three arguments: a string indicating a file path, the combination vector, and a vector containing the name of each bus segment file to be included inside the top-level SPICE netlist.* In the top netlist, the square voltage generators are instantiated, and their values are related to the actual combination of bits for the bus in the considered half-clock step. In Listing 46, an example of a top-level internal bus netlist for the BL is reported.

```
*Complete bus system netlist
*Global definitions
.GLOBAL Vdd
.INCLUDE ./InverterExample.sp
.INCLUDE ./bus_sectionBL0.sp
*...*
.INCLUDE ./bus_sectionBL31.sp
.param L_section = 2.42129e-06
.param sections = 256
.param R = 2.90555
.param C_sub = 1.23754e-26
.param C_cross_1 = 4.03823e-27
.param C_cross_2 = 1.34608e-27
.param supply_voltage = 1.1
.param pulse_duration = 6e-09
.param transition_time = 6e-11
```

```
.param simulation_duration = 9e-09
*supply generator
VCC Vdd 0 supply_voltage
Vd0 Vdd Vdd_dummy 0
V0 V_in_driver_0 0 PULSE (1.1 0 1E-12 6e-11 6e-11 6e-09 9e-09) DC 1.1
*...*
V15 V_in_driver_15 0 PULSE (1.1 0 1E-12 6e-11 6e-11 6e-09 9e-09) DC 1.1
*Input driver
X0 V_in_driver_0 V_out_inverter_0 Vdd_dummy 0 Inverter
X1 V_out_inverter_0 V_out_section_0_0 Vdd_dummy 0 Inverter
*...*
X30 V_in_driver_15 V_out_inverter_15 Vdd_dummy 0 Inverter
X31 V_out_inverter_15 V_out_section_0_15 Vdd_dummy 0 Inverter
*Bus sections
X33 V_out_section_0_0 V_out_section_1_0 BusSectionBL0
*...*
X48 V_out_section_0_15 V_out_section_1_15 BusSectionBL9
*Final capacitors
C0 V_out_section_1_0 0 C_sub
*...*
C15 V_out_section_1_15 0 C_sub
X49 V_out_section_1_0 V_out_load_0 Vdd_dummy 0 Inverter
*...*
X64 V_out_section_1_15 V_out_load_15 Vdd_dummy 0 Inverter
*analysis definition
.tran 1e-12 9e-09
.measure tran bus_delay_0 trig V_in_driver_0 val=0.55 FALL=1
+TARG V_out_section_1_0 val=0.55 FALL=1
*...*
.measure tran bus_delay_15 trig V_in_driver_15 val=0.55 FALL=1
+TARG V_out_section_1_15 val=0.55 FALL=1
.measure tran avg_i avg i(Vd0) from=6e-09 to=7e-09
.measure tran avg_power param='avg_i * supply_voltage'
.control
run
quit
.endc
.end

```

Listing 46 Example of an internal bus top-level spice netlist for the BL bus.

- *Calls the function* `runNetlist`*, passing in two string arguments indicating file paths: the first one is the file to simulate, and the second is the output file in which the simulation results will be written.* Inside this method, Ngspice is called:

```
busError Bus::runNetlist(const string filePath, const string
↪   outFilePath) {
        int state = 0;
    string cmd = "ngspice -b -o " + outFilePath + " " + filePath + "
↪   > " + "../../OUTPUT/Bus/.log.txt";
    system(cmd.c_str());
        return Error;
}
```

Listing 47 Command specified to Ngspice tool to run the generated netlist.

The **-b** flag stands for "batch mode". When this flag is used, Ngspice reads a list of commands from a file (or from standard input if no file is specified) and executes them without entering interactive mode. The **-o** flag stands for "output": Ngspice will write the simulation output to the specified file.

- *Adds an entry to the processed combinations map with the results of the simulated combination.*

An important distinction must be made between the simulations of the external and internal bus(es). These contributions are evaluated in two different instants: the first during the input **.dex** file parsing, while the second in the performance estimation phase, because it is necessary to compute all parameters of the modules located inside the LiM design before estimating the impact of the bus, especially the internal capacitances.

7.3.10 DExIMA output file

In Listing 48 is given an example of a DExIMA-Backend output file, which has the extension **.dof**.

```
##BUS RESULTS##
BL>>1000000111010011: Pavg = 0.00259578 (W); MaxDelay = 1.80485e-10 (s)
...
BL_LiM>>1000000111010011: Pavg = 0.00540404 (W); MaxDelay = 4.13644e-10 (s)

Simulation results

Clock period: 6 ns
Frequency: 166.667 MHz
Critical Path Instruction: timing_paths
Critical Path name: path[100]
Critical Path: 817.881 ps
Area: 386483 um^2
Dissipated dynamic energy: 580.816 pJ
Dissipated static energy: 274.489 pJ
Total dissipated energy: 855.305 pJ
Static power: 45.7482 mW
Execution time: 126 ns
Average dynamic power: 96.8026 mW
Total power: 142.551 mW
Total clock steps: 21

Instructions

Instruction: algorithm
Dissipated energy: 580.816 pJ
Static Power: 45.7483 mW
Area: 386484 um^2
Critical path: 0 fs
Critical path name:
Path delays
path[0] -> 0 fs

Instruction: timing_paths
Dissipated energy: 580.816 pJ
Static Power: 45.7483 mW
Area: 386484 um^2
Critical path: 817.881 ps
Critical path name: path[100]
Path delays
path[0] -> 514.59 ps
```

```
path[100] -> 817.881 ps
...
path[9] -> 514.59 ps

Technology internal parameters

Technology file: HP_45.txt
Interconnection overhead: 10%
Standard Cell overhead: 0%
Stack factor: 0
Vdd: 1.1 V
Aspect ratio: 3.09278
Cox: 30290 uF/m^2
Leff: 22.5 nm
Beta: 1.554
Diffusion lenght: 0 nm
C bottom n: 0 pF/m
C bottom p: 0 pF/m
C sidewall n: 432.674 pF/m
C sidewall p: 432.674 pF/m
C interconnections: 183.13 pF/m
Unitary Mos width: 0.09 um
Cin n mos: 0.0859322 fF
Gamma: 1.554
Rho: 1
Ion: 1340 uA/um
Ioff: 118.44 nA/um
Igate: 12.84 nA/um
Ion unitary mos: 120.6 uA
Ioff unitary mos: 10.6596 nA
Igate unitary mos: 1.1556 nA
```

Listing 48 Example of a DExIMA-Backend output file (**.dof**).

As it is possible to see, it is divided into sections:

1. *In the first part (labeled `##BUS RESULTS##`), the bus performance results are reported, both external/internal from/to the LiM.* For each combination, the delay and average power are given.
2. *The next part reports the performance results of the LiM architecture, with the most important data such as energy, power, area, critical path, etc.*

3. *After that, there is the section devoted to the declared instructions, then algorithm and timing_paths, needed for power estimation and timing, respectively.*

4. *The last section, on the other hand, summarizes the technology parameters used.*

7.4 Conclusions

In this chapter, the structure of DExIMA-Backend is analyzed in detail. Unlike the front-end, this discussion is carried out using the UML, as it provides a description of the classes and the relationships between them, resulting in being an extremely useful tool for explaining how the code works. In addition to the structure, the computational models for estimating the performance of the CMOS cells and the bus are also given, and an output file example containing the performance values provided by the tool is provided.

Chapter 8

Inserting LiM in a von Neumann system

Summary

In this short chapter, the procedure for evaluating the impact of LiM architecture in a classical CPU-Memory context is reported. The idea is to compare the performance of two systems, where the LiM accelerator is considered in one of them. Two C codes are simulated, one for the CPU-Memory architecture, where the CPU executes the algorithm, and the other for the CPU-Memory-LiM architecture, where the data transfer between main memory to LiM is emulated. Lastly, the LiM is responsible for the actual computation of the algorithm. Comparisons are proposed in terms of some figures of merit, such as energy and execution time, following the state-of-the-art models.

8.1 Comparison between CPU-Mem and CPU-Mem-LiM systems

Following the design of the LiM architecture, DExIMA software offers an environment for comparing two systems, CPU-Memory (CPU-Mem) and CPU-Mem-LiM. The first is a conventional von Neumann architecture consisting of a RISC-V CPU and a two-level cache system, while the second is a Beyond von Neumann design functionally comparable to the CPU-Mem but with a LiM co-processor. The emulated system is described within a Python file in the Gem5 root directory, located in `gem5/configs/learning_gem5/part1/two_level.py`, which is reported down below for reference purposes.

```python
# import the m5 (gem5) library created when gem5 is built
import m5
# import all of the SimObjects
from m5.objects import *

# Add the common scripts to our path
m5.util.addToPath('../../')

# import the caches which we made
from caches import *

# import the SimpleOpts module
from common import SimpleOpts

# get ISA for the default binary to run. This is mostly for simple testing
isa = str(m5.defines.buildEnv['TARGET_ISA']).lower()

# Default to running 'hello', use the compiled ISA to find the binary
# grab the specific path to the binary
thispath = os.path.dirname(os.path.realpath(__file__))
default_binary = os.path.join(thispath, '../../../',
    'tests/test-progs/hello/', isa, 'linux/program')

# Binary to execute
SimpleOpts.add_option("binary", nargs='?', default=default_binary)

# Finalize the arguments and grab the args so we can pass it on to our objects
args = SimpleOpts.parse_args()

# create the system we are going to simulate
system = System()

# Set the clock fequency of the system (and all of its children)
system.clk_domain = SrcClockDomain()
```

```
    system.clk_domain.clock = '1GHz'
    system.clk_domain.voltage_domain = VoltageDomain()

    # Set up the system
    system.mem_mode = 'timing'               # Use timing accesses
    system.mem_ranges = [AddrRange('512MB')] # Create an address range

    # Create a simple CPU
    system.cpu = TimingSimpleCPU()

    # Create an L1 instruction and data cache
    system.cpu.icache = L1ICache(args)
    system.cpu.dcache = L1DCache(args)

    # Connect the instruction and data caches to the CPU
    system.cpu.icache.connectCPU(system.cpu)
    system.cpu.dcache.connectCPU(system.cpu)

    # Create a memory bus, a coherent crossbar, in this case
    system.l2bus = L2XBar()

    # Hook the CPU ports up to the l2bus
    system.cpu.icache.connectBus(system.l2bus)
    system.cpu.dcache.connectBus(system.l2bus)

    # Create an L2 cache and connect it to the l2bus
    system.l2cache = L2Cache(args)
    system.l2cache.connectCPUSideBus(system.l2bus)

    # Create a memory bus
    system.membus = SystemXBar()

    # Connect the L2 cache to the membus
    system.l2cache.connectMemSideBus(system.membus)

    # create the interrupt controller for the CPU
    system.cpu.createInterruptController()

    # For x86 only, make sure the interrupts are connected to the memory
    # Note: these are directly connected to the memory bus and are not
    ↪  cached
    if m5.defines.buildEnv['TARGET_ISA'] == "x86":
```

```
    system.cpu.interrupts[0].pio = system.membus.mem_side_ports
    system.cpu.interrupts[0].int_requestor =
     ↪  system.membus.cpu_side_ports
    system.cpu.interrupts[0].int_responder =
     ↪  system.membus.mem_side_ports

# Connect the system up to the membus
system.system_port = system.membus.cpu_side_ports

# Create a DDR3 memory controller
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports

system.workload = SEWorkload.init_compatible(args.binary)

# Create a process for a simple "Hello World" application
process = Process()
# Set the command
# cmd is a list which begins with the executable (like argv)
process.cmd = [args.binary]
# Set the cpu to use the process as its workload and create thread
 ↪  contexts
system.cpu.workload = process
system.cpu.createThreads()

# set up the root SimObject and start the simulation
root = Root(full_system = False, system = system)
# instantiate all of the objects we've created above
m5.instantiate()

print("Beginning simulation!")
exit_event = m5.simulate()
print('Exiting @ tick %i because %s' % (m5.curTick(),
 ↪  exit_event.getCause()))
```

Listing 49 Implementation of the RISC-V-based two-level caches system in Gem5 [97].

This environment is offered in the "Comparison CPU-Memory" tab of Fig. 3.5 (b): it consists of a text editor, in which the user can create a C algorithm that will be automatically built and executed by Gem5 [97] software. The procedures used to examine the LiM effect on a conventional von Neumann architecture are similar to those taken in [127]:

1. *Definition of the CPU-Mem and CPU-Mem-LiM C codes.* In this phase, the user writes the algorithm that will be executed by the core of the classical CPU-Mem architecture. At the same time, DExIMA automatically creates the CPU-Mem-LiM algorithm that simply emulates the data movement from the main memories in the LiM, where the core of the computation is located. The code automatically generated by DExIMA is a simple for loop that accesses a number of memory locations equal to the memory size of the LiM accelerator. An example is given below:

```c
#include <stdio.h>
int main(){
        volatile int memory_content[1024] = {0};
        volatile int data;
        for(int i = 0; i < 1024; i++){
                data = memory_content[i];
        }
        return 0;
}

```

Listing 50 Example of a LiM code.

In this example, the main memory access of 1024 data items to be passed later to the LiM is emulated.

2. *Compile the C codes.* By clicking on "Gem5" button in Fig. 3.5 (a), the C codes for the CPU-Mem and CPU-Mem-LiM architectures are automatically compiled with riscv-gnu-toolchain [125] and executed with Gem5, that provides performance data such as CPU execution time, the total number of memory accesses, etc. To compile the C codes of the two CPU-Mem and CPU-Mem-LiM programs, the script contained in the SIMCnfg folder called `scriptGem5.sh` is used, which is explained in details in section 5.8. Data are then moved inside the appropriate working folder.

3. *Estimate the caches' performance.* At this point, the caches are characterized by means of Cacti [91]. By clicking on "Cacti by HP" in Fig. 3.5 (b) and

providing a few input arguments such as the cache size, the associativity, and the technology node, DExIMA executes Cacti, which outputs useful parameters such as the access time and the read/write energy per access.

4. *Estimate CPU-Mem and CPU-Mem-LiM performance.* Data from Gem5 simulation and Cacti evaluation are used to estimate the overall performance of the algorithms.

At this point, the user has several options. By clicking on "Plot instructions of the CORE" in Fig. 3.5, algorithms are profiled in terms of executed instructions, and the most recurrent instructions are reported: an example is shown in Fig. 10.9 (b-c). "Extrapolate caches info" reports useful cache parameters such as the miss rate, the overall accesses per cache type, the total number of hits, etc. Lastly, the "Compare CPU-Mem and CPU-Mem-LiM" option evaluates both systems by choosing a subset of figures of merit, such as the total memory accesses, the total caches access energy, the CPU execution time, LiM execution time, and LiM energy consumption. These last two values come from DExIMA-Backend.

8.2 Conclusions

This short chapter presents the methodology for comparing CPU-Mem and CPU-Mem-LiM solutions. To do this, it is necessary to use the Gem5 tool, which models a RISC-V architecture with two cache levels, reported in Listing 49. Examples that implement the procedures described in this chapter in a timely manner, starting with the definition of CPU-Mem and CPU-Mem-LiM code and ending with the estimation of instructions, cache parameters, and system energy, are provided in the chapter 10.

Chapter 9

DExIMA-Backend validation

Summary

Standard industrial software like Synopsys Design Compiler (DC) or similar can provide advanced tools and algorithms that precisely synthesize and estimate CMOS-based circuits. However, an extensive and complete description of the technology is compulsory to perform such complex routines, starting from the N and P transistors and moving toward the standard cell structure, layout, and characterization. The description of the transistors is usually reported in SPICE files, including a vast list of parameters that depend on the model (MOS levels, BSIM, etc.) [17] and on the technology node [26]. When characterized, the standard cells are automatically simulated and modeled, and the results are grouped in 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 the synthesizers and Place&Route tools to provide precise indications of the circuit performance. Even if companies or universities typically already provide all the files required to perform the synthesis and Place&Route, this process is long and time-consuming: a small modification of the starting transistor model requires the complete re-characterization of the entire library. DExIMA-Backend is introduced to lighten this drawback. In fact, it aims to provide relatively fast and approximated performance estimations of the architectures, with the possibility of integrating other technologies than standard CMOS in the future.

In this chapter, the idea is to validate the results of DExIMA-Backend by comparing them with those provided by standard industrial tools such as Cadence Liberate and Synopsys Design Compiler, thus providing an indication of how accurate DExIMA-Backend actually is in estimating the performance parameters.

9.1 Procedural steps for validation

Apart from the clear goal of reducing the efforts in the digital design flow, DExIMA-Backend must be able to provide a rough but clear indication of the system performance, so it has to be validated with respect to the standard EDA tools. The validation procedure of DExIMA-Backend consists of the following steps:

- *Choose the reference library.* As already stated before, the technological library used for the calculations in DExIMA-Backend is the FreePDK 45nm from North Carolina State University (NCSU) [149].

- *Simplification of the technology model.* As already said, the technology files are simplified imposing `capMod = 0`, `cvchargemod = 0` and `xpart = 0.5` to match the calculations used in DExIMA-Backend.

- *Characterization of the simplified library.* The FreePDK 45nm library already contains the SPICE netlists of each standard cell, which can be used by the characterization tool to characterize a new library with the modifications explained before. For this purpose, Cadence Liberate is used, a tool able to automatically generate the Liberty file description and a detailed datasheet of the library.

- *Stardard cells estimations: comparisons with Liberate datasheet.* In the datasheet generated by Liberate, there are important parameters like the internal dynamic energy of the cells, the timing, the leakage power, the capacitance of each pin, and the truth table. These results are useful to validate DExIMA-Backend in estimating the performance of the standard cells.

- *Complex designs: comparisons with Synopsys Design Compiler.* The datasheet alone is insufficient to estimate the accuracy of DExIMA-Backend in more complex designs. Synopsys DC covers this part, which can be used to have an indication of what happens to the DExIMA-Backend accuracy for designs involving a higher number of standard cells. The Liberty file generated by Liberate is compiled with Synopsys Library Compiler and converted into 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, estimated with the Backend, and the VHDL code generated is directly synthesized with DC, so the performance results of the two tools are directly compared.

9.1.1 Comparisons with Liberate datasheet

Cadence Liberate is a tool capable of characterizing the standard cells described in SPICE or Spectre languages and writing their performance results into a Liberty output file. A typical characterization flow is shown in Fig. 9.1. The steps start from the transistor-level view of the cells, designed in SPICE tools like Cadence Virtuoso, moving towards the layout view of the cell, the characterization into a Liberty file, and the generation of the abstract view of the layout, which can be used by Place&Route tools. While the layout view represents all the devices, interconnections, P/N wells, metal layers, contacts, etc., the abstract view contains only the essential elements needed by the Place&Route to correctly map and connect the standard cell, so only the metal layers and the contacts. In this instance, only a portion of this chain is used, taking the standard cells described in the FreePDK45nm and characterizing them with Liberate. It is important to underline that the considered netlists are taken from the schematic view and not the layout view, to avoid the inclusion of the parasitic elements of the layout. This is fundamental to have fairer comparisons with DExIMA-Backend since it does not include parasitics in the estimations. Liberate accepts in input a **.tcl** file, containing indications on the characterization procedure, such as the working temperature, the supply voltage value, the Look-Up Table templates for delay and power, the files to characterize, and so on. In Listing 51, the Liberate script is reported.

```
# Liberate Tcl File
set_var bus_syntax "<>"
set rundir $env(PWD)
#Allow short-circuit states for leakage. As a result, the short-circuit nodes are not initialized in the deck.
```

```
set_var mega_short_circuit_mode 2
#Resets the negative power
set_var reset_negative_power 3
set_var reset_negative_power_info 2
#Disables the search bound estimation
set_var constraint_search_bound_estimation_mode 3
#Set the maximum number of hidden vectors for power estimation
set_var max_hidden_vector 512
set_var conditional_expression_max_whens 1024
# Create the directories Liberate will write to.
exec mkdir -p ${rundir}/LDB
exec mkdir -p ${rundir}/LIBRARY
exec mkdir -p ${rundir}/DATASHEET
### Define temperature and default voltage ###
set_operating_condition -voltage 1.1 -temp 27
## Load template information for each cell ##
source ${rundir}/TEMPLATE/template_rechar.tcl
## Load Spice models and subckts ##
set spicefiles $rundir/MODELS/NMOS_VTL.scs
lappend spicefiles $rundir/MODELS/PMOS_VTL.scs
foreach cell $cells {
    lappend spicefiles ${rundir}/NETLIST/${cell}.spi
        set cell_name ${cell}.spi
}
read_spice $spicefiles
## Characterize the library for NLDM (default), CCS and ECSM timing.
char_library -auto_index -auto_max_capacitance -ccs -ecsm -cells ${cells}
## Save characterization database for post-processing ##
write_ldb ${rundir}/LDB/Nangate45_NOPEX.ldb
## Generate a .lib with ccs, ecsm ###
write_library -overwrite -ccs -bus_syntax "\[\]" ${rundir}/LIBRARY/Nangate45_NOPEX_ccs.lib
write_library -overwrite -ecsm -bus_syntax "\[\]" ${rundir}/LIBRARY/Nangate45_NOPEX_ecsm.lib
write_verilog ${rundir}/LIBRARY/Nangate45_NOPEX.v
## Generate ascii datatsheet ###
write_datasheet -format text ${rundir}/DATASHEET/Nangate45_NOPEX
```

Listing 51 Liberate tcl main script for library characterization.

First, constraints about short circuits, negative powers, maximum hidden vectors, and conditional expressions are applied to avoid cases with negative powers or limitations on the maximum number of test cases for more complex gates. Next,

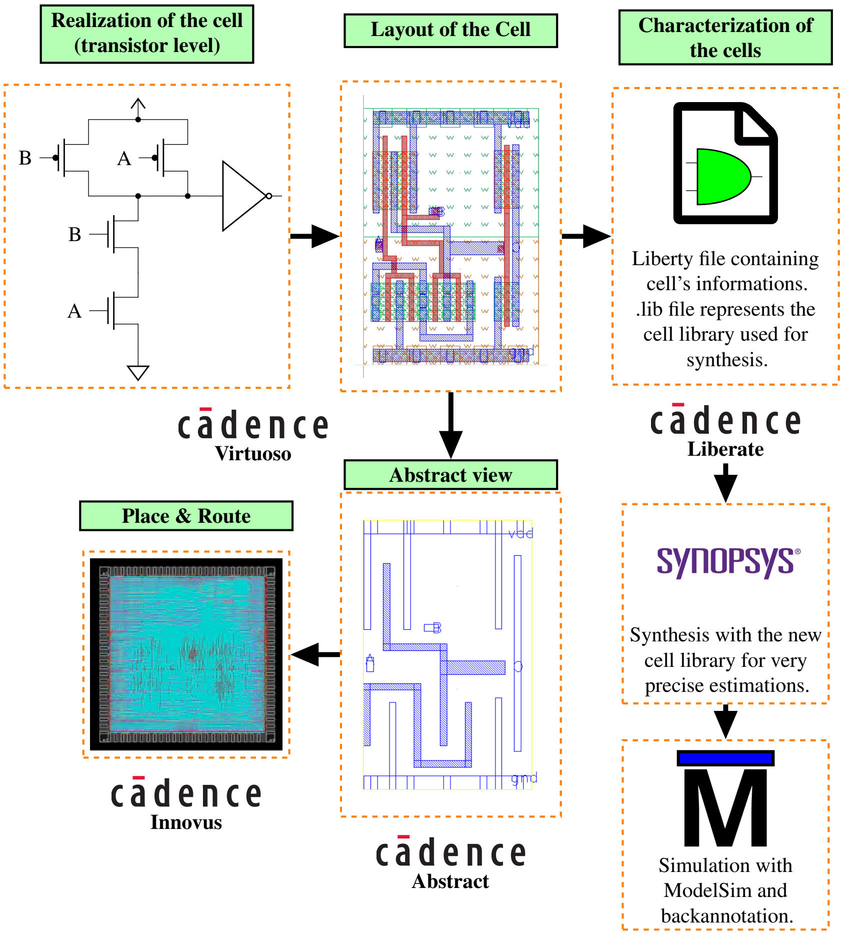

Realization of the cell (transistor level)
Layout of the Cell
Characterization of the cells
B
A
B
A
Liberty file containing cell's informations. .lib file represents the cell library used for synthesis.
cadence
Virtuoso
cadence
Liberate
Abstract view
Place & Route
SYNOPSYS
Synthesis with the new cell library for very precise estimations.
cadence
Innovus
cadence
Abstract
M
Simulation with ModelSim and backannotation.

Fig. 9.1 Example of a typical characterization steps for a standard cell library.

the operating conditions are set, with a supply voltage of 1.1V and a temperature of 27 °C. Then the information about each cell is loaded by sourcing another script file called `template_rechar.tcl`. This file is very important: it is obtained by exporting a template of data from an existing Liberty file. This operation can be accomplished with Liberate by reading the initial library using the `read_library` command and creating the template with `write_template`.

```
read_library NangateOpenCellLibrary_typical_ecsm.lib
write_template -verbose TEMPLATE/template_rechar.tcl
```

Listing 52 Liberate tcl script data template generation.

The file `template_rechar.tcl` contains the Look-Up Tables templates for power and delay and other constraints of each standard cell, as reported in Listing 53:

```
#Thresholds for delay measurements: 30% and 70% of the signal
set_var slew_lower_rise 0.3
set_var slew_lower_fall 0.3
set_var slew_upper_rise 0.7
set_var slew_upper_fall 0.7
set_var measure_slew_lower_rise 0.3
set_var measure_slew_lower_fall 0.3
set_var measure_slew_upper_rise 0.7
set_var measure_slew_upper_fall 0.7
#set the mix-max transition times and minimum output capacitance
set_var max_transition 1.98535e-10
set_var min_transition 1.17378e-12
set_var min_output_cap 3.65616e-16
#set the cells to characterize
set cells { \
  AND2_X1 \
  AND2_X2 \
  ...
}
#definition of the delay and power templates with input net transition and output capacitance.
define_template -type delay \
         -index_1 {0.00117378 0.00472397 0.0171859 0.0409838 0.0780596 0.130081 0.198535 } \
         -index_2 {0.000365616 0.00189304 0.00378609 0.00757217 0.0151443 0.0302887 0.0605774 } \
         delay_template_7x7

define_template -type power \
         -index_1 {0.00117378 0.00472397 0.0171859 0.0409838 0.0780596 0.130081 0.198535 } \
         -index_2 {0.000365616 0.00189304 0.00378609 0.00757217 0.0151443 0.0302887 0.0605774 } \
         power_template_7x7
#Definition of the inputs/outputs, templates, leakage and timing arcs of the cell.
if {[ALAPI_active_cell "AND2_X1"]} {
define_cell \
       -input { A1 A2 } \
       -output { ZN } \
       -pinlist { A1 A2 ZN } \
       -delay delay_template_7x7 \
       -power power_template_7x7 \
       AND2_X1

define_leakage -when "!A1 & !A2" AND2_X1
define_leakage -when "!A1 & A2" AND2_X1
define_leakage -when "A1 & !A2" AND2_X1
define_leakage -when "A1 & A2" AND2_X1

# delay arcs from A1 => ZN positive_unate combinational
define_arc \
       -vector {RxR} \
       -related_pin A1 \
       -pin ZN \
       AND2_X1
...
```

Listing 53 Liberate template_rechar.tcl script containing the standard cell templates.

After the template parsing, Liberate returns to the main script (Listing 51), reads the NMOS/PMOS technology model files and the standard cell netlists, and starts the characterization of the library by means of `char_library` command. It is important to underline that the NMOS/PMOS technology files are the ones that were simplified by setting `capMod = 0`, `xpart = 0.5`, and `cvchargemod = 0` options. At the end of the characterization process, Liberate outputs several files such as the **.ldb** database of the library that can be used in post-processing; two Liberty libraries, with Concurrent Current Source (CCS) and Non-Linear Delay Model (NLDM) descriptions; a Verilog file containing the behavioral description of the standard cells that can be used for post-synthesis simulations and, lastly, a datasheet file containing the human-readable characterization and performance results of each standard cell. From this datasheet, some cells are considered and directly compared with DExIMA-Backend, as shown in Table 9.1. In the datasheet, for each standard cell, the energy and timing are evaluated for each input, output transitions (input rising, input falling, output rising, and output falling), and different capacitance loads, differentiating between minimum value (corresponding to the minimum load and indicated as Min in Table 9.1), average (Avg) and maximum (Max). Choosing one combination as a reference, in Table 9.1, the minimum value for the dynamic energy and timing is considered because in this analisys the standard cells in DExIMA-Backend are evaluated with zero loads. For the static power, instead, the average value is taken. The absolute error is computed between DExIMA-Backend and Liberate, showing

Table 9.1 Standard cells performance comparison between Liberate and DExIMA-Backend.

Cell	Parameter	DExIMA-Backend	Liberate	AE*
INV_X1	Dynamic Energy (fJ)	0.61	Min 0.7	0.09
	Timing (ps)	1.97	Min 2	0.03
	Static Power (nW)	74.01	Avg 86.13	12.12
AND2_X1	Dynamic Energy (fJ)	1.28	Min 1.5	0.22
	Timing (ps)	5.43	Min 7.4	1.97
	Static Power (nW)	145.96	Avg 138.22	7.74
NAND2_X1	Dynamic Energy (fJ)	1.34	Min 1.4	0.06
	Timing (ps)	3.34	Min 3.4	0.06
	Static Power (nW)	133.40	Avg 88.09	45.30
OR2_X1	Dynamic Energy (fJ)	1.34	Min 1.6	0.26
	Timing (ps)	5.02	Min 3.2	1.82
	Static Power (nW)	116.65	Avg 118.02	1.37
MUX2_X1	Dynamic Energy (fJ)	2.46	Min 2.1	0.36
	Timing (ps)	10.55	Min 11.6	1.05
	Static Power (nW)	216.77	Avg 182.79	33.98
XNOR2_X1	Dynamic Energy (fJ)	2.93	Min 2	0.93
	Timing (ps)	11.78	Min 14.4	2.62
	Static Power (nW)	262.55	Avg 185.46	77.09
XOR2_X1	Dynamic Energy (fJ)	3.07	Min 2.3	0.77
	Timing (ps)	4.23	Min 4.7	0.47
	Static Power (nW)	219.29	Avg 174.04	45.25

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

a maximum difference of 0.93 fJ for the dynamic energy, 2.62 ps for the timing, and 77.09 nW for the static power. The intrinsic differences between the tools mainly cause these differences: Liberate performs several simulations, estimating the performance of each transition and measuring the contributions. In contrast, 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.

9.1.2 Comparisons with Synopsys Design Compiler

DExIMA-Backend should also be validated with more complicated blocks to have a clear indication of its precision. Since they are composed of multiple standard cells, a synthesis process is necessary. For this purpose, Synopsys Design Compiler is used, with the just characterized library as a database, to keep similarities with DExIMA-Backend computational models as much as possible. The estimations with both tools are made without the back-annotation process, with a fixed switching activity factor α of each net equal to 0.5. The synthesis script is reported in Listing 54.

```
#set the significant digits in the reports
set_app_var report_default_significant_digits 4
#set the target library
set t_lib Nangate45_NOPEX_ecsm
#get the library cells
set COLLECTION [get_lib_cell $t_lib/*]
set COLLECTION_CELLS [get_object_name $COLLECTION]
#set the cells to keep
set KEEP_CELLS {
        Nangate45_NOPEX_ecsm/NAND2_X1
        Nangate45_NOPEX_ecsm/AND2_X1
        Nangate45_NOPEX_ecsm/OR2_X1
        Nangate45_NOPEX_ecsm/MUX2_X1
        Nangate45_NOPEX_ecsm/INV_X1
        Nangate45_NOPEX_ecsm/DFFR_X1
        Nangate45_NOPEX_ecsm/FA_X1
        Nangate45_NOPEX_ecsm/HA_X1
        Nangate45_NOPEX_ecsm/NAND2_X1
        Nangate45_NOPEX_ecsm/NOR2_X1
        Nangate45_NOPEX_ecsm/OAI21_X1
        Nangate45_NOPEX_ecsm/TBUF_X1
        Nangate45_NOPEX_ecsm/TINV_X1
        Nangate45_NOPEX_ecsm/XNOR2_X1
        Nangate45_NOPEX_ecsm/XOR2_X1
}
#cells with dont_use attribute are not considered in the synthesis
foreach lib_cell $COLLECTION_CELLS {
        set flag_to_keep_cell 0
        foreach cell $KEEP_CELLS {
                if { $cell == $lib_cell } {
                        set flag_to_keep_cell 1

                }
        }
        if { $flag_to_keep_cell == 0 } {
                set_dont_use $lib_cell
        }
}
#STEP1: READING VHDL sources
analyze -f vhdl -lib WORK ../VHDLfiles/configpkg.vhd
analyze -f vhdl -lib WORK ../../CODE/LIBRARY/sources/Gates/AND2.vhd
...
analyze -f vhdl -lib WORK ../VHDLfiles/top_level.vhd
set power_preserve_rtl_hier_names true
elaborate top_level -arch structure
#-------------------------------------STEP2: CONSTRAINTS
#CREATE CLOCK
create_clock -name MYclk -period 6 CLK
set_dont_touch_network MYclk
#set the toggle rate equal to 0.5 with base clock
set_switching_activity -toggle_rate 0.5 -static_probability 0.5 -base_clock MYclk {*}
#--------------------------------------STEP3: SYNTHESIS
compile
##------------------------------------STEP4: results
report_timing > ./results/timingtop_level.txt
report_area > ./results/areaHIER_top_level.txt
report_power > ./results/power_noBA_top_level.txt
report_power -net > ./results/net_power.txt
##------------------------------------STEP5: BA SETUP
change_names -hierarchy -rules verilog
write_sdf ../netlist/top_level.sdf
```

```
62 write -f verilog -hierarchy -output ../netlist/top_level.v
63 write_sdc ../netlist/top_level.sdc
64 exit
```

Listing 54 Synthesis script used to compare DExIMA-Backend and Synopsys Design Compiler performance results.

To compare the two tools, using the same cells to implement the considered blocks is fundamental. DExIMA-Backend models the performance of the cells with a drive strength of 1, so all the cells with the suffix "_X1" at the end of the cell name should be selected in the Design Compiler as well. This is accomplished with the directive `set_dont_use`, indicating explicitly the cells not to use. Next, the script analyzes all RTL files of the implementation and elaborates the architecture. As reference, a clock period of 6 ns is used, but it can be changed by the user in DExIMA-CAD if needed. At line 51 of Listing 54, the static probability, i.e., the percentage of time in which the considered signal is at logic ’1’, and the toggle rate, expressed in transitions per second, are set to 0.5. When specified, the flag `base_clock` couples the value of the toggle rate passed with the `toggle_rate` flag with the clock period in the following way:

$$TR_{net_i} = \frac{TR_{\text{specified}}}{\text{Clock period}} \tag{9.1}$$

Where $TR_{\text{specified}}$ is the value passed with `toggle_rate` flag. In this case, the definition of the $TR_{\text{specified}}$ coincides with the switching activity factor.

$$TR_{net_i} = \frac{TR_{\text{specified}}}{\text{Clock period}} \equiv \frac{\alpha}{\text{Clock period}} \tag{9.2}$$

After the definition of the toggle rate, the design is compiled and synthesized with `compile` and the performance reports extracted, as well as the Verilog netlist of standard cells, the Standard Delay File (SDF), and Synopsys Design Constraint (SDC) files. Several blocks, such as registers, ripple-carry adders with 8, 16, and 64 bits parallelism, and simple logic gates are considered. For each case, the values of the Backend are directly compared with Synopsys DC, estimating the relative percentage error. However, to have a fair comparison between the two tools, Synopsys must synthesize and perform estimations on the same architecture as the DExIMA-Backend one, meaning that the same standard cells must be used in the two cases. Therefore, the following steps are followed to accomplish this constraint:

1. *Structural description of the test architecture.* Considering, for instance, adders test architectures, in DExIMA-Backend, they are implemented as ripple-carry adders, so the RTL code must be described in a structural way to match the models.

2. *Force the synthesizer to use the same standard cells as DExIMA-Backend's ones.* Most importantly, apart from the same structural description of the architectures, Synopsys DC must map the same standard cells as the ones included in the DExIMA-Backend structural blocks. To accomplish that, PRAGMAs are used in the VHDL source files: an example of a full adder RTL description is reported in Listing 55.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
entity FA is
        port(A: in std_logic;
        B: in std_logic;
        C: in std_logic;
        S: out std_logic;
        CO: out std_logic);
end FA;
architecture structure of FA is
-- synopsys translate_off
signal out_xor_1: std_logic;
signal out_and_1, out_and_2: std_logic;
-- synopsys translate_on
component FA_X1
port(
A,B,CI: in std_logic;
CO,S: out std_logic
);
end component;
begin
        -- synopsys translate_off
        out_xor_1 <= A xor B;
        out_and_1 <= out_xor_1 and C;
        S <= out_xor_1 xor C;
        out_and_2 <= A and B;
        CO <= out_and_1 or out_and_2;
        -- synopsys translate_on
        -- synopsys dc_script_begin
        -- set_dont_touch FA_instance
        -- synopsys dc_script_end
        FA_instance: FA_X1
        port map(A,B,C,CO,S);
end structure;

```

Listing 55 Full adder RTL code with PRAGMAs

> The directive `--synopsys translate_off` tells the synthesizer not to consider the subsequent code until it reaches `--synopsys translate_on`: in this way, the behavioral part of the code will not be synthesized. With the current code, however, the synthesizer possibly removes the `FA_X1` cell since it is mapped as a black box, so to avoid synthesis optimizations, the directive `--set_dont_touch FA_instance` is used, wrapped in `--synopsys dc_script_begin` and `--synopsys dc_script_end`, specifying that it is a Synopsys DC command.

The results and the test architectures are reported in Fig. 9.2 and Fig. 9.3, where dynamic, static, total powers, and critical path are compared with Synopsys DC. Highlighted in yellow, there is the relative error in percentage computed as:

$$\text{Relative error}_{\text{DExIMA-Synopsys}}(\%) = \frac{X_{\text{DExIMA}} - X_{\text{Synopsys}}}{X_{\text{Synopsys}}} \times 100\% \tag{9.3}$$

Where X_{DExIMA} and X_{Synopsys} are the considered measures for DExIMA and Synopsys, respectively. These results can be considered valid, remembering that DExIMA-Backend should provide a rough and fast estimation of the performance. Once the LiM circuits are evaluated with the Backend, the user can directly synthesize the design. As expected, the two tools provide different results because of the intrinsic differences between the computational models: DExIMA performs static evaluations on the circuit without the characterization process that includes measurements at the SPICE level of the standard cells' performance. This is not the only difference; in fact, synthesizers, in addition to having a different computation model, are also able to propagate switching activity within the nodes of the circuit, whereas DExIMA-Backend, when used in worst-case mode, imposes the same toggle rate for each node in the design. The switching activity greatly impacts the performance estimation, also greatly modifying the dynamic power value. It is therefore possible that, in estimations on more complex architectures, DExIMA-Backend overestimates the

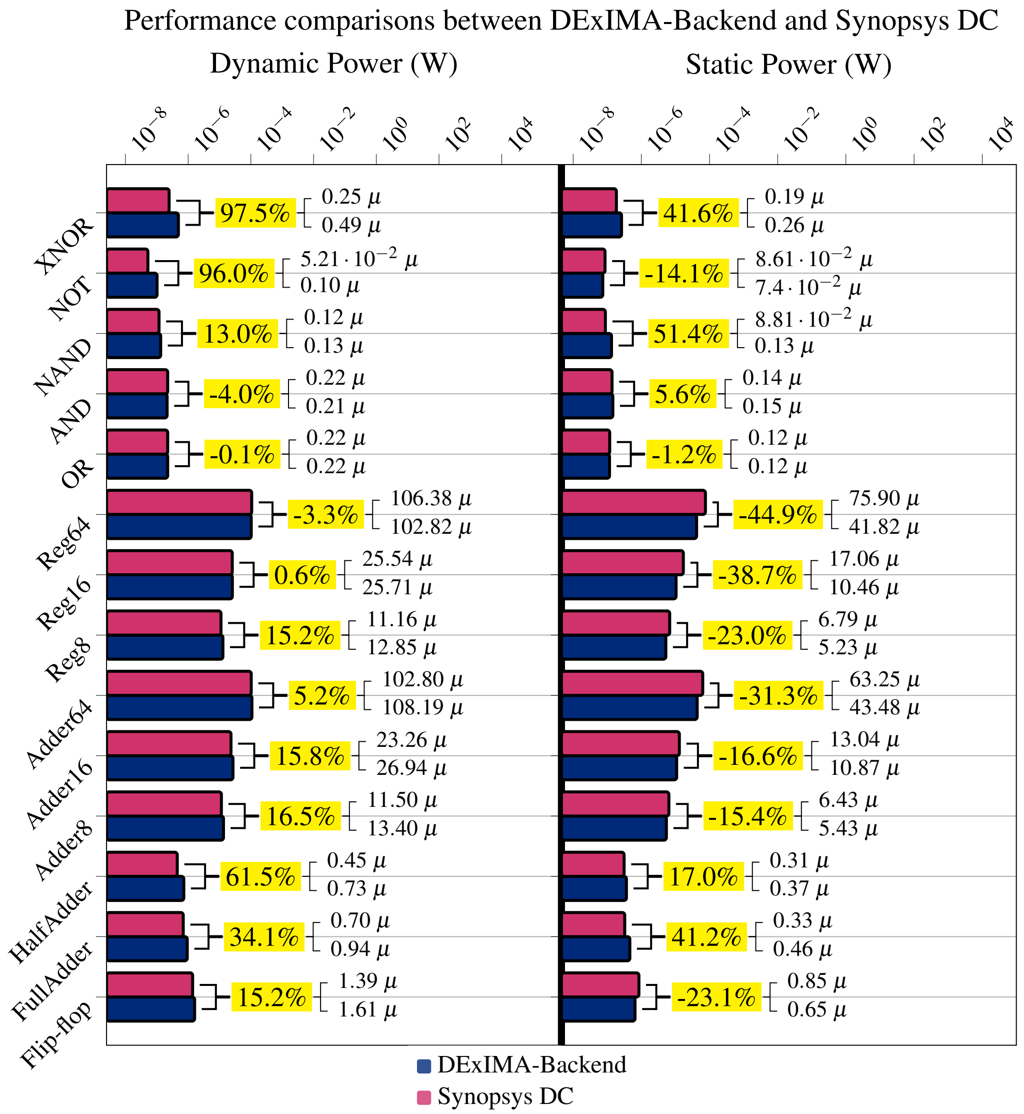


Fig. 9.2 Dynamic and Static powers comparisons between DExIMA and Synopsys Design Compiler.

dynamic power even considerably, but this is not a disadvantage because the user is still able to get an indication of the circuit consumption for a given technology, and once one is satisfied with the values obtained, he/she can move on to synthesis and Place&Route. However, the results obtained for each relevant parameter are close to the reference ones, suggesting that DExIMA can provide valid indications of system performance faster than classical synthesizers.

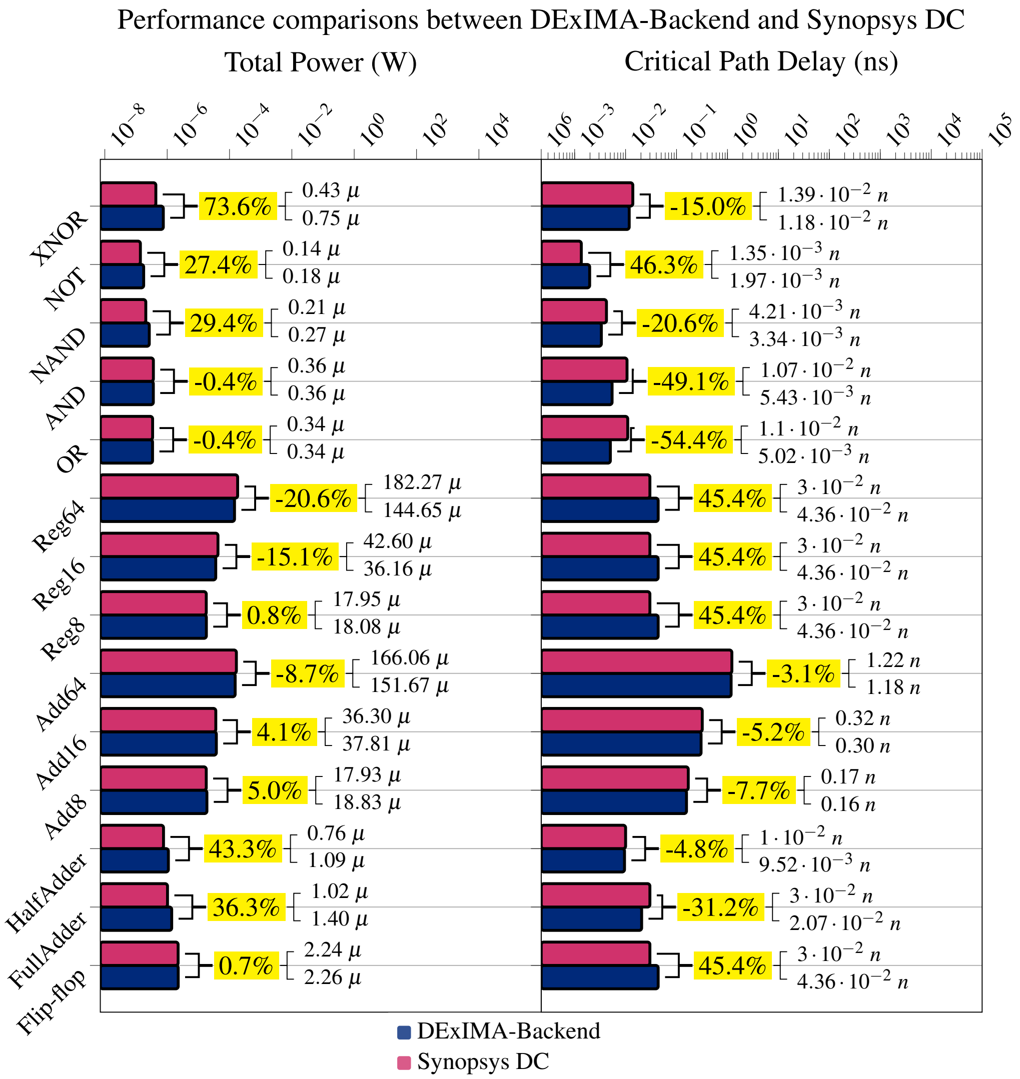


Fig. 9.3 Total power and critical path comparisons between DExIMA and Synopsys Design Compiler.

Another parameter to consider is the execution time of DExIMA-Backend in making performance estimates. To do this study, the execution times of DExIMA-Backend and Synopsys Design Compiler in providing the results were compared for the previously studied architectures. The values of execution times are printed on the command line by the two tools when the entire procedure finishes and they are reported in Table 9.2. Both tools are executed on the same PC's system architecture. The speed-up achieved by DExIMA-Backend for the proposed benchmarks varies from $\sim$ 21.4 to $\sim$ 263.2 times, confirming its capability to provide almost

Table 9.2 CPU usage time comparison between DExIMA-Backend and Synopsys DC.

Architecture	**CPU usage time (s)**		**Speed-up**
	DExIMA	**Synopsys**	
Reg64	0.281	6	21.4
Reg16	0.084	5	59.5
Reg8	0.086	7	81.4
Adder64	0.08	7	87.5
Flip flop	0.053	6	113.2
Adder8	0.041	6	146.3
HalfAdder	0.04	7	175.0
XNOR	0.028	5	178.6
Adder16	0.027	5	185.2
FullAdder	0.032	6	187.5
NOT	0.03	6	200.0
OR	0.024	5	208.3
AND	0.019	5	263.2
NAND	0.019	5	263.2

accurate estimations in a very short time. Further comparisons with more complex architectures are proposed in section 10.7.

9.2 Conclusions

In this chapter, the validation of the DExIMA-Backend tool is addressed, which is of fundamental importance in demonstrating the reliability of the tool. The validation steps require the use of Cadence Liberate (for characterization of the reference cell library) and Synopsys Design Compiler (for comparison with architectures composed of multiple standard cells). Comparisons are made in terms of performance and also in terms of execution time required by DExIMA-Backend in making the estimates. For the proposed subset of test architectures, DExIMA-Backend proved to be a fast and relatively accurate tool compared to commercial standards. In the next chapter, these comparisons will be repeated for much more complex LiM architectures, evaluating the reliability of DExIMA-Backend even on extremely complex architectures.

Chapter 10

Testing DExIMA: benchmarking and comparisons

Summary

In this section, DExIMA is tested with different LiM structures. In all proposed benchmarks, the LiM architecture is designed according to the algorithm to be implemented, starting from the LiM cell, the IRL, and finishing with the entire memory array. Finally, the "Comparison CPU-Memory" tool is exploited to compare the results obtained with the classical CPU-Mem and the beyond von Neumann CPU-Mem-LiM architecture.

10.1 XNOR-Net: a binary neural network

The first proposed benchmark is the XNOR-Net [7]. XNOR-Net working principle is recalled in the following:

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 into a series of bitwise-XNOR and pop-counting operations. Pop-counting consists of performing the difference between the number of ones and zeros in a bit string.

A generic i-th convolution operation is computed following the Equation 10.1:

$$Conv_{XNOR} \approx \text{pop-count}\left[!(I_i \wedge W_0), \ldots, !(I_{i+N-1} \wedge W_{N-1})\right] \qquad (10.1)$$

Where N is the kernel window size. This algorithm is implemented in [13], which proposes two LiM arrays made of basic LiM cells with an XNOR gate (to compute the product) and a half adder (to compute the pop-count), respectively. Only the first LiM Array dedicated to the XNOR products is implemented in DExIMA-CAD. The basic LiM cell is captured from DExIMA-CAD and depicted in Fig. 10.1. The

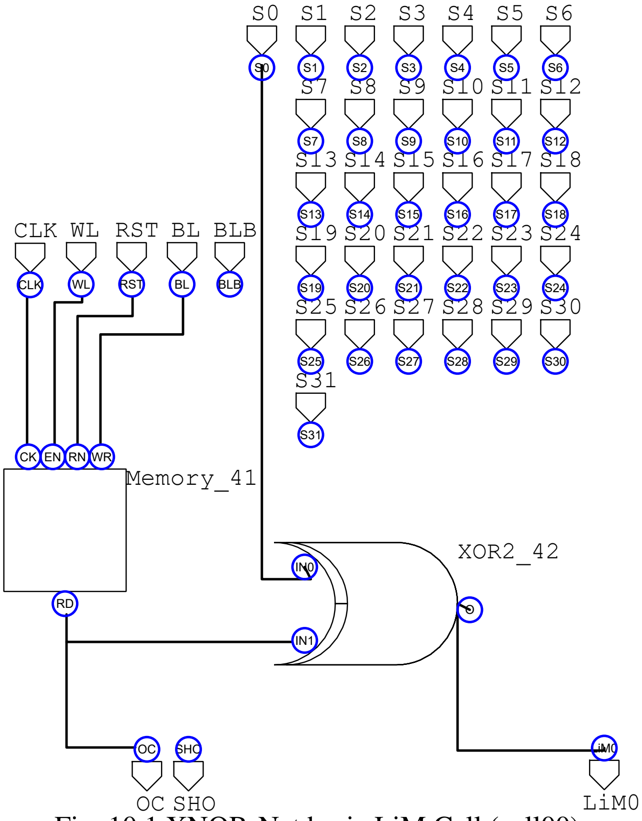


Fig. 10.1 XNOR-Net basic LiM Cell (cell00).

CPU-Mem code for the XNOR-Net (named xnor_net_cpu.c) is the following:

```
volatile int memory_content[N] = {0};
volatile int weights[K] = {0x3,0x2,0x3e,0x12,0x87,0xf2,...};
volatile int value = 0;
for(int j = 0; j < K; j++)
{
        for(int i = 0; i < N; i++)
        {
                value = !(memory_content[i] ^ weights[j]);
        }
}
```

Listing 56 XNOR-Net CPU-Mem algorithm.

The algorithmic steps to implement the LiM version of the Listing 56 are the following:

1. *Data precharging.* The values of the input matrix are binarized and saved in each memory row such that the 0th bit of row 0 corresponds to the binarized pixel (0,0) of the input image, the 1st bit of the row 0 corresponds to the binarized pixel (0,1), etc. This step requires `N` clock cycles.
2. *Binary convolution computation.* The binarized weights are streamed inside the memory array by means of the selectors `S00-S31` shown in Fig. 10.1. The array performs `N` parallel XNORs at the same time. This step requires `K` clock cycle.

10.2 Matrix-Vector Multiplication

By exploiting Intra Row Logic blocks, the user can realize more common operations such as Matrix-Vector Multiplication (MVM). The pseudocode is reported in Listing 57.

```
for(int i = 0; i < ROWS; i++)
    for(int j = 0; j < COLS; j++)
    {
        Y[i] = Y[i] + M[i][j] * V[j];
    }
```

Listing 57 MVM pseudocode.

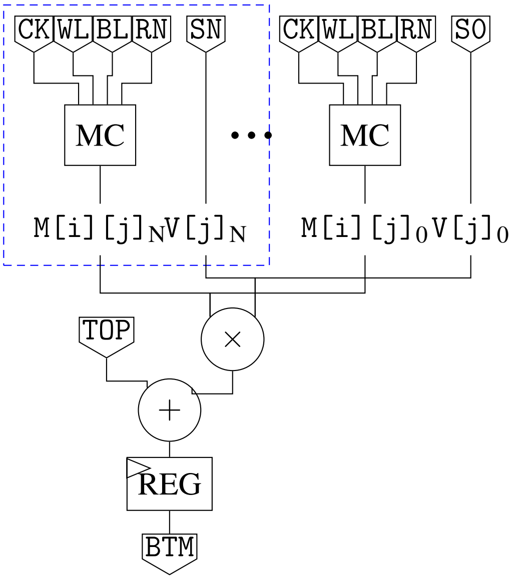


Fig. 10.2 LiM cell (bordered in blue) and IRL circuits for the MVM algorithm. IRL contains a multiplier, an adder and a register.

For each line of the memory array, the LiM cell, shown in Fig. 10.2, is made of a simple memory cell (MC in the figure) that stores 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]`$_n$ for each algorithmic step. The adder of the IRL takes in input the multiplied value of `M[i][j]*V[j]` and the signal coming from the previous memory line, that corresponds to `M[i][j-1]*V[j-1]`, since the TOP signal of the k-th row is connected to the BTM signal of the (k-1)-th row. Referring to Listing 57, the steps executed by the MVM algorithm are the following:

1. *Data precharging.* Matrix elements are saved in each line of the memory array so that inside row 0, there will be `M[0][0]`, `M[0][1]` in row 1, `M[0][2]`in row 2, `M[0][COLS-1]` in row COLS, `M[1][0]` in 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 LiM array for each clock cycle by means of `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, that 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, connected to the BTM of

the previous rows, assume the value of the previous accumulation that is summed 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.

10.3 K-Nearest Neighbor

The K-Nearest Neighbor (K-NN) is a classification and regression algorithm. Starting from a dataset, the K-NN calculates the distance between the points in the dataset and k centroids, returning those with the shortest distance. In DExIMA, this algorithm is implemented by considering the following pseudo-algorithm with 1 centroid:

```
#define N 1024
int D[N] = {0};
int X[N];
int Y[N];
int Xs,Ys;
for(unsigned int i = 0; i < N; i++)
{
    D[i] = abs(Xs-X[i]) + abs(Ys-Y[i]);
}
```

Listing 58 K-NN pseudocode.

The circuit that implements the IRL block is shown in Fig. 10.3, which can implement sums and absolute values. An IRL block must be inserted for each memory row to implement K-NN. The calculation of the absolute value is done by `Adder_19`, in fact, the operator A is the value 0 stored in the register `Reg_21`, while operator B takes the result of the subtraction (performed by `Adder_18`) and complements it with XORs based on the value of the MSB, exploiting the adders' sum/subtraction functionality.

The array will consist of two different types of LiM Cells, shown in Fig. 10.4. The cell shown in the Fig. 10.4 (a), is a variant of a standard memory cell, in fact it connects the values of the selectors to the SHO bus. The selectors convey the Xs and Ys coordinates of the centroid, which must be shared with the entire array. The memory structure will consist of all standard memory cells, except for the last row,

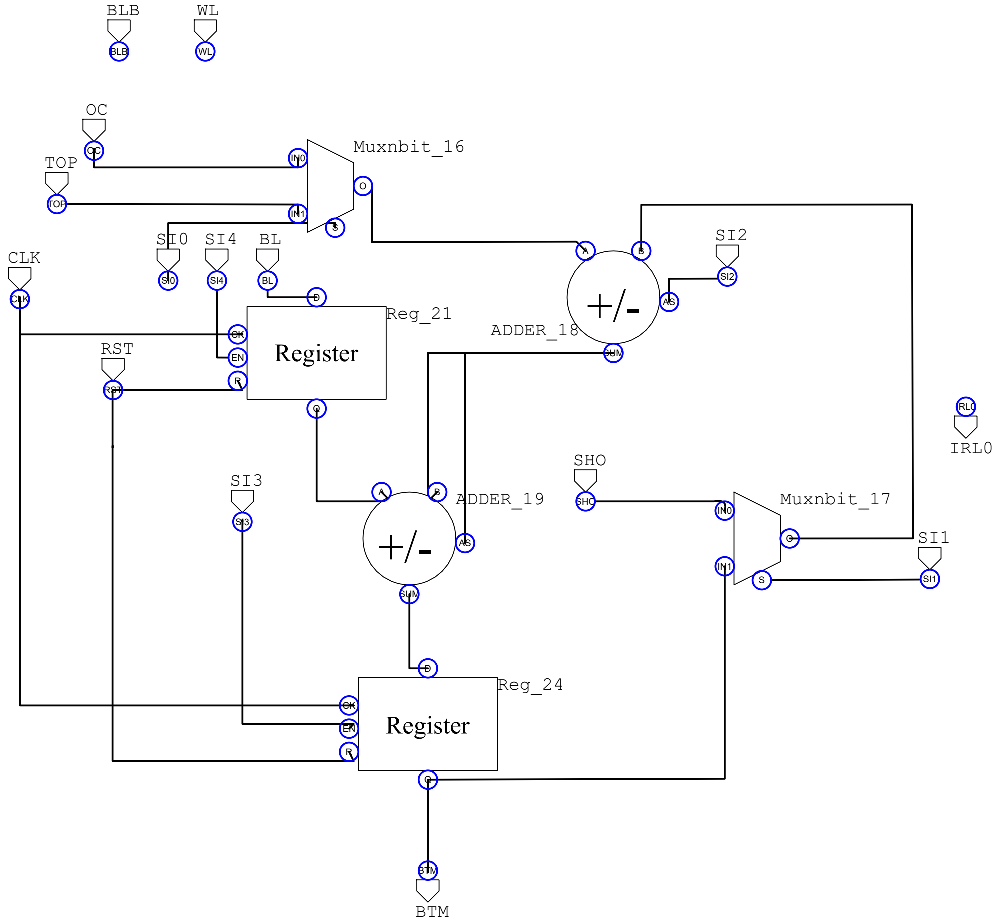


Fig. 10.3 IRL block implementing the K-NN computation.

where there will be special cells in the Fig. 10.4 (a). Specifically, each column will have a different cell: cell 00 in column 0, which connects the S0 switch to SHO; cell 01 in column 1, which connects the S1 switch, and so on. The algorithm is implemented as follows:

1. *Data precharging.* Within the even-numbered rows (0, 2, 4, 6,..., 1022), the values of X[i] are loaded, and in the odd-numbered rows (1,3,5,...,1021), the values of Y[i].

2. *First absolute value calculation.* At the beginning of the algorithm, only the even-numbered rows plus the 1023rd (containing the special LiM cells) are enabled, and the value of Xs is entered on the selector signals S0,...,S31. Then, in the same clock cycle, the IRLs of the even-numbered rows perform the subtraction between the value contained in the cells (thus the X[i]) and the

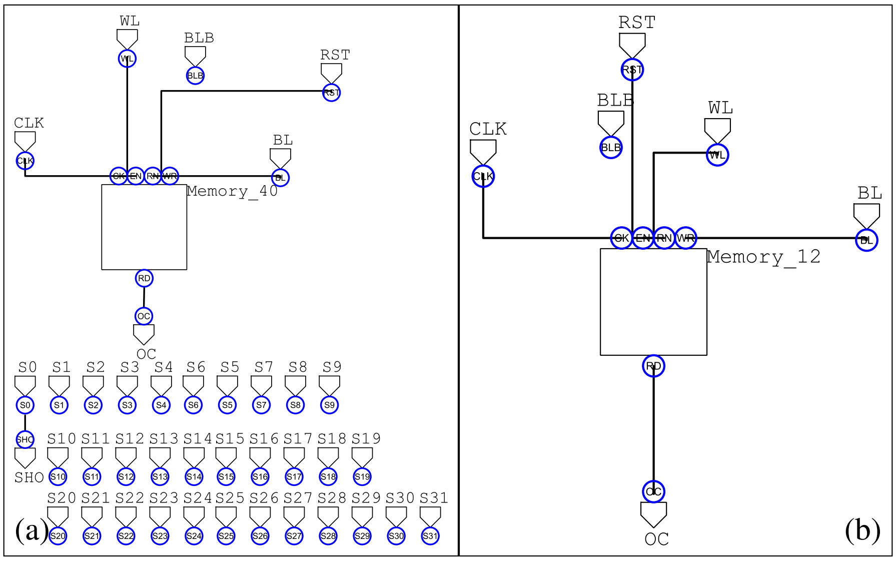

WL
BLB
RST
CLK
BL
Memory_40
OC
S0 S1 S2 S3 S4 S6 S5 S7 S8 S9
SHO
S10 S11 S12 S13 S14 S15 S16 S17 S18 S19
S20 S21 S22 S23 S24 S25 S26 S27 S28 S29 S30 S31
(a)
RST
BLB
WL
CLK
BL
Memory_12
OC
(b)

Fig. 10.4 LiM Cells used to implement the K-NN algorithm. (a) Cell 00, used to connect the selectors on the SHO bus. (b) Standard memory cells that simply hold data.

value of Xs (which will be on the SHO bus). On the subtraction, the absolute value is calculated with `Adder_19`, and the result will be saved in `Reg_24`.

3. *Second absolute value calculation.* The next step will be to enable the odd-numbered rows plus the 1023rd, simultaneously putting the value of Ys on the SHO bus. Following the same method as before, the calculation of subtraction and absolute value will be saved in `Reg_24`.

4. *Final sum.* In the final step, only the odd rows (1,3,...,1021) are enabled, and the sum between $|X[i]-Xs|+|Y[i]-Ys|$ is performed. This can be done using the mechanism offered by TOP-BTM signals: the BTM of the previous even-numbered row will be linked to the TOP of the next odd-numbered row. The result of the sum is saved again in `Reg_24`, where the final value can be found.

10.4 Bitmap Indexing

The concept of the Bitmap Indexing (BMP) algorithm was explained different times in this thesis. Here, a quick explanation from section 2.1.2 is reported for reference.

BMP is a strategy for quick database searches, according to [129]. It involves altering the data representation in order to search for information inside the database using simple bitwise operations. Each bit represents a field in bit arrays (also known as bitmaps), which may be true or false. The corresponding bit is set to true if the record corresponds to a certain field. The IRL structure includes the OneCounter, which implements the samples count.

Until now, benchmarks made use of a fairly large number of data elements, but in this context, the BMP algorithm implemented in this section employs a very small database (consisting of 15 elements) aiming to evaluate the impact of an extremely small LiM on the performance of a von Neumann architecture. The memory array will be 16×8 bits in size and is composed of two types of cells: standard LiM memory cells and computation LiM cells. The former store data and, when selected appropriately, send data to the SHO bus through the use of a tristate buffer. On the other hand, the latter also perform calculations coupled with IRL logic. The array will then be organized as follows:

```
-- File MEMORYARRAY_1.csv
st_memory_cell,st_memory_cell,...,st_memory_cell
st_memory_cell,st_memory_cell,...,st_memory_cell
...
compute_cell00,compute_cell01,...,compute_cell07

-- File MEMORYARRAY_1_intrarow.csv
NO-Logic
...
OneCountIRL
```

Listing 59 Organization of the LiM array for the BMP algorithm.

LiM Cells (standard and computation) are shown in the Fig. 10.5.

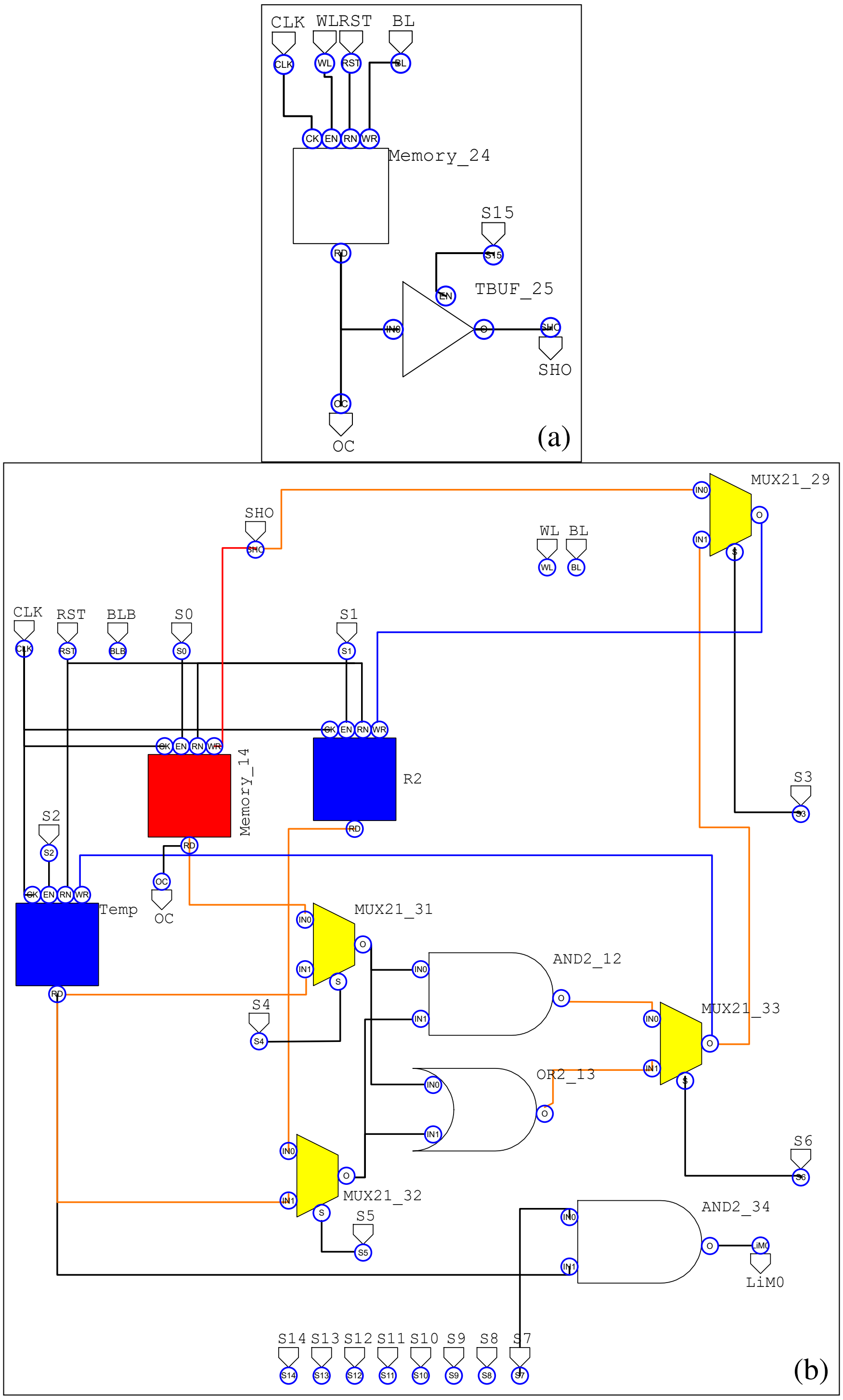


Fig. 10.5 LiM Cells for BMP implementation. (a) Standard memory cells (st_memory_cell) and (b) LiM computational cells (compute_cell00).

The algorithm is implemented as follows:

1. *Data precharging.* Bitmaps are loaded for each row. Specifically, in the proposed implementation, we have a sample of 8 elements with the following characteristics: category 1, category 2, type A, B, C, D, E,..., M. The characteristics are stored in the first 15 rows of the array, 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.

2. *Addressed question.* In the proposed example, the following query is to be answered: "how many elements are of category 2 and are they type A or type B?". The algorithm then results in a series of bitwise operations involving rows 1, 2, and 3. The algorithm will then be:

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#define ELEMENTS 8
#define BITMAPS 15
int OneCount(unsigned char element)
{
    unsigned char temp;
    int number_of_occurences = 0;
    for(unsigned int i = 0; i < ELEMENTS; i++)
    {
        temp = element >> i;
        temp = temp & 1;
        number_of_occurences += temp;
    }
    return number_of_occurences;
}
int main()
{
    unsigned char bitmap[BITMAPS];
    unsigned char result = bitmap[1] & (bitmap[2] | bitmap[3]);
    int counting = OneCount(result);
    return 0;
}

```

Listing 60 Pseudo-algorithm of the BMP query.

Taking the Fig. 10.5 as a reference, the steps performed on the LiM are as follows:

- *Save the contents of row 2 (corresponding to type A) within* Memory_14 *of the compute cells. This operation is done through the SHO bus, enabling the tristate buffers of row 2 containing standard memory cells.*
- *Save the contents of row 3 within* R2 *in the computation cells, exploiting the same principle used previously. Row 3 contains type B.*
- *Perform the OR between the contents of* Memory_14 *and* R2 *and save the result in* Temp. The result is the bitwise OR between type A and type B.
- *At this point, LiM takes the contents of row 1 (corresponding to category 2) and saves it in* Memory_14, *again exploiting the SHO bus.*
- *The AND between the contents of* Memory_14 *and* Temp *is done, thus obtaining the final query stored in* Temp.

These steps require 5 clock cycles. For the OneCounting operation, the IRL is also involved in combination with the S7-S14 computation cell selectors. The structure of the IRL is shown in Fig. 10.6: OneCounting is done by doing an AND & Right

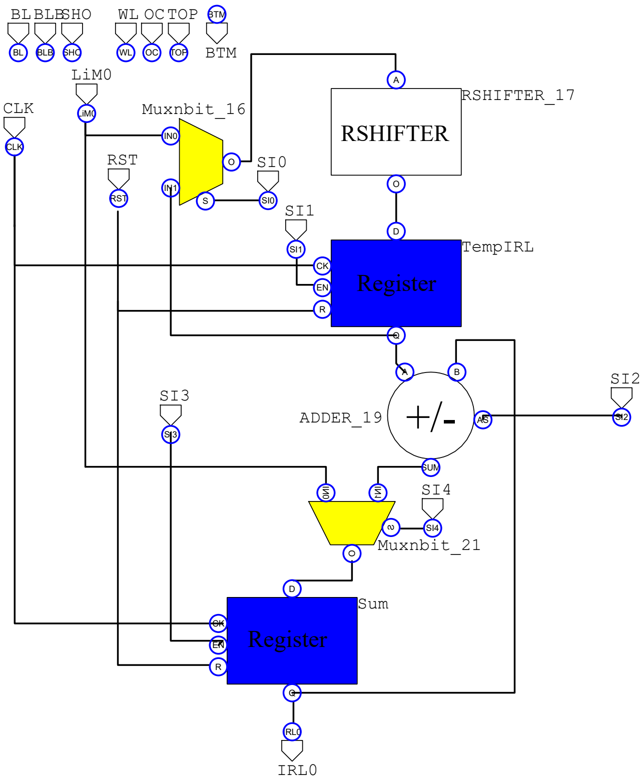


Fig. 10.6 IRL circuit for OneCounting operation.

Shift operation to count the number of ones within a sequence of bits. Then, the remaining part of the algorithm is carried out as follows:

- *Initially, the sequence "00000001" is inserted on the S14-S7 selectors.* The selectors are employed in the computation cells and transfer the mask to perform the bitwise AND between the contents of `Temp` and the mask itself. In cell 00, the AND will be done with S7, in cell 01 with S8, and so on. The value of the bitwise AND is supplied to the IRL via the LiM0 pin: the result is saved within the `Sum` register.
- *Next, the value "00..10" will be entered on switches S14-S7.* The AND is done again, but this time the result must be shifted to the right by 1 position by the use of `RSHIFTER_17`, which is a hardwired Right Shifter. The shifted value is saved within the `TempIRL` register.
- *At this point, a sum is performed between the value previously saved in the* `Sum` *register and the contents of* `TempIRL`. Then, the result is saved again in `Sum`.
- *The cycle starts again by providing the selectors the value "0..100" and performing the AND bitwise.* This time, however, the shift must be done in two positions to the right, requiring to iterate the procedure twice.
- *The algorithm is iterated until the mask combination of "1...0" is reached.*

The calculation of the OneCounting depends on the size of the bitmaps. No right shift is required at the beginning of the OneCounting algorithm, so the accumulation is performed in 1 clock cycle. At the second iteration, 1 clock cycle is required for the shift and 1 clock cycle for the accumulation. Next, 2 clock cycles are required for the shift and 1 clock cycle for the accumulation, and so on. Totally, the OneCounting procedure requires $\sum_{i=0}^{\text{Size bitmap}-1}(i+1)$ clock cycles.

10.5 Mean-Variance

The last proposed benchmark is the Mean-Variance computation, implemented as in Listing 61.

```
sum1, sum2, sum3 = 0;
for(int i = 0; i < N; i++) sum1 += X(i);
mean = sum1/N;
for(int i = 0; i < N; i++)
{
    sum3 += (X(i) - mean);
    sum2 += (X(i) - mean) * (X(i) - mean);
}
variance = (sum2-sum3*sum3/N)/N;
```

Listing 61 Mean-Variance pseudocode.

Mean-Variance computation is a sequential algorithm, meaning that in an i-th iteration, the calculations depend on values computed in the (i-1)-th iteration. However, the user can still use DExIMA to design a LiM array that works sequentially. To accomplish the Mean-Variance algorithm, the LiM array has only one IRL circuit at the last row that computes the values of sum1,sum2,sum3, mean, and variance, following the algorithm in Listing 61. The LiM cell, represented in Fig. 10.7, is simply composed of a memory cell and a buffer tristate on the shared output pin (SHO). Referring to Fig. 10.7 and Listing 61, the algorithmic steps are the following:

1. *Data precharging.* The samples are stored inside the array, with one sample for each memory line. This step requires N clock cycles.

2. *Sum1 computation.* The `sum1` value is computed by enabling the `sum1` register and by enabling the selector `S0` one row at a time. This step 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 equals a power of 2 since the hardware division is a very complex and intensive task that should be delegated to the CPU. The result is saved inside the `mean` register. This step requires 1 clock cycle.

4. *Sum3 and sum2 computations.* In the second for cycle, `sum3` and `sum2` can be computed by scanning the array again and by 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 the sum2 and sum3 are ready, the IRL circuit starts the variance computation. The multiplier inputs 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.

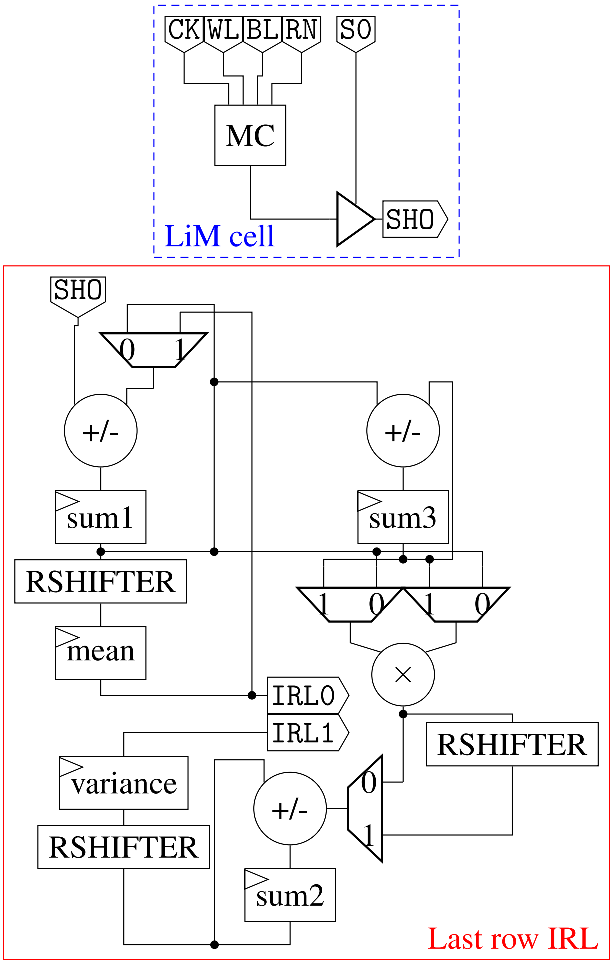


Fig. 10.7 Mean-Variance LiM cells and IRL circuits. Selectors pins are not shown for clarity.

10.6 Results comparison

In this part, the results obtained for the proposed benchmarks are reported. In all the cases, after the definition of the LiM array, generation of the VHDL code, clock period definition, and simulation, DExIMA-Backend performs the estimations, providing results in graphical (for example as shown in Fig. 10.8 and Fig. 10.9) or tabular formats (Table 10.1). Estimations in Table 10.1 are provided without the back-annotation process using 45nm CMOS technology. When back-annotation is not performed, DExIMA asks the user to provide a value of the toggle rate (TR) for the dynamic power estimation. In all cases, a TR of 0.5 is used for each node in the design, emulating a worst-case scenario. After that, the user can compare CPU-Mem and CPU-Mem-LiM architectures by exploiting the "Comparison CPU-Memory" section of DExIMA. Both the CPU-Mem and CPU-Mem-LiM algorithms are simulated with Gem5. Finally, by means of the "Compare CPU-Mem and CPU-Mem-LiM" option (see Fig. 3.5), DExIMA proposes a comparison between the two systems considering five figures of merits: total caches accesses, total caches energy, LiM energy, LiM algorithm execution time, CPU total execution time (memory read/write and computation). For all quantities, the lower value is better. The performance achieved by each benchmark is summarized in Table 10.1, which is divided into DExIMA-Backend results, i.e., the values referring to the LiM arrays, and CPU-Mem/CPU-Mem-LiM, i.e., the comparisons between the two systems. Last comparisons in terms of Execution Time/Memory Energy/Total Memory Accesses between CPU-Mem and CPU-Mem-LiM are proposed, which consider the two solutions having a core clock frequency of 1GHz. In this part, the three parameters analyzed are obtained in the following way: the Execution Time is calculated as the sum of the processor Execution Time (obtained as CPU Execution Cycles multiplied by the core clock period) and the LiM Execution Time in executing the algorithm; the Total Memory Energy is obtained as the sum of the caches energy plus the LiM Power-Delay product (obtained as the total LiM Power multiplied by the LiM Execution Time, also considering the data preload); finally, Total Memory Accesses is obtained as the sum of cache memory accesses and LiM accesses (considering data preload in the LiM and cache accesses to move data to the LiM).

Table 10.1 Performance values of each benchmark and comparison CPU-Mem and CPU-Mem-LiM.

Benchmark		**XNOR-Net**		**MVM**		**Mean-Variance**		**BMP**		**K-NN**		**Category**
LiM size		1024		256		1024		16		1024		**Backend results**
Parallelism		32		16		32		8		32		
Clock period (ns)		6.000		6.000		6.000		6.000		6.000		
Critical path (ns)		0.051		0.818		2.867		0.284		1.299		
Execution time (μs)		6.210		1.632		18.444		0.342		6.162		
Frequency (MHz)		166.667		166.667		166.667		166.667		166.667		
Area (mm^2)		0.285		0.386		0.290		0.002		1.146		
Dynamic Energy (nJ)		0.445		0.614		0.424		0.002		1.885		
Static Energy (nJ)		0.180		0.274		0.185		0.001		0.746		
Total Energy (nJ)		0.625		0.888		0.608		0.003		2.631		
Static Power (mW)		30.021		45.748		30.754		0.172		124.312		
Dynamic Power (mW)		74.111		102.310		70.649		0.394		314.242		
Total Power (mW)		104.132		148.058		101.403		0.566		438.554		
CPU Execution Cycles (Mticks)	CPU-Mem	1094.06	89.3%	88.97	25.5%	349.74	66.6%	53.48	0.0%	236.91	50.7%	**Comparisons CPU-Mem/CPU-Mem-LiM**
	CPU-Mem-LiM	116.76		66.28		116.76		53.47		116.76		
Caches Accesses (k)	CPU-Mem	526.53	92.8%	26.32	42.7%	154.34	75.5%	9.16	0.1%	92.23	59.0%	
	CPU-Mem-LiM	37.80		15.07		37.80		9.15		37.80		
Caches Energy (uJ)	CPU-Mem	68.79	92.6%	3.52	40.8%	20.24	75.0%	1.31	2.5%	12.20	58.8%	
	CPU-Mem-LiM	5.06		2.08		5.07		1.28		5.03		
LiM Power-Delay product (uJ)	CPU-Mem-LiM	0.6467		0.2416		1.8703		0.0002		2.7024		
LiM Execution cycles (ticks)		11.00		23.00		2050.00		41.00		3.00		
Final comparisons												
Execution Time (μs) (@f_{cpu} = 1GHz)	CPU-Mem	1094.06	88.8%	88.97	23.7%	349.74	61.3%	53.48	-0.6%	236.91	48.1%	
	CPU-Mem-LiM	122.97		67.91		135.20		53.81		122.92		
Memory Energy (μJ) (@f_{cpu} = 1GHz)	CPU-Mem	68.79	91.7%	3.52	34.0%	20.24	65.7%	1.31	2.5%	12.20	36.6%	
	CPU-Mem-LiM	5.7067		2.3250		6.9369		1.2772		7.7328		
Total Memory Accesses (k) (@f_{cpu} = 1GHz)	CPU-Mem	526.53	92.6%	26.32	41.7%	154.34	73.5%	9.16	-0.5%	92.23	57.9%	
	CPU-Mem-LiM	38.832		15.341		40.871		9.203		38.824		

Percentages indicate the improvement achieved in the CPU-Mem-LiM case. Benchmarks are evaluated without back-annotation.

- *XNOR-Net*: the algorithm performs the computation of the XNOR products reproducing the first layer of the Fashion-MNIST Convolutional Neural Network proposed in [13], having a size of 32x32 inputs of 1 bit each and a kernel size of 5x5 with 12 output channels `K`. The size of the LiM array, equivalent to `N` in Listing 56, is 1024 rows of 32 bits each. Performance is evaluated with a clock period of 6 ns. As shown in Table 10.1, the CPU-Mem-LiM architecture implementing the XNOR-Net algorithm turns out to be extremely advantageous, as it is capable of reducing energy, execution time and memory accesses compared to the classical von Neumann architecture. In fact, the LiM is able to significantly accelerate the XNOR-Net algorithm by performing all XNOR products in parallel, while maintaining low energy.

- *MVM*: in this case study, a matrix of ROWS $\times$ COLS $= 16 \times 16$ is chosen, so the LiM array has 256 rows of 16 bits each. Performance is evaluated with a clock period of 6 ns. Both the worst-case with a TR equal to 0.5 and the back-annotation processes are proposed to show the differences between the two estimations. The power in the back-annotation case can be expected to be lower than in the worst-case because the switching activity of the nodes, which impacts the dynamic power estimation, is for sure lower. The comparison is shown in Table 10.2. Moreover, all MVM results with back-annotation process during the evaluation phase are shown in graphical format in Fig. 10.8 and Fig. 10.9. In Fig. 10.9 (a-b), it is possible to see the effectiveness of LiM in reducing the von Neumann Bottleneck: apart from lightening the computational efforts of the CPU, it also reduces the number of instructions and memory operations. From the results in Fig. 10.9 (c), LiM-based MVM implementation considerably impacts the system performance, bringing improvements in execution time and energy. In the evaluation, apart from the estimation

Table 10.2 Back-annotation/worst-case power estimation comparison during algorithm evaluation.

Algorithm	**Process**	**Dynamic Power (W)**		**Total Power (W)**	
MVM	*Worst-case*	0.102	-4.90%	0.148	-4.70%
	Back-annotation (execution part)	0.097		0.142	

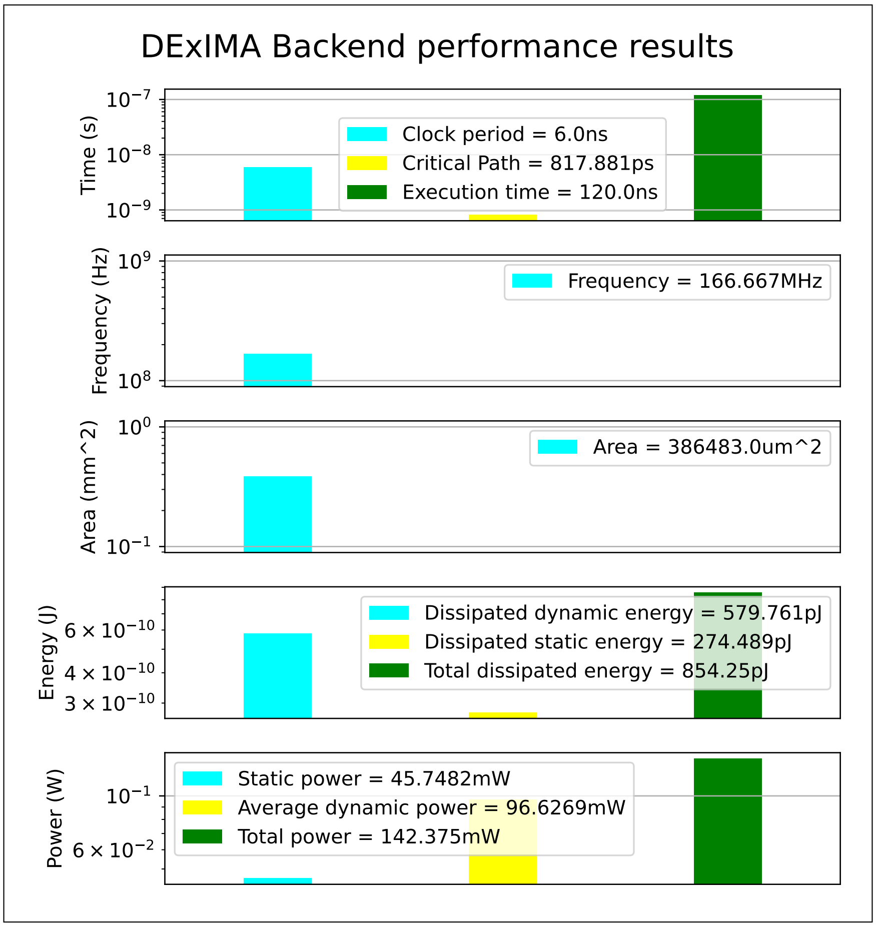


Fig. 10.8 DExIMA-Backend results for the MVM algorithm. The evaluation with back-annotation during the algorithm evaluation phase.

of the LiM array performance, also the bus contribution is estimated. In this case, the BL bus is considered during part of the data preload phase within the array. To do this, the performance estimation process with DExIMA-Backend is re-run, simulating only the preloading part of the first 10 data within the array and giving a clear indication of the impact of the bus. The results are shown in Fig. 10.10, in which the Average Power and Delay measured by Ngspice are reported for a set of combinations of bits (expressed in unsigned format). Interestingly, the impact of the external bus (BL) compared with the LiM's internal bus (BL_LiM) is much lower. This is because the capacitive load inside the LiM is significantly higher. In the specific case of the MVM, each LiM Cell is connected to the BL bus, but in other cases where the BL bus is also connected to the IRL blocks, the internal contribution can be much greater.

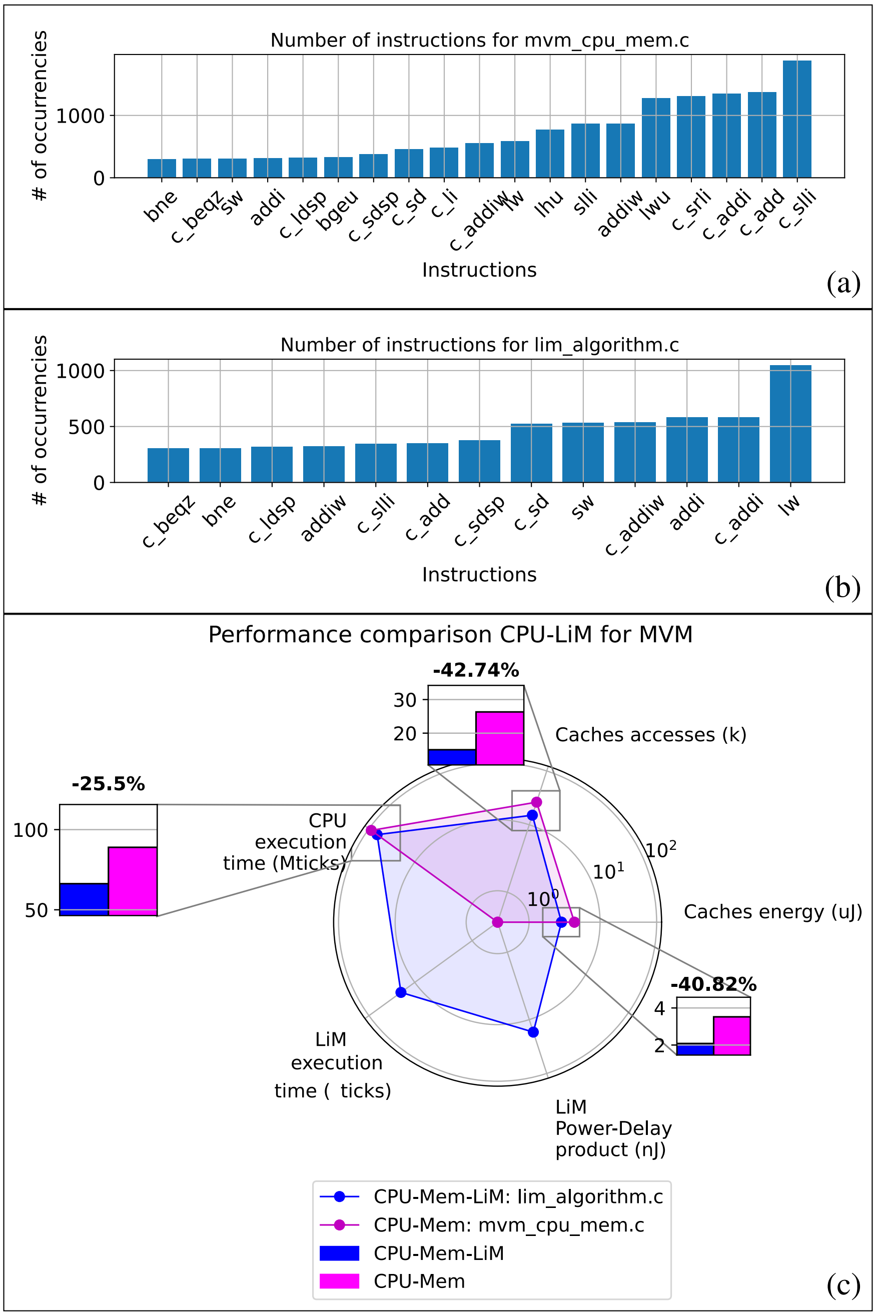


Fig. 10.9 Results of the MVM algorithm. The evaluation with back-annotation during the algorithm evaluation phase. (a-b) Instructions profiling for the MVM of the CPU-Mem and CPU-Mem-LiM architectures, respectively. (c) Comparison between CPU-Mem and CPU-Mem-LiM for the MVM (axes are in logarithmic scale).

- *K-NN*: the K-NN algorithm considers 512 pairs of 32-bit X,Y values. Performance is evaluated with a clock period of 6 ns. The LiM structure requires one IRL block per row, significantly impacting performance. The power value

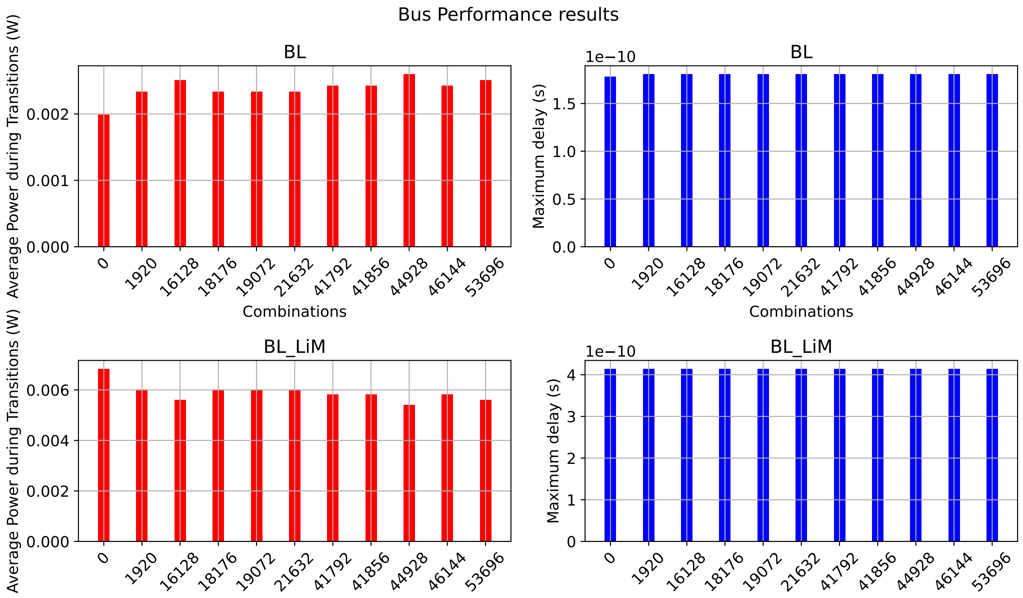


Fig. 10.10 BL Bus estimation of the MVM algorithm during data precharging.

obtained is very high, thus indicating the K-NN architecture as a disadvantageous solution in the LiM case. As can be seen from the Table 10.1, the K-NN implemented in LiM with the proposed solution leads to a worsening of the LiM Power-Delay product with respect to the other cases.

- *BMP*: the architecture implementing the BMP represents the smallest LiM solution compared to the others, as anticipated earlier. There are 16 entries of 8 bits each and, similarly to the Mean-Variance benchmark, the algorithm is executed in a pseudo-serial manner. The results are disadvantageous in the LiM case due to both the extremely small array size and serialization.

- *Mean-Variance*: the total number of samples N is equal to 1024, so the memory array has 1024 rows. The Mean-Variance algorithm is disadvantageous to be implemented in a LiM solution; in fact, the memory array runs in serial mode, requiring many steps to finish the computation. Specifically, considering 1024 elements, the total number of steps will be:

$$\begin{aligned}\#Steps_{Mean-Variance} &= \#Steps_{sum1} + \#Steps_{Mean} + \#Steps_{sum2,3} + \#Steps_{var} \\ &= 1024 + 1 + 1024 + 1 = 2050\end{aligned} \tag{10.2}$$

Despite this, the array's low power consumption leads to advantages from an energy perspective, demonstrating how LiM can still bring improvements as long as there are few hardware blocks within the array, keeping the power consumption low.

In almost all cases, the LiM approach demonstrates to be efficient in improving performance, bringing general system benefits. One of the most important figures of merit is the number of memory accesses: LiM paradigm reduces the memory accesses and consequently the energy in the analyzed cases, confirming the strong point of the Beyond von Neumann approach. The worst case is the BMP: since it is a very straightforward algorithm, the calculations exploited for the CPU-Mem case do not have a significant impact on the system performance, so delegating the computation to LiM does not bring a notable advantage compared to the other cases. Looking at the data, the following conclusions can be made:

- *The LiM concept is very beneficial, especially when used as an accelerator alongside the classical von Neumann system.*
- *LiM can bring advantages to the system as long as it can accelerate an application that contains a moderate amount of data.* In fact, LiM, if made too small, brings only disadvantages (BMP case) and is not worth using. Otherwise, the larger the LiM, the better the acceleration and execution time.
- *However, the complexity of the hardware placed inside the LiM must be taken into consideration.* Making a LiM too large with too many internal hardware blocks can negatively impact power, leading the LiM to be disadvantageous in terms of energy.

All of these considerations are critical and must be taken into account by the designer before starting the design flow of a LiM structure. With DExIMA software, it is possible to evaluate all these aspects quickly, automating the entire estimation and comparison process with von Neumann, thus avoiding design from scratch, lengthy and complex synthesis processes by-hand, and performance estimations.

10.7 DExIMA-Backend vs. Synopsys Design Compiler: complex architectures

In Fig. 10.11, comparisons between the worst-case results without back-annotation of DExIMA-Backend and Synopsys Design Compiler of XNOR-Net, MVM, and BMP architectures are proposed. These comparisons are very useful to understand how far the computational model implemented in DExIMA-Backend deviates for more complex architectures from those proposed in the validation part (chapter 9). It is possible to observe how, for all benchmarks, DExIMA-Backend overestimates performance, providing results up to 6.9 times worse than Synopsys in the dynamic power case. As explained earlier, this trend can be attributed to the different capacitance models (discussed in subsection 7.3.4), the approximation of the short-circuit power calculation (subsection 7.3.2), and the estimation of node switching activities. In DExIMA-Backend, the switching activity of the nodes is fixed at the toggle rate value set by the user (in these cases, equal to 0.5), while Synopsys Design Compiler can propagate the switching activity within the design, providing more accurate and faithful indications. The critical paths also differ from the reference, being an approximate model that also strongly depends on the capacitances of the design nodes. Static power values, on the other hand, are very similar to those provided by Synopsys Design Compiler, confirming the goodness of the models implemented within the Backend. These results, however, should not be seen as a weakness of the framework, as they can still provide worst-case estimates of LiM architectures that are then used to evaluate LiM in von Neumann systems. Based on these estimates, the user can do a full architectural exploration before moving on to the chip synthesis, Place&Route, and fabrication phase.

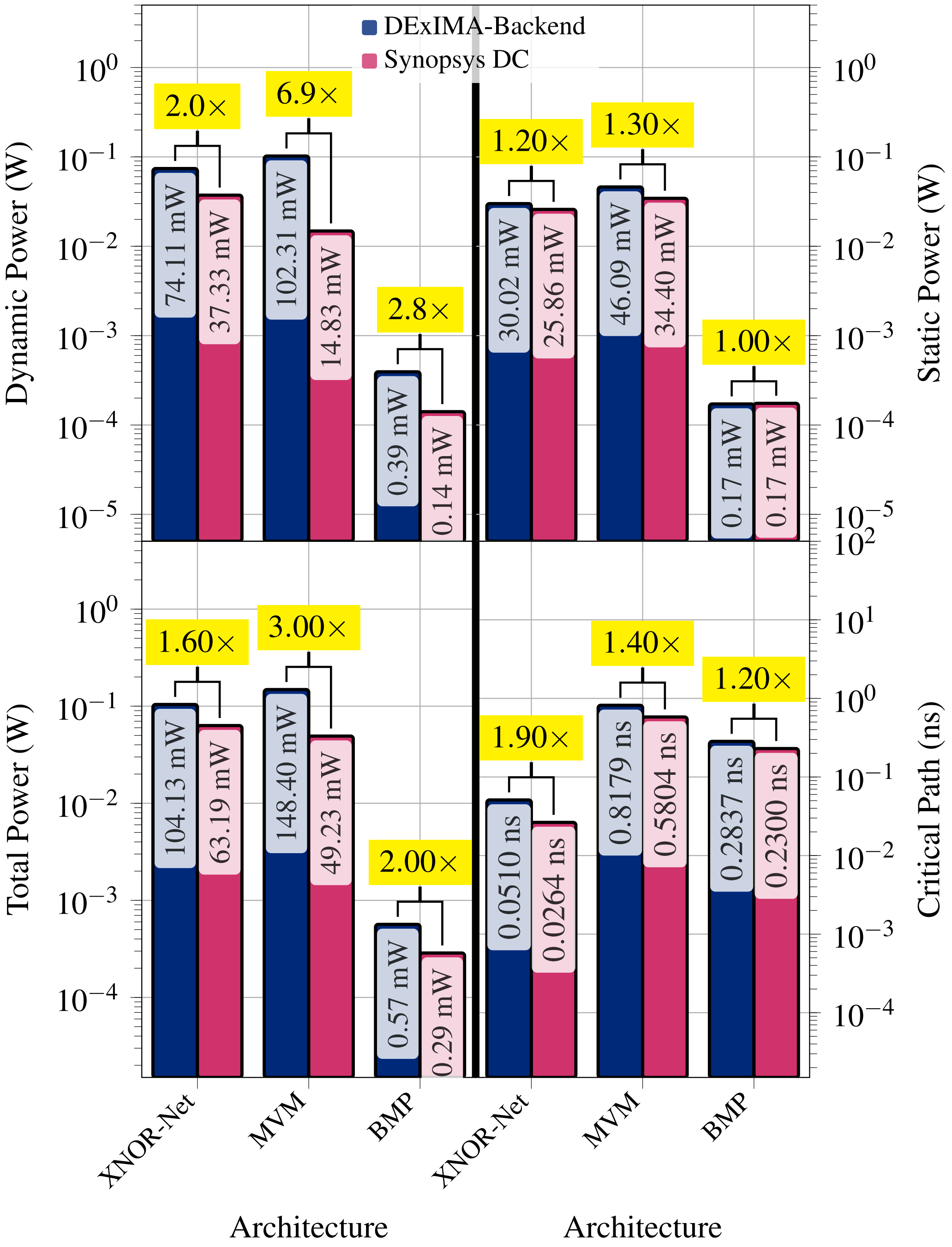


Fig. 10.11 Performance comparison between Synopsys Design Compiler and DExIMA-Backend for MVM, XNOR-Net and BMP.

10.7.1 Investigating the worst outcomes: MVM case

To give a more accurate explanation of the result obtained in the MVM case, comparisons are made between the performance calculated by DExIMA-Backend and Synopsys Design Compiler for MVM architectures with different parallelisms (4 bits to 32 bits) and different memory array depths (4 to 256 rows). The results of these analyses are the ratios between the parameters obtained with DExIMA-Backend and Synopsys Design Compiler, which are shown in Fig. 10.12. As the number of rows in the array increases, the three studied ratios of Dynamic Power, Total Power and Critical Path remain almost constant, while varying more as parallelism changes. These ratios vary in the ranges of (4.3×, 7.6×), (2.52×, 3.24×) and (1.39×,1.54×), respectively. The fact that the power and timing results do not change much as the number of rows varies is a decidedly positive result, because it confirms the goodness of DExIMA-Backend in modeling these architectures, scaling performance correctly by size. In the case of parallelism, on the other hand, there is a confirmation of what was anticipated earlier, i.e., DExIMA-Backend makes errors in modeling the individual hardware blocks for the propagation of switching activities and for the approximations made in the implemented computational model. Further analysis was performed on the switching activity, showing that Synopsys Design Compiler makes power estimations with different switching activity values that, unlike DExIMA-Backend, are not fixed but are obtained by propagating them within the design. Taking the MVM with 4 rows with 4-bit parallelism and the adder within the fourth IRL as an example, via the command `get_switching_activity {*}`, Synopsys Design Compiler reports the switching activity values of the adder's inputs and outputs and, by issuing a `report_power`, the associated power is provided. This value is compared with DExIMA-Backend and is reported in Table 10.3. The same adder is re-analyzed with Synopsys Design Compiler, but this time annotating by hand the value of the toggle rate, set equal to 0.5 for each pin of the design through the command `set_switching_activity -toggle_rate 0.5 -static_probability 0.5 -base_clock MYclk {*}`. At this point, the power estimation, shown in Table 10.3 under the heading "With forced activity," is carried out again. As expected, the power value estimated by Synopsys Design Compiler increases, thus reducing the difference between the two tools. It is worth noting that, again, switching activity is fixed on the input and output pins, but the internal adder pins (e.g., the carry chain) have switching activities estimated through propagation. Another interesting result is the Execution Time speed-up between the two tools, i.e., DExIMA-Backend and Synopsys Design Compiler required time to provide the estimations. This analysis was already proposed in Table 9.2, but it can be useful to evaluate this fundamental parameter for more complex architectures. The speed-up gives another clear indication of DExIMA-Backend strenghts compared to classical synthesizers, in fact Fig. 10.13 shows the results for the MVM architecture at varying parallelism and number of rows. It is possible to observe that the speed-up varies from a maximum of about 800 times (for smaller architectures) to a minimum of about 37 times (for more complex architectures, thus 16 bit MVM with 256 rows), revealing the efficiency of the Backend in providing close-to-reality estimations in a very short time. In the specific case of an array with maximum size (32bit × 256 rows), the Backend takes about 60 seconds, while Synopsys Design Compiler takes about 120 minutes.

Table 10.3 Toggle rate impact evaluation in terms of dynamic and total powers. Comparison between Synopsys Design Compiler and DExIMA-Backend.

Design	Pin	Toggle Rate		Dynamic Power (μW)		Ratio	Total Power (μW)		Ratio
		Synopsys DC	DExIMA-Backend	Synopsys DC	DExIMA-Backend		Synopsys DC	DExIMA-Backend	
/intrarow_logic_3/ADDER_16/	A[3]	0.009608	0.5	3.49	6.63	1.90	5.43	9.34	1.72
	A[2]	0.008581							
	A[1]	0.006866							
	A[0]	0.004618							
	B[3]	0.000509							
	B[2]	0.000488							
	B[1]	0.000529							
	B[0]	0.000376							
	AS	0.083333							
	SUM[3]	0.009984							
	SUM[2]	0.008957							
	SUM[1]	0.00706							
	SUM[0]	0.004812							
With forced activity	All	0.5	0.5	5.82	6.63	1.14	7.81	9.34	1.20

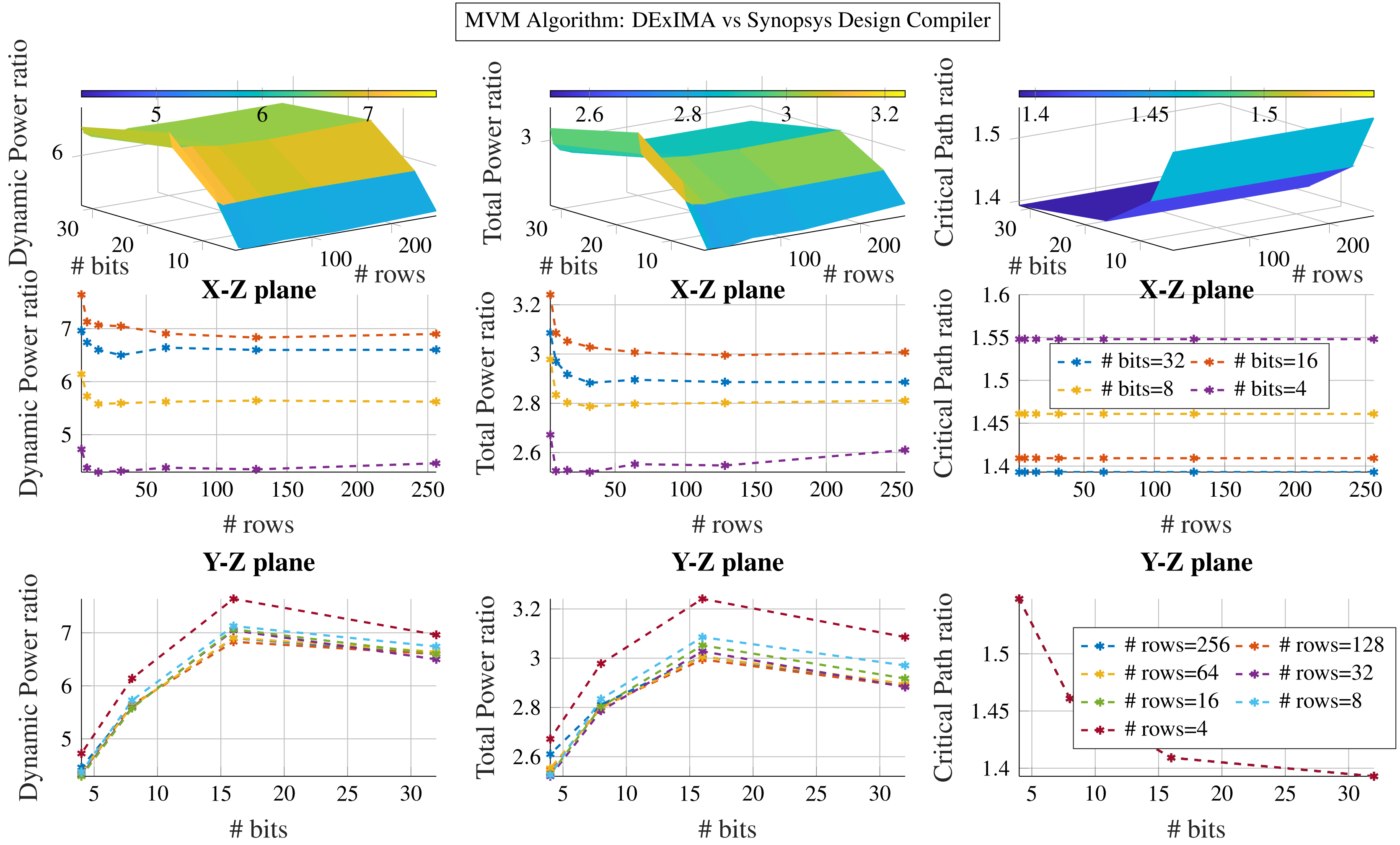


Fig. 10.12 Comparison between DExIMA-Backend and Synopsys Design Compiler performance results. The parallelism and the number of rows of the LiM array are swept.

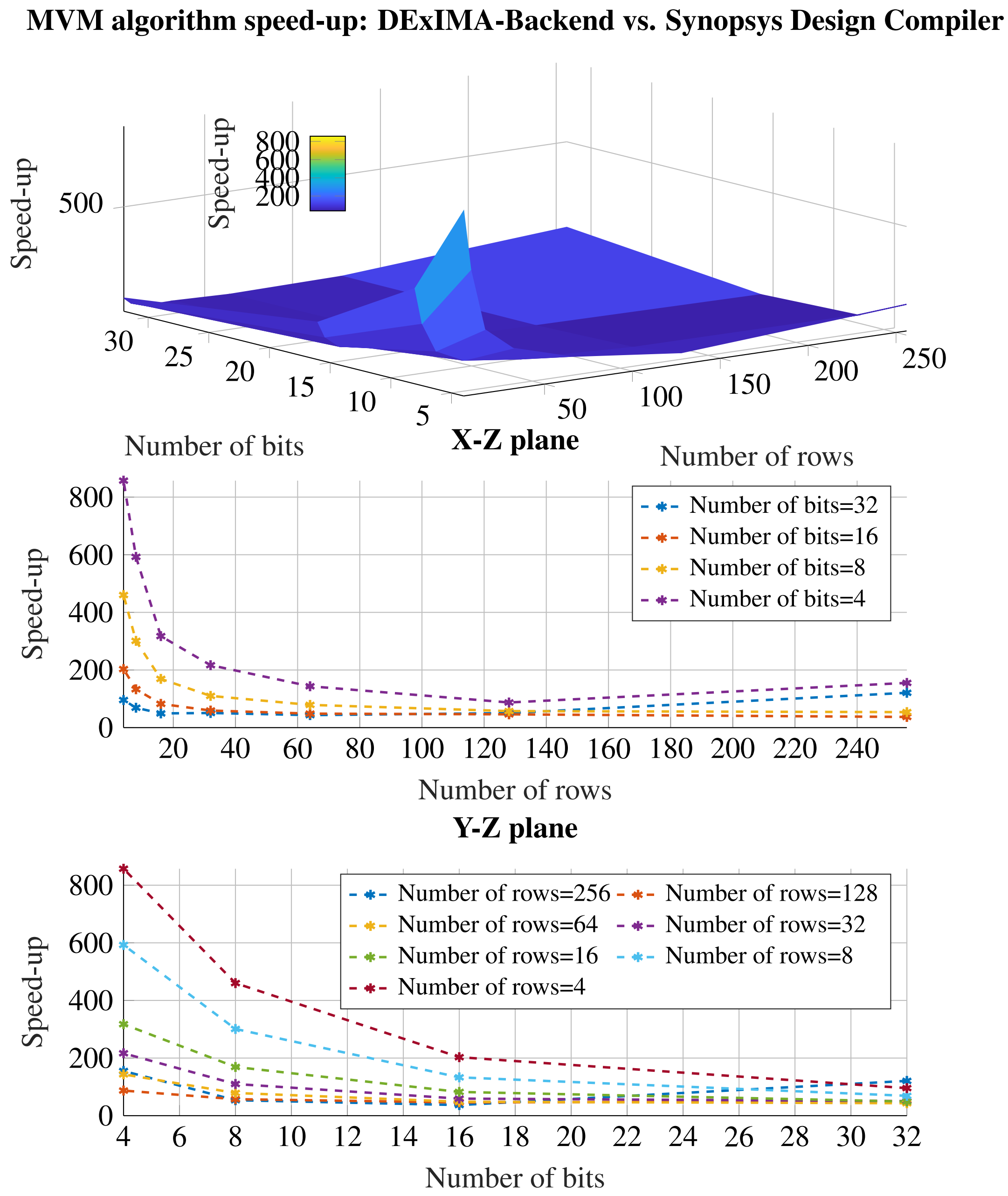


Fig. 10.13 DExIMA-Backend vs Synopsys Design Compiler Execution Time speed-up ratio.

10.7.2 Investigating even more the problem: DExIMA-Backend known issues

The comparisons conducted so far on the MVM and other architectures are still insufficient to provide quantitative proof of the cause of the errors on DExIMA-

Backend's estimations. For this reason, two examples are proposed that give a better idea of the explanations given above. The two examples deal with two architectures consisting of only 16-bit adders: the first has several adders connected in cascade, while the other has only one adder whose output is connected to multiple adders in parallel. The number of adders in both cases is varied, observing the effects on the parameters of interest. Fig. 10.15 shows comparisons in terms of Dynamic, Total Powers and Critical Path delay between DExIMA-Backend and Synopsys Design Compiler. The two architectures studied allow more accurate analysis of timing (the serial architecture) and dynamic power (the parallel architecture). In fact, in the case of the serial architecture, the worst result is obtained in the Critical Path delay case, because the timing model implemented in DExIMA-Backend differs from that of Synopsys Design Compiler: when a multibit block is instantiated on DExIMA-Backend, the timing is calculated as the Critical Path of the block itself, so when more are connected in succession (as is the case in the serial architecture),

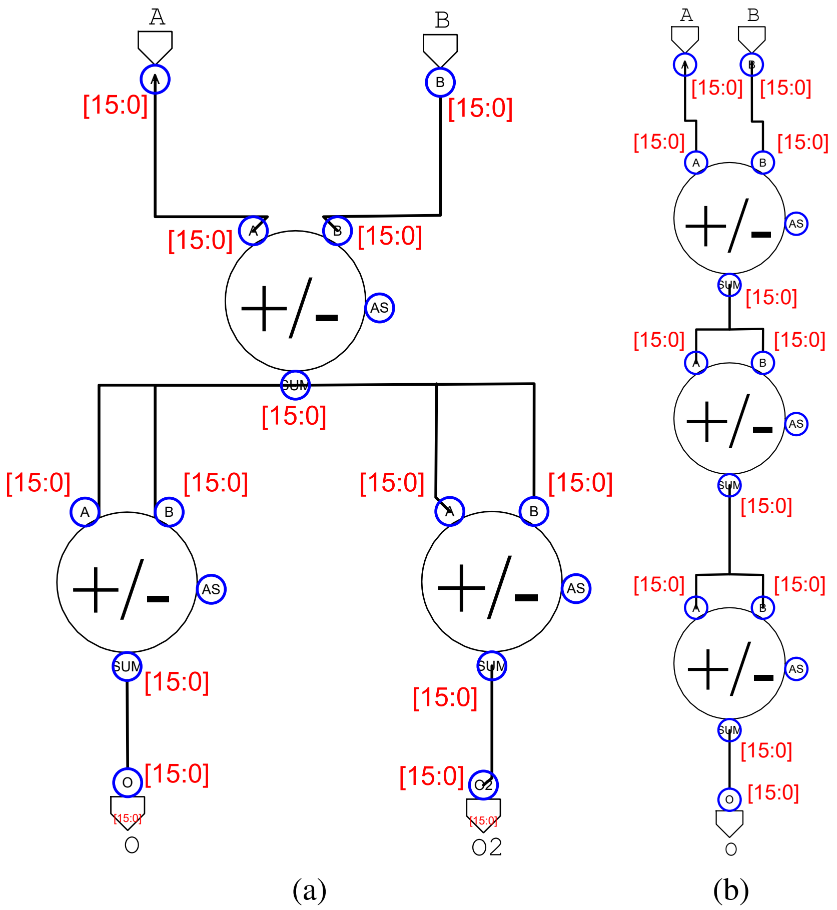
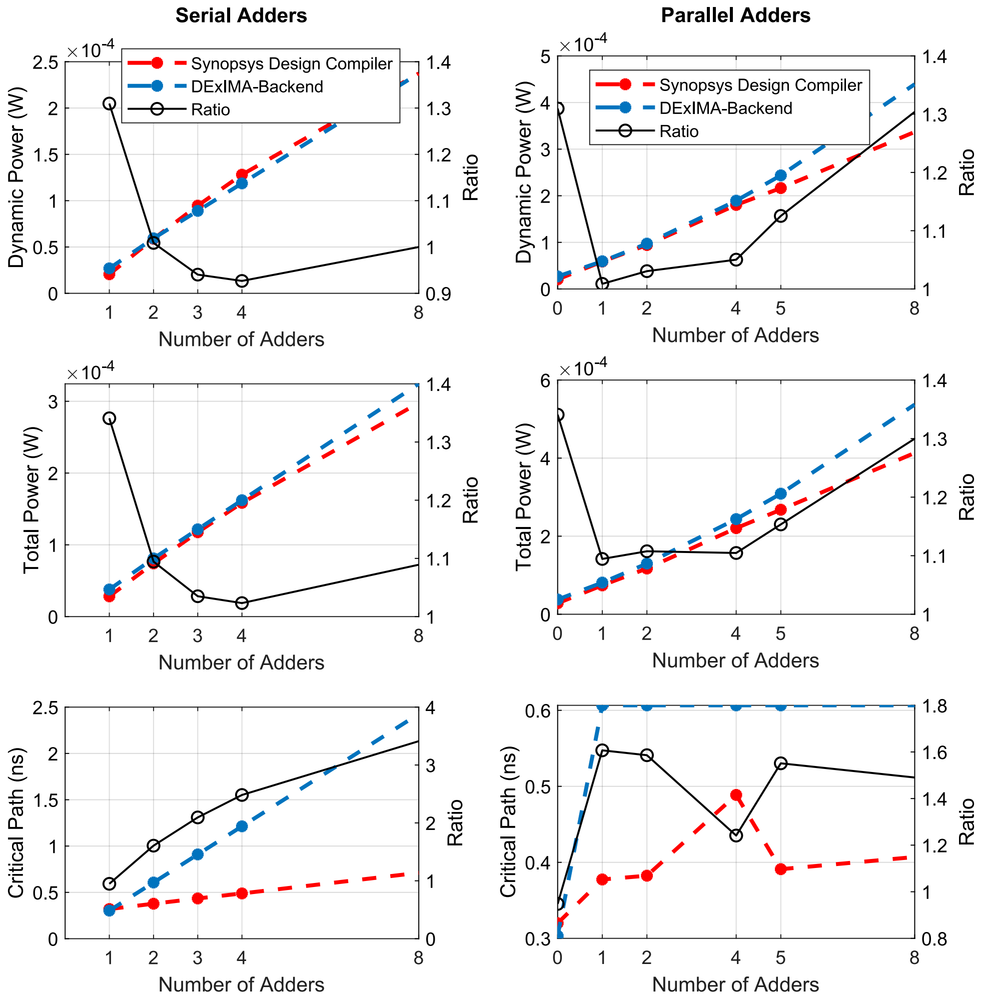


Fig. 10.14 (a) Two parallel adders circuit example. (b) Three serial adders circuit example.

the total timing will be $N \times Delay_{Adder}$, where N is the number of adders. In this case, however, this extremely worst-case estimation turns out to be different from reality because the Critical Path delay of the serial architecture is equal to $Delay_{Adder} + (N-1) \times (Delay_{FullAdder} + Delay_{XNOR})$, which is correctly calculated by Synopsys Design Compiler. Regarding dynamic power, the parallel architecture confirms the hypothesis of the differences between the capacity and switching activity models implemented in the Backend. As can be seen, the dynamic power ratio increases approximately linearly in the parallel case, but remains approximately constant in the serial case.



(a) (b)

Fig. 10.15 Dynamic, Total Powers and Critical Path comparisons between DExIMA-Backend and Synopsys Design Compiler. (a) Serial Adders analysis. (b) Parallel Adders analysis.

10.8 SRAM vs flip-flop memories

All the work presented so far is based on LiM arrays with a flip-flop as memory element, which is a good starting model for theorizing and developing In-Memory architectures. However, one might wonder about the impact of a more real memory array, so for example an SRAM-type array. As anticipated in the state-of-the-art of this thesis, In-Memory computation of SRAM-type arrays is decidedly different from that proposed in DExIMA: most SRAM approaches consist of analog operations with modified versions of sense amplifiers placed at the bottom of the memory array. Unfortunately, until now, DExIMA is not able to implement this type of array with associated computation. DExIMA does, however, offer the possibility of evaluating what the impact of an SRAM-based array would be by choosing an SRAM cell instead of a flip-flop as the memory element. By choosing the SRAM cell, DExIMA will build an array of SRAM cells exactly as is done with flip-flop arrays, and at this point, the user can make performance comparisons. The first proposed comparison is of area and power of two 16 kB arrays, one SRAM-based and one flip-flop-based, respectively. As can be seen from Table 10.4, both area and power are greatly reduced with an SRAM-type cell because the number of transistors of an SRAM cell is significantly less than that of a flip-flop. Integrating LiM logic within an SRAM array would therefore bring significant advantages, reducing the LiM impact more, as well as being a significantly more scalable array than a Register File. To follow a similar approach to what has been developed in this thesis, differential type logic could be placed within the SRAM array, doing something similar to the work presented in [154], thus introducing differential logic (also called DCVSL [155]) within DExIMA. It should be kept in mind that DCVSL-type logic requires doubling the pull-down network, thus having twice the number of transistors with respect to a static-CMOS logic. This could therefore negatively impact the performance of SRAM-based arrays, favoring those based on flip-flops. However, this study is beyond the scope of this thesis.

Table 10.4 SRAM-based and flip-flop based memory arrays performance comparisons.

Structure	Size (kB)	Area (mm^2)		Total Power (W)		Clock period (ns)	Toggle Rate
Flip-flop-based	16	0.878	-84.52%	0.313	-87.07%	6	0.5
SRAM-based		0.136		0.041			

The second comparison is made on the impact of internal bus performance on the two types of memories during a write operation. Results are shown in Table 10.5. As expected, the impact of the bus is also less significant in an SRAM-based memory. This can be justified by the lower capacitive load, which reduces both the power and the bus delay.

Table 10.5 BL bus performance comparison between SRAM-based and flip-flop-based memories.

Structure	Size (kB)	Average Power (mW)		Max delay (ps)		Bus name
Flip-flop-based	4	26.45	-70.84%	824.40	-70.01%	BL
SRAM-based		7.80		247.18		

10.9 Evaluating the impact of the bus on CPU-Mem and CPU-Mem-LiM

The benchmark results presented in the section 10.6 do not directly consider the effect the bus has on performance. The impact of the bus, as discussed earlier, can be estimated on the LiM architectures implemented on DExIMA, but unfortunately the same estimation cannot be made on a classical memory hierarchy in the CPU-Mem system, since Gem5 and Cacti do not implement a bus estimation like the one implemented in DExIMA. However, it is possible to make a rough estimation of bus performance in the CPU-Mem case as well, taking advantage of the results obtained in section 10.8 regarding the 4kB SRAM array. In this estimation, only L1-type caches, both data and instruction, are considered. The process of estimating CPU-Mem and CPU-Mem-LiM architectures' parameters with Gem5 and Cacti is performed by imposing an L1 cache size of 4kB, so as to have as close a resemblance as possible to the model implemented in DExIMA. At this point, the numbers of L1 data and instruction memory accesses are used to compute the total bus energy, obtained as:

$$\text{Total Bus Energy} = \text{Average Bus Power} \times T_{ck} \times \text{Number}_{\text{Accesses}} \tag{10.3}$$

Where T_{ck} is the clock period. The quantity Average Bus Power × T_{ck} defines the Energy/Access of the bus, since it represents the energy of a single access during a clock cycle. The Table 10.6 shows the results of the bus evaluated for the proposed

Table 10.6 Bus impact results and comparisons.

Architecture		**Memory**	**Size (kB)**	**Overall accesses**	**Clock Period (ns)**	**Bus Energy per Access (pJ)**	**Bus Energy (μJ)**	**Total (μJ)**	**Improvement (%)**
MVM	**CPU-Mem**	L1D	4	5036	1	7.80	39.28	202.43	37.72%
		L1I		20916			163.14		
	CPU-Mem-LiM	L1D	4	3475	1		27.11	126.07	
		L1I		11231			87.60		
		LiM	0.5	272	6	41.79	11.37		
XNOR-Net	**CPU-Mem**	L1D	4	100694	1	7.80	785.41	4103.70	91.81%
		L1I		425422			3318.29		
	CPU-Mem-LiM	L1D	4	8467	1		66.04	336.03	
		L1I		28912			225.51		
		LiM	4	272	6	163.52	44.48		
VAR	**CPU-Mem**	L1D	4	25887	1	7.80	201.92	1200.54	35.08%
		L1I		128029			998.63		
	CPU-Mem-LiM	L1D	4	8467	1		66.04	779.43	
		L1I		28912			225.51		
		LiM	4	3074	6	158.71	487.88		
BMP	**CPU-Mem**	L1D	4	1910	1	7.80	14.90	68.66	-0.59%
		L1I		6892			53.76		
	CPU-Mem-LiM	L1D	4	1913	1		14.92	69.06	
		L1I		6879			53.66		
		LiM	0.01563	57	6	8.52	0.49		

algorithms. The clock periods are equal to 1 ns in the case of the caches and 6 ns in the LiM case, in fact the clock frequency of the caches is assumed to be equal to that of the processor. As can be seen, since the cache memory accesses are significantly lower in the CPU-Mem-LiM in almost all cases, the bus energies are correspondingly lower. It is interesting to note that energies per access in the LiM cases are much larger, despite the fact that the memory sizes are smaller or equal to the L1 caches. These effects are due to the complexities of the LiM arrays compared to SRAMs and the higher clock periods. The BMP results are worse in the LiM case, since as discussed above, the proposed LiM implementation does not bring advantages over the classical case, requiring a similar number of memory accesses.

10.10 Conclusions

In this chapter, several benchmarks are proposed and implemented with DExIMA, applying all the procedures seen so far and providing architectures' performance and bus impact results and comparisons with the reference von Neumann system. The data obtained confirm the validity of the LiM concept as well as the flexibility of the tool in implementing very different algorithms by providing the user with guided support at every stage of the design. In addition, a comparison with the commercial Synopsys Design Compiler tool is again proposed to evaluate the difference in performance estimations provided by DExIMA-Backend for large and complex architectures. In addition, the execution times of the two tools for different LiM array sizes are evaluated. The speed-ups achieved by DExIMA-Backend for the MVM algorithm are reported, which is at least 37 times higher than Synopsys Design Compiler. Finally, evaluations on more standard memory arrays (based on SRAM cells) are proposed, and the limitations of the Backend are reported, which will be addressed as future work.

Chapter 11

Conclusions and future works

This thesis work proposes DExIMA, a tool dedicated to the architectural exploration of BvNC architectures. BvNC, particularly LiM architectures, are extremely versatile and, as widely demonstrated in the literature, effectively reduce the bottleneck of von Neumann structures. The design of LiMs can be based on different technological solutions, including SRAMs, DRAMs, or resistive elements, each implemented following completely different approaches. Some are based on analog operations, while others rely on the use of layers of logic interconnected with the memory array. From this, one can observe how heterogeneous the BvNC approach is, with no clear design flow: for each architectural solution, the designer needs to implement the structure from scratch. With the aim of trying to ease the procedure as much as possible, tools have been proposed in the literature that can provide estimations or simulate this new architectural approach, trying to standardize the methodology. These tools generally specialize in one or more types of BvNCs, some of them based on existing architectural templates. With DExIMA, the idea is to focus on LiMs and to define the whole architecture with high flexibility, going through the whole design flow up to performance estimation and comparison with von Neumann architectures. In fact, DExIMA maintains all architectural-level descriptions, so estimations can be done on any technology if implemented within DExIMA-Backend. The different benchmarks proposed show how, with DExIMA, the user has the possibility to control every step of the design with simplicity, greatly speeding up the whole procedure.

DExIMA, however, has some limitations that necessarily need to be addressed in the future. One mentioned earlier is the lack of support for technologies other than

CMOS and alternative structures from flip-flop-based arrays. Similarly, the CMOS estimation models implemented in DExIMA-Backend, as extensively discussed above, require corrections in order to provide estimations as close as possible to the ones made by commercial and industrial-use tools (e.g., Synopsys Design Compiler). Despite this, DExIMA-Backend has proven to be a good tool for making preliminary design estimations that indicate to the user how efficient a LiM architecture is in implementing a given algorithm. In addition to this, DExIMA-Backend is extremely faster than Synopsys Design Compiler, since it does not perform a real synthesis phase. The idea of DExIMA-Backend, in addition to guaranteeing the future possibility of implementing new technologies, is to give the user the opportunity to optimize the design as much as possible, and then move on to the actual implementation of the chip through synthesis and Place&Route phases with classic EDA tools.

In addition, DExIMA's interface requires enhancement to make the experience with the tool even more user-friendly, as well as enhancement of the interconnect capabilities within the array. A missing part also is the interface with the Octantis synthesis tool [131], developed at the Polytechnic of Turin, which is specific for LiM architectures and capable of providing an architectural implementation starting from an algorithm described in C.

Moreover, comparisons with standard architectures may in the future be extended to Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs), since LiM has a distinctly similar structure to these categories as massively parallel computing units. Therefore, the "CPU-Memory Comparison" section of DExIMA will have to be extended, integrating architectural modeling tools similar to Gem5, but reproducing GPUs and TPUs.

Currently, DExIMA is under development, with the idea of integrating all the shortcomings mentioned above.

References

[1] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2017.

[2] J Giceva. Lecture notes for data processing on modern hardware–lecture 3: Cache awareness for query execution models, 2020.

[3] Maha Kooli, Henri-Pierre Charles, Clément Touzet, Bastien Giraud, and Jean-Philippe Noël. Software platform dedicated for in-memory computing circuit evaluation. In *2017 International Symposium on Rapid System Prototyping (RSP)*, pages 43–49. IEEE, 2017.

[4] Jaeheum Lee, Jason K Eshraghian, Kyoungrok Cho, and Kamran Eshraghian. Adaptive precision cnn accelerator using radix-x parallel connected memristor crossbars. *arXiv preprint arXiv:1906.09395*, 2019.

[5] Mustafa F Ali, Akhilesh Jaiswal, and Kaushik Roy. In-memory low-cost bit-serial addition using commodity dram technology. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(1):155–165, 2019.

[6] Yann LeCun et al. Lenet-5, convolutional neural networks. *URL: http://yann. lecun. com/exdb/lenet*, 20:5, 2015.

[7] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European conference on computer vision*, pages 525–542. Springer, 2016.

[8] Di Gao, Dayane Reis, Xiaobo Sharon Hu, and Cheng Zhuo. Eva-cim: A system-level performance and energy evaluation framework for computing-in-memory architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):5011–5024, 2020. doi: 10.1109/TCAD.2020.2966484.

[9] Antonia Ieva. Speed-up of risc-v core using logic-in-memory operations, 2020.

[10] Andrea Coluccio, Antonia Ieva, Fabrizio Riente, Massimo Ruo Roch, Marco Ottavi, and Marco Vacca. Risc-vlim, a risc-v framework for logic-in-memory architectures. *Electronics*, 11(19):2990, 2022.

[11] Simone Domenico Antonietta. Weights in-memory neural network embedded ram, 2019.

[12] Simone Domenico Antonietta, Andrea Coluccio, Giovanna Turvani, Marco Vacca, Mariagrazia Graziano, and Maurizio Zamboni. Winner: a high speed high energy efficient neural network implementation for image classification. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 29–32. IEEE, 2019.

[13] Andrea Coluccio, Marco Vacca, and Giovanna Turvani. Logic-in-memory computation: Is it worth it? a binary neural network case study. *Journal of Low Power Electronics and Applications*, 10(1):7, 2020.

[14] L. Jiang, M. Kim, W. Wen, and D. Wang. Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, July 2017. doi: 10.1109/ISLPED.2017.8009163.

[15] Angela Guastamacchia. Gp-lima: A general purpose architectural model leveraging the logic-in-memory approach, 2021.

[16] Angela Guastamacchia, Andrea Coluccio, Marco Vacca, Fabrizio Riente, Mariagrazia Graziano, and Maurizio Zamboni. Mempa: a memory mapped m-simd co-processor to cope with the memory-wall issue. *Submitted to ACM Transactions on Computer Systems*, 2023.

[17] Spectre circuit simulator components and device models reference, 2020.

[18] Bruno E Forlin, Paulo C Santos, Augusto E Becker, Marco AZ Alves, and Luigi Carro. Sim2pim: A complete simulation framework for processing-in-memory. *Journal of Systems Architecture*, 128:102528, 2022.

[19] D. Fan and S. Angizi. Energy efficient in-memory binary deep neural network accelerator with dual-mode sot-mram. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 609–612, 2017.

[20] Woong Choi, Kwanghyo Jeong, Kyungrak Choi, Kyeongho Lee, and Jongsun Park. Content addressable memory based binarized neural network accelerator using time-domain signal processing. In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, pages 138:1–138:6, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5700-5. doi: 10.1145/3195970.3196014. URL http://doi.acm.org/10.1145/3195970.3196014.

[21] Jesper Knudsen. Nangate 45nm open cell library. *CDNLive, EMEA*, 2008.

[22] Richard T. Kouzes, Gordon A. Anderson, Stephen T. Elbert, Ian Gorton, and Deborah K. Gracio. The changing paradigm of data-intensive computing. *Computer*, 42(1):26–34, 2009. doi: 10.1109/MC.2009.26.

[23] Etienne Sicard and Alexandre Boyer. Impact of technological trends and electromagnetic compatibility of integrated circuits. In *EMC Compo 2019*, 2019.

[24] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017.

[25] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[26] International technology roadmap for semiconductors. http://www.itrs2.net/, 2022.

[27] John P Holdren and Shaun Donovan. National strategic computing initiative strategic plan. Technical report, National Strategic Computing Initiative Executive Council Washington United . . . , 2016.

[28] II Arikpo, FU Ogban, and IE Eteng. Von neumann architecture and modern computers. *Global Journal of Mathematical Sciences*, 6(2):97–103, 2007.

[29] Rudolf Eigenmann and David J Lilja. Von neumann computers. *Wiley Encyclopedia of Electrical and Electronics Engineering*, 23:387–400, 1998.

[30] John Von Neumann and Ray Kurzweil. *The computer and the brain*. Yale University Press, 2012.

[31] Wm A Wulf and Sally A McKee. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[32] Maurice V Wilkes. The memory wall and the cmos end-point. *ACM SIGARCH Computer Architecture News*, 23(4):4–6, 1995.

[33] Ashley Saulsbury, Fong Pong, and Andreas Nowatzyk. Missing the memory wall: The case for processor/memory integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, page 90–101, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917863. doi: 10.1145/232973.232984. URL https://doi.org/10.1145/232973.232984.

[34] Xingqi Zou, Sheng Xu, Xiaoming Chen, Liang Yan, and Yinhe Han. Breaking the von neumann bottleneck: architecture-level processing-in-memory technology. *Science China Information Sciences*, 64(6):1–10, 2021.

[35] Adrian Cristal, Daniel Ortega, Josep Llosa, and Mateo Valero. Out-of-order commit processors. In *10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 48–59. IEEE, 2004.

[36] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. Memory devices and applications for in-memory computing. *Nature nanotechnology*, 15(7):529–544, 2020.

[37] Kaya Can Akyel, Henri-Pierre Charles, Julien Mottin, Bastien Giraud, Grégory Suraci, Sébastien Thuries, and Jean-Philippe Noel. Drc 2: Dynamically reconfigurable computing circuit based on memory architecture. In *2016 IEEE International Conference on Rebooting Computing (ICRC)*, pages 1–8. IEEE, 2016.

[38] Zhiting Lin, Honglan Zhan, Xuan Li, Chunyu Peng, Wenjuan Lu, Xiulong Wu, and Junning Chen. In-memory computing with double word lines and three read ports for four operands. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 28(5):1316–1320, 2020.

[39] Hongyang Jia, Hossein Valavi, Yinqi Tang, Jintao Zhang, and Naveen Verma. A programmable heterogeneous microprocessor based on bit-scalable in-memory computing. *IEEE Journal of Solid-State Circuits*, 55(9):2609–2621, 2020. doi: 10.1109/JSSC.2020.2987714.

[40] Shihui Yin, Zhewei Jiang, Jae-Sun Seo, and Mingoo Seok. Xnor-sram: In-memory computing sram macro for binary/ternary deep neural networks. *IEEE Journal of Solid-State Circuits*, 55(6):1733–1743, 2020.

[41] Amogh Agrawal, Akhilesh Jaiswal, Chankyu Lee, and Kaushik Roy. X-sram: Enabling in-memory boolean computations in cmos static random access memories. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 65 (12):4219–4232, 2018.

[42] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 273–287. IEEE, 2017.

[43] Akhilesh Jaiswal, Indranil Chakraborty, Amogh Agrawal, and Kaushik Roy. 8t sram cell as a multibit dot-product engine for beyond von neumann computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(11): 2556–2567, 2019.

[44] William Andrew Simon, Yasir Mahmood Qureshi, Marco Rios, Alexandre Levisse, Marina Zapater, and David Atienza. An in-cache computing architecture for edge devices. *IEEE Transactions on Computers*, 2020.

[45] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. A 28 nm configurable memory (tcam/bcam/sram) using push-rule 6t bit cell enabling logic-in-memory. *IEEE Journal of Solid-State Circuits*, 51 (4):1009–1021, 2016.

[46] Dayane Reis, Michael T Niemier, and Xiaobo Sharon Hu. A computing-in-memory engine for searching on homomorphically encrypted data. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 5(2): 123–131, 2019.

[47] Leonid Yavits, Amir Morad, and Ran Ginosar. Computer architecture with associative processor replacing last-level cache and simd accelerator. *IEEE Transactions on Computers*, 64(2):368–381, 2013.

[48] Woong Choi, Kwanghyo Jeong, Kyungrak Choi, Kyeongho Lee, and Jongsun Park. Content addressable memory based binarized neural network accelerator using time-domain signal processing. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[49] Jae Seong Lee, Jisoo Yoon, and Woo Young Choi. In-memory nearest neighbor search with nanoelectromechanical ternary content-addressable memory. *IEEE Electron Device Letters*, 43(1):154–157, 2021.

[50] Mark Durlam, P Naji, M DeHerrera, S Tehrani, G Kerszykowski, and K Kyler. Nonvolatile ram based on magnetic tunnel junction elements. In *2000 IEEE International Solid-State Circuits Conference. Digest of Technical Papers (Cat. No. 00CH37056)*, pages 130–131. IEEE, 2000.

[51] Adnan Siraj Rakin, Shaahin Angizi, Zhezhi He, and Deliang Fan. Pim-tgan: A processing-in-memory accelerator for ternary generative adversarial networks. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*, pages 266–273. IEEE, 2018.

[52] Arman Roohi, Shaahin Angizi, Deliang Fan, and Ronald F DeMara. Processing-in-memory acceleration of convolutional neural networks for energy-effciency, and power-intermittency resilience. In *20th International Symposium on Quality Electronic Design (ISQED)*, pages 8–13. IEEE, 2019.

[53] Deliang Fan and Shaahin Angizi. Energy efficient in-memory binary deep neural network accelerator with dual-mode sot-mram. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 609–612. IEEE, 2017.

[54] Hong Wang and Xiaobing Yan. Overview of resistive random access memory (rram): Materials, filament mechanisms, performance optimization, and prospects. *physica status solidi (RRL)–Rapid Research Letters*, 13(9):1900073, 2019.

[55] Shahar Kvatinsky, Dmitry Belousov, Slavik Liman, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. Magic—memristor-aided logic. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 61 (11):895–899, 2014.

[56] Olga Krestinskaya and Alex Pappachen James. Binary weighted memristive analog deep neural network for near-sensor edge processing. In *2018 IEEE 18th International Conference on Nanotechnology (IEEE-NANO)*, pages 1–4. IEEE, 2018.

[57] Xinxin Wang, Mohammed A. Zidan, and Wei D. Lu. A crossbar-based in-memory computing architecture. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 67(12):4224–4232, 2020. doi: 10.1109/TCSI.2020.3000468.

[58] I Giannopoulos, A Sebastian, M Le Gallo, VP Jonnalagadda, M Sousa, MN Boon, and E Eleftheriou. 8-bit precision in-memory multiplication with projected phase-change memory. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 27–7. IEEE, 2018.

[59] Geethan Karunaratne, Manuel Le Gallo, Giovanni Cherubini, Luca Benini, Abbas Rahimi, and Abu Sebastian. In-memory hyperdimensional computing. *Nature Electronics*, 3(6):327–337, 2020.

[60] Berkin Akin, Franz Franchetti, and James C Hoe. Data reorganization in memory using 3d-stacked dram. *ACM SIGARCH Computer Architecture News*, 43(3S):131–143, 2015.

[61] Amir Morad, Leonid Yavits, and Ran Ginosar. Gp-simd processing-in-memory. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, 2015.

[62] Suhail Basalama, Atiyehsadat Panahi, Ange-Thierry Ishimwe, and David Andrews. Spar-2: A simd processor array for machine learning in iot devices. In *2020 3rd International Conference on Data Intelligence and Security (ICDIS)*, pages 141–147, 2020. doi: 10.1109/ICDIS50059.2020.00025.

[63] Giulia Santoro, Giovanna Turvani, and Mariagrazia Graziano. New logic-in-memory paradigms: An architectural and technological perspective. *Micromachines*, 10(6):368, 2019.

[64] B. Akin, F. Franchetti, and J. C. Hoe. Data reorganization in memory using 3d-stacked dram. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 131–143, June 2015. doi: 10.1145/2749469.2750397.

[65] Fabio Lorenzo Traversa and Massimiliano Di Ventra. Universal memcomputing machines. *IEEE transactions on neural networks and learning systems*, 26(11):2702–2715, 2015.

[66] Yuriy V Pershin and Massimiliano Di Ventra. Memcomputing implementation of ant colony optimization. *Neural Processing Letters*, 44:265–277, 2016.

[67] Fabio L Traversa, Fabrizio Bonani, Yuriy V Pershin, and Massimiliano Di Ventra. Dynamic computing random access memory. *Nanotechnology*, 25(28): 285201, 2014.

[68] Fabio Lorenzo Traversa, Chiara Ramella, Fabrizio Bonani, and Massimiliano Di Ventra. Memcomputing np-complete problems in polynomial time using polynomial resources and collective states. *Science advances*, 1(6):e1500031, 2015.

[69] Massimiliano Di Ventra. Memcomputing: When memory becomes a computing tool. *Physics Today*, 75(11):36–41, 2022.

[70] Nicholas Cunningham, Joseph Del Rocco, and Derrick Greenspan. Memcomputing: Leveraging memory and physics.

[71] Fabio L Traversa and Massimiliano Di Ventra. Polynomial-time solution of prime factorization and np-complete problems with digital memcomputing machines. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 27(2): 023107, 2017.

[72] Fabio L Traversa and Massimiliano Di Ventra. Memcomputing integer linear programming. *arXiv preprint arXiv:1808.09999*, 2018.

[73] S Agatonovic-Kustrin and R Beresford. Basic concepts of artificial neural network (ann) modeling and its application in pharmaceutical research. *Journal of Pharmaceutical and Biomedical Analysis*, 22(5):717 – 727, 2000. ISSN 0731-7085. doi: https://doi.org/10.1016/S0731-7085(99)00272-1. URL http://www.sciencedirect.com/science/article/pii/S0731708599002721.

[74] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *arXiv preprint arXiv:1811.03378*, 2018.

[75] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[76] Y. Wang, J. Lin, and Z. Wang. An energy-efficient architecture for binary weight convolutional neural networks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26(2):280–293, Feb 2018. ISSN 1557-9999. doi: 10.1109/TVLSI.2017.2767624.

[77] Dominik Scherer, Andreas Müller, and Sven Behnke. Evaluation of pooling operations in convolutional architectures for object recognition. In *International conference on artificial neural networks*, pages 92–101. Springer, 2010.

[78] P. N. Whatmough, S. K. Lee, G. Wei, and D. Brooks. Sub-uj deep neural networks for embedded applications. In *2017 51st Asilomar Conference on Signals, Systems, and Computers*, pages 1912–1915, Oct 2017. doi: 10.1109/ACSSC.2017.8335697.

[79] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[80] Eyyüb Sari, Mouloud Belbahri, and Vahid Partovi Nia. How does batch normalization help binary training?, 2019.

[81] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131, 2015.

[82] Y. Pan, P. Ouyang, Y. Zhao, W. Kang, S. Yin, Y. Zhang, W. Zhao, and S. Wei. A multilevel cell stt-mram-based computing in-memory accelerator for binary convolutional neural network. *IEEE Transactions on Magnetics*, 54(11):1–5, 2018.

[83] H. Yonekawa, S. Sato, H. Nakahara, K. Ando, K. Ueyoshi, K. Hirose, K. Orimo, S. Takamaeda-Yamazaki, M. Ikebe, T. Asai, and M. Motomura. In-memory area-efficient signal streaming processor design for binary neural networks. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, pages 116–119, Aug 2017. doi: 10.1109/MWSCAS.2017.8052874.

[84] X. Sun, S. Yin, X. Peng, R. Liu, J. Seo, and S. Yu. Xnor-rram: A scalable and parallel resistive synaptic architecture for binary neural networks. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1423–1428, March 2018. doi: 10.23919/DATE.2018.8342235.

[85] W. Wang, Y. Li, M. Wang, L. Wang, Q. Liu, W. Banerjee, L. Li, and M. Liu. A hardware neural network for handwritten digits recognition using binary rram as synaptic weight element. In *2016 IEEE Silicon Nanoelectronics Workshop (SNW)*, pages 50–51, June 2016. doi: 10.1109/SNW.2016.7577980.

[86] Dirk Jansen et al. *The electronic design automation handbook*. Springer, 2003.

[87] Eda software, hardware &amp; tools, 2022. URL https://eda.sw.siemens.com/en-US/.

[88] Eda tools, semiconductor ip and application security solutions, 2022. URL https://www.synopsys.com/.

[89] Computational software for intelligent system design™, 2022. URL https://www.cadence.com/en_US/home.html.

[90] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[91] Muralimanohar et al. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.

[92] Wilson Snyder. Verilator: Open simulation-growing up. *DVClub Bristol*, 2013.

[93] Matthew R Guthaus, James E Stine, Samira Ataei, Brian Chen, Bin Wu, and Mehedi Sarwar. Openram: An open-source memory compiler. In *2016 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6. IEEE, 2016.

[94] Paolo Nenzi and Holger Vogt. Ngspice users manual version 23, 2011.

[95] Geraldo F Oliveira, Paulo C Santos, Marco AZ Alves, and Luigi Carro. A generic processing in memory cycle accurate simulator under hybrid memory cube architecture. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 54–61. IEEE, 2017.

[96] John D Leidel and Yong Chen. Hmc-sim: A simulation framework for hybrid memory cube devices. *Parallel Processing Letters*, 24(04):1442002, 2014.

[97] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[98] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd annual ieee/acm international symposium on microarchitecture*, pages 469–480, 2009.

[99] Xiangyu Dong, Cong Xu, Yuan Xie, and Norman P. Jouppi. Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):994–1007, 2012. doi: 10.1109/TCAD.2012.2185930.

[100] Sheng Xu, Xiaoming Chen, Ying Wang, Yinhe Han, Xuehai Qian, and Xiaowei Li. Pimsim: A flexible and detailed processing-in-memory simulator. *IEEE Computer Architecture Letters*, 18(1):6–9, 2019. doi: 10.1109/LCA.2018.2885752.

[101] Sparsh Mittal, Rujia Wang, and Jeffrey Vetter. Destiny: A comprehensive tool with 3d and multi-level cell memory modeling capability. *Journal of Low Power Electronics and Applications*, 7(3), 2017. ISSN 2079-9268. doi: 10.3390/jlpea7030023. URL https://www.mdpi.com/2079-9268/7/3/23.

[102] Yannan Nellie Wu, Vivienne Sze, and Joel S. Emer. An architecture-level energy and area estimator for processing-in-memory accelerator designs. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 116–118, 2020. doi: 10.1109/ISPASS48437.2020.00024.

[103] Yannan Nellie Wu, Joel S Emer, and Vivienne Sze. Accelergy: An architecture-level energy estimation methodology for accelerator designs. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–8. IEEE, 2019.

[104] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *2019 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 304–315. IEEE, 2019.

[105] Mahdi Zahedi, Muah Abu Lebdeh, Christopher Bengel, Dirk Wouters, Stephan Menzel, Manuel Le Gallo, Abu Sebastian, Stephan Wong, and Said Hamdioui. Mnemosene: Tile architecture and simulator for memristor-based computation-in-memory. *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 18(3):1–24, 2022.

[106] Fei Gao, Georgios Tziantzioulis, and David Wentzlaff. Computedram: In-memory compute using off-the-shelf drams. In *Proceedings of the 52nd annual IEEE/ACM international symposium on microarchitecture*, pages 100–113, 2019.

[107] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. *ACM SIGARCH Computer Architecture News*, 44(3):14–26, 2016.

[108] Paulo C Santos, Geraldo F Oliveira, Diego G Tomé, Marco AZ Alves, Eduardo C Almeida, and Luigi Carro. Operand size reconfiguration for big data processing in memory. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 710–715. IEEE, 2017.

[109] Jeyhun Karimov, Tilmann Rabl, and Volker Markl. Polybench: The first benchmark for polystores. In *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence: 10th TPC Technology Conference, TPCTC 2018, Rio de Janeiro, Brazil, August 27–31, 2018, Revised Selected Papers 10*, pages 24–41. Springer, 2019.

[110] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, et al. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th international conference on Supercomputing*, pages 14–25, 2002.

[111] Zhaojun Lu, Md Tanvir Arafin, and Gang Qu. Rime: A scalable and energy-efficient processing-in-memory architecture for floating-point operations. In *Proceedings of the 26th Asia and South Pacific Design Automation Conference*, ASPDAC '21, page 120–125, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450379991. doi: 10.1145/3394885.3431524. URL https://doi.org/10.1145/3394885.3431524.

[112] Andrea Coluccio. In-memory binary neural networks, 2019.

[113] Shahar Kvatinsky, Guy Satat, Nimrod Wald, Eby G Friedman, Avinoam Kolodny, and Uri C Weiser. Memristor-based material implication (imply) logic: Design principles and methodologies. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2054–2066, 2013.

[114] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. ipim: Programmable in-memory image processing accelerator using near-bank architecture. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 804–817, 2020. doi: 10.1109/ISCA45697.2020.00071.

[115] Ke Chen, Sheng Li, Naveen Muralimanohar, Jung Ho Ahn, Jay B Brockman, and Norman P Jouppi. Cacti-3dd: Architecture-level modeling for 3d die-stacked dram main memory. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 33–38. IEEE, 2012.

[116] Purab Ranjan Sutradhar, Mark Connolly, Sathwika Bavikadi, Sai Manoj Pudukotai Dinakarrao, Mark A. Indovina, and Amlan Ganguly. ppim: A programmable processor-in-memory architecture with precision-scaling for deep learning. *IEEE Computer Architecture Letters*, 19(2):118–121, 2020. doi: 10.1109/LCA.2020.3011643.

[117] Waterman A. and Asanović K. *The RISC-V Instruction Set Manual Volume I: User-Level ISA*. EECS Department, University of California, Berkeley, 8 2016.

[118] Özlem Altınay and Berna Örs. Instruction extension of rv32i and gcc back end for ascon lightweight cryptography algorithm. In *2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS)*, pages 1–6. IEEE, 2021.

[119] Y. Chen, T. Krishna, J. Emer, and V. Sze. 14.5 eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 262–263, Jan 2016. doi: 10.1109/ISSCC.2016.7418007.

[120] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9, Aug 2016. doi: 10.1109/FPL.2016.7577308.

[121] S. Wang, D. Zhou, X. Han, and T. Yoshimura. Chain-nn: An energy-efficient 1d chain architecture for accelerating deep convolutional neural networks. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 1032–1037, March 2017. doi: 10.23919/DATE.2017.7927142.

[122] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. Yodann: An ultra-low power convolutional neural network accelerator based on binary weights. In *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 236–241, July 2016. doi: 10.1109/ISVLSI.2016.111.

[123] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[124] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. *ACM SIGARCH computer architecture news*, 23(2):24–36, 1995.

[125] Riscv-Collab. Riscv-collab/riscv-gnu-toolchain: Gnu toolchain for risc-v, including gcc, 2022. URL https://github.com/riscv-collab/riscv-gnu-toolchain.

[126] Umberto Casale. Programmable lim: a modular and reconfigurable approach to the logic in memory, 2020.

[127] Andrea Coluccio, Umberto Casale, Angela Guastamacchia, Giovanna Turvani, Marco Vacca, Massimo Ruo Roch, Maurizio Zamboni, and Mariagrazia Graziano. Hybrid-simd: a modular and reconfigurable approach to beyond von neumann computing. *IEEE Transactions on Computers*, 2021.

[128] Xin Jin and Jiawei Han. *K-Means Clustering*, pages 563–564. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_425. URL https://doi.org/10.1007/978-0-387-30164-8_425.

[129] Paul Lane, Viv Schupmann, and I Stuart. Oracle database data warehousing guide. *10g Release*, 2(10.2), 2005.

[130] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017. doi: 10.1109/TVLSI.2017.2654506.

[131] Andrea Marchesin, Giovanna Turvani, Andrea Coluccio, Fabrizio Riente, Marco Vacca, Massimo Ruo Roch, Mariagrazia Graziano, and Maurizio Zamboni. Octantis: An exploration tool for beyond von neumann architectures. In *2021 16th International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*, pages 1–5, 2021. doi: 10.1109/DTIS53253.2021.9505135.

[132] Milena Andrighetti, Giovanna Turvani, Giulia Santoro, Marco Vacca, Andrea Marchesin, Fabrizio Ottati, Massimo Ruo Roch, Mariagrazia Graziano, and Maurizio Zamboni. Data processing and information classification—an in-memory approach. *Sensors*, 20(6), 2020. ISSN 1424-8220. doi: 10.3390/s20061681. URL https://www.mdpi.com/1424-8220/20/6/1681.

[133] Tony Bybell. Gtkwave electronic waveform viewer, 2010.

[134] Burkhard Meier. *Python GUI Programming Cookbook: Develop functional and responsive user interfaces with tkinter and PyQt5*. Packt Publishing Ltd, 2019.

[135] Frank Rubin. The lee path connection algorithm. *IEEE Transactions on computers*, 100(9):907–914, 1974.

[136] Saharsh Laud. Lee algorithm: Shortest path in a maze, Feb 2021. URL https://www.codesdope.com/blog/article/lee-algorithm/.

[137] Sandeep Koranne. Hierarchical data format 5: Hdf5. In *Handbook of open source tools*, pages 191–200. Springer, 2011.

[138] Aliaksei Chapyzhenka and Jonah Probell. Wavedrom: rendering beautiful waveforms from plain text. In *Synopsys User Group (SNUG) Silicon Valley 2016 Proceedings*, 2016.

[139] Georg Brandl. Sphinx documentation. *URL http://sphinx-doc. org/sphinx. pdf*, 2021.

[140] Ray Salemi. *The Uvm Primer: A step-by-step introduction to the universal verification methodology*. Boston Light Press, 2013.

[141] Mariagrazia Graziano, Marco Vacca, and Nicola Piano. Dexima: a design explorer for in-memory architectures. 2019.

[142] Loris Mendola. Dexima a synthesis tool and performance estimator for logic-in-memory architectures, 2021.

[143] S.R. Vemuru and N. Scheinberg. Short-circuit power dissipation estimation for cmos logic gates. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 41(11):762–765, 1994. doi: 10.1109/81.331533.

[144] Spiridon Nikolaidis and Alexander Chatzigeorgiou. Analytical estimation of propagation delay and short-circuit power dissipation in cmos gates. *International journal of circuit theory and applications*, 27(4):375–392, 1999.

[145] L Bisdounis and O Koufopavlou. Short-circuit energy dissipation modeling for submicrometer cmos gates. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 47(9):1350–1361, 2000.

[146] Harry JM Veendrick. Short-circuit dissipation of static cmos circuitry and its impact on the design of buffer circuits. *IEEE Journal of Solid-State Circuits*, 19(4):468–473, 1984.

[147] Sergio Galdino. A family of regula falsi root-finding methods. In *Proceedings of 2011 World Congress on Engineering and Technology (CET 2011)*, volume 1, pages 514–517, 2011.

[148] Wikipedia contributors. Ridders' method — Wikipedia, the free encyclopedia, 2021. URL https://en.wikipedia.org/w/index.php?title=Ridders%27_method&oldid=1033092888. [Online; accessed 24-November-2022].

[149] Freepdk45 technology model, 2022. URL https://eda.ncsu.edu/freepdk/freepdk45/.

[150] Mohan V Dunga, X Xi, Jin He, Weidong Liu, Kanyu M Cao, Xiaodong Jin, Jeff J Ou, Mansun Chan, Ali M Niknejad, and Chenming Hu. Bsim4. 6.0 mosfet model. *University of California, Berkeley*, 2006.

[151] Longest path in an undirected tree, Jul 2022. URL https://www.geeksforgeeks.org/longest-path-undirected-tree/.

[152] Antoine Courtay, Olivier Sentieys, Johann Laurent, and Nathalie Julien. High-level interconnect delay and power estimation. *Journal of Low Power Electronics*, 4(1):21–33, 2008.

[153] Yehea I Ismail, Eby G Friedman, and Jose L Neves. Figures of merit to characterize the importance of on-chip inductance. In *Proceedings of the 35th annual Design Automation Conference*, pages 560–565, 1998.

[154] Junjie Mu, Hyunjoon Kim, and Bongjin Kim. Sram-based in-memory computing macro featuring voltage-mode accumulator and row-by-row adc for processing neural networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(6):2412–2422, 2022.

[155] Didem Z Turker, Sunil P Khatri, and Edgar Sánchez-Sinencio. A dcvsl delay cell for fast low power frequency synthesis applications. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(6):1225–1238, 2011.

Glossary

AlexNet AlexNet is a convolutional neural network, which competed in the ImageNet Large Scale Visual Recognition Challenge in 2012. The network achieved a top-5 error of 15.3% .. xviii, 18, 42–44, 46, 48

ASIC Application Specific Integrated Circuit. 21, 31

BCNN Binary convolutional neural network. 20

BL Bit Line. xvi, xvii, xx, 7, 10, 53, 95, 96, 148, 150, 161, 214, 217, 218, 220–222, 266, 268, 279

BNN Binary neural network. 46, 50

BSIM4 Berkeley Short-channel IGFET Model 4. 155, 166, 200–202, 205, 208, 210

BvNC Beyond von Neumann Computing. vi, 7, 9, 11, 15, 20, 21, 23–29, 44, 71, 282

CAD Computer Aided Design. 20, 22, 71, 82, 101, 161, 235, 243, 250

CAM Content Addreassable Memory. xviii, 7–9, 12, 49

CMOS Complementary Metal-Oxide Semiconductor. xix, 5, 7, 10, 40, 41, 43, 44, 49, 58, 60, 154, 155, 166, 199, 200, 233, 263, 278, 283

CNN Convolutional neural network. xi, 16, 17, 44, 47, 49

CPU Central Processing Unit. vii, x–xiii, xvii, xix, 5, 6, 11, 15, 25, 28, 29, 34, 37–39, 50–53, 58, 59, 61–63, 65–68, 74, 77–79, 82, 85, 97–101, 106, 107, 109, 110, 117, 118, 131, 134, 135, 171, 227–232, 248, 249, 251, 261, 263–265, 267, 269, 279, 280, 283

DC Synopsys Design Compiler. xix, 233–235, 243–245, 248, 273

DRAM Dynamic Random Access Memory. xi, 7, 8, 10–12, 20, 22, 24–26, 28, 44, 59

DRC Design Rule Check. 39

DUT Design Under Test. 125, 141, 142, 145

EDA Electronic Design Automation. 20, 21, 125, 234, 283

FinFET Fin Field-Effect Transistor. 5

FP Floating Point. 18

FPGA Field Programmable Gate Array. 21, 31

GAAFET Gate-All-Around Transistor. 5

GPU Graphics Processing Unit. 28, 29

HMC Hybrid Memory Cube. 10, 11, 22

HTML HyperText Markup Language. xiv, 130, 131

IC Integrated Circuit. 5

IFMAP Input feature map. 16, 17, 19, 47, 48

IRL Intra Row Logic. vii, xii, xiii, xvi, xvii, 52, 53, 55–57, 72–75, 79–81, 85–88, 96, 101, 113, 116, 133, 134, 150, 160, 164, 171, 188, 214, 215, 249, 252–254, 256, 259–262, 266, 267, 272

LEF Library Exchange Format. 40

LeNet LeNet is a type of convolutional neural network. xi, 16

LiM Logic-in-Memory. vii, viii, x–xiii, xv–xix, 7, 10–12, 17, 23–42, 46–57, 61, 67, 68, 71–77, 79–83, 85–90, 92–101, 107, 109, 110, 112, 113, 116, 118, 120, 124, 129–135, 138–143, 146, 147, 149–151, 153–155, 158, 160, 161, 164, 170, 171, 188, 189, 214, 217, 218, 220, 223, 225, 227–232, 245, 248–259, 261–269, 274, 278–283

LVS Layout-Vs-Schematic. 39, 191

MAC Multiply and Accumulate. 23, 26, 28, 42

MAGIC Memristor-Aided Logic. 27

ML Match Line. 7, 8

MLC Multi level cell, more then one bit can be hold into a single cell. 22

MLP Multilayer perceptron is a class of artificial neural network. Each node is a neuron that uses a nonlinear activation function, except for the inputs. MLP uses backpropagation for training.. 16, 17

MNIST The MNIST database (Modified National Institute of Standards and Technology database) is a dataset of handwritten digits with 60000 images in B/W.. 38, 265

MRAM Magnetoresistive random-access memory (MRAM) is a non-volatile random-access memory technology. Data in MRAM is not stored as electric charge or current flows, but by magnetic storage elements. 22, 44, 46

MTJ Magnetic Tunnel Junction is a component composed by two ferromagnets separated by an insulator. Electrons can tunnel from one ferromagnet into the other.. 10, 20

NN Neural Network. 15–20, 44–46, 48, 49, 65

NVM Non-volatile memory. 12, 22

OFMAP Output feature map. 17, 47

OOM Out of memory implementation.. 46, 48–50

OOO Out-Of-Order. 6

PCM Phase Change Memory. 10, 22, 26

PE Processing Element. 42, 63

PEX Parasitics EXtraction. 39

PIM Processing in Memory. xviii, 22, 24, 25, 28, 29

ReLU Rectified linear unit, a type of neuron's activation function which consists into $ReLU(x) = max(0,x)$. In terms of training time, it is the best choice.. 16

RRAM Resistive switching random access memory. 20, 22, 26, 27

RTL Register Transfer Level. viii, 20–22, 37, 67, 77, 81, 101, 128, 129, 138–140, 142, 144, 146, 148, 150, 152, 243–245

SIMD Single Instruction Multiple Data. xii, xix, 11, 52–55, 57–68, 71, 73, 75

SL Search Line. 7–9

SLC Single Level Cell. 22

SOT Spin-orbit torque: a type of magnetic RAM. 22, 44, 46

SRAM Static Random Access Memory. x, xi, xix, xx, 6–12, 20, 22, 23, 26, 119, 128, 129, 170, 188, 278–280

stride stride, in the context of CNNs, is the distance between the receptive field centers of neighboring neurons in a kernel map. 16, 17, 47

STT Spin-transfer torque is an effect in which the orientation of a magnetic layer in a MTJ can be modified using a spin-polarized current.. 22, 26

TOP5 top-5 error is measured by checking if the target label is one of your top 5 predictions (the 5 ones with the highest probabilities).. xi, 18

TSV Through Silicon Via. 11, 20

UML Unified Modeling Language. xiv, xv, 156, 157, 165, 174, 188, 226

UVM Universal Verification Methodology. ix, xiv, 81, 90, 124, 138–143, 146–149, 151, 153

VCD Value Change Dump. viii, xiii, xiv, 81, 92, 93, 107, 111, 125, 126, 130, 139

VLSI Very Large Scale Integration. 21

WL Word Line. 7, 9, 10, 148