POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

DExIMA: a Design Explorer for In-Memory Architectures

Relatori:
Prof. Mariagrazia Graziano
Prof. Marco Vacca

Candidato:
Nicola Piano

Ottobre 2019

Acknowledgments

Questa tesi mi ha richiesto quasi un anno di duro lavoro. Fatto di momenti pesanti, quando non ne potevo più e non vedevo l'ora che finisse. Ci sono state tante persone che mi sono state vicine. Ma vorrei ringraziare tutte le persone che ci sono state in questi tre anni.

Prima di tutto vorrei ringraziare la Professoressa Graziano. Prima di tutto per avermi proposto un lavoro impegnativo, complicato e nuovo, qualcosa di diverso rispetto a ciò che conoscevo già. In secondo luogo vorrei ringraziarla per la fiducia che non ha mai fatto mancare, nonostante per lunghi periodi non portassi i risultati sperati.

Un grosso ringraziamento anche a Marco Vacca. È stato un riferimento con la sua impagabile disponibilità. Una guida senza cui non avrei realizzato tutto ciò che ho realizzato.

Grazie anche alla mia famiglia per il supporto economico ma non solo durante questi tre anni. Loro mi hanno sempre supportato nei miei sogni, nelle mie ambizioni.

Ai miei amici, Gianlu, Riki, Fra. L'esperienza di questi tre anni non sarebbe stata la stessa senza di loro. Mi sono stati vicini come nessun altro nei momenti più difficili che ci sono stati.

Non c'è stato esame che non abbia preparato insieme a Gianluca nelle lunghe giornate con pranzi a base di *aglio, olio e peperoncino*. Ho mille ricordi che mi porterò nel mio bagaglio personale, dalla nottata sulla PCB (e birra successiva) fino al nostro caro processore. Sono innumerevoli gli episodi che non scorderò mai.

Grazie a Riki, una persona su cui ho potuto sempre contare. Certo non ci va delicato nel dirti le cose, ma è quello che serve in certi momenti. Tre anni di coinquilinanza non sono facili, ma sicuramente non sono mai stati noiosi.

Infine Fra. Una persona con cui ho condiviso un percorso in questi anni, quello di Icarus. Un percorso in cui ci siamo trovati a dover collaborare. Nonostante questo potesse portarci a litigare sono soddisfatto invece di quello che abbiamo creato insieme. Oltre a questo un percorso di tipo personale in cui ci siamo supportati a vicenda.

Uno speciale ringraziamento ai ragazzi del team, specialmente ai miei amici Elettrodomestici. Esperienza fantastica che mi ha portato tanto sia dal punto di vista umano che dal punto di vista professionale. Non è facile avere la responsabilità di guidare un gruppo di persone ma è stato più facile con delle persone con tanta voglia di lavorare e di seguire me e Francesco con le nostre idee.

Un ultimo grande ringraziamento va alla persona più vicina di tutte in questo periodo. Agata. In questi mesi ho ricevuto un grande supporto da lei. Momenti pesanti sono stati alleggeriti dalla sua presenza. Nei momenti di stress io ho trovato tranquillità. Nei momenti di felicità ho trovato complicità. Grazie per il grande supporto e e per la grande sopportazione.

Contents

List of Figures

Introduction

In the last years the deep submicron integration has carried out a reduction of the gate delay. At the same time the memory delay reduction has been significantly less resulting in the so called *memory bottleneck*. The Figure 1 shows the increasing of memory and processor performance from 1980 to 2010. Although the data are pretty old the trend is clear. In these 30 years in front of a strong processor performance increase, the memory performance have not kept pace. It is clear that technology scaling is no longer enough to increase system performance. A new design paradigm is then necessary.

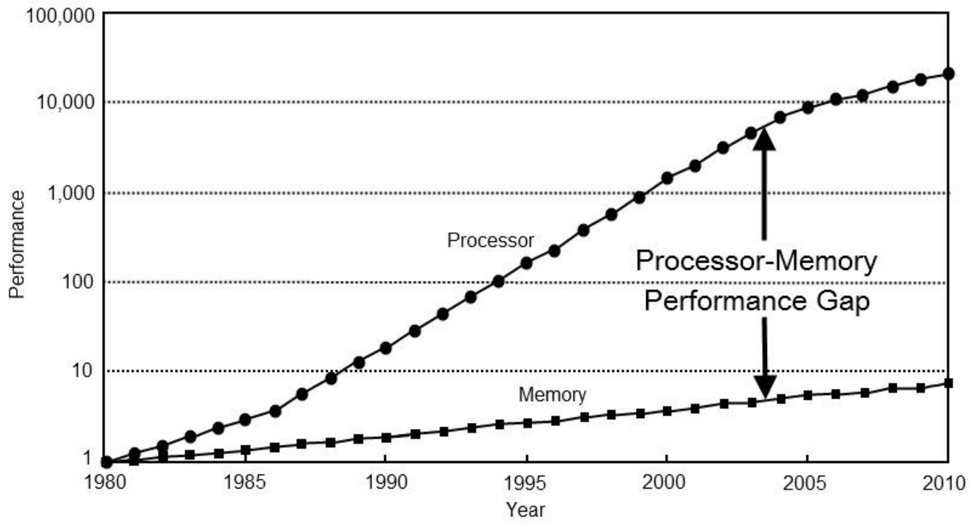


Figure 1: **Memory-Processor performance gap**. The figure shows the difference in performance increase between memory and processor over 30 years.

In the last years research has tried to reduce the impact of the memory on computation in different ways. The strategies are manly two. The first one is to try to reduce the physical distance between memory and computation using new technologies such as 3D stacking. This leads to a reduction of the delay and power consumption of the memory bus.

The second approach is to take the in-memory computing way. This is achieved by migrating from a standard von Neumann architecture, in which the memory only stores data and the logic elements perform the computation, to a new paradigm, that is the in-memory computing, in which the memory also performs computation. In this way the data are moved from and to memory a less number of times. The memory bottleneck has been circumvented avoiding a large number of memory access.

A lot of papers in the last years present a variety of implementation marked by several degrees of in-memory computing. The typical approach has been to analyze an application typically implemented with an ASIC approach and to search a way to reduce the impact of the memory. This produced infinite non-homogeneous structures that are even widely different from one another. In this scenario, in-memory computing includes a large number of different techniques.

The purpose of this thesis work is to determine a fixed high-level structure for Logic-in-Memory based architectures. This structure has to be as generic as possible, with some common boxes, let's call it containers, in which different types of in-memory computing can be allocated. However, the structure leaves an infinite number of unanswered questions. What should a specific box contain? Where is it better to bind a given operation? Should this implementation be fine also for a larger amount of data involved in computation? The answer to these questions is hard to find simply designing by hand. On the other hand a detailed working RTL implementation could be too demanding in a preliminary design. A solution has been found in a tool, which has been called *DExIMA*.

DExIMA stands for **D**esign **Ex**plorer for **I**n-**M**emory **A**rchitectures. As the name suggests it is a performance estimator mainly for in-memory architectures. The purpose of this tool is precisely to answer to the previous design questions. In other words, DExIMA helps the designer to populate the design space.

In the first chapter an overview of what research has developed in the field of in-memory computing is given. In order to start to developed a general structure, a classification of these work is fundamental.

In the second chapter it is proposed a generic structure for Logic-in-Memory architectures. This should be the starting point for any in-memory computing design.

Then the shortest, but not the least important one, is the third. In this chapter the reasons that led to develop DExIMA are extensively explained.

The fourth one is entirely dedicated to bus. The bus is undoubtedly important in the system performance estimations. In this chapter a specific explanation of how to estimate bus performance is given.

The next three chapter are focused on the tool itself. The first one is an overview of the tool, the second one shows how to interact and describe the hardware while the last one is focused on the simulator core and the hardware models.

The eighth chapter presents an example of usage. Three version of Binary Neural Network are implemented and compared with DExIMA.

In the last chapter, some considerations on the tool and on the results are presented showing its strengths and weakness. Finally some consideration on further improvements are brought to the reader.

Chapter 1

State of the art

The problem of memory bandwidth and power consumption is becoming more and more relevant in digital design and then in computation. As transistors are getting smaller and smaller the speed of the logic gates increases. In data intensive algorithm the limitation to the computing capability is the memory performance. For this reason new design solutions have been explored in the last years with the aim to overcome the so called *memory bottleneck*.

While in von Neumann architecture memory and processing elements are strictly separated, these new design paradigms have been explored various degrees of integration between memory and processing elements.

According to [1] four categories can be identified:

- Computing-near-Memory (CnM);
- Computing-in-Memory (CiM);
- Computing-with-Memory (CwM);
- Logic-in-Memory (LiM).

This classification is based on the role the memory plays in the computation.

Computing-near-Memory Since in the first approach, *Computing-near-Memory*, memory and processing elements are still well separated, it is the most similar to the von Neumann one. In fact, the processing elements are simply put closer to the memory. This can be achieved by stacking memory layers and processing elements in a 3D structure and connecting them exploiting *through-silicon vias (TSVs)* technology [2].

The memory bottleneck problem is addressed reducing the length of the bus and thus reducing the power consumption and the delay resulting from the bus usage.

Moreover, stacking memory layers on top each other leads to a reduction of the memory access cost. This solution is strictly technological since there is no change at the system level.

Some works have exploited this solution. An important work coming from this class of architectures is Hybrid Memory Cube [3]. In this work it is proposed a 3D stacked DRAM in which the lower layer contains the logic which performs data routing, refresh, DRAM sequencing and so on. This lead to an increment of memory bandwidth, lower power consumption and reduced area.

An other significant work proposes a structure of stacked processor-memory [4]. It is a multiprocessor system in which each core is a structure with memory and logic stacked one on top of the other. A 2D mesh network allows communication between cores.

Computing-in-Memory Architectures belonging to the category of *Computing-in-Memory* approach exploit the memory structure to perform computation. Either non-volatile, such as MRAM (Magnetoresistive Random-Access Memory) or RRAM (Resistive Random-Access Memory), or volatile memory, such as SRAM or DRAM, are employed.

In this approach the memory is not modify to allow computation, but the analog elements such as the sense amplifier or the match lines for a CAM are exploited to perform row-wise or column-wise operations.

The memory is left as is. From a technology perspective this is a big advantage since the production of a *special* memory is not necessary. On the other hand the computation is limited to a small group of operations.

A work belonging to this category is an architecture called PIMA [5]. PIMA allows row-wise and column-wise computation just exploiting the functionality of the sense amplifier. Obviously the memory controller is totally modified to perform these operations. CiM can be applied in some memory-intensive computation, such as neural network. An example is PRIME [6] which is a solution to accelerate NN applications in ReRAM based main memory.

Computing-with-Memory The third approach, *Computing-with-Memory*, exploits the non-volatile memory technologies. However, the computation is performed using the memory as look-up table (LUT). In the memory, computation is stored as precomputed results. As the previous approach the memory array is left as is, the structure is exploited to perform this simple logic operations.

For example, an architecture called RADAR, uses a ReCAM (ReRAM-based CAM) to compute DNA alignment [7]. In this case CAM, implemented in resistive RAM technology, are employed to compute results. Authors of [8] present a design for the same purpose. In [9], instead, it is proposed an associative processor. It is a processor, with its own instruction set, in which the results are precomputed. A CAM allows the retrieve of outputs.

Logic-in-Memory *Logic-in-Memory* approach is the most advanced integration between memory and logic. Differently from all other approaches, Logic-in-Memory allows to perform computation without moving data from one block of the memory to an other one. Data are moved to the peripheral circuits only when they are read. This result is achieved by integrating small logic elements in every single memory cell, strongly modifying the memory array.

In [10] it proposes a memory cell integrated with CMOS logic gates. The cell is realized in MTJ technology [11]. The memory cell in [10] is employed in [12] for search operations. An other architecture, proposed in [13], interleaves rows with logic and memory and rows with only memory elements to store results.

Some works try to integrate both LiM and other approaches. The system proposed in [14] present LiM elements but also a CnM approach. The architecture presents two layer, a memory layer and a logic layer. The memory is not a standard memory but it integrates some logic, exploiting Logic-in-Memory approach.

1.1 A further step: CLiMA (Configurable Logic-in-Memory Architecture)

The concepts behind *CLiMA (Configurable Logic-in-Memory Architecture)* [1] are essentially the approaches briefly outlined in the previous section. The purpose is the definition of a flexible architecture that can be adapted to any given application.

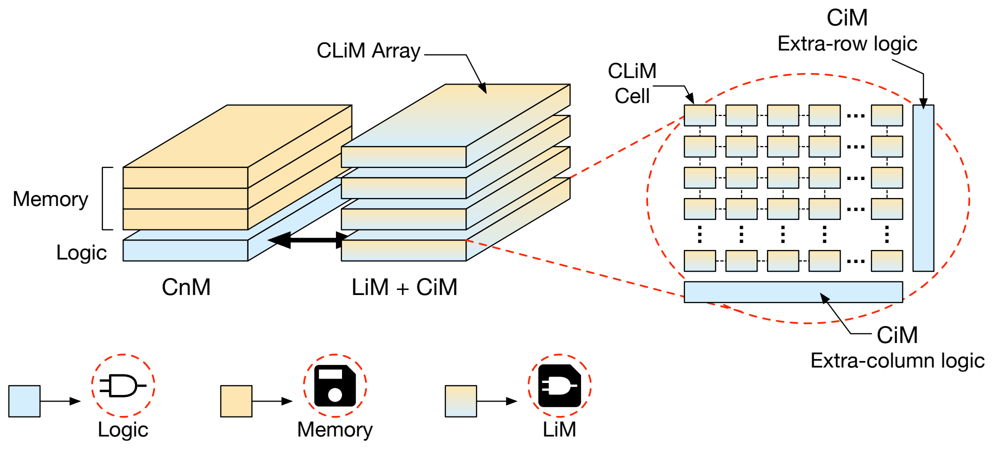


Figure 1.1: **CLiMA in-memory computing appraches**. The figure shows the coexistence of the four approaches in CLiMA. [1]

As suggested by the name, the architecture is designed to be configurable according to the type of computation to be performed. Even if the approach mainly exploited in CLiMA is the LiM one (again, as suggested by the name), CnM, CiM and CwM approaches are sometimes exploited.

Finally, CLiMA is an expression of the four approaches to solve the memory-wall problem. Since the applications of in-memory computing differ greatly from one another, only this multi-approach architecture can suit all the requirements of a complex system.

In Figure 1.1 it can be noticed that the system is heterogeneous. All the four approaches coexist and contribute to the computation. The Computing-near-Memory approach shorten the physical distance between logic and memory leading to a faster data transfer between logic and memory. In the meanwhile the LiM unit embeds both LiM and CiM approaches.

Of course the designer can choose where to map each and every operation of an algorithm. This potentially leads to a large number of different implementation of the architecture even for the same application. Obviously, the freedom of the designer's fantasy is not unlimited. To be more precise a closer look to CLiMA can show its limits.

LiM, the main approach used by CLiMA, is exploited by simply adding logic in the single memory cell, which is called CLiM cell. It is a simple storage cell

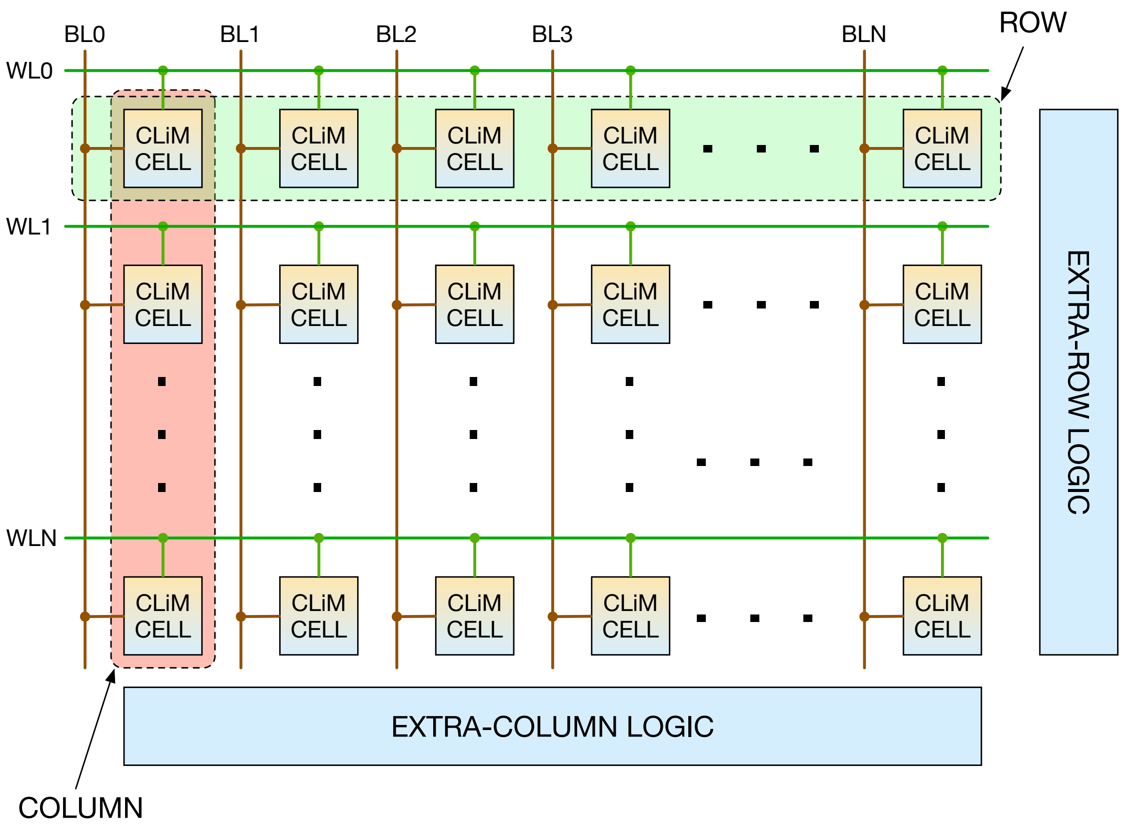


Figure 1.2: **CLiM array high level structure**. CLiM cells are arranged in rows and columns highlighted in green and red respectively. Extra-row and extra-column logic represent the peripheral circuits which performs operation with a CiM approach. [1]

implemented in one of the technologies mentioned above (SRAM, DRAM, MRAM, RRAR etc.). The content of this cell can be manipulated locally, i.e. inside the cell, by executing bit-wise operation such as OR, AND, XOR or slightly more complex operation such as two bit additions.

The LiM cell is the base unit of CLiMA. The Figure 1.3 shows the cell organization. The two main elements are the memory cell and the full adder. The full adder is choosen for its flexibility. In fact, it can perform all the essential logic operations. Table 1.1 shows the truth table also in case of a fixed $C_{in} = 0$ or $C_{in} = 1$. In these cases the full adder can perform the basic logic operation such as AND, OR, XOR and XNOR.

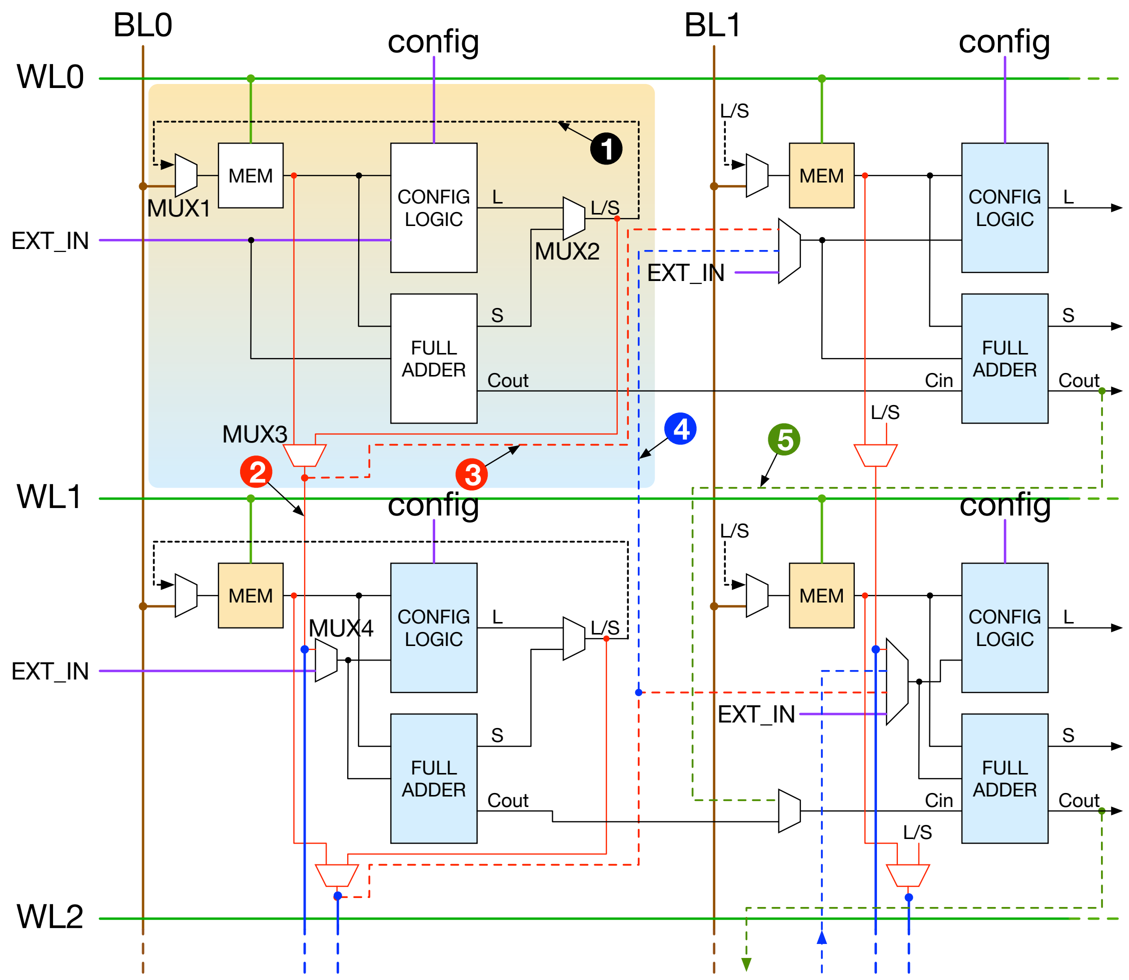


Figure 1.3: **CLiM cell structure and interconnections**. A full adder is embedded in the cell. Some logic allows the configuration of the full adder and of the interconnections. Some multiplexers allow to configure the interconnections with the surrounding cells. [1]

This memory organization allows to perform different logic functionality, which are also shows in Figure 1.2:

- local: the computation is performed inside the cell;
- intra-row: the computation involves two or more cells in the same row;
- intra-column: the computation involves two or more cells in the same column;
- inter-row: the computation is executed between two rows;

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$C_{in} = 0$			
A	B	C_{out} $A \cdot B$	S $A \oplus B$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

$C_{in} = 1$			
A	B	C_{out} $A + B$	S $\overline{A \oplus B}$
0	0	0	1
0	1	1	0
1	0	1	0
1	1	1	1

Table 1.1: **Full adder truth tables**. In the upper part a table shows the standard FA truth table. The other two tables show the truth table in case of fixed $C_{in} = 0$ and $C_{in} = 1$ respectively.

- inter-column: the computation is executed between two columns.

The inter-row/column are perfect for bitwise operations between rows or columns. These operation are frequent in parallel algorithm and memory intensive application. The inter-row/column computation fits other applications, even more complex. An example is the ripple carry adder (RCA) which can be implemented cascading the full adders of an entire row or column.

The interconnections, as mentioned before, are guaranteed by several multiplexers. The 1.3 shows several possible connections highlighted by circled numbers:

1. write back the logic result in the same cell;

2. cell output to south cell;

3. cell output to east cell;

4. cell output to north-east cell;

5. carry output to south cell.

Clearly the architecture provides a huge number of computation capabilities. In this scenario finding a method to simulate and compare different implementation of the same algorithm in CLiMA is a hard job.

Chapter 2

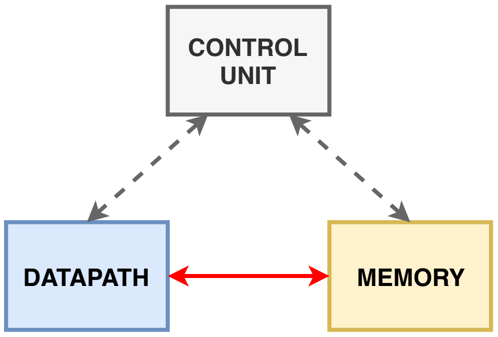
Generic structure for an in-memory architecture

In this chapter it will be discussed the generic structure for an in-memory architecture. The discussion starts from the system level (or high level in this chapter) and goes deeper to lower level up to the bus analysis and the in-memory interconnects.

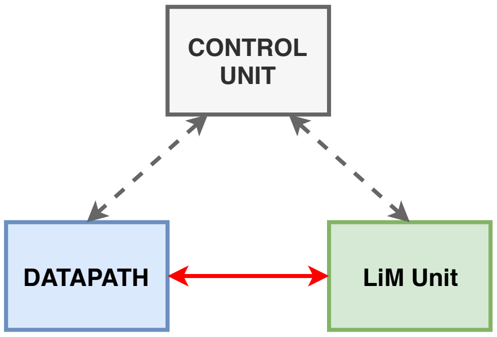
2.1 High-level structure

As shown in Figure 2.1(a), in von Neumann architecture there are three actors:

- datapath;
- memory;



(a) Standard von Neumann architecture



(b) Logic-in-Memory architecture

Figure 2.1: **Von Neumann and LiM architecture structures**. In (a) it is shown the well known structure of the von Neumann architecture, with memory and datapath well separated and a bus (red arrow) that allows data transfer. On the contrary, in (b) it is shown the structure of a in-memory architecture in which the memory acts also as computing element.

- control unit.

The *memory* is the part in which the data are stored. This section is populated by a large number of memory type, from caches to simple SRAM, from main memories to register files. In von Neumann architecture all these types of memories coexist constituting the so called memory hierarchy. Even if these different layers of memory are very different from each another, they only have storage purpose.

On the contrary the *datapath* only performs computation. The data stored in the memories are read, processed and written back. This is the computing sequence of every von Neumann architecture. As can be seen, the computation and the storage are two well separated processes performed by two well separated entities. In von Neumann architectures the datapath acts as computing unit and the memory acts as storage unit.

An other fundamental element is the *bus* which is indicated with the red arrow. It connects memory and datapath. The bus usage strongly affects the computation performance, especially for those algorithm which may be classified as *memory intensive*.

The grey element, the *control unit*, is responsible for the control signal for both memory and datapath. As can be seen it is present in both architecture and has the same tasks.

In the Figure 2.1(b) is shows an architecture structure which, at first glance, seems similar to the von Neumann one (Figure 2.1(a)). In fact, it presents a datapath block (blue box in both structures), a control unit, (grey block in both structures) and a bus (the red arrow in both structures). The difference lies in the block located in the bottom right corner: the memory for von Neumann and the *Logic-in-Memory unit (LiM unit)* in the LiM-based architecture.

As highlighted by the different color, the LiM unit has not only storage capability but it is also a computing element as well as the datapath. It can be stated that a portion of computation is moved from datapath to memory resulting in a *magic* unit which performs both storage and computation.

The amount of computing elements in the LiM unit can varies significantly, from simple logic gates inside some memory cell to large and complex circuits performing inter row and inter column operations. Then an algorithm can be implemented mapping some operations in-memory and some others out-of-memory.

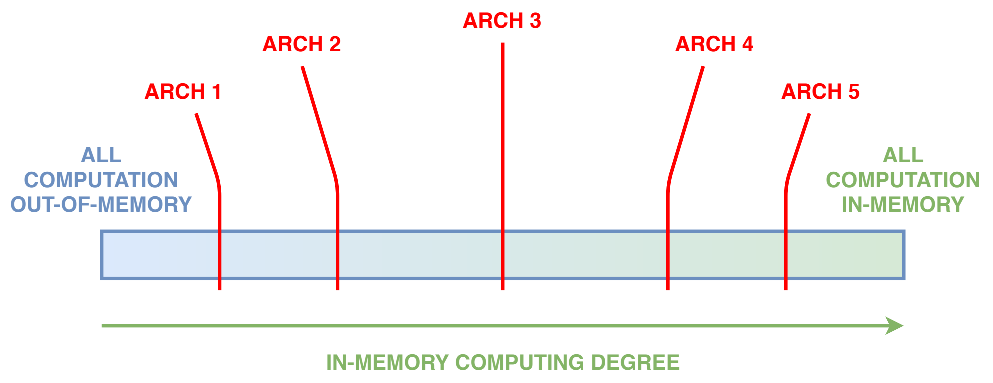

ARCH 1
ARCH 2
ARCH 3
ARCH 4
ARCH 5
ALL COMPUTATION OUT-OF-MEMORY
ALL COMPUTATION IN-MEMORY
IN-MEMORY COMPUTING DEGREE

Figure 2.2: **In-memory computing scale**. Different architectures performing the same algorithm can be classified by their degree of in-memory computing.

Clearly different solution can be found in the design space and the resulting architectures can be defined by their degree of in-memory computing. This parameter can be defined as *the amount of operations which are executed in-memory compared to the total amount of operations*.

The Figure 2.2 shows five architectures which are ordered by this factor. The leftmost architectures are characterized by a lower degree of in-memory computing, while the rightmost architectures are characterized by a higher degree of in-memory computing. The two extremes of the scale are the von Neumann architecture (left side) and an architectures in which all operations are performed in-memory (right side).

2.2 LiM unit

The LiM unit is the central element for an in-memory architecture. The Figure 2.3 shows a general configuration of this block.

The basic element is the LiM cell, highlighted by yellow boxes, which can be seen as a modified memory cell. These elements perform local computation with maximum degree of in-memory computing.

The red and violet boxes instead are defined as inter-row computation. Often the computation between two or more rows is essential to perform a large number of operations.

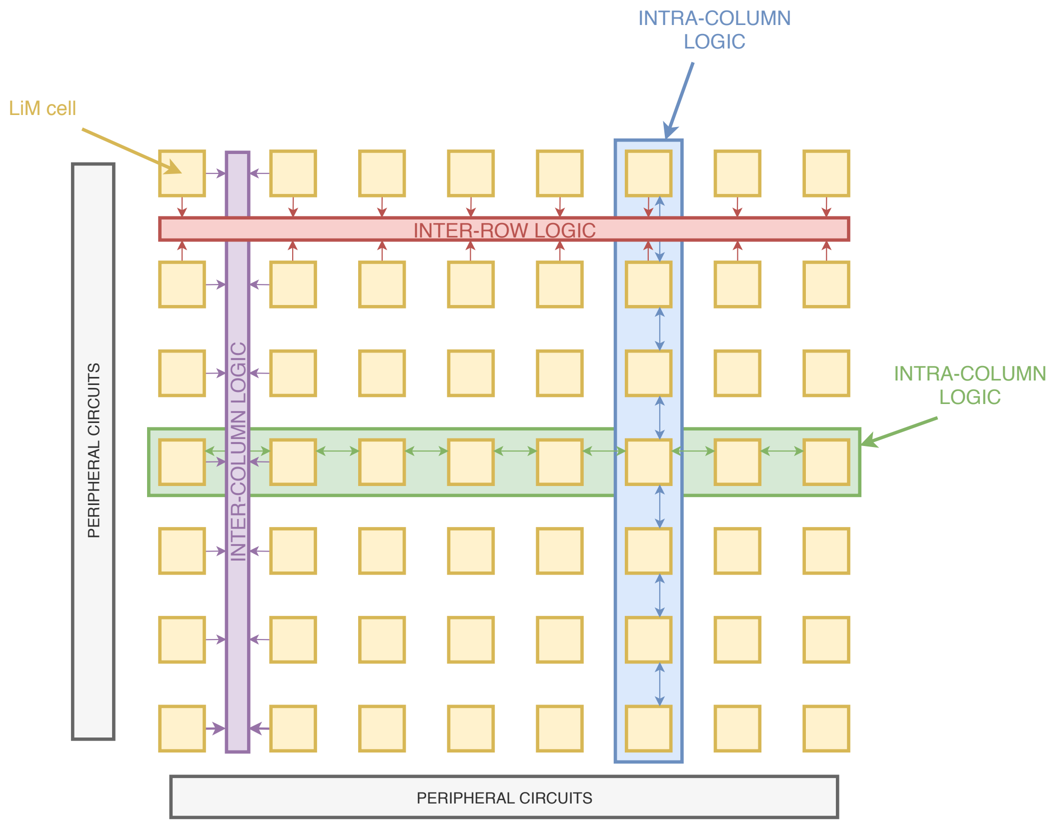


Figure 2.3: **Logic-in-memory unit**. The yellow boxes are LiM cells which may or may not include logic. The red and violet boxes represent logic operations that involved two columns and two rows respectively. The green and blue boxes represent inter rows and inter columns operations respectively.

Finally the blue and green boxes represent computation that involves different cells of the same row or column. These configuration allows to perform even complex computation.

There are no limits in the selection of computing elements, although a good design should comply with these guidelines.

2.2.1 LiM cell

As said before, the LiM cell is the base element of the LiM unit. The structure of the cell is shown in Figure 2.4.

The yellow box is a *memory cell*. It can be implemented with different technology,

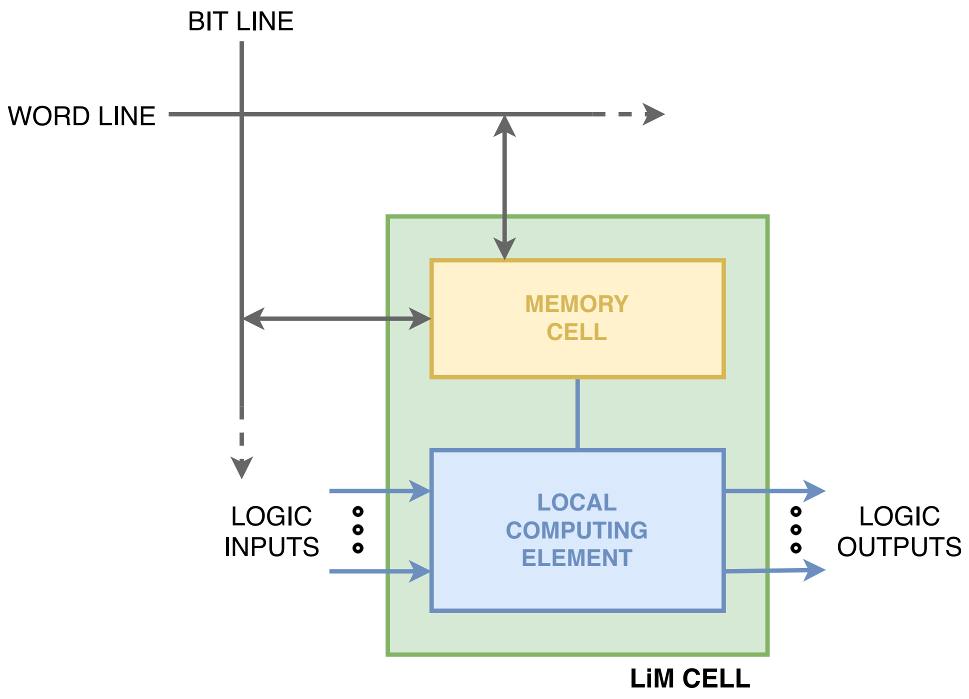


Figure 2.4: **LiM cell internal structure**. In yellow the memory cell, in blue the computing element.

from standard CMOS SRAM to the newest technology such as MTJ, Resistive RAM and so on.

The *local computing element* instead is what makes the cell a LiM cell. It can be a simple gate such as AND, OR and so on or an arithmetic circuit such as full adder and half adder. The Figure 2.4 shows that a fixed connection between the memory cell and the local computing element is always present. The other ports have no fixed connection and then they can be configurable.

Note that the *configurability* doesn't mean that the connection can be configurable at run time, but instead, the configurability is thought as a design choice. The designer can explore different configurations at design time. The resulting architecture indeed is not a programmable logic system like an FPGA, but it is an architecture specifically designed for a given application.

As mentioned before common logic computing elements are simple gates such as NOT, AND, OR and so on or slightly more complex circuits such as full adder and half adder. Since one of the input ports of computing element is fixed connected, to ensure locality of computation, the other inputs are left unconnected by default. It is a designer task to choose the local computing element and the connections of its free input-output ports. The possibility are then endless, even considering a single application.

2.2.2 Generic inter-cell computation

Despite what it can be seen in the Figure 2.3, the local logic element can perform different types of operations among those defined in Section 2.2. In fact, since the local logic elements has some free connections it can be used to perform also inter-row or inter-column logic operations.

Figure 2.5 shows an example of inter-row computation. Note that the computation is performed by the local logic elements of one of the two rows involved in the computation. The other one doesn't integrate logic, i.e. it is a simple memory row.

On the other hand, in Figure 2.6 it is depicted a configuration in which the FAs and HAs of an entire row are cascaded forming a RCA. Here also the local computing

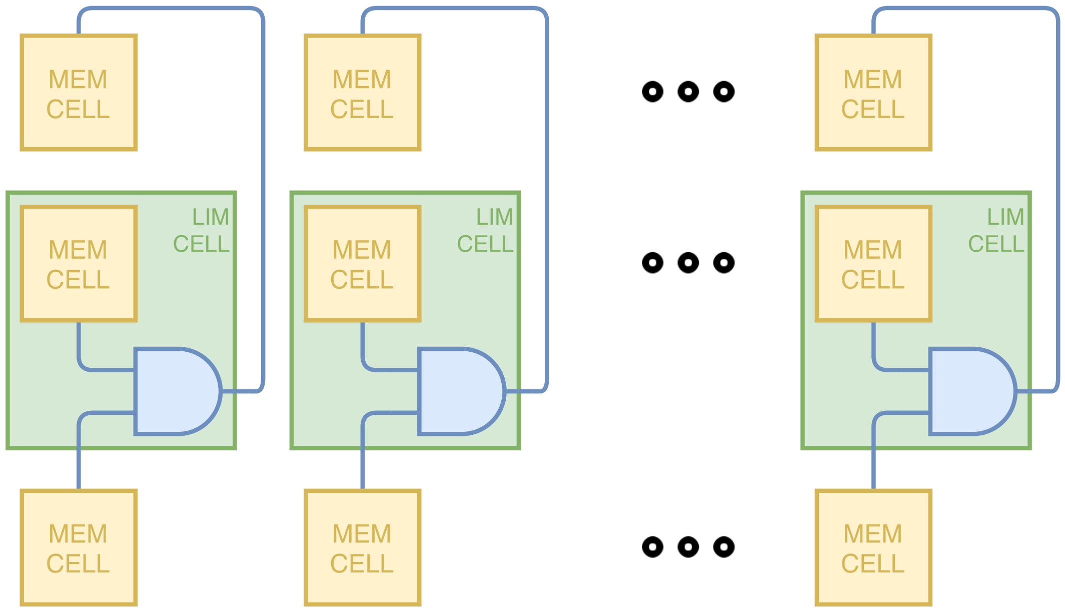


Figure 2.5: **An example of inter-row computation**. As it can be seen in the figure, the computation consists in a bit-wise AND between two adjacent rows. To do this the local computing element of one row is employed.

element is used to do computation that involves two or more cells.

In some cases it would be an understatement to divide between inter-row, inter-column, intra-row and intra-column computation since in a computation can coexist two or more computation type. In Figure 2.6 the addition is at the same time inter-row but also intra-row. All the cell of the row contribute to the final computation but at the same time the data involved in the computation comes from two different rows.

In some other cases, instead, the computing element is not inside the LiM cell. Suppose to have to sum up the bit of an entire row, or part of a row. Probably the smarter way is to compute the sum with a pop counter. The pop counter in Figure 2.7 does exactly this computation and it can be classified as intra row-computation because the data involved in the computation come all from the same row.

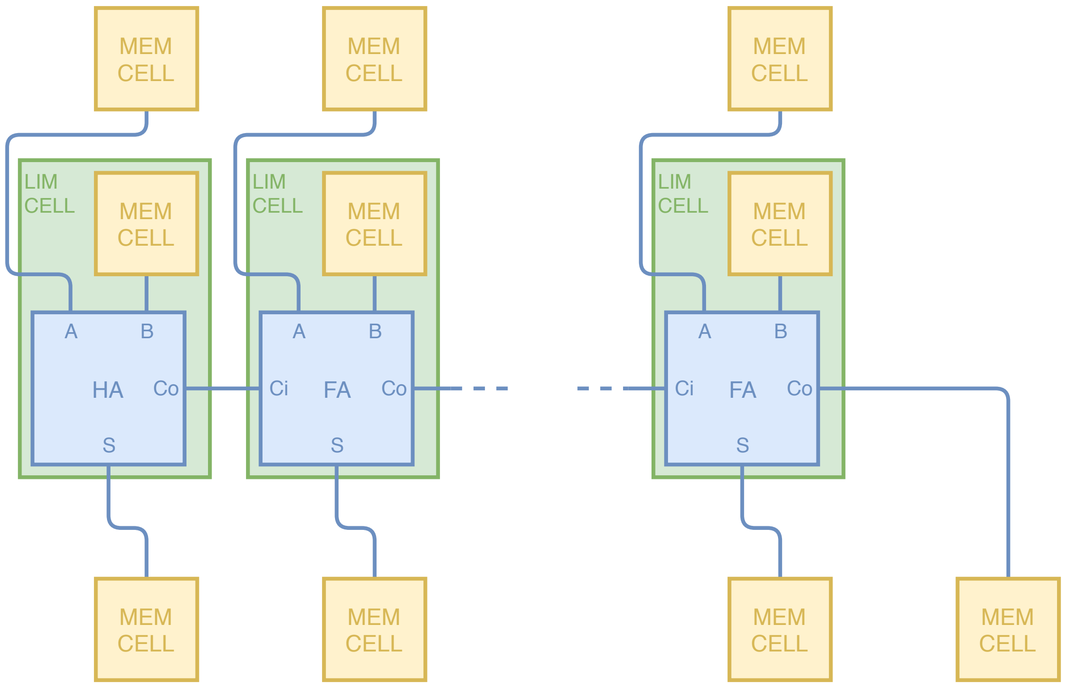


Figure 2.6: **An example of intra-row/inter-row computation**. As it can be seen in the figure, the computation consists in addition between two adjacent rows. To do this the local computing elements (FAs and HAs) of one row are cascaded to made up a ripple carry adder.

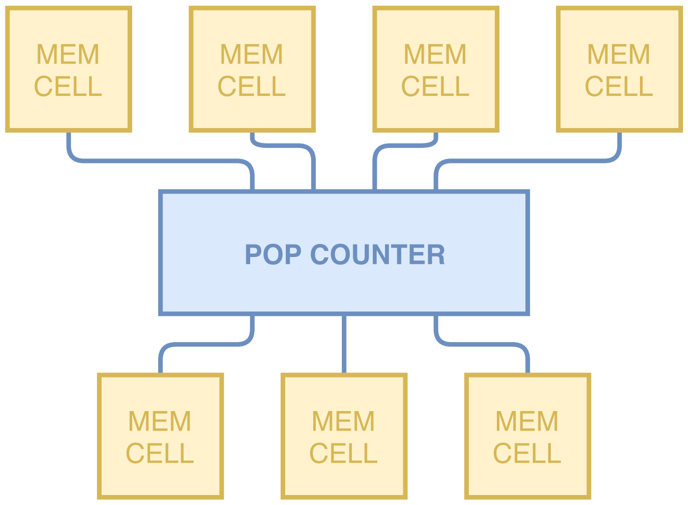


Figure 2.7: **An example of intra-row computation**. The data stored in a row are summed up by a pop-counter and stored in an other row. Only data coming from one single row are involved in the computation.

2.2.3 Interconnections complexity

The interconnections between logic elements are very important. They are fundamental to do some computations. It is also fundamental to handle them with care. In fact, as discussed in the previous chapters, the Logic-in-Memory paradigm aims to overcome the memory wall problem is some way.

One of the main principle is to shorten the distance between storage and computation and to reduce how often the data are moved. Every time a long interconnection is used to move data the advantages of in-memory computing are reduced.

It is then a good design practice to limit the number of interconnections and to shorten their length. Computation between near data are preferred with respect to computation involving distant data. Pay attention not to turn local interconnections into an internal bus. This leads to lose all the benefits due to Logic-in-Memory.

The willingness to limit data movement can also influence the way in which the data are stored in the memory the first time.

In some cases it is better to map some operations outside the memory, i.e. on the datapath block in Figure 2.1(b).

Chapter 3

Motivations

In this chapter it will be discussed the motivations that have led to the development of DExIMA.

In Chapter 1 some works are analyzed and classified according to the classification proposed in [1], while in Chapter 2 it is given a generic structure for in-memory architectures starting from a generic organization of the LiM unit, the *magic* memory proposed in that chapter.

Due to a large number of possible design choices some considerations have to be done. In Chapter 2 several computation possibilities have been analyzed such as intra-cell, local computation and so on. It is not very clear which are the advantages of one choices against one other. Even if it is clear that, it is difficult to quantify these advantages.

In digital design the simulators play this role, i.e. quantifying the performance and the difference between one design solution with respect to an other one. Moreover, it should be good to simulate the design at higher-level than the Register-Transfer Level (RTL).

The idea is to simulate at architectural level by selecting high-level modules and connecting them to build the model of the architecture. Then cycle-based simulation is to be performed in order to extract dynamic performance (manly dynamic energy). Note that at this moment it is not mandatory to verify the correctness of the algorithm execution, then it is not necessary to do a functional simulation.

Some architectural level simulator found in literature are exposed in the following.

3.1 Architectural level simulator

The field of architectural simulator is very large and cover multiple abstraction level.

Gem5 [15] is defined as *"a modular platform for computer-system architecture research"*. It provides 4 CPU models: a simple single CPI CPU, a detailed in-order CPU, a detailed out-of-order CPU and higher level model of CPU. The first three use high-level ISA description. Moreover Gem5 provides also an advanced GPU model.

The memory is simulated with an event-driven system including different level of caches, different memory configurations (DDR3/4 GDDR5, HBM1/2/3 and so on). In addition, Gem5 allows multi-core simulations.

Currently Gem5 supports different instruction sets such as Alpha, ARM, SPARC, MIPS, POWER, RISC-V and x86. A particular feature is the capability to simulate full systems which means that also the operating system can be simulated.

This simulator is very advanced but lack of some essential components to simulate architectures. In fact, the configuration of the hardware is limited by the simulator models. There's no way to describe and simulate module such as Logic-in-Memory cells or even simple logic circuit, but only well-known CPU architectures.

An other important simulator is *Wattch* [16]. Wattch is framework for architectural-level power analysis and optimization. This architectural simulator provides a configurable model of an out-of-order processor.

The purpose of this work is to simulate different software (assembly codes) and to evaluate power consumption at high-level. A lower level simulation achieves a better accuracy but the computation cost is very high.

In [16] authors use *SimpleScalar* which keeps track of which part of the processor is active at a given cycle. It provides a simulation environment for a 5-stage modern processor.

Also this simulator provides models for only a specific class of circuit, which is the out-of-order processors. However the simulation process, which consider active and non-active part of the circuit is interesting to estimate power consumption for in-memory architectures.

3.2 Simulator features

As discussed in previous section neither Gem5 [15] nor Wattch [16] meet the requirements of high-level simulation. Then, a new tool has to be designed.

The first step is to identify what are the features that must embed:

- configurability;
- ability to describe the execution of an algorithm;
- a sufficient number of available models;
- mantainability.

Configurability means that the model of the architecture has to be configurable. The easier way to achieve this goal is to configure the model from the outside, for example describing the circuit and the algorithm execution by means of some files.

The simulation has to be also based on the execution of an algorithm which has to be describe carefully. As mentioned before it is not important to verify the correctness of the algorithm, which is a different task with respect to the performance evaluation. Then the description of the algorithm has to be focused on the power and the delay resulting from the execution of a specific step of the algorithm.

The last required feature, i.e. the availability of a sufficient number of hardware models, is fundamental. Only in this way all the possible complex circuit can be described.

The maintainability is a crucial features: this tool has to be expandable. As far as the set of models are large, it may be not sufficient for some architectures. The possibility to implement a new model has to be guaranteed, organizing in a clever way the models inside the code.

The architectural blocks shown in Figure 2.1(b) need to be reflected by the tool. So the separation between LiM unit and datapath should remain also in the tool.

The bus is a different question. It has to be simulated carefully because in memory-intensive applications it affects the performance significantly.

Chapter 4

Bus Spice simulation

Today, in *von Neumann* architectures the *Bus* often represents the most power consuming element of the system. In state-of-the-art SoCs it consumes up to the 50% of total power consumption. Moreover, the technology scaling leads to a reduction of gates delay, making the bus the real bottle-neck of the *von Neumann* architecture. As mentioned in Chapter 1, the primary aim of the Logic-In-Memory paradigm is to overcome this bottle-neck.

In this perspective, an accurate estimation of the bus performance (area, delay and power consumption especially) is fundamental to populate the area-latency-power design space with the different configuration of the architecture. In other words, a good evaluation of bus performance allows the designer to make architectural choices based on reliable values. Unfortunately some phenomena, such as cross-talk, cannot be neglected. Due to the complexity of the circuit *SPICE* simulations are needed.

4.1 Electrical model

A bus is a group of parallel wires. In Figure 4.1 is shown the physical structure of the bus. Three wires ara shown but the structure can be scaled up to N wires. First the single wire model is considered computing the wire section resistance and capacitance. After that, the multiple wire model is developed taking into account the crosstalk capacitances.

4.1.1 Single wire model

The base element of the bus is the wire. A wire can be modelled with three parasitic characteristics [17]:

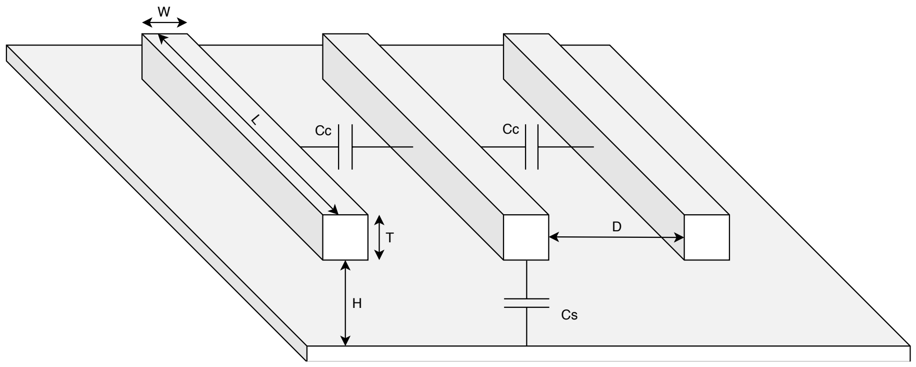


Figure 4.1: **Physical structure of the bus**. C_S is the total wire-to-substrate capacitance. C_C is the crosstalk capacitance between two adjacent wires. W, L and T are respectively the width, the length and the thickness of the wire respectively. D is the distance between two adjacent wires.

- *resistance* R (Ω);
- *capacitance* C (F);
- *inductance* L (H).

According to what Ismail et al. exposed in [18] the inductance is most of time neglected. Two figure of merit can be considered and they are mixed in an inequality:

$$\frac{t_r}{2\sqrt{LC}} < L_{wire} < \frac{2}{R}\sqrt{\frac{L}{C}} \tag{4.1}$$

where t_r is the rise time of the of the input signal of the CMOS bus driver, R, C and L are the resistance, capacitance and inductance per unit length respectively.

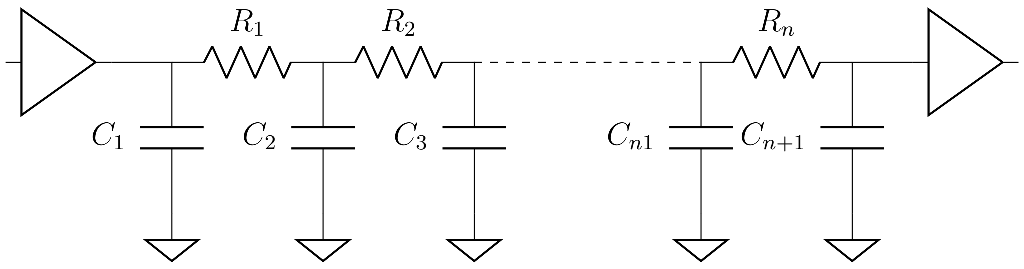


Figure 4.2: **Bus single line simulation circuit.** The model of single line bus is composed of a series of RC circuit.

L_{wire} is the length of the wire. If L_{wire} is inside the range, the inductance cannot be neglected. This range does not exist when the following inequality is satisfied.

$$t_r > 4\frac{R}{L} \tag{4.2}$$

Since wire length does not appear, the 4.2 is valid whatever the value of L_{wire}. The following model is based on the assumption that 4.1 is valid.

In order to compute R and C three parameters can be defined:

- *resistance per square* $R_\square$ *(Ω/square)*;
- *wire underside capacitance* $C_\square$ *(F/m)*;
- *wire edge capacitance* C_e *(F/m)*.

As shown in Figure 4.1 the single wire is defined by 4 physical dimensions: the width of the wire (W), the length of the wire (L), the thickness of the wire (T) and finally the distance between wire edge to substrate (D). The parameters $C_\square$ and C_e depend on the distance D.

The resistance of a wire section is given by the following equation:

$$R = R_\square \frac{L}{W} \tag{4.3}$$

The capacitance is the sum of two contributions: the capacitance between the bottom surface of the wire and the substrate (C_{S0}) and the capacitance due to the wire edges, known as fringe capacitance (C_F).

$$C_{S0} = C_\square WL \tag{4.4}$$

$$C_F = 2C_eL \tag{4.5}$$

So, the total substrate capacitance is given by the sum of these two contributions.

$$C_S = C_{S0} + C_F = L(C_\square W + 2C_e) \tag{4.6}$$

The model of the entire wire is a lumped element one. A number N of wire section are connected in series (with a final capacitor) and each one represent a wire section of length L/N. The Figure 4.2 shows the electrical lumped model of a single wire. In [17], Courtay et al. fix the number of sections to 3. In this way a precision of better than 5% is guaranteed.

4.1.2 Multiple wire model

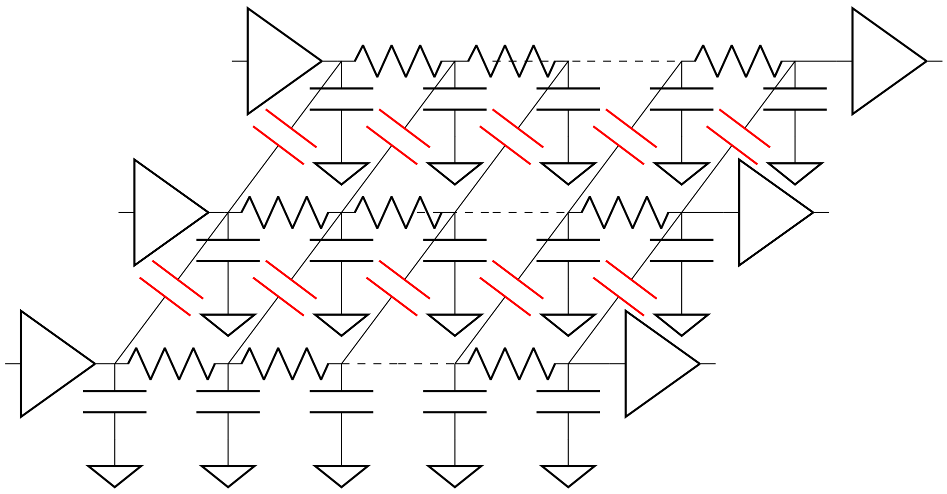


Figure 4.3: **Bus multiple wires simulation circuit.** In this figure is shown the generic form of multiple wires circuit.

The single wire model can be extended to derive the entire bus model. From electrical point of view the bus is not simply a composition of multiple independent wires. Since the crosstalk effects cannot be neglected, several parallel wires must be considered.

As shown in the Figure 4.1 the crosstalk capacitance is the parasitic capacitance between two adjacent wires. It depends on the facing area. The larger the lateral surface of the wire the greater the crosstalk capacitance. So the crosstalk capacitance is given by:

$$C_C = \varepsilon_0 \varepsilon_R \frac{TL}{D} \tag{4.7}$$

where $\varepsilon_0 \varepsilon_R$ is the dielectric constant of the material between the two wires (probably SiO_2) and T, L and D are the physical dimension shown in Figure 4.1.

The effect due to crosstalk can be summarized into three categories.

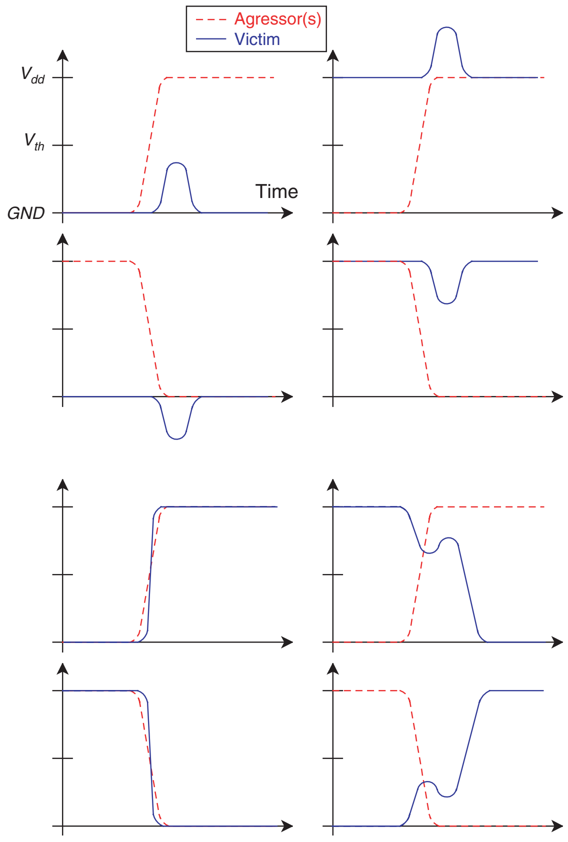


Figure 4.4: **Crosstalk capacitance effect**. In the first four graphs the victim is not switching while in the last four ones the victim is switching. The voltage change on the victim depends on the switch of the aggressor wire.

- **Noise**. The coupling can induce a voltage change on the wire due to a transition on the adjacent wire. In the first four graphs in Figure 4.4 is shown the effect of a switch on the aggressor wire. This leads to a voltage increase or decrease on the victim wire depending on the transition direction of the aggressor wire. There are two critical situations:
  - the victim is fixed at GND and a positive transition occurs on the aggressor;
  - the victim is fixed at VDD and a negative transition occurs on the aggressor.

  These two cases can lead to an error. Indeed, if the voltage increase is so large that the signal becomes greater than V_{TH} (the threshold voltage of the load buffer of the bus), the data bit is incorrectly recognized as 1 (instead of 0). Viceversa, if the voltage decrease is so large that the signal becomes smaller than V_{TH}, the data bit is incorrectly recognized as 0 (instead of 1). The first row - first column and the second row - second column graphs in Figure 4.4 show the first and the second case.

- **Timing performance degradation**. When victim and aggressor switch at the same time a voltage increase or decrease occurs on the victim. This leads to an increase or a decrease of propagation delay on the victim. Indeed, if the transition on the wires are in the same direction the propagation delay slightly decreases (look at the third and fourth graphs of the first column in Figure 4.4). On the other hand, if the transition on the wires are in the opposite direction, the propagation delay increases (look at the third and fourth graphs of the second column in Figure 4.4).

- **Greater power consumption**. Crosstalk capacitances means much capacitance involved in the transition. Since the power consumption is directly proportional to the capacitance the power consumption increases as the capacitance increases. The larger the crosstalk capacitances the greater the power consumption on the bus.

The multiple wire model is the starting point to estimate the entire bus delay

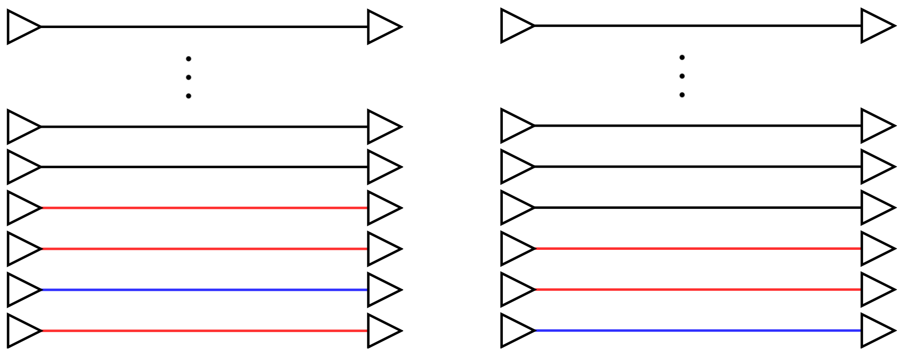

(a) Last wire aggressor (b) Second-to-last wire aggressor

Figure 4.5: **Wire aggressors on last and second-to-last wire of the bus**. The victim is the blue wire, while the aggressors are represented by the red wires.

and dynamic power. To better estimate the performance the model is extended to 5 wires, i.e. 2 aggressors per side. With regards to the last and the second-to-last wires the model is slightly different. The last wire has only two aggressors, because on one side there are no wires. The second-to-last wire has only three aggressors, two on a side and one on the other side. This is make more clear in the Figure ??. The three models will be 5-wires-model, 4-wires-model and 3-wires-model.

4.2 Spice simulations

In order to estimate the bus performance, Spice simulations have been executed. As discussed in the previous section the circuits to simulate are essentially 3: the 5-wires, the 4-wires and the 3-wires. A C++ software runs parametric simulations.

4.2.1 Spice model

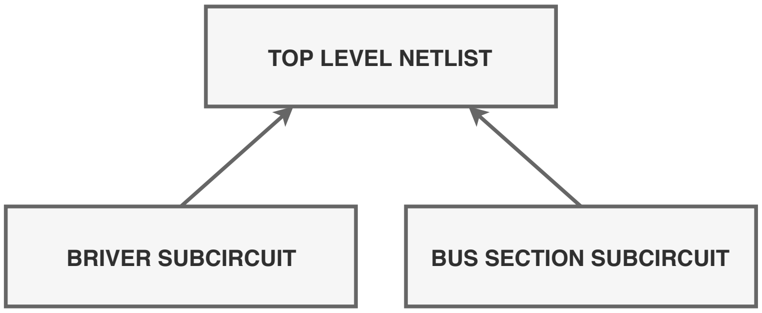


Figure 4.6: **Bus Spice simulation structure**. The driver netlist defines the driving circuit of the bus. The bus-section netlist defines the RC single bus section with cross talk capacitances. The top level netlist contains the instance of the two subcircuits and the measure command.

NGSpice, a free open-source Spice version, has been used. The general netlists organization is shown in Figure 4.6.

There are two subcircuits, bus driver and bus section, defined in two different netlists. The first one is simply the bus driver, implemented with two inverters. The second one is the bus section. It contains the line resistance, the substrate total capacitance for each wire and the crosstalk capacitances between wires. The last one, the top level netlist is the structural description of the circuit. A number N of bus section are connected together. The bus drivers are connected to the input of the first bus section. Bus drivers are also used as bus loads connecting them to the outputs of the last bus section.

Since the bus performance depends on the the transition on the aggressor wires, a complete simulation has to be performed. In other words every bus transition has to be simulated. Considering a 5-wires model the number of simulation to be performed should be $2^5 \cdot 2^5$ (number of possible initial state of the bus times number of possible final state of the bus). Assuming that the noise due to crosstalk is not enough to modify the data bit which is propagating through the victim wire (see Section 4.1.2), the number of simulation can be halved. In fact the transition in which the data on the victim wire is constant has not to be considered.

The number of simulations to be performed can be generilized with respect to the number of wire considered.

$$N_S = \frac{2^{2N_{wire}}}{2} = 2^{2N_{wire}-1} \tag{4.8}$$

Since the simulations have to be performed for 5-, 4- and 3-wires models the total number of simulations is pretty high.

$$N_S^{(tot)} = N_S^{(5-wires)} + N_S^{(4-wires)} + N_S^{(4-wires)} = 512 + 128 + 32 = 672 \tag{4.9}$$

A C++ program has been developed to generate the netlists and to run the simulations in a parametric fashion.

4.2.2 C++ implementation

The general structure of the program is depicted in Figure 4.7. A class called `BusSimulation` provides several methods to read the configuration file, to generate the netlists and to run the simulation.

- `parse_parameter_file(const string file_path)` parses an input file containing the bus and simulation parameter which are:
  - number of bus lines;

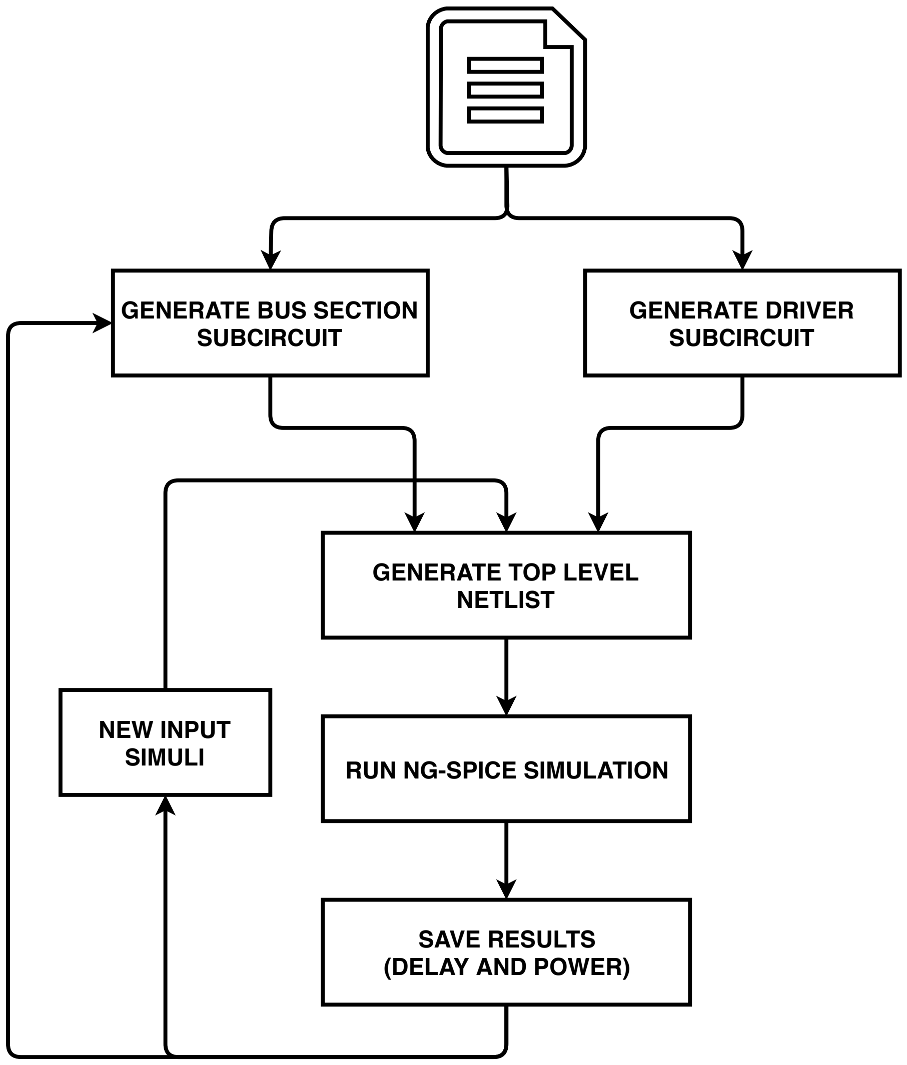

GENERATE BUS SECTION SUBCIRCUIT
GENERATE DRIVER SUBCIRCUIT
GENERATE TOP LEVEL NETLIST
NEW INPUT SIMULI
RUN NG-SPICE SIMULATION
SAVE RESULTS (DELAY AND POWER)

Figure 4.7: **C++ software graph**

– number of section in which the bus is divided;

– wire length;

– wire width;

– wire thickness;

– wire distance from substrate;

– distance between two adjacent wires;

– input signal rise time;

  - input signal fall time;
  - input signal pulse duration;
  - simulation time;
  - bus supply voltage;

- `compute_electrical_parameter()` computes bus electrical parameters such as wire resistance, total substrate capacitance and crosstalk capacitances for a single bus section;

- `generate_inverter_subckt(const string file_path)` generates the bus driver subcircuit netlist;

- `generate_bus_segment_subckt_5_wire(const string file_path)` generates the 5-wires bus segment;

- `generate_top_netlist_5_wire(const string file_path, uint8_t initial_bus_state, uint8_t final_bus_state)` generates the top level netlist containing the 5-wires bus segments, the bus drivers and loads and the voltage sources according to the initial and final bus states;

- `generate_bus_segment_subckt_4_wire(const string file_path)` generates the 4-wires bus segment;

- `generate_top_netlist_4_wire(const string file_path, uint8_t initial_bus_state, uint8_t final_bus_state)` generates the top level netlist containing the 4-wires bus segments, the bus drivers and loads and the voltage sources according to the initial and final bus states;

- `generate_bus_segment_subckt_3_wire(const string file_path)` generates the 3-wires bus segment;

- `generate_top_netlist_3_wire(const string file_path, uint8_t initial_bus_state, uint8_t final_bus_state)` generates the top level netlist containing the 3-wires bus segments, the bus drivers and loads and the voltage sources according to the initial and final bus states;

- `run_netlist(const string file_path, const string out_file_path)` runs the spice simulation and saves the results in an the output file;
- `parse_bus_delay(const string out_file_path)` returns the bus propagation delay parsing it from the simulation output file;
- `parse_driver_current(const string out_file_path)` - it returns the integral of the current parsing it from the simulation output file.

Chapter 5

DExIMA: a first overview

```
################################################################################
#                                                                              #
#                                    DExIMA                                    #
#                                                                              #
#                  Design Explorer for In Memory Architectures                 #
#                                                                              #
#                                                                              #
#                       Developed by Nicola Piano @ PoliTO                     #
#                                                                              #
################################################################################
```

DExIMA (**D**esign **Ex**plorer for **I**n **M**emory **A**rchitectures) is a powerful tool developed to analyze the dynamic performance of a given architecture by means of execution-based performance computation. Moreover, the tool is designed manly for In-Memory-Architectures (as the acronym suggests) which is the new frontier for an increasingly number of application.

As discussed in the previous chapters this tool has to be developed with some specific features: it must be configurable and dynamic.

Configurable means that the designer has the ability to describe its personal architecture. Some mathematic models describe the most frequently used digital circuits performance. In this way a given architecture can be modelled.

On the other hand, it has to be dynamic which means that the estimation of the performance is done taking into account what the circuit has to perform to execute a specific algorithm. To be more clear, the execution time as well as the dynamic power of the circuit depend on the operations executed by the circuit and not only on the circuit structure.

Moreover, suppose to have to estimate performance of a LiM-based architecture. In this architecture probably a standard logic circuit is placed side by side by a Logic-in-Memory block. A bus connects the logic with the memory. It stands to reason that the presence of the Logic-in-Memory block allows a lower number of read/write memory operations and than a lower bus usage with respect to a standard von Neumann architecture in which the memory acts only as storage element and the computation is all performed outside the memory.

Obviously a static analysis, not based on the execution of algorithm steps, doesn't allow to estimate the difference between the two implementations. The main goal of this estimator is to evaluate the difference between a standard von Neumann implementation and a variety of LiM-based solutions. For these reasons the tool has to perform dynamic analysis.

To ensure the first main feature, i.e. the configurability, DExIMA presents an interface, i.e. a *DExIMA frontend*. In other words the core, called the Simulator Core is surrounded by a sequence of classes aim to take some external informations and instruct the Simulator Core to reconfigure itself according to these external informations. These external informations are provided to DExIMA by means of two different files. A much detailed description is provided in Chapter 6.

The second main feature, i.e. the ability to perform dynamic analysis, is ensure by the structure of the Simulator Core, i.e. the *DExIMA backend*. The Simulator Core can be defined as a cycle-based performance evaluator in which the cycle is an execution step. Every step the simulator evaluates which elements takes part in computation and computes the dynamic performance according to this analysis.

5.1 Structure

In Figure 5.1 is shown a high-level structure of DExIMA. The front-end is based on three compilers: the *Architecture Compiler*, the *Logic-in-Memory Compiler* and the *ASIC Operation Compiler*. The back-end is composed of the *Simulator Core* and the *Hardware Models*. In particular the Simulator Core makes use of the Hardware Models to perform both static and dynamic analysis on the circuit.

In the following two chapters a more detailed description of both the front-end and the back-end is provided.

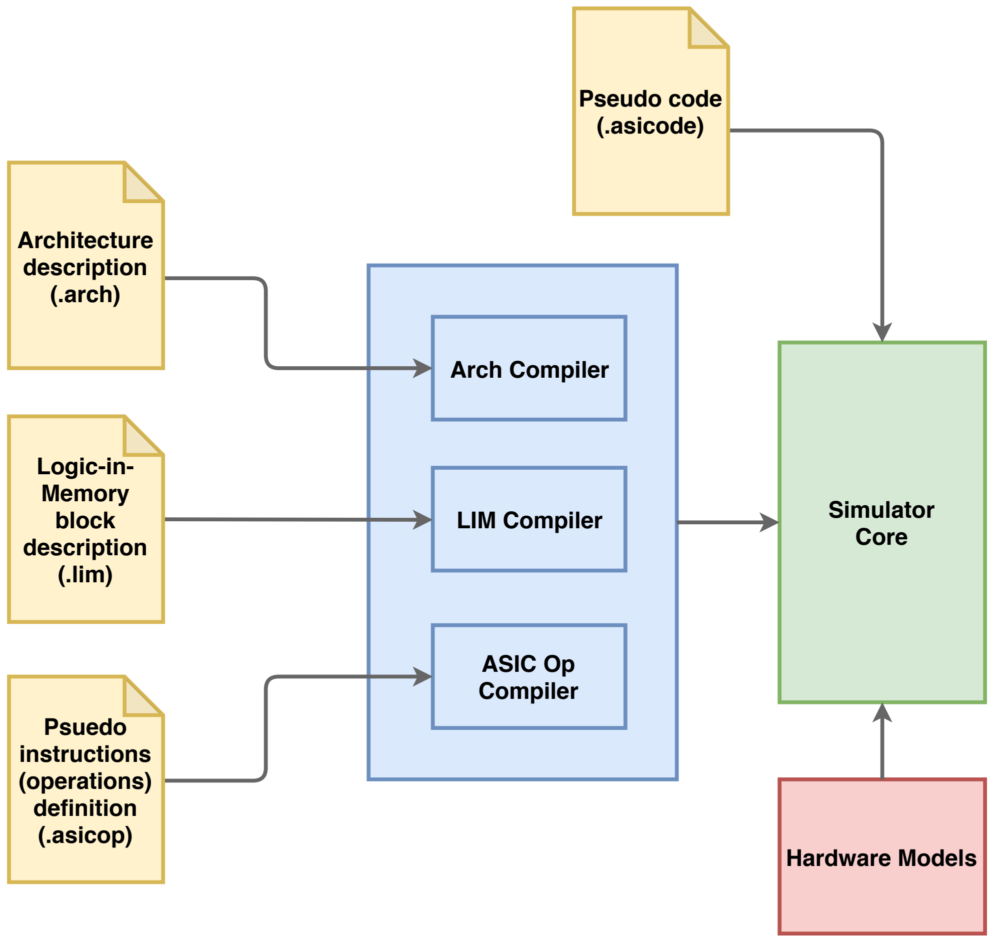

Pseudo code (.asicode)
Architecture description (.arch)
Arch Compiler
Logic-in-Memory block description (.lim)
LIM Compiler
Simulator Core
Psuedo instructions (operations) definition (.asicop)
ASIC Op Compiler
Hardware Models

Figure 5.1: **DExIMA high-level structure**. Four files, highlighted in yellow, contain hardware description and algorithm description. The interface, highlighted in light blue, connects the simulator core with these four files. The simulator core, in green, executes the pseudo code and extracts the performance from the hardware models.

5.2 Mode of operation

The first step is to design an architecture by hand. After that it is necessary to write down the description of the hardware with a particular syntax. The description is purely structural. The declaration of the hardware models is followed by the definition of the interconnections between them.

Same goes for the Logic-in-Memory block description. In this case both definitions of memory structure and logic structure are given. The logic inside every cell is selected among elementary logic circuits such as *NOT*, *AND*, *NAND*, *OR*, *NOR*, *XOR*, *XNOR*, *Full Adder* and *Half Adder*. The description of inter-cell, inter-columns and inter-rows logic is described exactly in the same way the out-of-memory logic elements are described.

The further step is oriented to make the tool dynamic. As discussed above the performance depends on the execution of the algorithm. To achieve this goal a new concept is introduced: the *pseudo instruction*.

5.2.1 Pseudo instruction

The concept of the *pseudo instruction* derives from the need to describe the steps of the algorithm. Since every processor has its own instruction set, a generic architecture described with DExIMA must have its own pseudo instruction set. The definition of these pseudo instructions are defined in two files: the `.asicop` file for the pseudo instructions related to the out-of-memory hardware and the `.lim` for the in-memory pseudo instruction. The `.lim` file contains also the structure of the LiM-block.

In this way the functionalities of the circuit are defined and the algorithm can be described by listing a sequence of these pseudo instructions. Both the ASIC Operation Compiler and the LiM Compiler parse the `.asicop` and the `.lim` files to populate the pseudo instruction set.

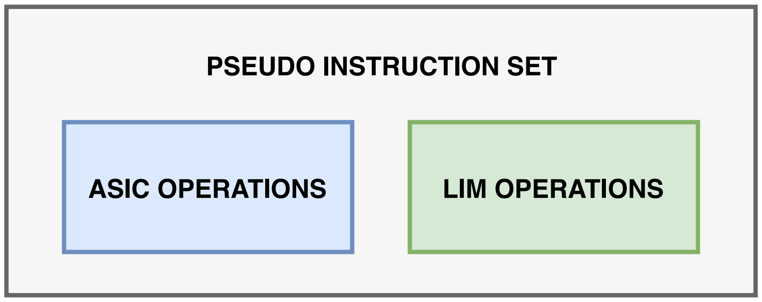


Figure 5.2: **DExIMA pseudo instruction set**. The pseudo instruction set is composed by both the ASIC Operations and LiM Operations.

Chapter 6

DExIMA front-end: the three compilers

In this chapter a full description of the simulator front-end is presented.

A number of C++ classes make up the so called *Simulator front-end*. Since the simulator has been designed to be configurable, the hardware configuration and the description of the algorithm represent the input data for DExIMA. Hence the need to develop the simulator interface.

The Figure 6.1 shows the structure of the front-end. Three compilers, for the three input files, produce data for the Simulator Core.

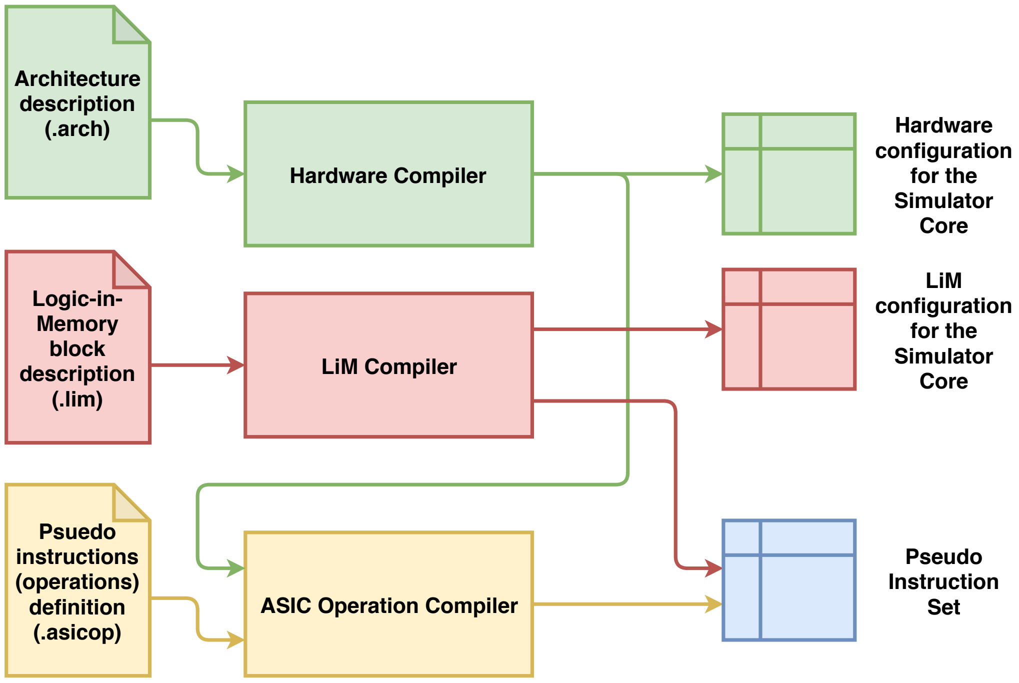

Architecture description (.arch)
Hardware Compiler
Hardware configuration for the Simulator Core
Logic-in-Memory block description (.lim)
LiM Compiler
LiM configuration for the Simulator Core
Psuedo instructions (operations) definition (.asicop)
ASIC Operation Compiler
Pseudo Instruction Set

Figure 6.1: **DExIMA front-end structure**.

6.1 Out-of-memory hardware description

The out-of-memory hardware configuration is described in the .arch file. This file is structured in two sections: the init section and the map section.

The first one, the init section, contains the declaration of the hardware module present in the circuit that has to be described. These modules can be selected among the available models, which are the most frequently used digital circuits such as the standard logic gates (AND, NAND, OR, NOR, XOR, XNOR), full-adder, half-adder, ripple-carry-adder, flip-flop and so on.

Each and every model needs some parameter. In Table 6.1 a list of models is represented together with their related parameters. The initialization needs also to pass these parameters to the Simulator in order to set the models as a consequence.

Model	Keyword	Parameters	Input ports	Output ports
AND	AND	# of inputs	IN1, IN2, ...	OUT
NAND	NAND	# of inputs	IN1, IN2, ...	OUT
OR	OR	# of inputs	IN1, IN2, ...	OUT
NOR	NOR	# of inputs	IN1, IN2, ...	OUT
XOR	XOR	-	IN1, IN2	OUT
XNOR	XNOR	-	IN1, IN2	OUT
Full Adder	FA	-	A, B, Cin	S, Cout
Half Adder	HA	-	A, B	S, Cout
RCA	RCA	# of bits	A, B, Cin	S, Cout
Multiplier	MUL	# of bits	A, B	P
D Flip-Flop	DFF	-	D	Q
Register	REGISTER	# of bits	D	Q
Multiplexer	MUX	ways, # of bits	IN1, IN2, ... , S	OUT
Decoder	DECODER	# of input bits	IN	OUT
Up Counter	UPCOUNTER	# of bits	EN	CNT
Pop Counter	POPCOUNTER	# of inputs	IN1, IN2, ...	OUT

Table 6.1: **Model and related parameters and ports**. The table shows the list of the available models and their related parameters, input and output ports.

The second section, the map section, contains the mapping between the modules. Obviously the available ports, for some of the models depends on the parameters mentioned before. A 4-inputs NAND will have 4 input ports called `IN1`, `IN2`, `IN3` and `IN4` and an output port called `OUT`. A 4-bits 2-way Multiplexer will have 2 4-bits inputs called `IN1` and `IN2`, a 2-bits input called `S`, the selector bits, and an output port called `OUT`.

6.1.1 A first .arch input file example

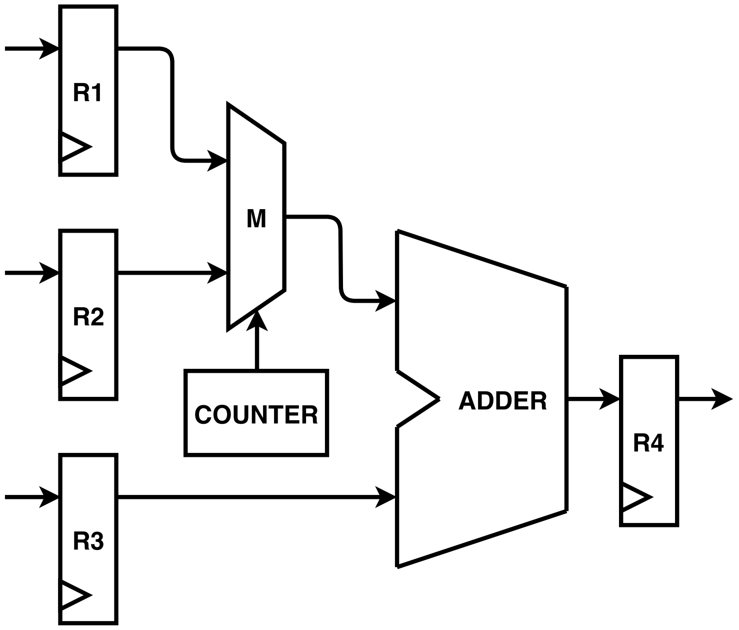


Figure 6.2: **A simple circuit example to implement in DExIMA**. A simple circuit made up of 4 registers, a multiplexer, a counter and an adder.

In this section a very simple circuit is described with DExIMA. The Figure 6.2 shows a simple circuit made up of 4 Registers, a Multiplexer, a 1 bit counter and an RCA.

As discussed before, the init is the section in which the modules are declared and the parameters are passed to the simulator. According to the Table 6.1 the register, the RCA and the counter models takes one parameter, the number of bits, while the multiplexer takes two parameters, the ways and the number of bits. The init section of the .arch file for this circuit is shown in the following.

```
begin init

        REGISTER R1(8)
        REGISTER R2(8)
        REGISTER R3(8)
        REGISTER R4(8)
        MUX M(2, 8)
        UPCOUNTER COUNTER(1)
        RCA ADDER(8)

end init
```

The syntax is very simple. To declare a module the designer has to write the model keyword followed by the instance name and the possible parameters. In the first declared module, the register R1, REGISTER is the model keyword for the register, R1 is the instance name and the 8 in brackets is the number of bits.

The second section, i.e. the map section, is used to interconnect the modules defined in the init section. Suppose to have the same circuit, the one shown in Figure 6.2. The map section related to this circuit, according to the instance names declared in the init section, will be:

```
begin map

        R1.Q -> M.IN1
        R2.Q -> M.IN2
        COUNTER.CNT -> M.S
        M.OUT -> ADDER.A
        R3.Q -> ADDER.B
        ADDER.S -> R4.D

end map
```

The operator `->` allows to connect an output port of the left module to an input port of the right module. In order to indicate a specific port the designer has to write down the instance name of the module followed by the dot, i.e. the port operator, and then by the port name. The module ports are the ones shown in Table 6.1.

The mapping of the modules is very important for a correct estimation of the performance of the circuit. Since the dynamic power as well as the delay is strongly affected by the fanout of each logic gates, a precise circuit structure description allows the Simulator Core to compute accurately both these metrics.

Knowing that, it is clear that a smart way to annotate these data would be nice. The solution was to keep track of these data in a list of objects of type

CompiledModule.

6.1.2 CompiledModule class

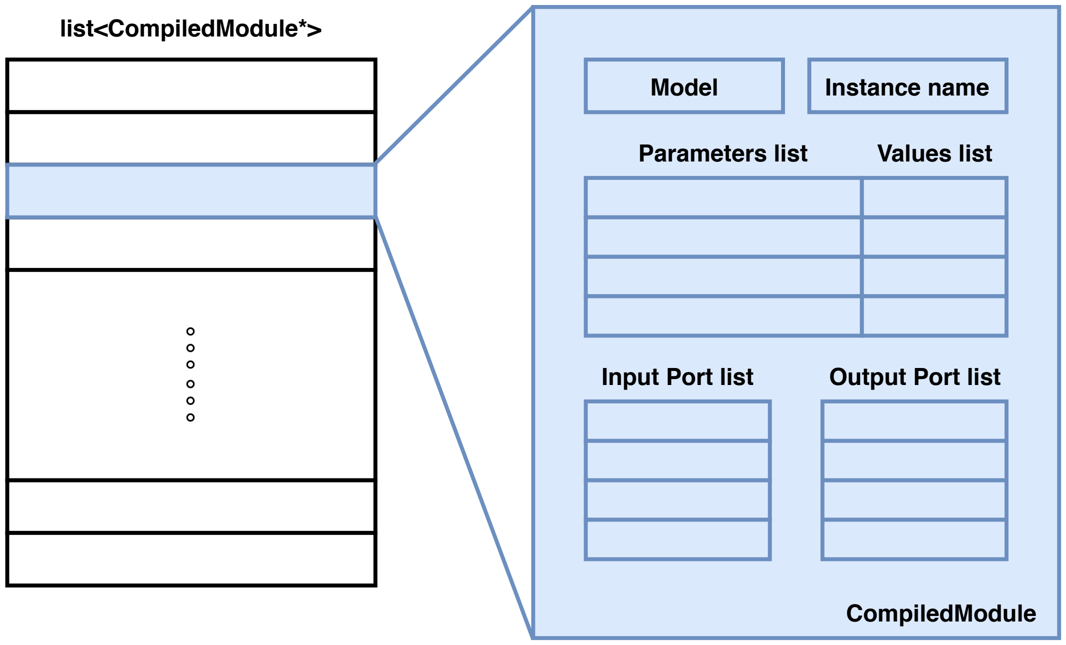


Figure 6.3: **CompiledModule overview**. The fields are used to take note of module configuration. The parameters are sometimes needed to generate the right number of ports.

The model to simulate, its parameters and its input/output ports are the data contained by an object of type `CompiledModule`. CompiledModule connects the front-end to the back-end of DExIMA. A brief overview of what a CompiledModule object contains is shown in Figure 6.3. On the left side of the figure a list of these objects is shown. According to this list the Simulator Core generates the model of the circuits. As shown by the figure, a CompiledModule object has several fields:

- model name;
- instance name;
- parameters;
- input and output ports.

According to the Table 6.1 the model name is the keyword. The instance name is the one declared in the init section. The parameters are coded with strings: a list of couple name-value are created when the module is initialized according to the passed parameters. For the input and output ports a different approach has been adopted. Two classes called CompiledModuleInputPort and CompiledModuleOutputPort are defined in order to better handle the fanout computation for each output ports.

6.1.3 CompiledModuleInputPort and CompiledModuleOutputPort classes

`CompiledModuleInputPort` and `CompiledModuleOutputPort` classes are used inside the CompiledModule class to keep track of the connections between the current module and the other modules in the design.

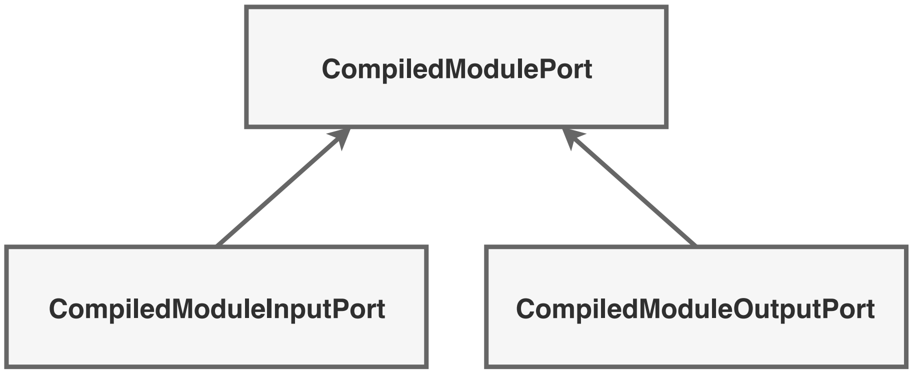


Figure 6.4: **CompiledModuleInputPort and CompiledModuleOutputPort inheritance**. CompiledModuleInputPort and CompiledModuleOutputPort classes inherit from CompiledModulePort class all the methods and variables. Only the `set_connection()` method implementations differs from each other.

The structure of both classes is similar. In fact they inherit from the same superclass, i.e. the CompiledModulePort. Each and every port is marked by a name and the bit width. The `set_connection()` method for the CompiledModuleInputPort class simply set the connection as true, while for the CompiledModuleOutputPort class also increment the fanout. The variable fanout is present only for the output ports and it is recorded as equivalent NAND input capacitance unit. A much exhaustive explanation will be given in Chapter 7. In simple words every time an output port is connected to an input port the fanout of that port is incremented.

6.1.4 Compiler

The .arch file is compiled by the *Hardware Compiler*, or simply *Compiler*. The Compiler parses the input file and generates a list of pointers to CompiledModule objects. There is a one-to-one match between the module defined in the init section of the source file and the elements of this list.

The output of the Compiler is precisely this list which is forward to the Simulator Core. It is also passed to an other element of the front-end which is the ASIC Operation Compiler described later in this Chapter.

The steps to compose this list are very simple. First of all the init section is parsed row by row by creating the CompiledModule objects and pushing them in a dynamic list. At this moment the model name, the instance name, but even the parameter values are known.

Some errors can be generated by the Compiler in case of incorrect syntax. In order to better keep track of these error, a class called `CompilerError` has been developed. This class is responsible to print out a message in case of syntax or semantic error. Common errors can be wrong number of parameters or unknown model name. According to the parameters, each CompiledModule object can create its own CompiledModuleInputPort and CompiledModuleOutputPort objects.

After that, the Compiler parses the map section. For each and every mapping there are a left and a right port. The left port is always an output port while the right port is always an input port. The port is indicated using the instance name of the module and the port name. The Compiler reads the left port, searches the instance name among the CompiledModule objects and calls the `set_connection()` method on the port. The same procedure is followed for the right port. An error is generated when the left or the right port doesn't exist.

6.2 Logic-in-Memory unit description

The Logic-in-Memory unit (LiM unit) is what differentiates a von Neumann architecture from an in-memory architectures. DExIMA, as suggests the acronym, is first and foremost a tool to estimate in-memory architectures performance. The description of this block is therefore critical.

A special file contains the configuration of LiM array. As described in the Chapter 2 this unit may be rather complicated. The cells may contain simple logic, such as OR-gate, AND-gate and so on. The content of an entire row of memory may be the inputs of a pop counter. A column-wise AND can be then stored on an other column. These are only few examples of what a LiM unit can do and how it is made up.

The file is compiled by a second compiler, called the *LiM Compiler*. The description of the LiM unit has to be quite accurate. The file is structured in 5 sections: init, memdef, cells, map and operation.

The first one, i.e. the init section, is dedicated to defined those modules that perform intra-row, intra-column and every non-local logic. The syntax is identical to the init syntax for the Hardware Compiler. Also the available modules are the same (see Table 6.1).

The memdef section is focused on the memory definition. Here the memory size is declared together with the number of ports and the instance name.

In the third section it is defined the local structure of each cell. A cell can embed an AND gate or a full adder or other simple digital circuits or can be even left without computing capabilities.

The fourth one is similar to the map section of the .arch file. The interconnections between modules and memory cells are defined here. A deep difference lays in how the interconnections are interpreted. A more detailed description will be provided later in this Chapter.

The operations section is very different from the other ones. While the previous sections are oriented to the hardware description, the operations section is focused on what the LiM unit can perform, i.e. on the pseudo instruction that can execute.

6.2.1 Memory array description

Suppose to have a simple LiM array similar to the one depicted in Figure 6.5. The first parameters to define is the memory size and the memory ports. The memdef section includes these informations.

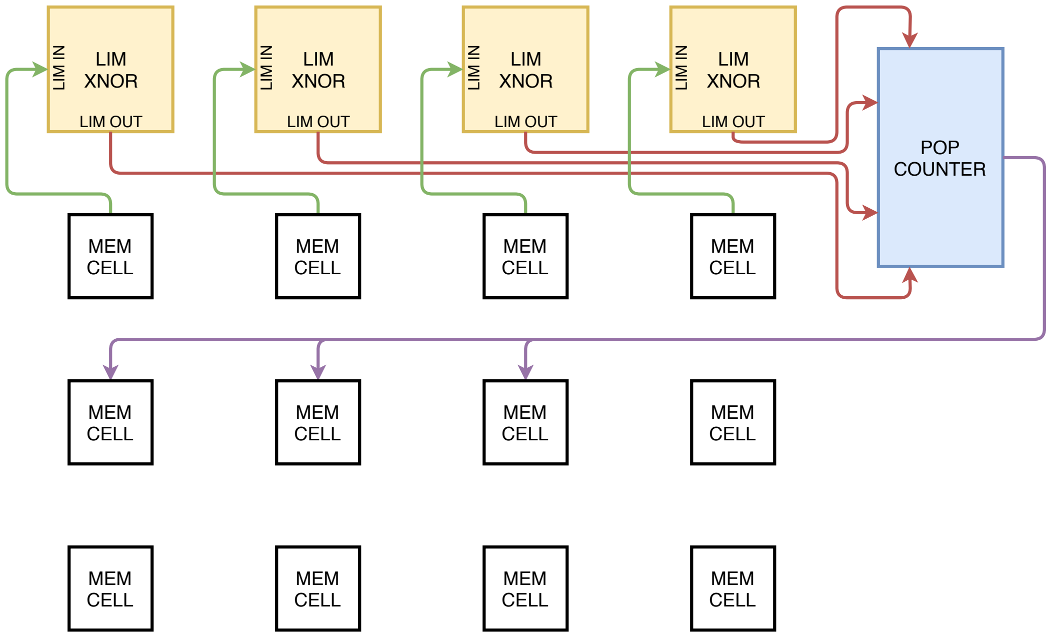


Figure 6.5: **A simple example of LiM unit**. The first row cells embed XNOR-gates which performs bit wise XNOR between the first and the second rows. The outputs have been summed up with a pop counter and then the result is stored in the third row.

An example of memdef section is shown below.

```
begin memdef

        row = 4
        column = 4
        read_port = 1
        write_port = 1
        ports_bit_width = 4

end memdef
```

Essential data are collected in this section. The memory size is expressed as rows and columns number. The memory interface is described with four parameters: number of read ports, number of write ports, number of read/write ports and their bit width.

In this case the memory size is 4x4 bits. A so small size is just a matter of ease of drawing. A bigger memory has been difficult to represent in a schematic such as the one in Figure 6.5. A further optional parameter is the instance name of the LiM unit.

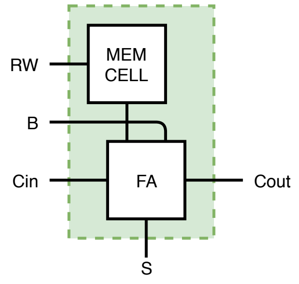
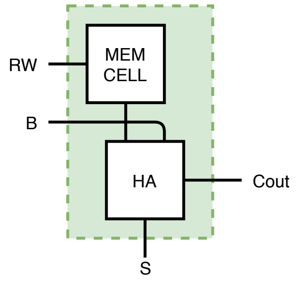
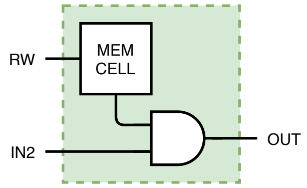
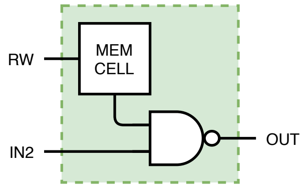
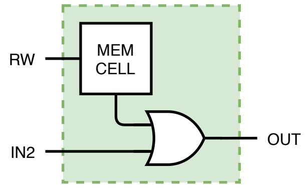
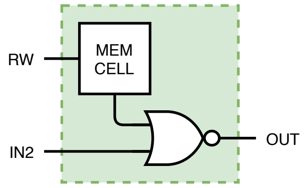
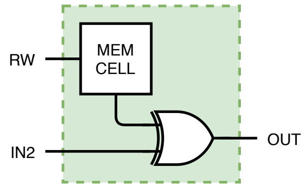
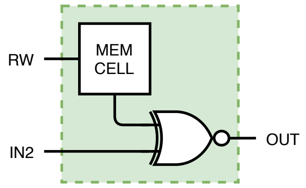
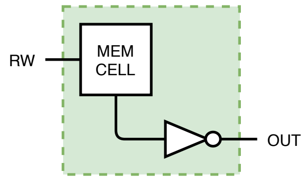
lim-fa			lim-ha
lim-and			lim-nand
lim-or			lim-nor
lim-xor			lim-xnor
lim-not			

Table 6.2: **A brief summary of local computing elements**. In this table the available intra-cells LiM are shown together with the their related keywords.

An other fundamental section is the cells section. Once defined the structure of the memory (size and input/output connectivity) the next step is to define the intra-cell logic. The Table 6.2 shows the possible intra-cell logic. In addition to the standard logic gates such as AND, OR, XOR and so on, full adder and half adder are available LiM-cell models. In the cells section the designer can set the cell model among these ones. As regards the example in Figure 6.5, all the cells are simple memory cells except for the first row. The latter is made up of XNOR lim cells. The circuit structure of this cell is shows in the second elements of the fourth row in Table 6.2.

The associated keywords are used in the cells section as can be seen in the following example code.

```
begin cells

        (0, 0 to 3).lim-xnor

end cells
```

The reference to the cells position is expressed into the round brackets. The first one is the row index while the second one is the column index and a comma separates them. The index can be expressed as a single value or as a range. In the example above the row index is a single value while the column index is a range. This piece of code represents exactly what is shown in Figure 6.5. The first *0* stands for the first row of the memory. The range *0 to 3* represents the columns from 0 to 3.

6.2.2 Inter-cells logic description and in-memory interconnects

Intra-columns, intra-rows and other degrees of Logic-in-memory fall in this category. In order to describe this design solution a different approach has been adopted, an approach very similar to one used to describe out-of-memory hardware.

The first section of a .lim file is the init section. The structure is the same as the init section of an .arch file. Again suppose to have the LiM unit represented in Figure 6.5. A 4-bits pop counter perform a row-wise bit counting. The following code represents the init section in this case.

```
begin init

        POPCOUNTER bit_counter(4)

end init
```

As well as in the .arch file, a map section is present also here. In this case the in-memory interconnects are described. The syntax is similar to previous map section with some differences. This is due to the fact that the whole of the available ports include also the input and output ports of the inter-cell logic elements as well as the direct connection to the memory cell itself to read/write the content. The code below is the map section referred to the LiM unit in Figure 6.5.

```
begin map

        (1, 0 to 3).RW -> (1, 0 to 3).IN2

        (0, 0).OUT -> bit_counter.IN1
        (0, 1).OUT -> bit_counter.IN2
        (0, 2).OUT -> bit_counter.IN3
        (0, 3).OUT -> bit_counter.IN4

        bit_counter.OUT -> (2, 0 to 2).RW

end map
```

Selecting a specific cell or a group of cells has the same syntax of the cells section. The contribution of the interconnect on the power and delay performance of the LiM unit is annotated in some objects of type CompiledInterconnect. A much exhaustive description is provided later.

6.2.3 CompiledLimCell and CompiledLimUnit class

According to the Figure 6.1 an object of type `CompiledLimUnit` is one of the outputs provided by the LimCompiler. The Figure 6.6 shows its structure. The instance name, the memory size and so on are recorded here.

The first focus is on the LiM array. It is a 2D array of pointers to objects of type CompiledLimCell which will be discussed later. For now it's enough to know that an object of this type contains the local logic element of the cell together with its output interconnects. The array has a number of rows equal to the number of memory rows, as well as the number of columns is equal to the number of memory columns. An element points to a non-null object only when some logic is present in the cell.

Again consider the example in Figure 6.5: the first row of the array will be not-null pointers while the others three rows will be actually null. Each of the elements

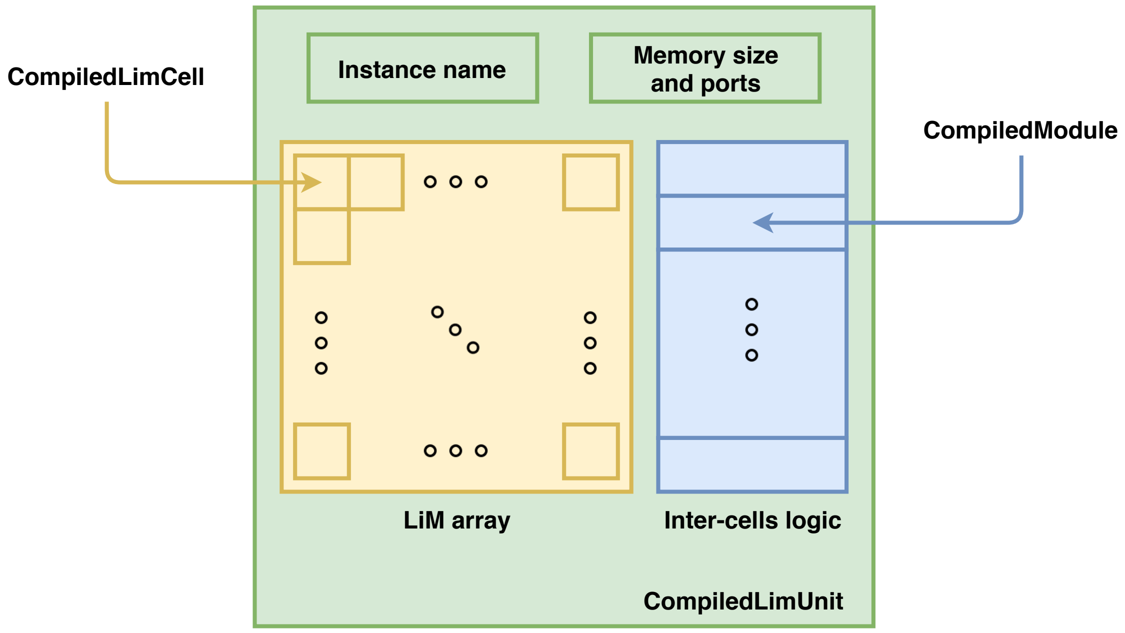


Figure 6.6: **Structure of CompiledLimUnit class**. The class contains informations about the memory itself but also includes in-memory logic description. The LiM array keeps the inter-cell logic description while a list of CompiledModule objects contains inter-cells logic elements.

of the first row will have also an interconnects defined.

The second element to focus on is list of CompiledModule objects. This is no different from the list described in Section 6.1.2. Every time an inter-cells module is defined in the init section a CompiledModule object is created and pushed into this list. An interconnects on the output port is defined here too.

Going more in detail a description of the `CompiledLimCell` class explains how intra-cell logic description is bring to the Simulator Core. A variable called `_cell_type` can assume a value among the available LiM cell described in Table 6.2. According to this value a CompileModule object is created. A further information is contained here that is the output interconnects.

6.2.4 CompiledInterconnect class

`CompiledInterconnect` objects describes the interconnects inside the LiM unit. Every time intra-cell logic element output is connected to any input ports an object of this type is created. The length of the interconnects is expressed by x-y coordinates of starting point and end point. The x coordinate represents the column number. On the contrary, y coordinate indexes the row.

Being unaware of memory physical dimensions, it is impossible to estimate a physical length of the interconnect. That's why the length in annotated as starting cell position and end cell position in the LiM array. Later the Simulator Core will compute the effective length of the interconnect according to the height of the rows and the width of the columns.

6.2.5 Logic-in-Memory pseudo instructions definition

The last section of a .lim file is the operations section. While in the previous section the description has involved the hardware structure, this section describes what the hardware can do, defining the so called LiM pseudo instructions.

As discussed in Chapter 5, the operations, or psuedo instructions, form the so called pseudo instruction set. This set is composed of the Logic-in-Memory pseudo instructions and the ASIC pesudo instructions which will be discussed later in this Chapter.

An example of code is reported below.

```
operation convolute: (0, 0 to 3).lim, bit_counter; [(0, 0 to 3).lim, bit_counter]
```

The keyword `operation` precedes the operation name, in this case `convolute`. The body of the instruction, i.e. the code after the colon, is divided in two parts. The first one defines which are the parts of the circuit involved in computation. It can be said that these modules are active when this pseudo instruction is executed. This will be useful later in dynamic power computation.

The second part indicates the various paths involved in the computation. In the case of the LiM unit depicted in Figure 6.5 the path referred to the operation just described is only one. The signal flows from the first row and second row to the third going through a XNOR gate and a pop-counter. Knowing that the LiM Compiler interprets the sequence `(0, 0 to 3).lim` as a parallel computation of four XNOR, the execution time of this pseudo instruction is the sum of the XNOR gate delay and the pop-counter delay.

An example of multiple path instruction will be presented later.

6.2.6 LimCompiler

The *LimCompiler* provides two outputs: the hardware configuration of the LiM unit by means of a CompiledLimUnit object and a list of pointers to LimOperation. This data are collected parsing the .lim file.

The first step is to parse the init section. At this moment the list of Compiled-Module present in CompiledLimUnit object is populated by the inter-cells logic elements. The procedure is identical to one followed by the Hardware Compiler.

The following section defines the memory dimension and the memory ports. The LiM Compiler iterates on the parameters in this section and pushes them into the CompiledLimUnit object. At the end of this section the LiM unit is simple memory array with a defined size and a certain number of ports according to the parameters parsed from the .lim file. Some errors can be generate by an incorrect source file. The most common are related to a wrong parameter name or missing parameter. For instance, the number of rows and columns is mandatory in the memory definition. Without these essential values the model cannot be run. Moreover, if any port is defined, whether it is a read, write or even read/write port, the bit width has to be defined too.

The cells section is parsed row by row. For every row an intra-cell logic definition is added to the CompiledLimCell array.

The mapping operation is very similar to the one performed by the Hardware Compiler. The main difference is in the cell mapping. In this case the LimCompiler tries to connect the output of the intra-cell logic, or even the memory cell itself, to one of the other input ports. An error is raised whenever an unknown port is used.

All this section contributes to the composition of the CompiledLimUnit object. The LimOperation list generation is entirely operations section's responsibility. Every time the LimCompiler finds a pseudo instruction it reads the active cells and the active modules and pushes them in a new LimOperation object together with the delay paths. Once the compiler finishes to parse the operation, the LimOperation object is pushed into the output list of pseudo instructions.

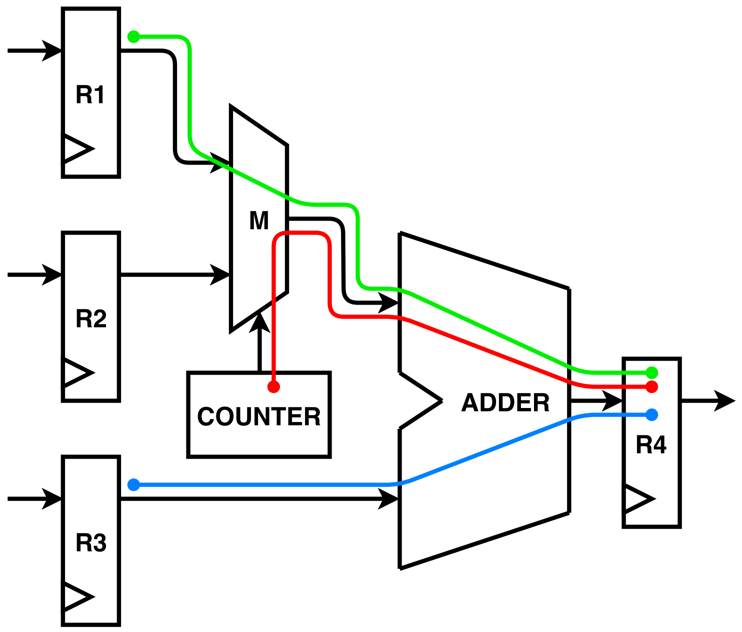


Figure 6.7: **A simple circuit example with its delay path**. The circuit is the same shown in Figure 6.2. The three possible paths are highlighted in green, red and blue

.

6.3 Out-of-memory pseudo instruction definition

Similar to the LiM Operations, the out-of-memory pseudo instructions contribute to the pseudo instruction set. They are also called ASIC Operations in opposition to the processor instructions. The syntax is similar to the one used for the definition of a LiM Operation. Consider the circuit shown in Figure 6.7 and suppose to define a pseudo instruction that perform the addition.

```
operation add: { COUNTER, M, ADDER, R4; [M, ADDER, R4] [ADDER] [COUNTER, M, ADDER,
        R4] }
```

Obviously each and every operation is marked with a name, which is `add` for the operation above. The content of the curly brackets is the operation body which can be divided in two section. The fist one lists the part of the circuit involved in the computation. The second section instead shows the possible path present in the part of the circuit involved in the computation.

6.3.1 Active modules

The modules involved in the computation can be defined as active modules. This modules are active in the sense that its dynamic power is added to the total power of the circuit. If a given module is not involved in computation its dynamic power is not considered. A more detailed explanation will be provided in Chapter 7.

The ASIC Code Compiler checks if all the modules are part of the circuit. If a module is not in the circuit an error is generated.

6.3.2 Critical path computation

As mentioned before, an operation is defined by some active modules but also by a certain number of paths. The purpose of this paths declaration is to compute the execution time, or critical path delay, of the operation. In the case under consideration there are three paths. These paths can be seen in Figure 6.7.

In the operation body these three paths are listed. Clearly the critical path is the red one, but in a lot of situations it is not so clear which is the critical path among all the paths.

The computation of the execution time is a Simulator Core task which knows the value of the delay for each and every module.

Chapter 7

DExIMA backend: hardware models and cycle-based simulator

The core of the estimator is the cycle-based simulator, or *Simulator Core*. The structure is shown in Figure 7.1. The Simulator executes a sequence of operations among those in the pseudo instruction set.

The pseudo instructions are collected in a list of parallel operations. This means that every location of the vector is a list of parallel instructions. The intermediate data generated by the three compilers described in Chapter 6 feed the Simulator Core (on the left side in Figure 7.1).

The first step is to generate the model of the circuit. The list of CompiledModule objects is read element by element and the models of each and every element is created. These models are designed starting from a framework called TAMTAMS. A description is given later in this Chapter.

Then the LiM model is created. The CompiledLimUnit object contains the guidelines to create the model. The structure of the model, called LimUnit, is similar to the CompiledLimUnit object. Like the latter it presents an instance name, a matrix with local computing elements, a list of inter-cell logic elements and so on. The memory performances are estimated by *CACTI*, which is an open source tool.

The bus simulation, instead, is based on look-up tables. The data for these ones are extracted starting from the simulation described in Chapter 4.

7.1 TAMTAMs-derived models

TAMTAMs is a simulation framework developed at the Polytechnic University of Turin [19]. The purpose it has been designed for is to better understand what the

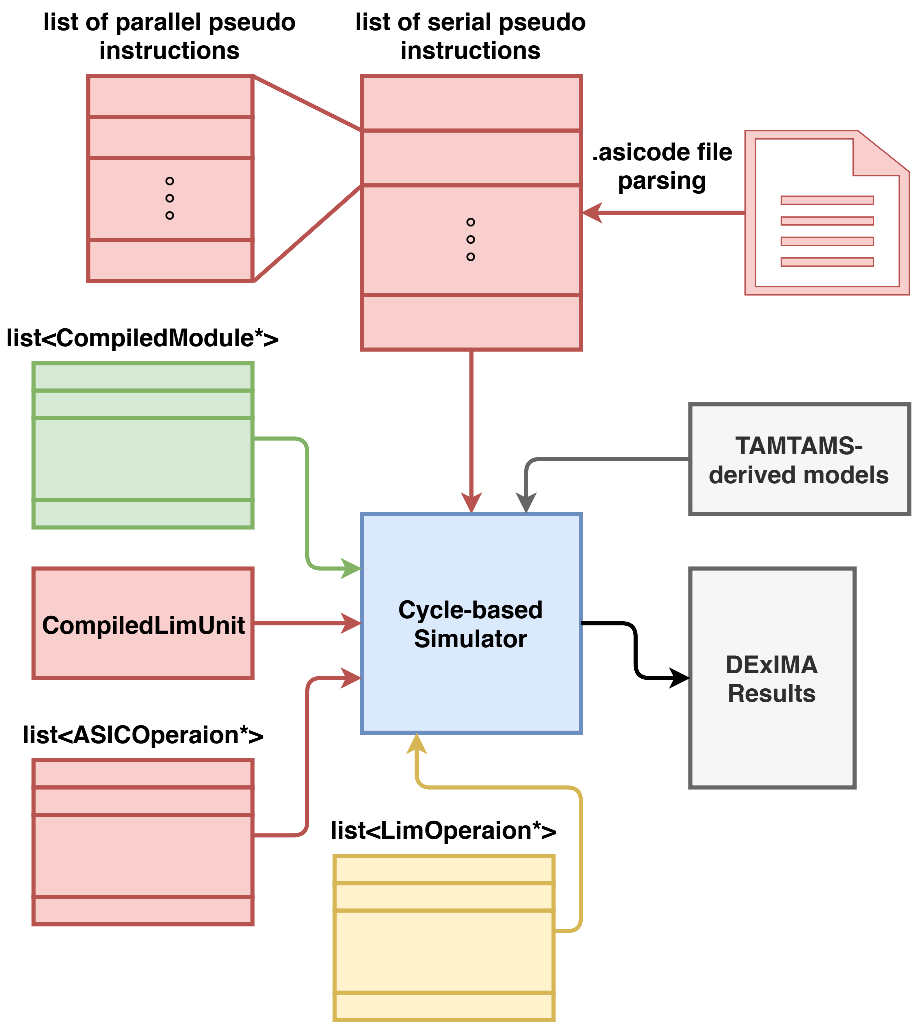


Figure 7.1: **DExIMA backend structure**. The back-end includes first of the the Simulator Core, i.e. the cycle-based execution core.

technology scaling leads. The framework has a main features that make it a perfect candidate to model in a simple and accurate way the hardware of a generic digital circuit: the modularity.

Modularity means that every hardware model is designed based on lower level models. These low level models are, in turn, based on a lower level model until the transistor level is reached. Let's suppose to have to model a simple 2-way multiplexer. A multiplexer is one of the simplest digital circuit. It is made up of two AND gates, one NOT gate and one OR gate. The high level model uses the models of the gates to estimate the multiplexer performance. The AND, OR and NOT gates are, in turn, developed on the model of the transistor which is the lower level model of the framework.

The modularity is a powerful features. In fact, it allows to easily expand the software. Moreover, if the model of the transistor is not accurate enough it can be changed just substituting the relative module without impacting on other modules.

At this moment the NAND is implemented in standard CMOS technology. Nothing prevents to explore new technology such as MTJ (Magnetic Tunnel Junction).

TAMTAMs was initially designed in Octave, a free version of MATLAB. In this thesis, for convenience, the models have been implemented (as the rest of the code) in C++ in a simple hierarchical structure. Instead of the the transistor, it was chosen the NAND-gate model implemented in 27-nm CMOS technology as the lower level TAMTAMs model.

The choice to start with the NAND-gate model derives from the fact that it can be used to implement whatever gate you want. Moreover, it is not necessary to go at a lower level, i.e. the transistor level, because the NAND model seems to be accurate enough to perform the analysis the simulator has been designed for.

7.1.1 Models design

Given a digital circuit, the first design step is to depict it by hand. Then every gate is substituted by its NAND-based equivalent circuit. Since the description must be parametric, also the critical path, power consumption, energy and area must be parametric too. The model is then an abstraction of a digital circuit and the performance are computed starting from some parameters. To be more clear, suppose to have a ripple carry adder. The ripple carry adder is characterized by the number of bits and by the presence, or not, of the carry input bit that turned it in a adder/substractor. Clearly the critical path, as well as the other performance indicators, must be parametrically defined.

At this moment only few models are integrated in the software. Despite this, a large number of quite complex digital circuits can be described. In the following these models are described starting from the lowest level model: the NAND.

NAND As discussed above, the lowest level model is the NAND model. This model is surely the most complex in terms of number of equations. Moreover, it is also the most critical as regards the accuracy of the entire estimation of the performance. In fact, given the gate delay for example, the computation of the multiplexer delay is quite simple.

The model is based on the physical characteristics of the NAND gate. Starting from the physical parameters the capacitances are computed and then the performance are derived from these capacitances. The model provides 5 metrics:

- area;
- delay;
- static power;
- dynamic energy;
- dynamic power.

The area computation is straightforward, it is simply given by the physical area occupied by the 4 transistors of the NAND gate. The delay is given by the following equation.

$$t_{nand} = C_{TOT}\frac{V_{DD}}{I_{nand}} \tag{7.1}$$

where C_{TOT} is the sum of the output capacitance and the load capacitance of the NAND gate as it can be seen below.

$$C_{TOT} = C_{out} + C_{load} = C_{out} + fanout \cdot C_{in} \tag{7.2}$$

It can be seen easily that the total capacitance involved in the port switch, and then in the gate delay, depends on the fanout which is measured in terms of number of NAND gates driven by the current gate. The current I_{nand} is the gate current.

The static power is computed as the average of the 4 input combinations:

$$I_{static} = \frac{1}{4}[(I_{off_n}+I_{gate_p})+(2I_{off_n}+I_{gate_p})+(2I_{off_n}+2I_{gate_n}+I_{gate_p})+(I_{off_p}+4I_{gate_n})] \tag{7.3}$$

I_{static} is then multiplied by the V_{DD} to obtain the static power. The dynamic energy is given by the following equation.

$$E_{dynamic} = \alpha \cdot C_{TOT} \cdot V_{DD}^2 \tag{7.4}$$

The dynamic power is obtained multiplying the energy with the frequency:

$$P_{dynamic} = E_{dynamic} \cdot f \tag{7.5}$$

Each and every model is based on the NAND gate or on derived models such as full adder, half adders, flip flops and so on.

Logic gates The first step to model the standard logic gates is to draw its equivalent NAND based circuit. The Figure 7.2 shows the equivalent circuits for the NOT, AND, OR, NOR, XOR and XNOR gates.

After that the performance are extracted. The area is simply the total number of NAND gates multiplied by the single NAND area. The static power is computed in the same way, i.e. the total number of NAND gates multiplied by the static power of the single NAND gate.

The critical path delay is computed by simply adding together the delay of the NAND gates present in the critical path. The dynamic power and energy are similar to the static power and the area. It is the the sum of all NAND gates dynamic power and energy.

Table 7.1 shows the logic gates delay. The other performance are very simple to compute.

Note that in some NAND delay a superscript indicates a $x2$. It stands for the fanout of the gate. In fact, according to the Figure 7.2, some NAND gates drive two other NAND inputs. Of course the NAND load capacitance increases and in turn the total capacitance increases (see Equation 7.2). This results in an increased gate delay. As a consequence of the increased fanout, the dynamic energy and power increase too.

Other models The other models are implemented in the same way. The step are summarised in the below:

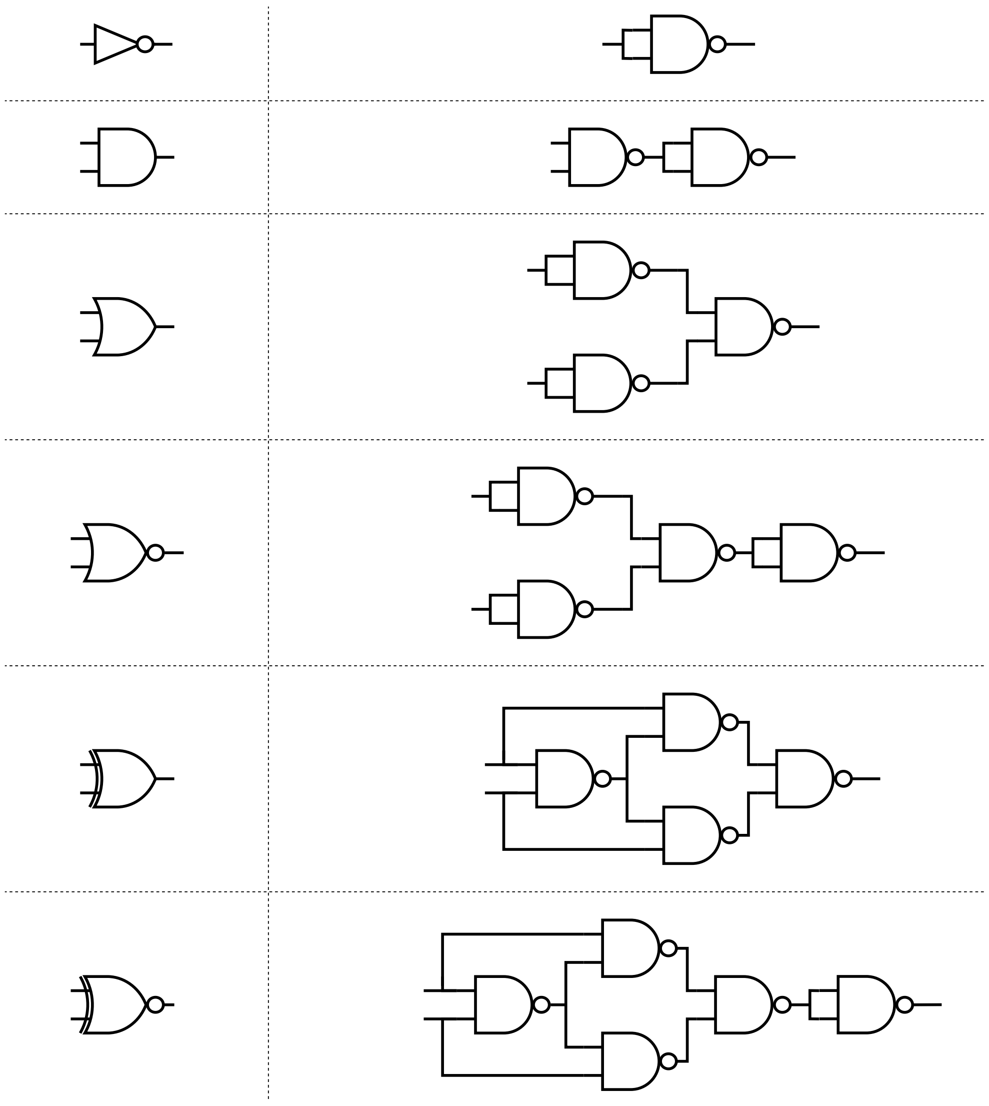


Figure 7.2: **Equivalent NAND based circuit for all the standard logic gates**.

- draw the equivalent NAND gates circuit;
- count the number of NAND gates in the the circuit dived per fanout;
- compute the static power and the area according with the total number of gates;
- compute the dynamic power and energy considering the gates fanout;

	t_{gate}
NOT	$t_{NAND}^{(x1)}$
AND	$t_{NAND}^{(x2)} + t_{NAND}^{(x1)}$
OR	$2 \cdot t_{NAND}^{(x1)}$
NOR	$2 \cdot t_{NAND}^{(x1)} + t_{NAND}^{(x2)}$
XOR	$t_{NAND}^{(x2)} + 2 \cdot t_{NAND}^{(x1)}$
XNOR	$2 \cdot t_{NAND}^{(x2)} + 2 \cdot t_{NAND}^{(x1)}$

Table 7.1: **Summary of standard logic gate delays**.

- compute the delay identifying the critical path and considering the gates fanout.

These are general guidelines to develop a TAMTAMS derived model. For the parametric circuit, i.e. those ones in which the circuit size depends on some parameters, it is slightly different. In fact, the first step is to identify a general equation to compute the number of NAND gates and their fanout. A simple example can be the *multiplexer.*

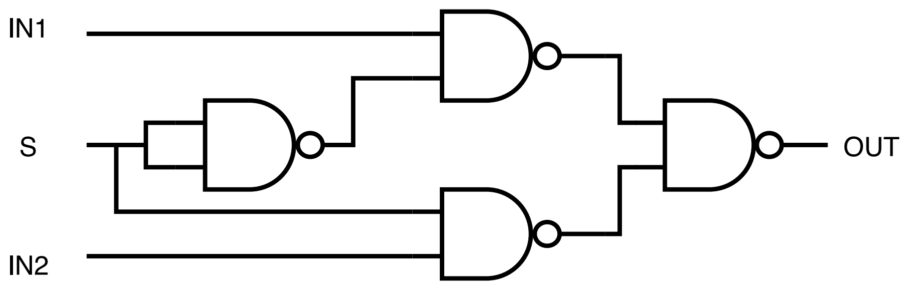


Figure 7.3: **NAND based 2-way multiplexer**.

A simple 2-way 1-bit multiplexer is shown in Figure 7.3. The circuit is composed of 4 x1 NAND gates. The model of this simple multiplexer is easy to implement. If the number of ways of the multiplexer increases, the design of the model becomes different. In Figure 7.4 is shown a 4-way multiplexer made up by 3 2-way multiplexers.

The class `Mux` has a fixed implementation for the 2-way multiplexer while to implement a much wider multiplexer the constructor is called again to generate the model of the 2-way multiplexer, i.e. the real NAND implementations shown in Figure 7.3, and it is used to generate a structure similar to the one shown in Figure 7.4.

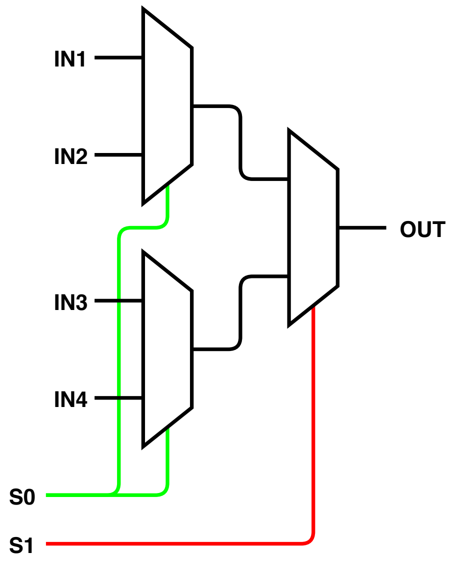


Figure 7.4: **4-way multiplexer**. A 4-way multiplexer is made up of 3 2-way multiplexer.

To be more clear, suppose to want to create a 4-way (the same shown in Figure 7.4) multiplexer. The object of type Mux is constructed passing the number of ways as parameter (4). This objects creates an other object of type Mux, but this time it is created a 2-way mux and its performance are extracted. The performance of the 4-way mux are expressed in terms of 2-way mux performance. The Table 7.2 summarizes and generalizes for a generic number of ways.

7.1.2 Model class organization

In Figure 7.5 it is shown the class structure of the models. For convenience only some of them are shown, but the following considerations apply to all the models.

Area	$(n-1) \cdot A_{MUX^{(2)}}$
Delay	$\log_2(n) \cdot t_{MUX^{(2)}}$
Static power	$(n-1) \cdot P^{static}_{MUX^{(2)}}$
Dynamic energy	$(n-1) \cdot E^{dynamic}_{MUX^{(2)}}$
Dynamic power	$(n-1) \cdot P^{dynamic}_{MUX^{(2)}}$

Table 7.2: **Summary of n-way multiplexer performance**.

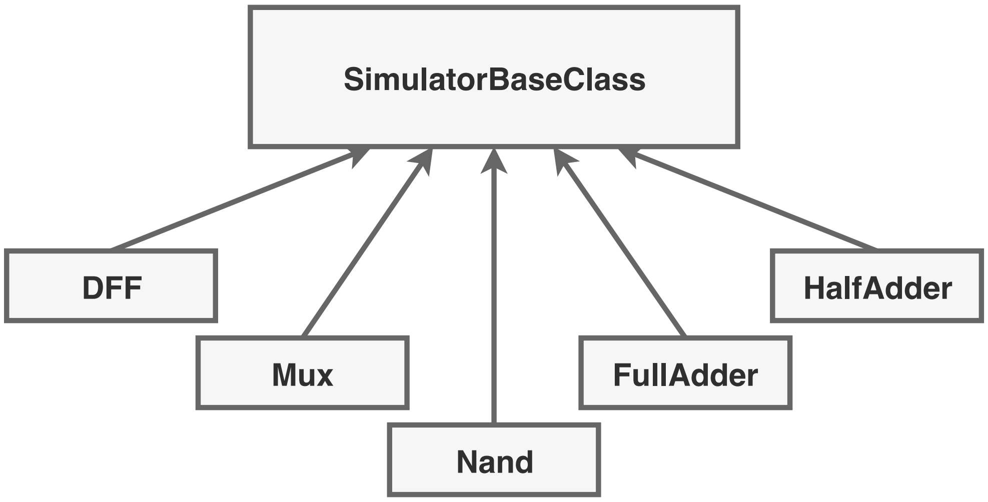


Figure 7.5: **Model class organization**. All the model classes (some of them shown in the figure) inherit from `SimulatorBaseClass`.

The class `SimulatorBassClass` is an abstract class inherited by the model classes. It has 5 implemented public methods:

- `float SimulatorBassClass::get_area();`
- `float SimulatorBassClass::get_delay();`
- `float SimulatorBassClass::get_static_power();`
- `float SimulatorBassClass::get_static_energy();`
- `float SimulatorBassClass::get_static_power()`.

This methods are already implemented by SimulatorBaseClass and are inherited as they are by the model classes. They return the protected variables associated, which are `_area`, `_delay`, `_static_power`, `get_static_energy` and `_static_power` respectively.

The virtual method `void SimulatorBassClass::compute_performance()` is implemented by the model class. In fact, this method contains the hardware models discussed in the previous sections, then it can be implemented only by the subclass.

For a good number of models the method `float SimulatorBassClass::get_delay()` is overloaded by the model class. Suppose to have a ripple-carry-adder which is made up of half adders and full adders. As shown by the Figure 7.6, the critical path is the one starting from the LSB of one of the inputs to the carry out passing through the C_{in}-C_{out} path of each full adder, except the first one. This path is not the critical one of the full adder. For these cases the `get_delay()` method in overloaded and receive the path as parameter.

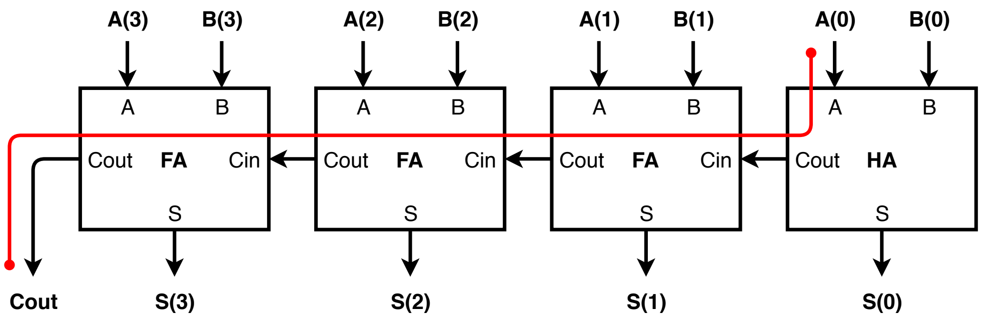


Figure 7.6: **A 4-bits ripple carry adder (RCA)**. The critical path is highlighted in red.

7.2 From memory to LiM unit simulation

The evaluation of memory performance, bus performance and LiM unit performance are very important. A change of design paradigm, from von Neumann to Logic-in-Memory, impacts manly on these three elements. It stands to reason that these elements are the most critical.

The memory performance are computed by an open source memory simulator called *CACTI*. It allows to simulate simple Static RAM, caches and even main memories providing access time, read and write energy, static power and area.

The LiM unit is simulated extending CACTI by merging it with TAMTAMS. In particular, the standard memory operations are simulated by CACTI while the logic inside the memory array is simulated using the models mentioned above (see section 7.1.1).

The bus performance estimation is covered by the last section of this Chapter.

7.2.1 CACTI: an open source memory simulator tool

CACTI is a memory simulator designed by HP (Hewlett-Packard). Muralimanohar et al. has exposed in [20] the capabilities of CACTI. In this thesis work only some of these features are employed. In particular the scratch RAM simulation is the most frequently used mode of operation of CACTI. The scratch RAM indeed is a standard static RAM. This memory is used to simulate the memory performance of the Logic-in-Memory block.

The tool is run by DExIMA and the output file is then parsed. The data provided by CACTI are the access time, the area, the dynamic energy for read and write operations and the static power. After a more detailed analysis some other data can be extracted. In particular the delay is given by the sum of several contributions, including the bitline delay, the sense amplifier delay and so on. These metrics are very important to compute LiM unit performance. In the next section a more detailed analysis of these features will be provided.

As mentioned before, CACTI is run by DExIMA that takes care of composing the input file. The first lines are the most important ones and an example is reported below.

```
-size (bytes) 16384
-block size (bytes) 4
-associativity 1
-read-write port 0
-exclusive read port 1
-exclusive write port 1
```

According to the characteristics of memory that has to be simulated, an object of class `ComposeMemoryConfiguration` is created and it generates the entire CACTI input file (.cfg). While the size and the number of ports are easy to understand, the block size and associativity need a clarification. In fact those data are not defined for a scratch RAM, but they are exclusive characteristics of caches. For a scratch RAM the associativity is set to 1, and the block size is arbitrary.

The other two important parameters to set are the bit-width of the ports and the cache type which must be set to *sram*.

7.2.2 LiM unit

The LiM unit can be defined as a magic memory. The evaluation of this unit is very important since it is a central part of in-memory architectures.

It is made up of a part which simulate the memory behaviour and an other part that evaluate the logic performance. The memory read/write operations simulation is a task demanded to CACTI, while the logic simulator is done exactly in the same way in which the out-of-memory performance are evaluated, i.e. using TAMTAMS models.

The computing elements are divided in two main categories:

- local computing elements;
- inter-cell computing elements.

The former category contains all the logic present inside the LiM cell. To be more clear, the logic defined in the section *cells* of the .lim file. The latter one regards the logic defined in the section *init* of the .lim file.

The distinction between these two categories reflects also the class organization. The LiM cell is described by the class `LimCell` which model the local logic element with a TAMTAMS model. The intra-cell computing elements are modelled as a list of `CompiledModule` just like the out-of-memory hardware.

The delay and the dynamic power of the LiM unit are computed according to the pseudo instruction definition. In particular, as discussed in Chapter 6, an operation is defined by the active part of the circuit and its signal paths. The critical path delay is computed as the highest one among all the path from all the pseudo instructions.

The dynamic power is computed simply adding together the dynamic power of the active model for each operation. In other word each and every operation is characterized by its dynamic power.

The performances of the memory are obviously affected by the presence of logic inside the memory array. In order to consider this worsening in performance the memory access time and power are multiplyied by a corrective factor. This factor depends on the area occupied by the computing elements.

Furthermore, since the interconnects are very important in LiM unit performance evaluation, they are modelled carefully. In the next section it will be explained how they are simulated.

7.2.3 In-memory interconnects

The delay and power due to *in-memory interconnects* are modelled by their capacitance. Since the port fanout is measured in NAND input capacitance unit, also the interconnects contribution should be expressed in this unit.

The interconnect contribution is computed in three steps:

- computation of wire length;
- computation of wire capacitance;
- express the capacitance in NAND input capacitance unit.

The computation of wire length is computed considering a manhattan interconnect. Suppose to have to interconnect the cell (0,0) with the cell (15,7). The wire will start from (0,0) running vertically until the row 15. Here it continues horizontally until column 7. The total length depends on the rows height and columns width.

Knowing the length the capacitance to substrate can be computed with the following equation:

$$C = C_{SUB_0} + C_{fringe} = \varepsilon^2 \frac{WL}{H} + 2\pi\varepsilon^2 \frac{L}{\log \frac{H}{T}} \tag{7.6}$$

where W is the wire width, L is the wire length, T is the wire thickness and H is the distance from the substrate. These parameters are obviously a consequence of the technology.

Once the capacitance is computed, express it in NAND input capacitance unit is just a matter of scaling the value.

$$N_{C_{NAND}} = \frac{C}{C_{NAND}} \tag{7.7}$$

In this way it is expressed simply as an addition of load capacitance.

7.2.4 Bus

An entire chapter is dedicate to the bus (Chapter 4). The simulation of the bus is crucial and it is also costly. For these reason, the performance of the bus are gathered in a look-up table (LUT). In this way the access to delay and power related to a bus transaction are easy to find.

The bus simulation is associated to the LiM unit. Every time a LiM unit is created a bus is created too.

7.3 Cycle-based simulator

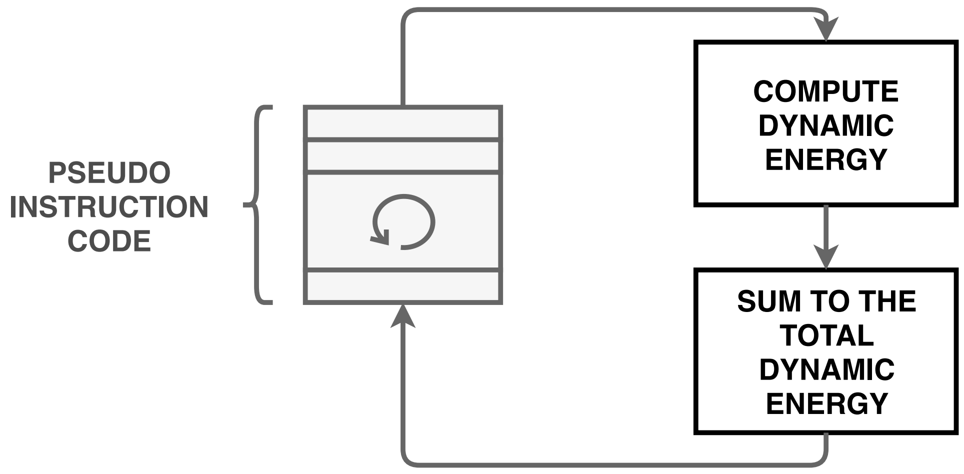


Figure 7.7: **Cycle-based simulation**. The simulator iterates on the operations in the code by computing its dynamic energy and adding to the total dynamic energy of the architecture.

The simulator can be classified in the cycle-based simulators. A cycle is represented by a pseudo instruction. At every cycle the simulator extracts the dynamic energy related to the current pseudo instruction and sums it to the total dynamic energy.

Furthermore the dynamic energy is collected by categories: Logic-in-Memory, memory, bus and out-of-memory hardware. In this way a better performance analysis can be done. In fact, it is fundamental to know where the power are consumed so that some considerations about the implementation can be done. If too much power is consumed in the bus the idea may be to move some computation inside the memory. Moreover the LiM unit read and write operation can be too slow due to an excessive presence of logic. All these consideration can be done only if performance metrics are divided in categories.

Since the simulation cycle corresponds to the circuit cycle, the execution time is straightforward computed. It is equal to the critical path, which correspond to the cycle period, multiplyied by the number of instruction in the code.

$$T = cp \cdot code\_lenght \tag{7.8}$$

Suppose to have a pseudo instruction set composed of 3 instructions. The instruction delays of the three instructions are $1ns$, $1.5ns$ and $0.8ns$. Since the critical path delay is $1.5ns$, the execution time will be computed just multiplying this delay by the number of instruction in the code.

The other parameters, area and static power, are computed before the execution of the code. In fact they are static and therefore independent of the code.

Chapter 8

BNN: a case study

A *Binary Neural Network (BNN)* is a neural network in which both the weight and the input matrix has binary value [21]. Starting from a standard Convolutional Neural Network (CNN) the input are binarized together with weights. The procedure consists of two main steps:

- normalization of the input matrix: it means that the input matrix is normalized with respect to the mean value;
- binarization: it consists of substituting the positive values with a logic 1 and the negative ones with a logic 0.

In this way the convolution layer which is normally a multiply and accumulate circuit is turned into a XNOR between the filter and the input matrix and then a bit counting. This reduces the computation cost significantly.

Some important parameters characterize a binary neural network:

- input matrix size $n \times n$;
- filter size $m \times m$;
- stride s.

8.1 Von Neumann implementation

A von Neumann implementation is shown in Figure 8.1. The input matrix is stored in the memory (blue box) which has 1 read port and 1 write port. The XNOR-net, which computes the binary multiplication, is composed of $m \times m$ XNOR gates. The output of the XNOR-net feeds the pop counter inputs. The pop counter performs the bit counting and the MSB, which represents the sign, is stored in a register.

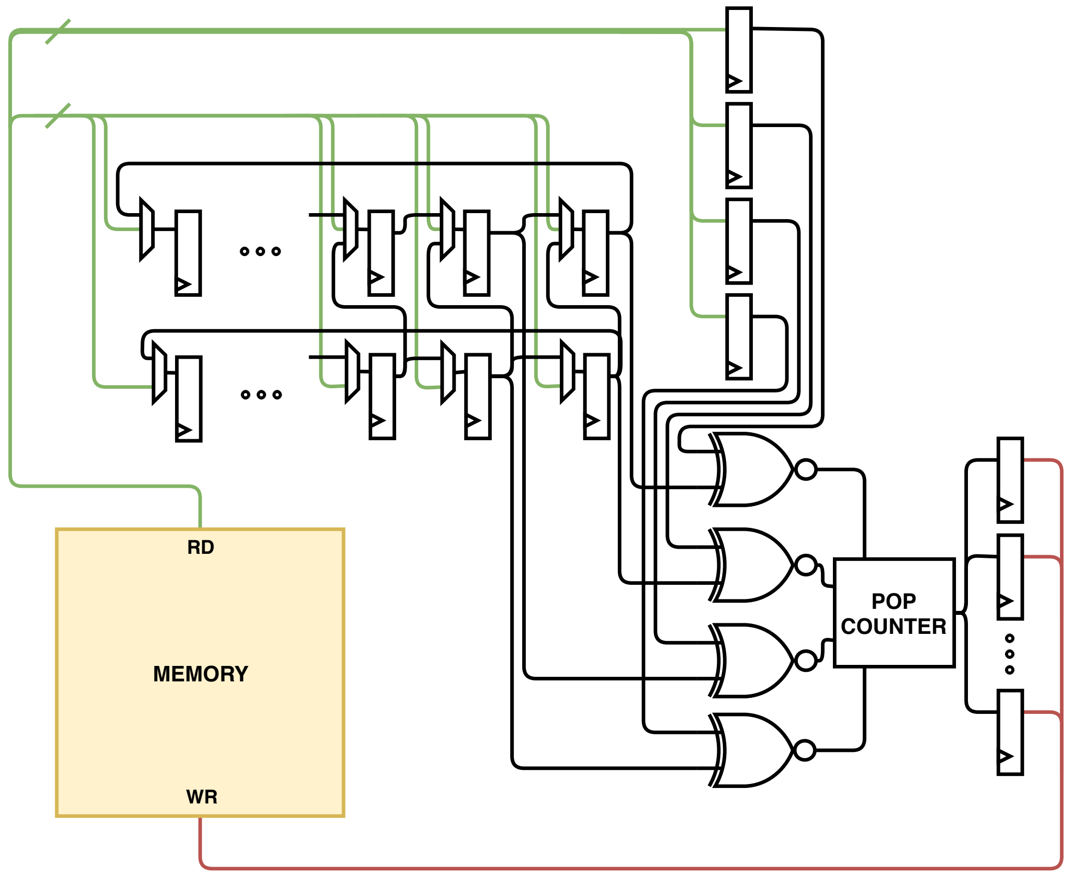


Figure 8.1: **A von Neumann implementation a the convolutional layer of BNN**.

The convolution is done in several step. At the beginning the portion of the matrix shown in Figure 8.2 is read. This portion of matrix has m rows and has a bus width number of columns. This portion is stored in the register outside the memory. A first convolution is done and then the matrix is shifted left. Then an other convolution is done and so on until the first portion is totally convoluted. The output data are then stored in the memory.

A new portion of matrix is read from memory and the same procedure is followed until the matrix is completely convolutes and the output matrix is stored in the memory.

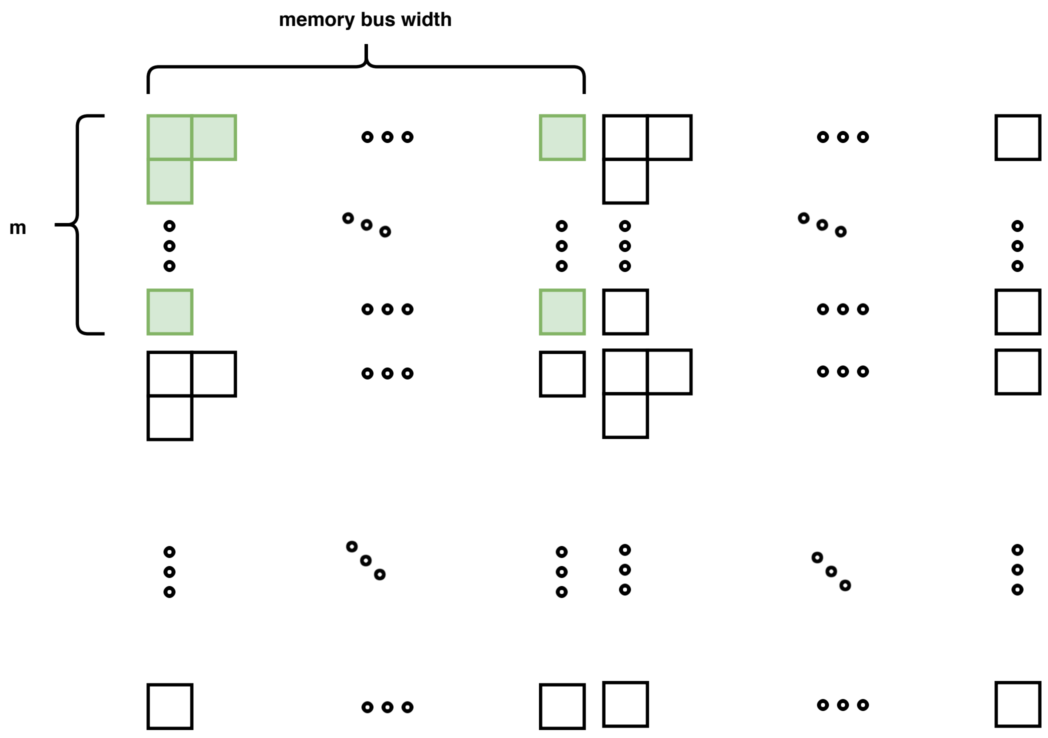


Figure 8.2: **Input matrix blocks**.

8.2 In-memory XNOR-net, out-of-memory bitcounting

In the second implementation the XNOR-net is implemented in memory. The matrix is stored in the memory as rows and columns just like it is for the von Neumann implementation.

In Figure 8.3, the input matrix (white boxes on grey background) is moved step by step to the filters (green boxes). The filters are composed of a $m \times m$ matrix of *lim-xnor* cells (see Section **??**). The number of filters per matrix extended row is a design parameter. For the sake of simplicity in the circuit shown in Figure 8.3 this parameter is 4, but is most often far greater. Note that the matrix extended row is a group of m input matrix rows, which is the filter dimension.

Each and every filter cell has a multiplexer on the read/write port, which selects among the possible inputs of the filter. The Figure 8.3 shows also which matrix section is moved to the filter. The equation below computes the number of inputs

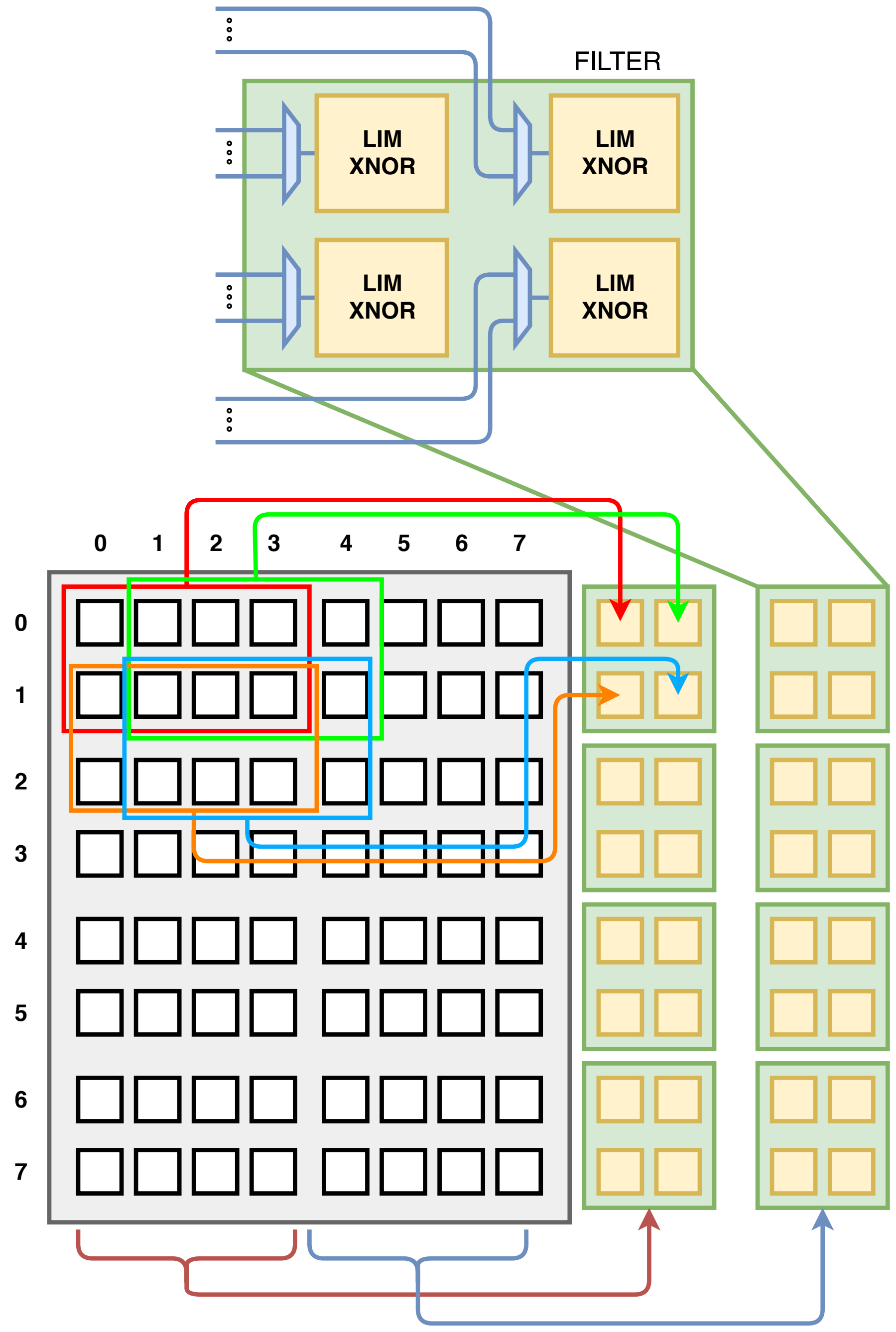


Figure 8.3: **A Logic-in-Memory implementation of the convolutional layer of BNN**.

per mux:

$$N_{inputs}^{(MUX)} = \frac{n}{filters\_per\_matrix\_extended\_row} \cdot m \tag{8.1}$$

The content of the red, green, orange and blue boxes goes to the the multiplexer inputs of the first, the second, the third and the fourth cell of the first filter respectively. The columns from 4 to 7 will feed the the second filter of the first row and so on.

After XNOR-computation the results are stored in a different part of the memory. The results occupy a larger amount of memory with respect to the input matrix. In fact for each and every convolutional layer output bit the memory has to allocate $m \times m$ bits for these intermediate results.

Once the intermediate results are stored, the memory is read and the bit-counting is performed out-of-memory by means of a pop counter.

This is a partial in-memory architecture in which part of computation is performed in memory. The convolutional layer of a BNN can be also implemented entirely in memory.

8.3 In-memory XNOR-net, in-memory bitcounting with extended matrix

The last architecture is implemented totally in memory. The data are read from or write to the memory during the computation. For these architectures the power consumption due to both the memory read/write and the bus usage get zero.

To perform the convolution in parallel the matrix has to be store in memory in slightly different way with respect to the previous two implementations. The matrix is extended and input matrix element is present more than once in the memory. Soppose to have a matrix of 8x8 bits and a 2x2 filter. The output matrix of the convolution layer has 7x7 bits. Each and every bit is computed performing the XNOR between 4 input bits and the four wights.

The Figure 8.4 shows how the matrix is stored in the memory. The memory cells containing the matrix are all *lim-xnor*. They are divided into 7x7 groups which represent the number of output bits. The outputs of the XNOR gate of every cell inside a group goes to the inputs of a pop counter as shown in Figure 8.5.

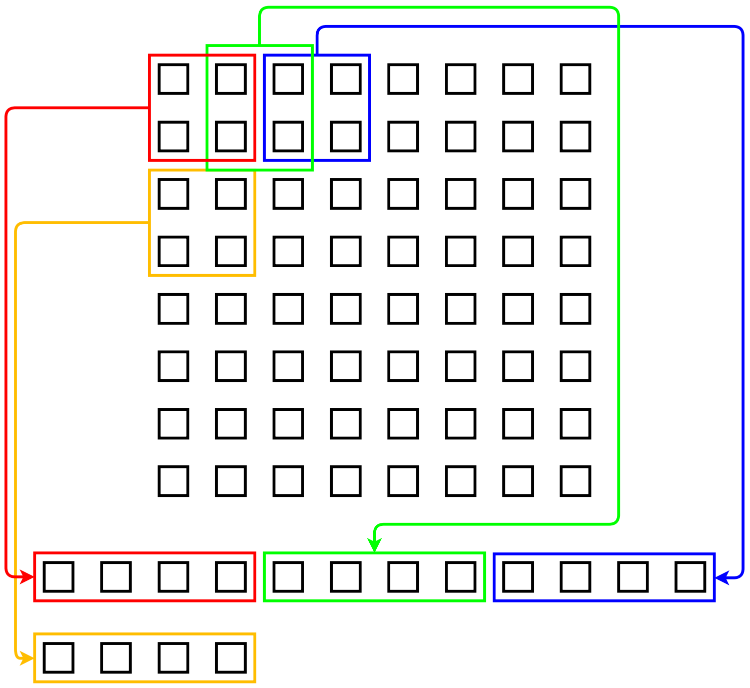


Figure 8.4: **Matrix organization of a LiM based BNN**. The matrix is extended to allow the parallel computation of the XNOR operation.

8.4 DExIMA simulations

	Von Neumann	LiM 1	LiM 2
64x64	0.295601 ns	0.361886 ns	1.60356 ns
128x128	0.295601 ns	0.520913 ns	2.73599 ns
256x256	0.373833 ns	0.792391 ns	2.44902 ns

Table 8.1: **Critical path**

	Von Neumann	LiM 1	LiM 2
64x64	1.301237 μs	0.3828751 μs	1.603565 ns
128x128	5.177164 μs	0.9058676 μs	2.73599 ns
256x256	26.11221 μs	2.450864 μs	2.44902 ns

Table 8.2: **Execution time**

The three implementations are described to be simulated in DExIMA. They

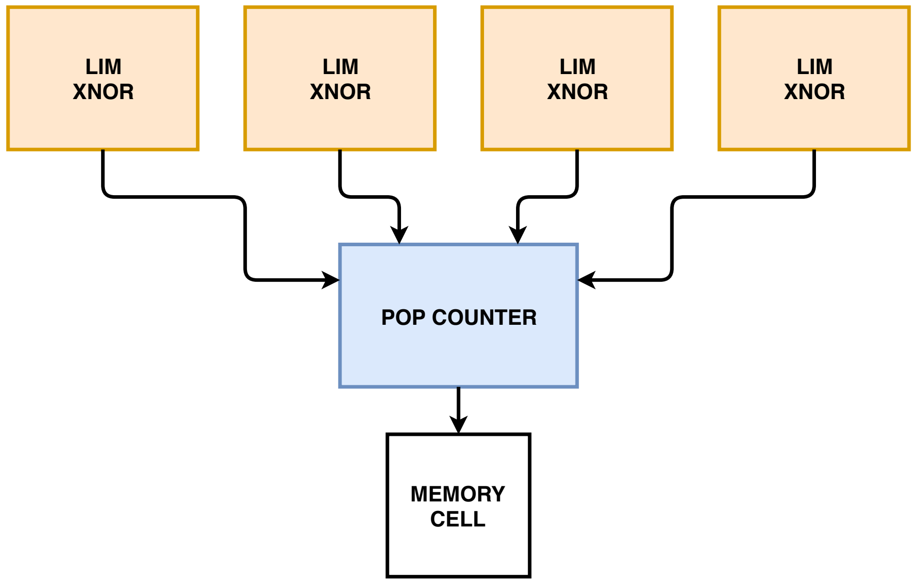


Figure 8.5: **BNN convolution layer in memory processing elements**. The figure shows the computing element of an in-memory implementation of a BNN convolutional layer.

	Von Neumann	LiM 1	LiM 2
64x64	$1.382623 \cdot 10^4 \mu m^2$	$4.787402 \cdot 10^4 \mu m^2$	$2.539689 \cdot 10^5 \mu m^2$
128x128	$4.667965 \cdot 10^4 \mu m^2$	$1.755846 \cdot 10^5 \mu m^2$	$1.028480 \cdot 10^6 \mu m^2$
256x256	$1.668747 \cdot 10^5 \mu m^2$	$7.351387 \cdot 10^5 \mu m^2$	$1.036889 \cdot 10^6 \mu m^2$

Table 8.3: **Area**

	Von Neumann	LiM 1	LiM 2
64x64	62.42631 mW	216.2110 mW	756.7076 mW
128x128	59.89511 mW	357.8932 mW	1818.343 mW
256x256	42.42832 mW	528.9544 mW	2031.410 mW

Table 8.4: **Dynamic power**

	Von Neumann	LiM 1	LiM 2
128x128	4.540415 mW	62.99049 mW	172.2904 mW
128x128	10.59488 mW	246.8088 mW	289.4778 mW
256x256	32.17326 mW	985.2453 mW	707.4029 mW

Table 8.5: **Static power**

	Von Neumann			LiM 1			LiM 2		
	Mem	LiM	Bus	Mem	LiM	Bus	Mem	LiM	Bus
64x64	0.95	0	11.47	1.72	204.0	8.66	0	756.7	0
128x128	1.66	0	10.70	4.02	344.9	7.37	0	1818	0
256x256	2.81	0	7.39	6.42	515.9	5.45	0	2031	0

Table 8.6: **Dynamic power per category (mW)**

are simulated for different input matrix dimensions but the same filter size. The simulation parameters are listed below:

- input matrix 64x64, 128x128, 256x256;
- filter size 3x3.

So the different architecture generated are 9. Every architecture is described with the 4 canonical files: the architecture description, the LiM unit description, the ASIC pseudo instruction definition and the pseudo code.

The results are reported in the tables below (Tables 8.1, 8.2, 8.3, 8.4, 8.4 and 8.6).

8.4.1 Results

Starting from the Table 8.1 and Table 8.2 it is clear that the critical path delay is not a significant parameter. In fact, the faster architecture, regardless of the number of bits, is also the architecture with the higher critical path delay.

This is because in the last architecture (LiM 2) the computation is performed all in parallel.

Although the last architecture is also the most power consuming, apparently. In fact, it consumes from 3.5 to 5 times the other architectures. Since it outperforms the other two architecture by at least 190 times there are room to decrease supply voltage to decrease the dynamic power consumption.

Even though the architecture LiM 1 has some of computation performed in memory it consumes at least 25 times the power consumption of von Neumann architecture, but there is a small advantage in terms of execution time.

Probably because in architecture LiM 1 it is necessary a bigger memory. The memory size impacts on the memory performance and the read/write operation are costly. Against this the parallelism is not sufficient to speed up the computation.

An important point is that the critical path in von Neumann architectures is given by the out-of-memory circuit for the two smaller design, while for the last one, i.e. 256x256, the critical path becomes the access time.

Even this simple application, the convolutional layer of a BNN, presents some unexpected results.

Conclusion

The thesis goal was initially to find a generic structure that complies with the requirements of an in-memory architecture. Starting from CLiMA, the architecture proposed in [1], it has been develop a LiM-based architecture model as generic as possible. In this architecture they are taken into account almost all the possible form of computation. This model has to be thought as a set of empty boxes. The task to fill in the boxes is demanded to the designer.

Soon, the need to develop an architectural-level simulator specifically design to evaluate in-memory architectures performance came out. A tool that can answer some design question. *What should a specific box contain? Where is it better to bind a given operation? Should this implementation fine also for a larger amount of data involved in computation?*

DExIMA should assist the designer and should help him or her to answer the above questions.

In Chapter 3 there are exposed the required features that this tool was supposed to have:

- configurability;
- ability to describe the execution of an algorithm;
- a sufficient number of available models;
- mantainability.

The *configurability* has been achieved introducing 3 input files that are used to describe the out-of-memory hardware, the LiM unit configuration, the operations that can perform and the definition of the out-of-memory pseudo instructions.

The fourth file guarantees the *ability to describe the execution of an algorithm.* It describes the execution of an algorithm step by step using the pseudo instructions selecting them among the pseudo instruction defined by the pseudo instruction set.

The number of available models is sufficient to model most of circuits but they will never been enough. Although, the most frequently used circuit are implemented:

flip-flop, full adder, half adder, ripple carry adder, counters, logic gates, booth multiplier and so on.

However, the implementation of an additional model would change mainly the front-end of DExIMA. In fact it should be defined a proper keyword, its parameters and its ports. Furthermore, the model has to be design similar to the other models. It should be a class inherited from the `SimulatorBaseClass` class and the performance equation has to be included in the method `compute_performance`.

Further improvements

DExIMA is composed of about 65 C++ classes. This is the first version of the software. It can be said that it is the β-version. Like all the β-version there are, of course, several problems that should be solved in the next versions.

Parallelization The tool is really too slow when the circuit becomes large. Both the parsing of the input files and the execution of the pseudo instructions have high computational cost. In fact the input files can be even very large reaching many hundreds of thousands of lines generating list of objects, manly `LimCell`, `CompiledModule` and `SimulatorBaseClass`).

A solution can be to parallelize the parsing of the file in which most of times a line is independent of all others. Similarly the execution of the code can be done in parallel because the execution of an instruction doesn't compromise the execution of an other instruction.

Syntax improvements In order to speed up the simulation somethings can be done on the input file syntax. As mentioned before the input files can be very large. This is manly due to the minimal syntax. In the code description it is impossible to use any kind of loop. Also in the description of hardware, both in-memory and out-of-memory, there's no possibility to generate the same module multiple times.

It is not rare to have tens of parallel identical flip flop that would be easily described with a loop with some lines of code. Now, they have to be defined using one line per flip flop.

Better integration between CACTI and LiM unit simulation A strong limitation is the poor integration between the model of the memory and the model of logic inside the memory. In fact some consideration about how the memory performance gets worse by including computing elements in memory are not precise.

At this moment the memory delay and power consumption are evaluated according to the memory area increment. Although this is a good idea, that gets closer to the reality, it is difficult to evaluate how much is this contribution.

Fortunately, CACTI is open source and it can be changed to meet the Logic-in-Memory requirements. It would be a good idea to turn CACTI in a LiM unit simulator and not just a conventional memory simulator.

Better front-end In some cases, the three compilers doesn't help the user. When long compilation processes are running, DExIMA seems to do nothing even when it is working well. Maybe some outputs showing the progress of the compilation can be helpful.

At the same time when long simulation are started there are no information about the number of instructions executed among all. It would be better to show some minimal indication about the simulation.

Bibliography

[1] Giulia Santoro. *Exploring New Computing Paradigms*. PhD thesis, Politecnico di Torino, 2019.

[2] 2009 international technology roadmap for semiconductors (itrs). 2009. URL https://www.semiconductors.org/wp-content/uploads/2018/09/Interconnect.pdf.

[3] J Jeddeloh and B Keeth. Hybrid memory cube new dram architecture increases density and performance. In *2012 Symposium on VLSI Technology (VLSIT)*, pages 87–88. IEEE, 2012. ISBN 9781467308465.

[4] Dae Hyun Kim, Krit Athikulwongse, Michael B Healy, Mohammad M Hossain, Moongon Jung, Ilya Khorosh, Gokul Kumar, Young-Joon Lee, Dean L Lewis, Tzu-Wei Lin, Chang Liu, Shreepad Panth, Mohit Pathak, Minzhen Ren, Guanhao Shen, Taigon Song, Dong Hyuk Woo, Xin Zhao, Joungho Kim, Ho Choi, Gabriel H Loh, Hsien-Hsin S Lee, and Sung Kyu Lim. Design and analysis of 3d-maps (3d massively parallel processor with stacked memory). *IEEE Transactions on Computers*, 64(1):112–125, 2015. ISSN 0018-9340.

[5] Shaahin Angizi, Zhezhi He, and Deliang Fan. Pima-logic: A novel processing-in-memory architecture for highly flexible and energy-efficient logic computation. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018. ISBN 9781538641149.

[6] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. Prime: a novel processing-in-memory architecture for neural network computation in reram-based main memory. *ACM SIGARCH Computer Architecture News*, 44(3):27–39, 2016. ISSN 01635964.

[7] Wenqin Huangfu, Shuangchen Li, Xing Hu, and Yuan Xie. Radar: A 3d-reram based dna alignment accelerator architecture. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018. ISBN 9781538641149.

[8] Roman Kaplan, Leonid Yavits, Ran Ginosar, and Uri Weiser. A resistive cam processing-in-storage architecture for dna sequence alignment. *IEEE Micro*, 37 (4):20–28, 2017. ISSN 0272-1732.

[9] Leonid Yavits, Shahar Kvatinsky, Amir Morad, and Ran Ginosar. Resistive associative processor. *IEEE Computer Architecture Letters*, 14(2):148–151, 2015. ISSN 1556-6056.

[10] Shoun Matsunaga, Jun Hayakawa, Shoji Ikeda, Katsuya Miura, Haruhiro Hasegawa, Tetsuo Endoh, Hideo Ohno, and Takahiro Hanyu. Fabrication of a nonvolatile full adder based on logic-in-memory architecture using magnetic tunnel junctions. *Applied Physics Express*, 1(9):091301, 2008. ISSN 18820778.

[11] Tunneling between ferromagnetic films. *Physics Letters*, A 54.3:225–226, 1975. ISSN 0375-9601.

[12] Hooman Jarollahi, Naoya Onizawa, Vincent Gripon, Noboru Sakimura, Tadahiko Sugibayashi, Tetsuo Endoh, Hideo Ohno, Takahiro Hanyu, and Warren J Gross. A nonvolatile associative memory-based context-driven search engine using 90 nm cmos/mtj-hybrid logic-in-memory architecture. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 4(4):460–474, 2014. ISSN 2156-3357.

[13] Kai Yang, Robert Karam, and Swarup Bhunia. Interleaved logic-in-memory architecture for energy-efficient fine-grained data processing. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*, volume 2017-, pages 409–412. IEEE, 2017. ISBN 9781509063895.

[14] Lei Jiang, Minje Kim, Wujie Wen, and Danghui 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. IEEE, 2017. ISBN 9781509060238.

[15] The gem5 simulator. http://www.gem5.org/Main_Page.

[16] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. *SIGARCH Comput.*

Archit. News, 28(2):83–94, May 2000. ISSN 0163-5964. doi: 10.1145/342001.339657. URL http://doi.acm.org/10.1145/342001.339657.

[17] Antoine Courtay, Oliver Sentieys, Johann Laurent, and Nathalie Julien. High-level interconnect delay and power estimation. *Journal of Low Power Electronics*, 4:21–33, 2008.

[18] Y.I Ismail, E.G Friedman, and J.L Neves. Figures of merit to characterize the importance of on-chip inductance. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 7(4):442–449, 1999. ISSN 1063-8210.

[19] Fabrizio Riente, Izhar Hussain, Massimo Ruo Roch, and Marco Vacca. Understanding cmos technology through tamtams web. *IEEE Transactions on Emerging Topics in Computing*, 4(3):392–403, 2016. ISSN 2168-6750.

[20] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P. Jouppi. Cacti 6.0: A tool to model large caches. Technical report, Hewlett-Packard Laboratories - School of Computing, University of Utah.

[21] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. volume 9908, pages 525–542. Springer Verlag, 2016. ISBN 9783319464923.