POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master Thesis

GP-LiMA: A General Purpose Architectural Model leveraging the Logic-in-Memory Approach

Supervisors:
Prof. Mariagrazia Graziano
Prof. Marco Vacca
Prof. Giovanna Turvani

Candidate:
Angela Guastamacchia

July 22, 2021

Summary

Nowadays, the old technological drivers underpinning the unceasing enhancement of processing systems performance seem to be no longer sufficient to support the usual growth exponential trend. One of the main issues is linked to the standard Von-Neumann architectural organisation, based on a clear division between the CPU and the memory. This layout implies a constant data stream between CPU and data memory that, in turn, are affected by a significant performance gap. Then, the resulting massive number of memory accesses entails wastes of both time and power. This issue, known as *memory wall*, is even exacerbated in the lately demanded data-intensive applications, leading to prohibitive energy expenses.

This thesis work presents the design of a novel architectural model based on an alternative computing approach, i.e. the Logic-in-Memory (LiM). This paradigm leverages the integration of processing elements inside the memory to reduce both execution times and energy consumptions, entrusting parts of the required elaboration directly to the memory itself. In particular, the LiM approach allows cutting the number of memory accesses and the energy involved in the data travel between memory and CPU and exploiting the total memory bandwidth to elaborate in parallel all the stored data. However, the already existing systems that embed the LiM processing kind are customised for one specific application; therefore, new LiM structures must be designed from scratch for each new demanded application.
Thus, during this thesis development, an efficient model to speed up the design of LiM solutions was engineered by taking a cue from the already existing Programmable LiM (PLiM) template proposal developed at the Polytechnic University of Turin. The PLiM suggests a pre-defined skeleton for a LiM system, which is easily adaptable by the hardware designer to the specific tasks at hand. The basic structure comprises the LiM Array, i.e. the memory array composed of a sequence of word locations, called Smart Rows, integrating both storing and computing elements, and a control part handling the data processing inside the array. Furthermore, the Smart Rows structure can be quickly modified by inserting custom blocks to tailor the basic LiM structure to the desired algorithm. Then, all the Smart Rows are driven by the same control signals, so the PLiM devices work following the Single Instruction

Multiple Data (SIMD) computing model, particularly suitable for the execution of data-intensive algorithms.

This thesis study started with an analysis of the investigated PLiM model goodness in generating efficient LiM devices. Five different benchmarks, i.e. K Nearest Neighbour (K-NN), Matrix-Vector Multiplication (MVM), K-means, mean and variance ($\mu\&\sigma^2$), Discrete Fourier Transform (DFT), were chosen as target algorithms and mapped onto the retrieved PLiM systems customised for each of them. The performance achieved by the five devices pointed out the model suitability for producing LiM systems capable of efficiently running highly parallel tasks while highlighting the PLiM devices inadequacy for executing the more sequential procedures.

Hence, during this thesis, an improved architectural model, returning LiM systems characterised by high programming generality and processing efficiency even for sequential tasks, was conceived and referred to as GP-LiMA (General Purpose Logic-in-Memory Architecture). The GP-LiMA model was set up along the lines of the PLiM, as shown in Figure 1. However, it moves away from the PLiM paradigm, providing a grid-like arrangement of the Smart Blocks that further accommodates a dense routing network to implement complex data exchanges. Moreover, the single Smart Block is supplied with a default structure that, coupled with the flexible interconnections, speeds up the processing times and maximises the GP-LiMA programming generality, further easing the hardware designer's task. Then, the LiM device, generated from this model, can be inserted into a standard system, interacting with the CPU either as data memory or as a Multiple-SIMD co-processor enabling different Smart Blocks subsets to run different instructions simultaneously.

Finally, to verify the achieved performance and the programming generality, different comparisons were made between a specific synthesis of the GP-LiMA structure and the PLiM models for all the previously mentioned benchmarks. In particular, the GP-LiMA Unit analysed was implemented through the 45 nm technology node. It included 1344 bytes of memory addressable space and could elaborate up to 512 16-bits data in parallel, working at a maximum frequency of 232.55 MHz. Then, for each of the mapped benchmarks, the results in terms of execution time and energy consumptions were compared on a per-sample basis with the ones accomplished by the customised PLiM devices. The GP-LiMA reached almost the same performance as the customised PLiM architectures for the more parallel algorithms (K-NN and K-means). In contrast, for the algorithms involving sequential procedures ($\mu\&\sigma^2$, DFT), the GP-LiMA showed an outstanding reduction of the execution times of about 67% and 37% and energy savings of about 80% and 63%, respectively.
Ultimately, to highlight the benefits brought by the insertion of the GP-LiMA paradigm inside a standard system, the energy consumed by the GP-LiMA was

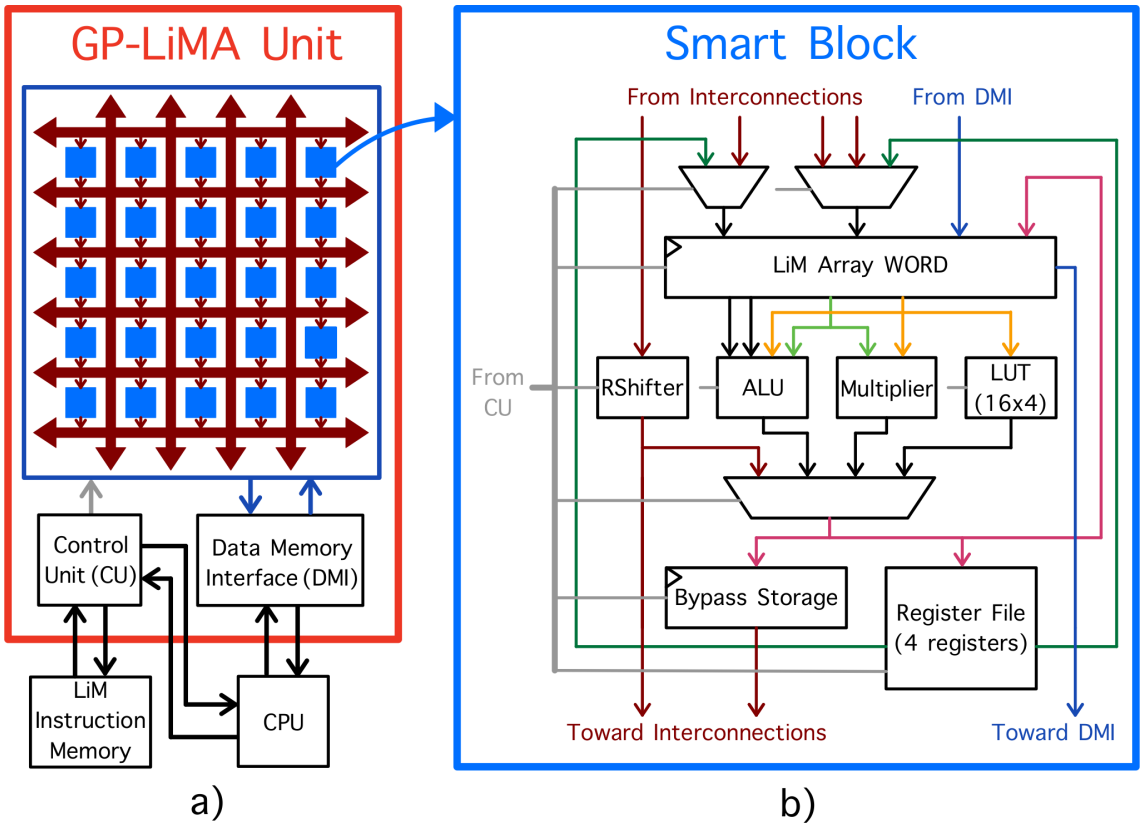


Figure 1: a) High-level view of the GP-LiMA Unit connected to the CPU. b) Insight of the Smart Block structure.

compared with the one spent by a classical RISC-V architecture while performing the memory accesses to the connected memory hierarchy. A clear gap between the two systems stood out from results comparison on the same five benchmarks. Running both systems with the same clock frequency (232.55 MHz), the GP-LiMA showed massive energy savings with respect to the classical RISC-V memory hierarchy ranging from about 49% in the worst case (DFT) to 91% in the best case (K-means). Whereas, running the standard RISC-V architecture at a frequency four times higher than the GP-LiMA one (1 GHz vs 232.55 GHz), the results still pointed out competitive values concerning the GP-LiMA energy expenses, demonstrating to win over the RISC-V memory hierarchy for some benchmarks. For algorithms that can be easily parallelised (K-means), the GP-LiMA showed energy savings of 76%, while for applications involving sequential operations that can be performed in a reduction tree-like fashion ($\mu\&\sigma^2$), it provided energy savings of about 52%.
It follows that the insertion inside a standard system of the LiM co-processor, generated from the GP-LiMA model, can successfully cope with the memory wall issue.

Acknoledgements

At the end of this Master Thesis work, I would like to express my gratitude to all those who helped me during these last months and all along my years of study.
First of all, I would like to thank my supervisors. Thanks to Prof. Mariagrazia Graziano, Prof. Marco Vacca, Prof. Giovanna Turvani, and Prof. Maurizio Zamboni, who were always ready to give me advice whenever needed. A special thanks to Andrea Coluccio, who, during these months, was always there for me, supporting me in every step of my thesis and making me grow both professionally and as a person.
Also, I would like to thank all the PhD students of the LiM group for helping me during my work and for welcoming me to their group during all the thesis.
A big thanks to my parents, who demonstrated to be always by my side even in the most insecure and unexpected moments.
Another important thank you to Delia, who proved to me that real friendship can be seen with time and requires love and effort, making me understand that I would miss one of the greatest parts of my life without her.
Then, a massive thanks to Valentina, who after 15 years made me believe in things that go beyond what we can simply see and touch, forging a spaceless and timeless bound on which we could always count on.
I want to say thank you also to Letizia, who surprised me with an unexpected friendship that taught me home can be found in any place if you just meet the right people, and to Fabrizio, who always inspires me with its everlasting initiative and bravery to embrace life as it comes. I would also like to thank Angelo, Giuseppe, Antonio and Domenico that in different moments were able to endure me during the projects making time lighter and even remaining with me after the end.
Finally, a Special thanks to Marco, who decided to bet on me from the first moment and patiently taught me how I could overcome obstacles, getting me this far.

Sincerely,
Angela

Table of contents

Chapter 1

Introduction

For some years now, we are witnessing the sharp growth of a new field of applications based on machine learning algorithms. This field is meant to become more and more pervasive in our daily life. Its main feature lies in the massive amount of data to be processed, ideally in parallel, to complete a single task. However, this poses a real challenge for the hardware that is now more than ever required to support a heavy data crunching rate. On the other side, applications like data search algorithms, data encryption, or image compression could benefit from making the computer architectures able to deal with this obstacle. Therefore, in the last decades, many efforts have been spent to empower the architectures, integrating more and more transistors into a single chip. Nevertheless, soon this solution is going to be no longer sufficient to pursue this goal, as we are going toward the expected *Moore's law* ending.

In 2017, Thomas N. Theis and H.-S. Philip Wong came up with a careful discussion [1] highlighting how, over the years, technological developments in Integrated Circuits (ICs) design and manufacturing have managed to keep up with the far-sighted *Moore's law* (1965) [2]. This law stated that, for the same minimum cost per component, the integration complexity would have increased by a factor of two per year, resulting in an exponential increase of performance and a decrease in die cost and dimension. Unfortunately, they also concluded that the same technological drivers, which have underpinned this law up to now, will be no more valid to prevent this trend from approaching a plateau, so alternative solutions will need to

be explored.

To better understand why old technological drivers can no longer be exploited and which are the new proposals the industry is going toward, it is helpful to recap the evolution of the techniques implemented over the years.

The first mechanism to increase the integration density of silicon wafers was a barely geometrical scaling protocol. It consisted of reducing the physical dimension of the devices through the shrinking of the CMOS channel length. The gate length reduction did not translate into a slowdown in systems performance but rather to their improvement. The channel length shrinking involved an increase of the drain current and a lowered gate capacitance, leading to a heavy drop of the CMOS charging times with an exponential enhancement of the clock frequency and a heavy reduction of circuits dissipated power.

On the other hand, the decrease of the device dimensions alone gave birth to many issues, mainly due to a higher electric field along the channel, caused by a non-scaled voltage drop on a smaller region. Examples are a lowered lifetime and reliability due to breakdown occurrences and an increase in the dissipated power. Therefore, the scaling procedure required to act, at the same time, on another factor to compensate for the electric field growth, namely the operating voltage. Thus, the actual scaling was performed following some predefined rules, named after Dennard, who was one of the first researchers to mathematically claim how to limit the short channel effects [3]. However, this way could not be pursued beyond a given bound. The gate oxide thickness reduction coupled with the scaling of the voltage swing could not be carried out indefinitely, since the former would have lead to excessive exponential growth of the gate leakage current (due to tunneling effects), while the latter would have resulted in either the switching speed slowdown or the rise of the static power value [1].

Nevertheless, due to the unceasing technological research, the exponential trends regarding the integration density and the device switching speed continued to be further sustained by innovating both material and device structure. Howbeit, at that point, the operating voltage had almost reached its scaling limit, so, to avoid running into an unsustainable power and heat generation, the clock frequency was prevented from keeping its exponential climbing. In order to overcome this obstacle

in the performance escalation trend, the industry started to focus on architectural solutions, taking advantage of the new achievements in lithography and integration capabilities. More cores started to be embedded in the same chip, aiming at improving the overall performance by enhancing the throughput rather than the single-core clock frequency. In this way, the architecture parallelism was exploited to speed up the algorithms' execution, splitting the workload onto multiple cores, active simultaneously.

Although it allowed the process scaling economics to still hold for a while, the multicore scaling led to increasingly power-constrained systems, to the point that, today, the energy consumption of systems is doomed to keep worsening from one technology generation to the next. This phenomenon is directly linked to the so-called *dark silicon* issue, i.e. the amount of transistors under-utilization caused by the non-ideal scaling [4]. On the one hand, most of the algorithms fail to fully exploit the availability of hundreds, or even thousands, of cores. On the other hand, even if some algorithms could ideally use the entire system, the dynamic and static power consumption together with the related heat generation, caused by the enabling of all the embedded devices at the same time, would be prohibitive, at least at full frequency. Thus, only a subset of the multicore system can be safely switched on at a given time [5]. Besides, this requires developing a set of methods to enable only the parts needed for the algorithm execution efficiently. Moreover, the kind of application and the required performance level must be taken into account while carefully designing these procedures. Alternatively, to tackle this issue, also referred to as *utilization wall*, other solutions exploit the die area availability to customize a portion of the system for the applications at hand, to reduce the energy consumption when the chip must be underclocked [6].

Despite these attempts and the improvements in the integration density and amount of memory embedded in a die, since 2005, the performance gain across technology generations has muted, pointing out a slowdown in the systems performance exponential trend [1]. The cause is traceable to both architectural and technological aspects concerning the physical structure and the usage of the memory inside a system. Nowadays, the most diffused system organization is a more complex adaption of the Von Neumann paradigm, which is based on a clear division between processing and storage elements [7]. On one side, there is the CPU, which is in charge of

executing all the computations involved in a given application, while, on the other side, there is the memory, which holds both input and output data that the CPU has to elaborate. So, this structure relies on a heavy data flow connecting the memory to the CPU and vice versa, implemented through a bus system. This data exchange is the source of another performance-limiting issue, known as the *memory wall* [8]. Even though memories have undergone the scaling process over the years, increasing their speed and storage capacitance, their performance improvement rate has failed in keeping up the same pace of the microprocessors enhancements. It follows that the CPU receives data at a rate much lower than the one at which it can run. It means that all the efforts spent to empower the systems are partially wasted because of the limited bandwidth of the memory (*Von Neumann bottleneck*).

To deal with this bottleneck, one of the leading solutions in modern systems is the memory hierarchy implementation [9]. Since having larger memories means slower data access time, instead of placing a single huge and slow memory that interfaces with the CPU, data are distributed over a chain of different sized memories (physically organized in progressive order: the smallest and fastest memory close to the CPU, while the biggest one on the opposite side). During the algorithm execution, they are dynamically copied from one memory into another, according to their access frequency and their relative physical position in the main memory. So, the attempt is to have all the data that the CPU needs at a time stored in the smallest memory to minimize the data access time as much as possible. However, it is not always possible to meet this condition; that is the reason why the *memory wall* has not been overcome yet [8]. Moreover, especially for the latest data-intensive applications, the *miss rate* (the rate at which the CPU needs to access data that are not stored in the closest memory) can be very high. This because they require to process a considerable amount of data (high data transmission bandwidth needed), which, in addition, are typically distributed over an ample memory address space. It follows that memory latency heavily affects the overall system performance. Furthermore, this data exchange also negatively impacts power consumption, as each memory access involves an energy expense. This energy changes according to the memory size (grows as its square root), so the high *miss rate*, proper of heavy data crunching applications, makes the memory access the main contributor for the dynamic power consumption.

Hence, to minimize the power expense, the memory at the lower level (the main one, typically a DRAM, Dynamically Random Access Memory) should be kept, as much as possible, in the idle state (issue referred to as *dark memory* [10]), so several techniques regarding the algorithmic optimization, aiming to reduce the DRAM accesses, are being developed. While, from the technology side, another proposed item is the *Hybrid Memory Cube* (HMC) [11], [12]. It consists of the memory bandwidth enlargement for the data transmission and the reduction of memory access energy, carried out thanks to the design of a 3D DRAM memory. This memory is characterized by a stack of heterogeneous die layers, each optimized for speed and concurrency. The storage layers are interleaved with application-specific logic dies in charge of efficiently handling the DRAM functionality. The 3D structure allows having a memory with a large storage capacity coupled with a denser network of interconnections, which helps in reducing the distance signals travel, resulting in dropped memory access time (memory latency) and lowered energy consumption. Besides, the compact nature of this memory suggests bringing as much memory as possible close where the data manipulation occurs, for approaching even more low power and high bandwidth systems.

However, again this approach is not conclusive to counteract Moore's trend slowing as a stand-alone solution, but, following this path, the last glimmer of light seems marked by the strict integration and interaction between advancements in both technological and architectural aspects. Concerning the former, today, research is looking for alternative technologies based on different operating principles which do not suffer from the voltage scaling limit and allow, at the same time, the monolithic 3D integration of processing and memory elements [1]. Nevertheless, this integration implicitly requires the design of new architectural paradigms able, among others, to effectively exploit the merging of computing and storage items.

In here, only the architectural side will be investigated, particularly deepening solutions as the *Processing-in-Memory* (PiM) approach and the *Coarse Grained Reconfigurable Architectures* (CGRAs) (see chapter 2). Both ideas move away from the classical Von Neumann organization, heading for a more distributed and heterogeneous way of performing the computations inside the new system, which can count on various processing blocks of different nature, each fitting a set of functions

marked by some well-known features.

CGRAs represent a more conservative way to tackle the performance scaling issue. They fall in the middle way between general-purpose and Application Specific Integrated Circuit (ASIC) solutions and do not require a complete revolution in the system technology and organization. In general, there are different kinds of CGRAs. However, most of them are typically exploited by inserting them in a more complex system and used as hardware accelerators. They are dedicated to executing the heaviest portions of an application, characterized by the frequent repetition of the same set of complex operations applied on a large dataset. Most of them are organized in regular structures, made up of different small blocks or several instances of the same basic block, which can exchange data throughout an interconnections network, whose properties vary according to the actual CGRA. The elementary blocks can perform a variable set of functions and can run in parallel on different data. From the systems performance enhancement perspective, the CGRA winning weapon lies in the reconfigurability attribute of the single blocks (also called Reconfigurable Cells) and interconnections. Thanks to the embedded statically or dynamically configurable elements, it is possible to customize the hardware from time to time, according to the application that the whole system has to execute. In this way, the more specific time and power-consuming functions can be mapped and run by the CGRA block, which can speed up their processing by relying on the hardware implementation of these tasks. Then, further acceleration can be obtained if, in parallel, the microprocessor in the system is forced to take care of the simpler sequential algorithm parts for which it has been designed and optimized. Acting in this direction, the execution time for a given application results lowered, leading to a reduced total energy expense. Thus, up to some extent, to achieve better performance, it becomes not strictly necessary to keep increasing the number of transistors per unit area (which, by the way, would lead the power density to explode), but it is sufficient to spatially reprogram the silicon to fit the requested elaboration in time [13].

As opposed to CGRAs, *Processing-in-Memory* solutions strongly lean on the projected technology advancements in terms of convergence between logic and storage. PiM represents one of the main themes research and industry have been focusing on over the years; therefore, in literature, there is plenty of material and declinations

of this concept (more detailed classifications are drafted in chapter 2). However, the common ground lies in getting away from the CPU-centric computational paradigm to move towards architectures where the boundary between memory and processing becomes blurred. By either bringing the memory very close to the computational logic or directly implementing a memory block provided with processing capabilities, PiM systems aim at reducing the latency due to the data exchange between the storage and the computing blocks and the related energy expense. Regarding the more "extreme" PiM approaches, which see simple logic functions performed directly inside the memory cells, the goal is to bypass the data transmission. Data do not need to be moved from the memory to the processing unit, which results in the full exploitation of the internal memory bandwidth and the heavy drop of latency and energy expense contributions, given by the memory accesses and the data travel along with large distances [14]. Additionally, all the memory cells would work simultaneously, processing in parallel a massive array of data, which could provide further improvements in terms of total algorithms execution time, especially for data-centric applications [15].

Yet, all of this could be gained at the expense of generality loss, on the contrary of well-established programming models (mainly developed for sequential algorithms) which aim at abstracting the instruction set from the hardware, simplifying the software designer task, the backward code compatibility, and so on [1]. For this reason, it is important to underline that the final goal is not to replace the old frameworks with the new proposals completely but to integrate both items into the same system and make them interact. In this way, the advantages coming from both worlds could be exploited to achieve the best performance ever, hoping for the birth of a new era beyond Moore's law one.

This thesis focuses on the design of a new architecture, called *General Purpose Logic-in-Memory Architecture* (GP-LiMA), which embeds and combines both the strengths of the mentioned approaches. The proposed system core is based on a set of programmable elementary blocks (called Smart Blocks), organized in a mesh-like fashion made of both processing and storage elements. So, following the PiM paradigm, the GP-LiMA behaves as a smart memory array capable of elaborating the data it holds, each time choosing the operation within a set defined by the

processing elements included in the single blocks. However, these instructions can be as complex as the ones performed by the Reconfigurable Cells. Then, similarly to CGRAs, part of these processing elements can be reconfigured at will, e.g. each Smart Block can be provided with a Look-Up-Table (LUT), whose content can be statically configured before the memory is started in the processing mode. Moreover, always taking a cue from the CGRAs structures, the mesh of blocks (called GP-LiMA Matrix) is filled with various programmable interconnections, aiming to enhance the data exchange inside the smart memory. The interconnections and the Smart Block reconfigurability result in a more flexible and finely customizable datapath. Indeed, the achieved flexibility makes the GP-LiMA instructions set even wider than the one usually affordable by common CGRAs, making it suitable to be used as an actual co-processor. Hence, the ultimate idea is to insert the GP-LiMA in a standard CPU-centric system environment, where the CPU can interact with the GP-LiMA by accessing it as a standard data memory or enabling it to run in the processing mode. An example of the general system behaviour could be the following: when the whole system is required to execute a heavy data crunching application, at the beginning, the GP-LiMA content is initialized with the data to be massively processed, then the CPU starts the algorithm and entrusts the heaviest parallel parts of the code to the GP-LiMA, while it keeps running the simpler and conditional parts. For this reason, the GP-LiMA Matrix is supported by a control part which enables the architecture to work in the processing mode, running a real Multiple-Single Instruction Multiple Data (M-SIMD) sub-program by itself without the need for intervention from the CPU or other control blocks. Concerning the benefits the GP-LiMA usage aims to provide, they are the same as the PiM solutions, i.e./ reduced memory accesses and data traffic between CPU and memory, meaning lower total energy consumption and execution time, but without losing in programming generality, thanks to the reconfigurability attribute and the co-processor functional mode. Furthermore, the GP-LiMA stands out just for the expected massive drop of the algorithm execution time that is carried out by relying on multiple mechanisms: the straightforward Logic-in-Memory attribute almost zeros the latency, due to the memory access time and the long-distance signals travel, and allows to exploit the full data memory bandwidth for the heavy parallelization

of data-intensive computational tasks (M-SIMD mode); then, the programmability and reconfigurability features speed up the more complex functions, by making feasible their hardware implementation, and cut down the number of instructions needed for the algorithms execution; at last, the further processing parallelism due to the tasks split between CPU and GP-LiMA adds up, still curtailing the total processing time. Moreover, as a consequence of the decreased algorithm execution latency, by using the GP-LiMA, even a further cutback in the total energy expense could be reached.

Summing up, the idea behind this thesis is to design an M-SIMD co-processor based on a Logic-in-Memory array, which is characterized by semi-reconfigurable macrocells that, in turn, are fully interconnected to guarantee a dense exchange of data and the highest possible degree of programming flexibility, with the purpose of achieving a massive reduction in the algorithms execution time and energy consumption.
Here, the thesis is organized according to the following outline:

- ***Chapter 2* - State of the Art** is divided into two macro sections dedicated to the taxonomy of CGRAs and PiM solutions, respectively. Both start with a brief explanation of the general concept and then list the main developments present in literature, dwelling on the more meaningful ones in view of the new proposed architecture. The advantages and disadvantages linked to each approach are highlighted to give a roadmap on the main points the GP-LiMA focuses on to reach the final goal.

- ***Chapter 3* - Programmable LiM** details an already existing architecture (called PLiM), belonging to the PiM world, which represents the starting point for the development of the actual GP-LiMA paradigm. At the beginning, the PLiM general structure is described (referring to the original model), deepening all the composing blocks and the related aims, strengths, and weaknesses, which were taken into account during the design of the GP-LiMA. Then, it follows a sections showing how this architecture was handled at the beginning of this thesis work so as to implement a set of benchmarks used to evaluate the PLiM behaviour and the linked performance accomplished. This part will

be addressed at the thesis work end while dealing with the GP-LiMA performance and used as a comparison term to estimate the goodness of the achieved results.

- ***Chapter 4* - GP-LiMA Paradigm** shows the idea behind the GP-LiMA developed during this thesis, starting from the environment in which it is supposed to work in, then explaining the LiM Matrix composition, together with the Smart Block structure, and deepening the programmable interconnections and their possible implementations, taking into account the timing and power constraints issue. The chapter ends by dealing with the control part and explaining the provided Instruction Set Architecture (ISA) and how the whole architecture can be handled in the different functional modes (M-SIMD or SIMD).

- ***Chapter 5* - GP-LiMA Performance** focuses on a specific implementation of the GP-LiMA paradigm in order to retrieve a synthesized structure on which an estimation of the performance can be carried out. Here, all the techniques applied to obtain an optimized netlist both after the synthesis and the place&route are discussed. Then, the performances reached by the are reported and compared with the ones accomplished by the PLiM architecture on the same benchmarks, as already anticipated.

- ***Chapter 6* - Conclusions** summarizes the main features of the GP-LiMA and how it succeeds in reaching the target goals. At the very end, some open questions are proposed that may lead to future enhancements for the designed architecture.

Chapter 2

State of the Art

This chapter focuses on the proposals existing in literature to cope with the *memory wall* issue. It deepens two macro architectural solutions from which this thesis work lets inspiration, such as CGRA and PiM.
Both of them aim at reducing the execution time and the energy expense but tackling different aspects of the standard Von-Neumann paradigm. The former introduces a coarse level of hardware parallelism in a common CPU-centred system by inserting a re-configurable block working alongside the CPU, while the latter directly heads the memory issue by changing the role and functionality of the memory itself. This last idea relies on a new kind of memory embedding processing capabilities that, in the first place, allows to strongly reduce the number of memory accesses (main cause for dynamic power consumption and latency) and provides a deeper level of hardware parallelism. Both of these achievements contribute to strongly lower the overall algorithm execution time, which, in turn, leads to further energy savings besides the ones gained from the drop in memory accesses number.

Thus, this chapter revolves around the specific features of both CGRA and PiM architecture, so laying the first foundations for understanding the GP-LiMA framework.
The discussion is organized as follows:

- ***Section 2.1*** **- Reconfigurable Architectures** deepens the Reconfigurable Architectures principle, starting by briefly overviewing Field Programmable

Gate Arrays (FPGA) and then proceeding by showing some examples of different types of CGRAs developed over time. The section concludes with a specific CGRA, called Morphosys, which played a relevant role in the design of this thesis architecture proposal.

- ***Section 2.2* - Processing-in-Memory** deals with the PiM idea, pointing out the different interpretations of this concept. As in the previous section, a list describing the consequent implementations is drafted, emphasizing the PiM derivation on which the GP-LiMA is based.

2.1 Reconfigurable Architectures

Reconfigurable Architectures systems category involves devices that tend to work like ASIC but exploit, at the same time, the silicon plasticity to provide dynamic functional reconfigurability [13]. Reconfigurable architectures aim to achieve the computational efficiency proper of ASIC solutions while still retaining some functional flexibility by adapting the hardware at compile time when the task to be run changes. In this way, the limit of ASIC systems, which is the constraint to run always a single specific application set during the design phase, can be mitigated up to some extent while exploiting the ASIC potentiality to directly implement complex functions in hardware, saving energy and time. Thus, thinking at the system's design solution space, built looking at programming flexibility and computational efficiency, these systems are placed next to ASIC solutions (very high computational efficiency but no programming flexibility) one step towards the general-purpose systems (maximized programming flexibility and low computational efficiency).

2.1.1 FPGAs: Field Programmable Gate Arrays

One of the well-known kinds of reconfigurable architectures is the FPGA. It is a prefabricated electrically programmable device, which can be set to implement almost any kind of digital function [16]. The programmability attribute consists of the ability to decide the function to be executed after the end of the silicon chip fabrication, and it is retrieved by switching the technology (standard cells) used to implement most of the common devices. In particular, one of the leading technologies employed

in FPGAs is the SRAM (Static Random Access Memory). Moreover, to properly fit a large set of tasks, the granted reconfigurability is at a very fine-grained level, mostly at single bit granularity. For this reason, an FPGA is structured as an array made up of different types of blocks. These ranges from programmable blocks (Configurable Logic Blocks - CLBs), aimed at emulating simple logic operations, to memory components and to more complex elements like DSP blocks (Digital Signal Processing blocks), which include multipliers, or even to a small microprocessor [16]. All of these are immersed in a dense programmable routing network which allows the blocks to be fully inter-connectable (see Figure 2.1). At the boundaries, this array is surrounded by a frame composed of configurable input/output blocks used to interface the chip with the outside world or even to make it work inside a larger system. However, the main element on which the FPGA is built is the CLB, which is the finer-grained component used to implement a simple combinational or sequential function that defines a single bit signal, returned at the block output. The insertion of a small SRAM LUT, typically on 4-6 inputs, makes the function programmable and then runnable. This small memory stores all the values the 1-bit output signal has to assume, one for each of the possible combinations of inputs values, defined by the function the LUT is associated to [13]. Besides, the value of the final CLB output can be further chosen by programming the selection signal of a multiplexer which takes as inputs the LUT output, its complement, the output of a memory element taking the LUT output, and again its complement. Once more, to be programmed, the value of the control signal is stored in a 2-bit SRAM element. The generic scheme of a CLB is depicted in Figure 2.1. Thus, by interconnecting several CLBs, it is possible to accomplish an arbitrarily complex logical function. The routing network comprises a considerable number of wires whose connections are established thanks to programmable switch matrices put at each wires intersection for this scope. The content of small SRAM blocks drives the configurations of the switches. It follows that the behaviour of an FPGA device is programmed at compile-time by initializing the content of all the SRAM components, so almost like a simple memory writing operation. Therefore, the function implemented by the FPGA can be changed a considerable number of times after fabrication, always before the start of the actual algorithm execution.

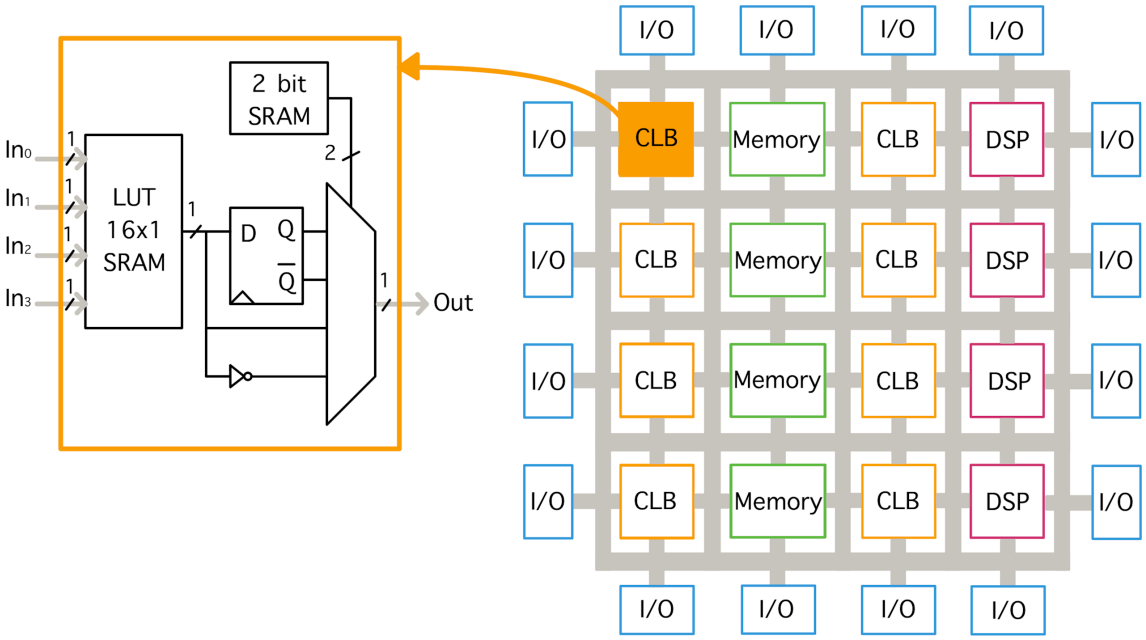


Figure 2.1: The arrangement of the blocks surrounded by the routing network in the FPGA framework is shown on the right. It is a simplified scheme that does not include the actual number of CLBs, which is way higher. On the left, the internal structure of a generic CLB is detailed.

Since FPGAs implement circuits running at a frequency two orders of magnitude lower than the respective standard-cell ASIC solutions, they were initially thought of only for prototyping purposes as a preliminary step before producing ASIC devices. Nevertheless, in early 1990, they started to be also considered as a possible stand-alone solution to implement real working devices, especially employed in high-performance computing, and even for general-purpose computing [13]. However, they are affected by several limitations which prevent them from being suitable for all the possible kind of applications. First of all, the single bit-level granularity makes the operations among multiple-bits data intricate to be performed. These must be implemented through the composition of several processing units, which, in turn, requires a massive usage of the programmable interconnections, leading to a significant routing overhead and so to a device with low area usage and power efficiency. Furthermore, the high number of programmable elements (including CLBs and routing switches) involves the need for a massive amount of configuration data that must be permanently stored in a separate memory and loaded into the FPGA

each time the device is switched on. This required data movement implies a further overhead in the power consumption and slow configuration times compared with the standard general-purpose systems. These drawbacks make them strongly unsuitable for algorithms demanding a frequent change in the system configuration. At last, the development of applications to be mapped on FPGAs cannot be carried out using common high-level languages, like the one used for microprocessor-based system, but it still requires the knowledge of hardware description languages (HDLs) used for ASIC designs, which are prerogative of hardware experts [17]. Besides, the FPGA Electronic Design Automation (EDA) flow takes longer compilation times than the ones taken by software compilation tools [13]. Then, the low operational frequency, intrinsic to the programmable technology, adds up to all of these limitations.
Therefore, since the major problem seemed linked to the extreme fine level of bit processing, another branch for the Reconfigurable Architecture was derived from the FPGA paradigm, i.e. CGRAs.

2.1.2 CGRAs: Coarse Grain Reconfigurable Architectures

CGRAs are devices belonging to a particular branch of the more general reconfigurable architectures systems classification relying on a multiple-bit wide datapath. CGRAs give up part of the flexibility given by the single bit-level configurability to favour a more direct mapping of a more extensive set of applications on the provided datapath by including already specialized complex operators directly in silicon. At first, the goal is to avoid the routing overhead due to the connection of multiple processing elements required to implement instructions running on common length-wide data (more than 1-bit) [17]. Secondly, it also turns into the improvement of the performance in terms of higher clock frequency allowed [13], due to the use of logic macro blocks already optimized. However, consequently, the interconnections are built upon multiple bits, which means a higher area usage. Howbeit, this con is overcome by decreasing the number of processing elements inside the device, which implies scaling back the size of the routing network [17]. This downsizing can be done without affecting the number of applications that can be successfully mapped on the CGRA since the single instances are more powerful at the processing level

than FPGAs; therefore, fewer blocks are required to implement the same algorithm. Indeed, the specialization of the small set of blocks composing the CGRA results in the overall enhancement of both device generality and silicon utilization compared with the FPGAs ones. Moreover, the drop in the number of processing blocks leads to a smaller set of configuration data to be permanently stored and then loaded into the device before it starts working. The implied data movement reduction results in faster configuration times[13] which bring benefits to the device static configuration process, but, more important, they even enable, in specific cases, dynamic reconfigurability useful for algorithms asking for a steady change of the runnable instructions. Lastly, the more defined composition of the datapath does not strictly require the knowledge of HDLs to properly program the device when developing and uploading the addressed applications.

It follows that CGRAs are reconfigurable devices made up of a limited set of specialized processing units running on multiple-bit data, which can be programmed and interconnected at the user's will both dynamically (the instructions they implement can change during the application lifetime) or statically (the device behaviour is fixed and set before the application starts). Similarly to the FPGAs, the programmability attribute refers both to the single processing elements and the routing network in which they are immersed. Just because of the actual specialization of the basic blocks, different CGRAs have been developed over time, each more customized for a particular target of applications. In general, they are based on an array of programmable processing blocks, called Reconfigurable Cells (RCs), whose actual composition and interconnections vary according to the specific CGRA, and are, in practice, primarily used as hardware accelerators. They run in parallel to the CPU of the system they belong to and are dedicated to the more complex and specific algorithm parts or data-intensive portions, which can benefit from the availability of multiple processing units (operating on different data at the same time) to speed up the overall program execution. Therefore CGRAs can be used as co-processors, with a customizable working mode.

A generic example of the internal structure of an RC is shown in Figure 2.2. The core of the processing is represented by the ALU whose two operands are retrieved from the multiplexers, driven by the configuration register, selecting among the data coming from the interconnections with other RCs and the content stored in the small

(SRAM) memory embedded in the same RC. The RC programmability is given by the ability to choose which of the available operations the ALU must perform at a specific time and to select the current operands it elaborates by exploiting the configuration register. This register plays the same role as the SRAM elements composing the CLBs of the FPGAs, so it is the element to be initialized during the device configuration, aimed at customizing the CGRA for the application at hand. The data stored in this register is divided into multiple fields, each associated with a different element in the RC block, which needs to be appropriately driven (see Figure 2.2). Furthermore, early CGRAs often include a LUT configured together with the configuration register to enlarge the set of implementable operations. Besides, it is worth specifying that the configuration register can be made up of multiple registers in order to save different configurations which can be dynamically applied at run-time in sequence (cycled), or at will, according to the specific CGRA functioning [13], or better to the particular flow-control of data it implements [13].

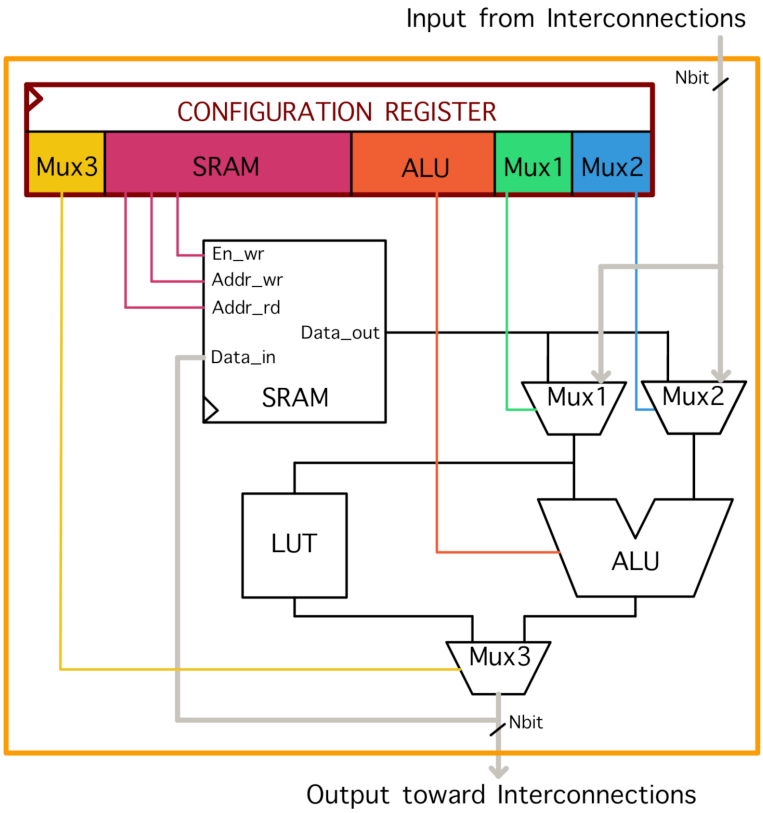


Figure 2.2: Generic internal architecture of a RC.

In general, CGRAs can be classified relying on the following features [17], [18]:

- Granularity, namely the datapath width, which roughly ranges from 2-4 bits to 32 bits (typical values are 16 and 32 bits). Some CGRAs are provided with a mechanism to facilitate the composition of a datapath with higher width like it happens for FPGAs, so, in practice, the user can choose among a set of possible data length;
- Interconnection Structure, which can be shaped as
  - a *linear array*, where all RCs are organized along a line and can interact only with their neighbours (mainly thought of for mapping pipelines without forks onto the CGRA);
  - a *mesh*, that is the most diffused arrangement which sees all the blocks placed according to a mesh-like topology where only blocks belonging to the same row or column can exchange data, favoring efficient parallelism;
  - a *crossbar switch*, that is the most flexible kind of interconnection which enables all the possible routings among all the array blocks, which, however, leads to a high overhead in terms of area occupation;
- Programmability, i.e. the depth of the configuration registers, also called context registers, embedded in the architecture. According to this parameter, CGRAs can be distinguished in
  - *single-context systems*, which are devices that can hold only one configuration at run-time, so they are always limited to perform the same task after the context registers are initialized during the context loading phase;
  - *multiple-context systems*, whose context registers can store several configurations at a time for the same CGRA. So doing, these kinds of devices can change the function they implement even without repeating the context loading phase, by simply pointing to a different location in the configuration registers;
- Reconfigurability, that is when the CGRA can change its configuration and so the instructions it performs. It can be:

  - *static*, the reconfiguration can only occur if the CGRA interrupts the program execution; in which the RCs can be programmed only at compile-time, before the start of the program execution;
  - *dynamic*, the configuration registers content can change even at run-time; that is typically feasible in multiple-context systems, where it is possible to write locations that are not currently used to drive the CGRA logic components;

- Computation Model, namely depending on its peculiar structure, each CGRA can adapt to implement a certain set of functional modes, including *SISD* (Single Instruction Single Data), *SIMD* (Single Instruction Multiple Data), *MIMD* (Multiple Instruction Multiple Data), *M-SIMD* (Multiple Single Instruction Multiple Data), *linear array*, *VLIW*, *pipelined* mode or other variants.

In the following, an overview of three specific CGRAs implementations is reported together with Table 2.1, which summarises their features according to the mentioned CGRA parameters. All architectures are examples of the early influential CGRAs, which are more relevant for this thesis design. Notably, the last faced architecture is the one that paved the way for the current GP-LiMA structure.

CGRA	**Granularity**	**Interconnection Structure**	**Programmability**	**Reconfigurability**	**Computation Model**
DPGA [19], [20]	1 bit	crossbar	multiple context	dynamic	SISD, SIMD
PADDI [21]	16 - 32 bit	crossbar	multiple context	static	VLIW, SIMD
MorphoSys [18]	8 - 16 bit	mesh	multiple context	dynamic	M-SIMD

Table 2.1: Classification of the dealt CGRAs according to the listed features.

DPGA: Dynamically Programmable Gate Array

The DPGA [19], [20] is the link between FPGAs and CGRAs. It still supports a bit-level computation paradigm, but, unlike FPGAs, it follows the multiple-context programming paradigm coupled with a dynamic type of reconfiguration capability. In practice, the DPGA can be seen as a hybrid architecture standing in the middle between an FPGA and a SIMD array. In each of the programmable components, they can concurrently reside up to 4 different configurations that can be switched

at run-time thanks to the same global instruction, called context identifier (CID), which is forwarded to all those elements. Notably, each reconfigurable element is associated with its own DRAM context memory, composed of 4 registers, which is addressed by the 2 bits CID signal. Then, the CID value can eventually change at each clock cycle, and it points to the configuration data that have to drive the logic elements inside the DPGA currently. These lasts are organized into nine macro sections, called SubArrays, which are placed following a regular grid-like path and exchange data through the first level of interconnections (called global interconnects) implemented with crossbar switches (see Figure 2.3).These crossbar interconnects (see Figure 2.4) are a set of multiplexers (one for each signal output by the crossbar switch), each taking as inputs all the signals entering the crossbar switch and returning one of the signals composing the final crossbar output. The selection performed by the multiplexers can be programmed thanks to the context memory element included in each interconnect block and driven by the broadcast CID.
Furthermore, all SubArrays are made up of 16 RCs and are placed on composing a 4x4 matrix which is internally fully routed thanks to a mesh-like topology of connections (called local interconnects), as shown in Figure 2.5. Each array element, i.e. the RC, works on a single bit and, like FPGAs, relies on a 4 inputs 1 output LUT to implement the programmable logic function. Indeed, since the DPGA must allow the user to choose at run-time the context to be run among four configurations, inside each RC, 4 LUTs are inserted. All the LUTs outputs are forwarded to a multiplexer driven by the context memory addressed with the CID value. Then, the final RC output is chosen again by the context memory output that selects among the multiplexer output and the output of a flip-flop that takes as input the same signal coming from the previous multiplexer. Then, the 4 LUTs inputs are given by the inputs selector 15-to-4 taking 15 1-bit signals: 8 coming from the global interconnects, 6 from the local ones, and one directly from the inside of the RC itself. Again, the selector is driven by the related field in the pointed context memory location. The RC internal architecture can be seen in Figure 2.6.
Summing up, thanks to the different context memories present in each of the configurable components, the RCs can concurrently run a different function on a different data, as it happens for FPGAs, but the specific function they execute is pointed

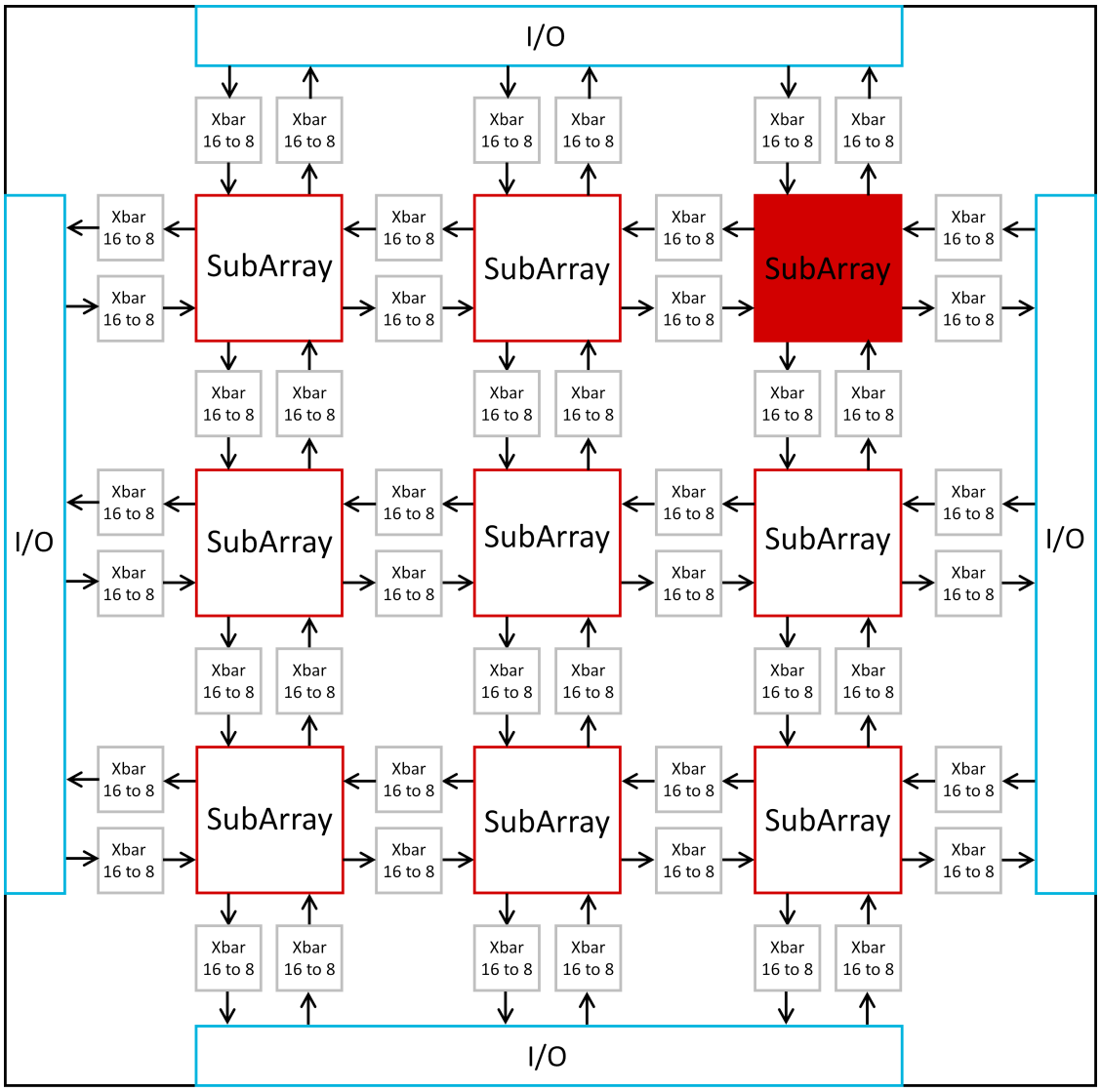


Figure 2.3: DPGA platform arrangement overview. The shown crossbar blocks filling the SubArrays grid make up the global interconnects.

for all the RCs by the same CID instruction, as SIMD architectures behave. Moreover, concerning the programmability, to update the configuration memories content (which can occur even at run-time for the unused locations), programming lines are distributed all over the DPGA. At last, the presence of 2 levels of interconnections confers on the DPGA a reasonably high degree of connectivity, which, in turn, implies further programming flexibility.

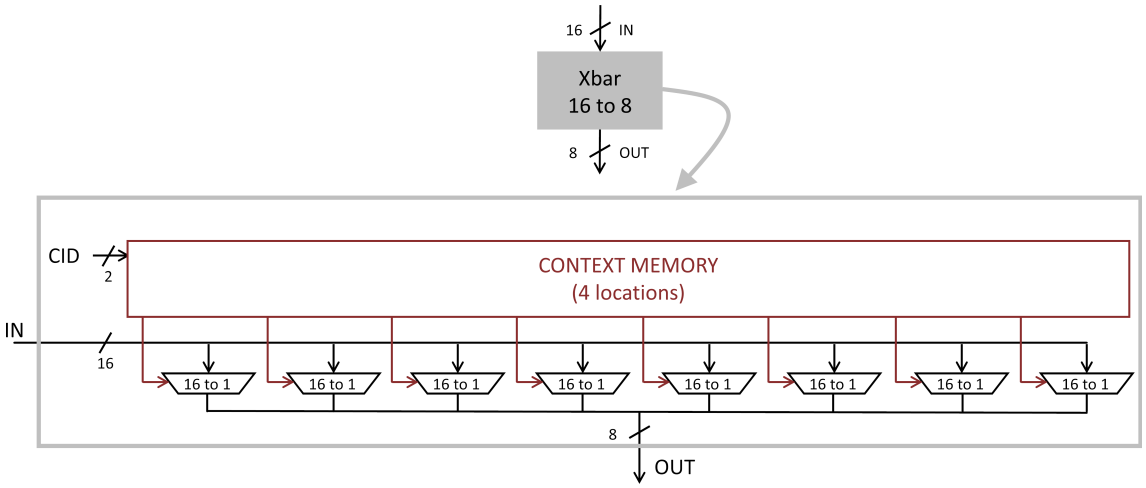


Figure 2.4: Internal structure of a programmable crossbar switch.

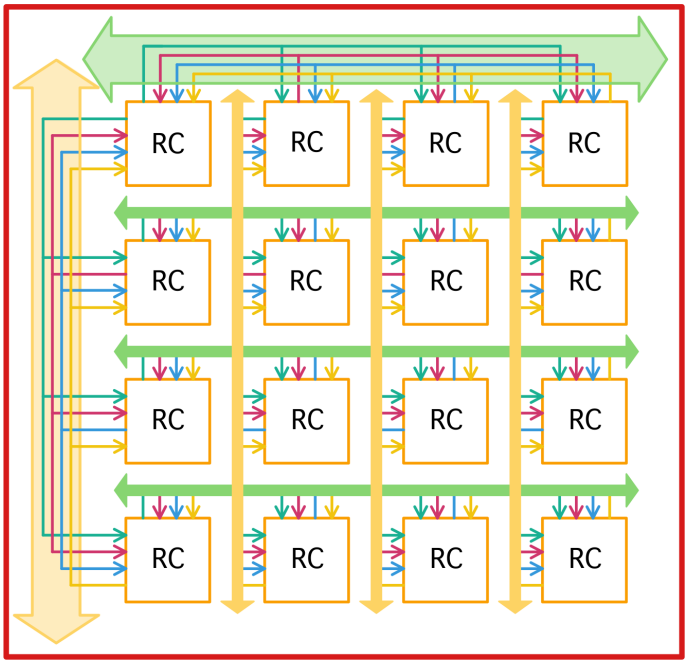


Figure 2.5: Internal layout of a SubArray composed of the RCs mesh and the local interconnects.

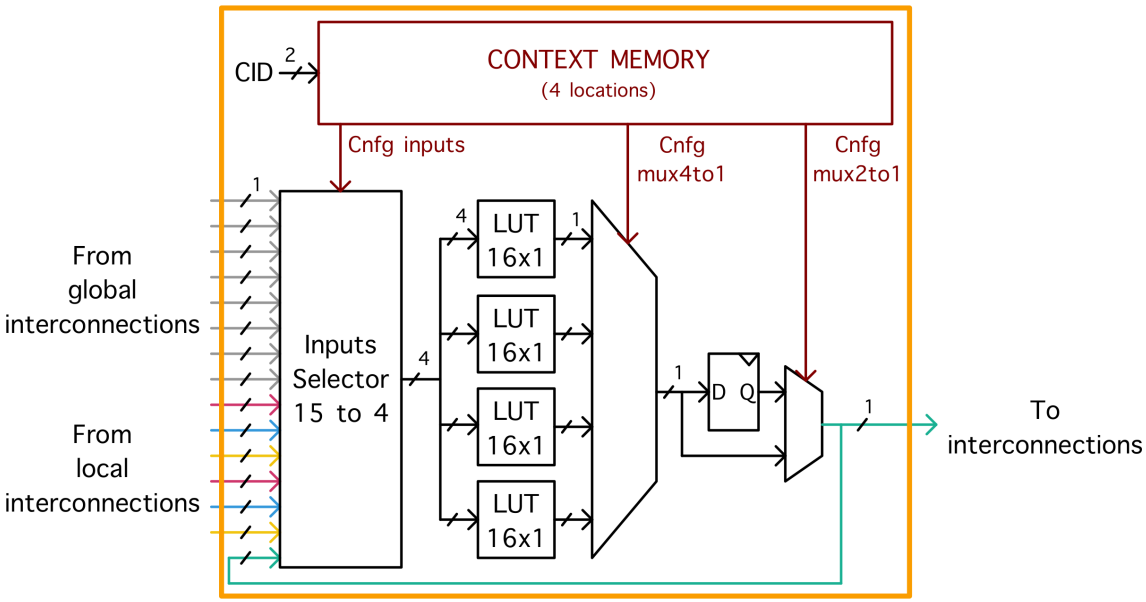


Figure 2.6: RC scheme.

PADDI: Programmable Arithmetic Devices for High-Speed Digital Signal Processing

PADDI [21], [17] has been mainly thought of as a stand-alone system for prototyping purposes of DSP applications that require multiple fast arithmetic units working in concurrency and sharing a dense and flexible routing network. As DPGA, the chosen interconnections paradigm leans on the dynamically controlled crossbar switch type, which ensures a conflict-free data exchange aimed at maximizing the usage efficiency of the programmable execution units (called EXUs, referred to as RCs in a generic CGRA). The hardware platform has a granularity of 16 bits and is divided into 4 clusters of 8 EXUs and eight local controllers (CTLs, i.e. the context memories) each, as shown in Figure 2.7.
The EXU is made up of 2 dual-port register files (RFs containing six registers each) for concurrent read and write operations. Both RFs take their inputs from the related multiplexer that selects among the data coming from the communication network and the result of the previous operation from the same EXU. Then, the RFs provide the inputs for the logical core (ALU), which implements a set of functions given by the carry select adder (sum, subtraction), the logarithmic shifter

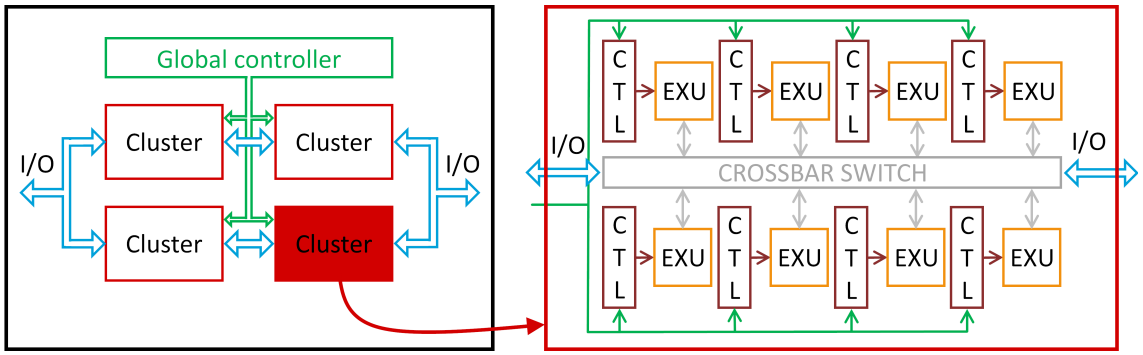


Figure 2.7: On the left, the PADDI system overview is depicted. On the right, the inside of a single cluster is zoomed.

(right shifts), and the hardwired comparator. Additionally, thanks to the reconfigurability properties of the architecture, each EXU is provided with an optional pipeline register which can be connected to the final output signal holding the selected result from the ones gathered from the mentioned logical blocks. Moreover, even though the datapath width is 16 bits, it is possible to concatenate two EXUs to achieve an accuracy of 32 bits.

Regarding the programmability of the EXUs, the PADDI framework behaves like a dynamically reconfigurable multiple-context system. Therefore, each EXU is associated with its own context memory, which drives all the embedded components and can store up to 8 different configurations. Unlike the DPGA, all CTLs are SRAMs that support static reconfigurability. Thus, they can be serially reconfigured only during the context loading phase. To do this, CTLs are all connected into the same scan chain, which behaves like a big serial shift register, so that, to configure the whole chip, few external pins are needed (more in detail, eight scan chains are instantiated, one for each different configuration data). While, similarly to the DPGA, these memories are addressed by a 3 bits global instruction, handled by the external global controller, which can change its value at run time even past every clock cycle. It means that the overall PADDI functionality can vary during the application lifetime in 8 different ways, even if these particular behaviours cannot be changed at run time. Thus, the actual value of the eight configuration data must remain fixed till the next setup phase. So, looking at the entire architecture, by unifying the configuration data, output by all the single context memories in

the EXUs in the face of the same broadcast global instruction, a VLIW operating the PADDI functionality is obtained. Therefore, on the one hand, thinking of the EXUs functional mode individually, each of them can implement at the same time a unique function as it works for a MIMD system organization. While, on the other hand, all the EXUs are globally driven by the same macro instruction, pointing out a behaviour similar to SIMD architectures.

MorphoSys: Morphoing System

The Morphosys [18] framework represents the architectural solution that comes closest to the GP-LiMA philosophy. It is proposed as an alternative hardware model to support data-intensive algorithms, characterized by high regularity, intense throughput demand, and especially heavy data parallelism. Examples for these applications are: multimedia, data encryption, DSP, and discrete cosine transforms. In order to fulfill this goal, the MorphoSys paradigm aims at adding to a standard general-purpose system SIMD capabilities, helpful in shortening the execution times for routines requiring the handling of a significant amount of data. Therefore, the idea behind MorphoSys is to build a complete system combining a simple TinyRISC [22] with a reconfigurable (SIMD-like) co-processor and a high-bandwidth memory interface, used to properly move data between the external data memory and the reconfigurable array. Here, the 32-bit RISC processor is in charge of executing the most sequential and straightforward parts of a target application while properly driving the co-processor to perform the high data-parallel portions. The arrangement of the components embedded in the Morphosys chip and their logical goals are outlined in Figure 2.8. Notably, the TinyRISC makes use of the context memory associated with the reconfigurable array, together with new additional instructions introduced in its ISA, to control the instruction flow the co-processor has to carry out for a given application. Moreover, to load the memory configurations and start all the data transfer procedures between the external memory and the co-processor, the RISC also controls the high data bandwidth memory interface, composed of the DMA (Direct Memory Access) controller and the frame buffer (FB). This last embodies the real novelty brought by the MorphoSys framework. The frame buffer acts as a data cache to make the data transfers transparent to the reconfigurable

processor by accomplishing them in parallel during the regular computations.

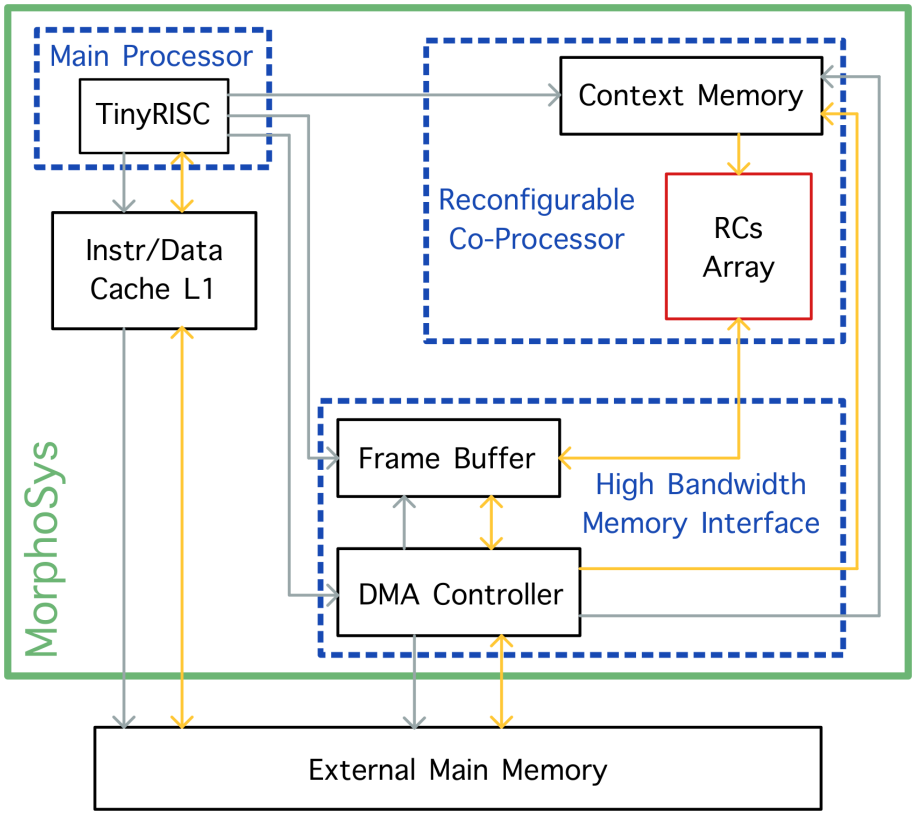


Figure 2.8: General layout of the macro blocks composing the MorphoSys chip. The grey wires indicate the control signals, while the yellow ones carry the actual data.

From the reconfigurable array standpoint, the overall structure follows the guidelines of CGRAs. The core is based on a mesh of RCs with 16 bits granularity and is filled with multi-level programmable interconnections. It is divided into four macro-quadrants, each enclosing an 8x8 array of RCs, as shown in Figure 2.9. In order to allow as much as possible a dense programmable data exchange while preventing incurring an excessive routing area overhead, the communication network has been divided into three hierarchical levels of interconnectivity (physically implemented using three different metal layer technology):

- *nearest neighbour (NN)* level, where each RC is enabled to access the output of the adjacent RCs, at most four signals;

- *mesh* (or *intraquadrant*) level, which also includes parts of the interconnections classified as the previous level, where RCs of the same quadrant can read the data of the other RCs belonging with their same row or column (see Figure 2.9). In this case, each RC can access any of the provided signals, independently on which are the data the other RCs in the same interconnection want to make access to at the same time;

- *bus-like* (or *interquadrant*) level, that is implemented exploiting another hardware technique aiming at global connectivity among RCs of different quadrants, always included in the same row or column. Specifically, each row and each column are associated with two buses, which the RCs are connected to employing tristate buffers. Each bus has a preferred direction for the data transfer, so one bus is written only by the RCs of one quadrant and is read by the RCs of the other quadrant, and vice-versa for the other bus. Only one RC at a time is enabled to write on the bus, while the 4 RCs of the other quadrant connected to the same bus can concurrently take this data as input. For this reason, during the program execution, the tristate buffers need to be properly activated by the configuration data selected from the context memory.

The basic reconfigurable processing elements are designed to resemble conventional microprocessor models, so they include a 16x12 multiplier, too. In order to accommodate MAC (multiply-accumulate) instructions, the ALU block is followed by an output register and operates on 28 bits to be compliant with the possible overflow occurrences due to the product's execution. Among the usual logic and arithmetic functions (it can perform up to 25 different functions), the ALU can also compute the absolute value of a difference and the MAC instruction in a single clock cycle. The RC is provided with some storage capabilities, i.e. it contains a register file (RF) of 4 locations to save temporary data. Finally, the data the RC works on are picked through two multiplexers taking data coming from the entire routing network, the RF and the FB (for the RC internal structure see Figure 2.10).
Moreover, the driving of the processing components that constitute the RC is accomplished by leaning on a context register connected to the global context memory

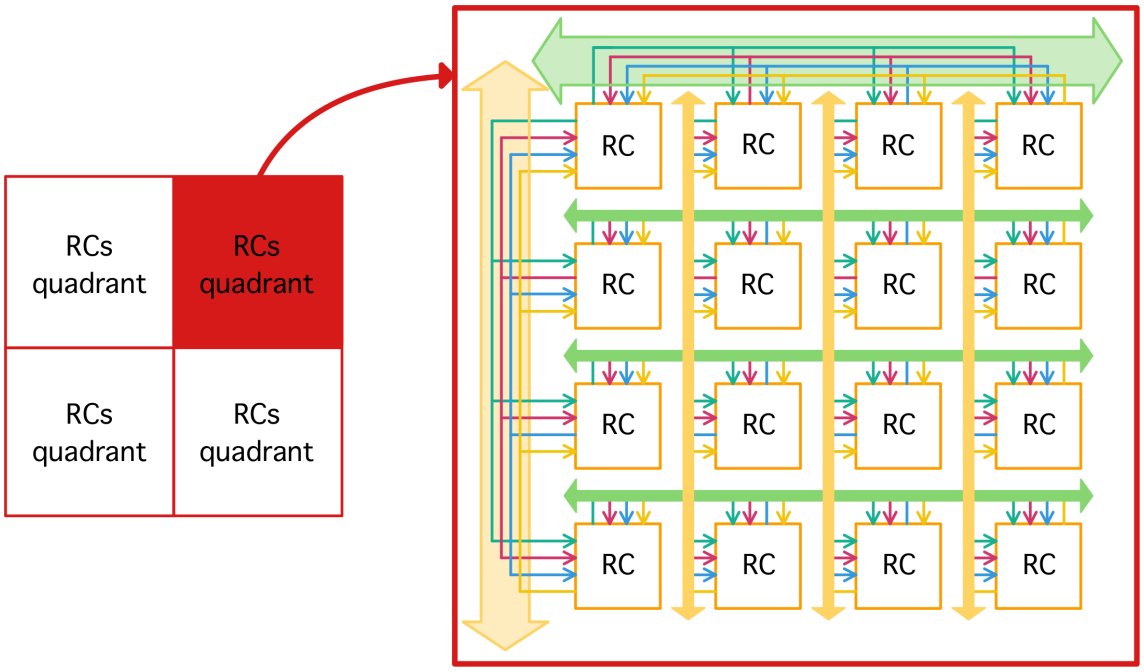


Figure 2.9: On the left, the RCs array partition into quadrants is shown. On the right, there is a zoom on the internal arrangement of the RCs in a quadrant. The delineated interconnections are only the mesh level type.

embedded in each RC. At every clock cycle, a new instruction output by the context memory is loaded into this register. It means that the Morphosys architecture can adapt its functional behaviour at run-time by picking the instruction to be performed from a finite set of directives. Furthermore, this set, available in the context memory, can be dynamically changed simply by overwriting the inactive locations. Deepening the context memory organization, this was designed to be compliant with the M-SIMD computation model. RCs belonging to the same row all compulsorily perform the same instruction, while, at the same time, RCs of different rows can run other instructions. Indeed, at compile-time, the user can decide whether he wants to keep this arrangement or modify the context broadcast so that RCs belonging to the same column are driven by the same instruction. For this reason, the context memory is divided into two macro sections, each dedicated to one of these operating modes. However, this means that the co-processor can perform up to 8 different instructions on different data at a time, which is carried out by a further split of every memory context block into eight sets, each connected to a different RCs row

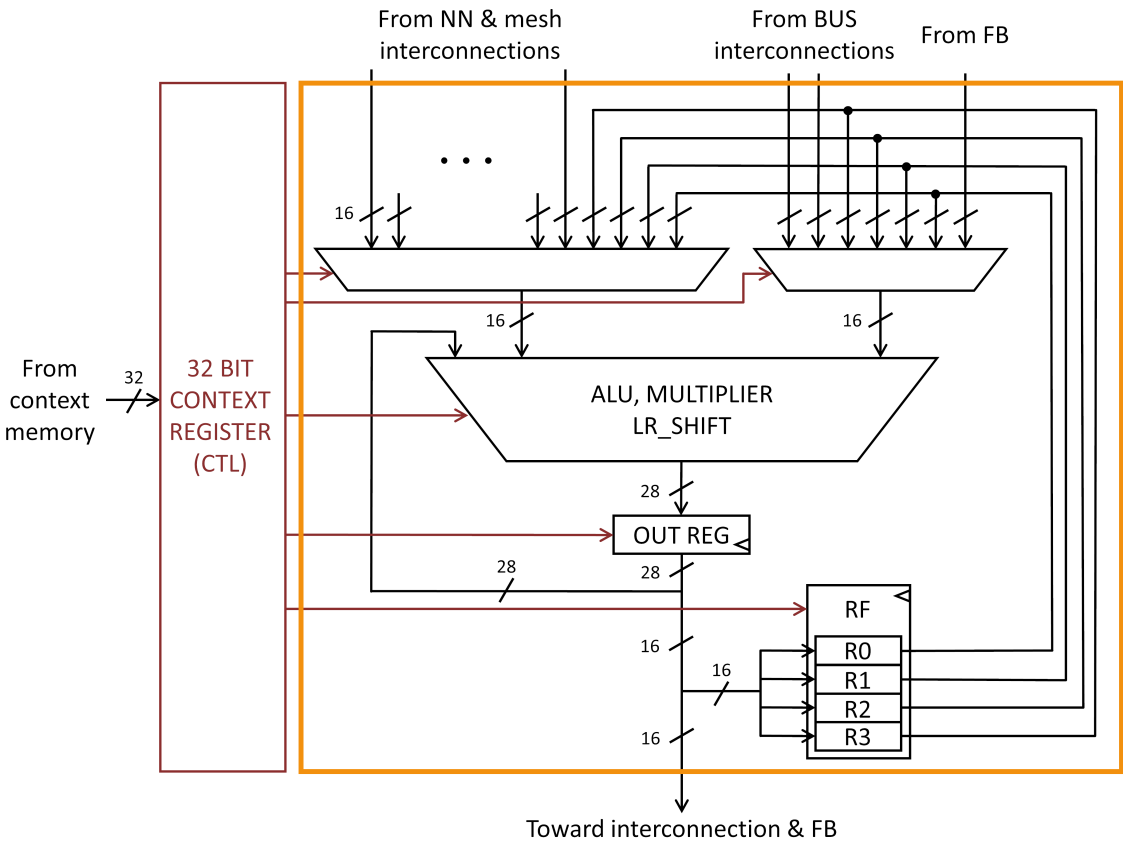


Figure 2.10: Internal structure of the single RC.

(or column). Then, in turn, each set can hold up to 16 different context words, from which the RISC can draw out the current instruction that the related row (or column) must execute. Besides, further functional flexibility is given by the possibility to enable even one single column for a certain time.

To conclude, the MorphoSys architecture has been one of the landmarks while designing this thesis proposal. Indeed, the GP-LiMA framework has borrowed from it the row-wise implementation of the M-SIMD computing model, the coupling of the reconfigurable co-processor with a standard RISC dedicated to its control, the coexistence of different types of interconnections inside the RCs array, and the internal composition of the RCs themselves.

2.2 Processing-in-Memory

The *memory wall* issue, as the name suggests, is mainly linked to the required data exchange between CPU and memory. Indeed, the memory brings along with it some criticalities, like slow access times together with a notable power consumption overhead, limited data transmission bandwidth, and working frequency slower than the CPU one. For this reason, the generic PiM principle aims at shifting the research focus on the use being made of the memory itself inside the system. However, how this research and the consequent models are refined changes depending on the different variations into which the PiM concept can turn. Some solutions take advantage of the physical memory attributes to modify its actual functionalities, while others concentrate on the arrangement of the memory inside the system.
Therefore, to bring some order and facilitate the understanding of the different developments the PiM paradigm has undergone, [9] proposes a taxonomy that is reported here. Four approaches are distinguished:

- ***Computing-with-Memory*** *(CwM)*: it is not exactly a branch of the PiM classification since it is basically an alternative technique that makes use of memory elements to implement logic functions. However, it is worth mentioning CwM to clarify even more by contrast OF what PiM concepts consist of. Indeed, CwM is what most FPGAs and CGRAs are based on, i.e. the LUT item. A memory component (like the SRAM block in the FPGA case) is employed as a processing element to perform the computation required by a generic logic function. The memory is used to store all the values the output of any function can assume for all the possible combinations of the inputs values. Note that, in this case, the memory (LUT) does not hold the data to be elaborated, but these, i.e. the inputs used to address the memory to gain the computation results, must always be taken from another memory. Thus, none of the cons related to the data movement between processing and storage elements is dented.

- ***Computing-near-Memory*** *(CnM)*: it is the first approach that starts going toward the PiM principle. CnM is the most conservative proposal that queries the memory position inside a system rather than its logical purpose. It aims

at physically bringing the memory closer to the processing units so that both the time and energy spent to transfer data from the memory to the CPU and vice-versa are strongly lowered. In works [23] and [24], this is accomplished by embedding the memory in the same CPU die. However, the most representative devices for this PiM branch are the ones achieved thanks to the recent advancements in the die stacking technology, as the commercialization of the HMC confirms [15]. As already anticipated in chapter 1, the HMC is a memory device that stacks multiple layers of memory and control logic to form a 3D structure, where the distances among all the components embedded in the same device are shortened. Then, further upgrades of this concept have been carried out, leading to the current CnM devices. In these systems, layers of storage elements are interspersed with layers characterized by processing capabilities but still identifying separate entities. The 3D integration allows exploiting the vertical interconnections (*through-silicon vias*) so that not only the speed of the data fetching from memory is enhanced, but also the memory bandwidth and the amount of implemented functionalities per silicon area are. One of the most known examples for this kind of system is the Active Memory Cube (ACM) [25], which inserts below the DRAM dies of the HMC a layer filled with sophisticated processing elements [26].

- ***Computing-in-Memory*** *(CiM)*: is what mainly is meant by PiM. This architectural solution leverages the physical structure of the memory itself, like the cell composition and the read/write control logic, to perform some simple data computations directly inside the memory. The idea is to take advantage of its analogue components (e.g. the sense amplifier in the peripheral circuitry of SRAMs or DRAMs) by simply accessing the memory standard functionalities differently so that it is possible to retrieve at the memory output the result of a certain logic operation applied on the data stored in the activated cells. In general, the set of functions a memory can emulate depends on the specific technology it is made of. For instance, in the case of DRAM memories, by just enabling the reading of multiple locations concurrently, the returned output value represents the bitwise-and or the bitwise-or (depending on the stored selection signal) of the data connected by the same bit line and held

by the activated memory cells [27], [28]. Similarly, an SRAM memory can be handled to execute bitwise-and or bitwise-nor operations by reading the involved data locations concurrently [15]. Besides, it is possible to even carry out a matrix-vector multiplication by properly driving with the vector values the word lines associated with the cells storing the matrix values [29], [30], [**?**]. Then, other memory technologies as resistance-based ones can be further exploited for this purpose, such as Resistive RAM (RRAM) [31], [32], Phase Change Memory (PCM) [15], [33], Magneto Resistive RAM (MRAM) [34], [35]. Moreover, after a certain function has been computed, the CiM paradigm provides for the memory to be handled so that the returned result is written back in the same memory. It implies that, for some simple operations, there is no need to bring the data out of the memory and inside the CPU, and then vice-versa, but the processing of a set of data (which can even be as big as the memory size) can occur directly inside the memory itself. Therefore, unlike the previous solution, here the cons due to the memory accesses are almost cut off, leading to lots of energy and latency savings together with a drop in the total algorithm execution time, further pushed by the possibility to exploit the whole internal memory bandwidth. The high data memory parallelism enables to perform the same operation concurrently on different data, resulting in a sharp speed-up, especially of data-intensive algorithms. An example of a device categorizable as CiM is the In-Memory Intelligence (IMI) [25]. It is a SIMD architecture that, starting from a DRAM block, connects one basic processing element (which performs XOR and NAND operations) to each sense amplifier in the memory peripheral circuitry, all implemented through standard DRAM structures.

- ***Logic-in-Memory*** *(LiM)*: it is a branch derived from the CiM one, so it brings pretty much the same advantages. The LiM approach consists of physically modifying the internal memory cell structure by inserting a small set of logic ports connected to the storage part. In this way, besides the standard storage functionalities, the single cell is also provided with computing capabilities, allowing building a more extensive set of logic functions inside the whole memory. The goal is to introduce limited customized logic blocks

each time, which can suit the operations required by the target application in an ASIC fashion. Dissimilarly from the CiM systems, it comes easy to execute functions working on data even belonging to different bit lines. Here the data computations are enabled without moving data down to the memory periphery (as, instead, it happens in the CiM devices), which translates into even shorter processing times and reduced energy expense. However, like CiM architectures, memory read and write operations are always required, but only to enable some specific computations by shifting data between cells, and to update the cells content with the new results, respectively. An example of LiM architecture is the one introduced in [36], where a LiM model customized for the implementation of a binary neural network (BNN) is examined. In each memory cell (1 bit), the storage element output is connected to a xor gate together with a further input signal, while the signal returned by the xor is forwarded to an external logic block that completes the elaboration of the data required by the target BNN.

An evolution of the LiM branch is represented by the PLiM architecture described in chapter 3 which marked the start for the design of the LiM exposed in this thesis. It unifies the LiM concept, namely the inclusion of logic elements inside each memory cell, to the programmability attribute. The PLiM organizes the logic part to be embedded in the cell so that the functions it performs can be changed during the algorithm execution. In this way, there is no need to physically fabricate a different LiM architecture for each application to be run, but the system can be initialized at compile-time with a different program each time.

In order to learn more about PiM architectural solutions, two other systems proposed in the literature are explored. Both hold within them and combine different qualities of the PiM concept, meaning that they assemble more than one technique among the upper listed ones, especially the last framework reviewed. Besides, the first examined architecture is also stated because it gathers some functional similarities with the GP-LiMA framework.

GP-SIMD: General Purpose SIMD

The GP-SIMD [36] is an architecture aimed at seamlessly running machine-learning algorithms, which are composed of a sequence of highly parallelizable subroutines that elaborate a large set of data at a time. Here, the SIMD computing paradigm is exploited to perform parallel tasks efficiently and is coupled with a standard core that takes charge of the sequential parts. However, the massive amount of data that these two parts need to exchange negatively impacts performance and power dissipation issues. The GP-SIMD copes with this data synchronization issue by implementing the SIMD processing part leveraging the PiM paradigm. It comprises a standard CPU and a SIMD co-processor that communicate with the same shared memory having two-dimensional accesses, which solve the data synchronization issue. Thus, the GP-SIMD can be mainly categorized as CnM since, through the SIMD co-processor that physically resides close to the memory array, a set of bit-serial processing units, each associated with a different memory row, is available. Therefore, the SIMD co-processor and the SRAM with modified memory cells combination acts as a large memory integrating computing elements.
As the MorphoSys ecosystem, the GP-SIMD architecture provides a multi-level parallel processing capability. Besides the SIMD parallel operating mode applied on the data in the shared memory, the CPU may start the SIMD co-processor, in a non-blocking manner, to perform a task while it keeps running some sequential instructions. Concerning the actual layout of the GP-SIMD system, this is composed of a sequential processor that exchanges data with the shared memory array through an L1 cache and controls the SIMD co-processor. This co-processor is divided into two sections. One is the datapath made up of all the processing units directly connected to the shared memory array, which, in turn, are interconnected to each other through a reduction tree network. The other section concerns the control of the SIMD array itself performed by a microprogrammed sequencer, which is appropriately driven by the CPU to start the programmed parallel tasks. In this way, no coherency problems are encountered since the CPU can issue the wait() instruction, through which the synchronization is achieved. The CPU waits for the sequencer to finish the demanded tasks before proceeding with the processing of the remaining sequential parts.

Concerning in-memory-like kinds of operations, the related bit-serial processing units (PUs) include a Full Adder (FA), a single bit block for the logic functions, and four 1-bit registers each. The operations occur in a bit-serial manner for each data, but all data are elaborated by the PUs alongside. More in detail, the data are organized in the shared memory so that one or more words can compose a single row to which a different PU is connected. In general, if a function must be performed on all data belonging to two different datasets, these datasets are stored so that all rows contain one word from each of them. So the number of rows gives the dataset dimension. The SIMD processor can read/write a bit slice at a time, corresponding to a 1-bit wide memory column. An example of a possible computation is the vectorial sum between A and B, each composed of N elements of K bits each. In this case, all A elements are stored in the first K columns of the memory (one data for each row), while the B ones in the following k columns, and, lastly, the results are stored starting from the 2Kth column. The N elements sums are computed simultaneously, but one bit of the result is elaborated by each PU at a time. Thus, the final N elements resulting vector is obtained in about K clock cycles. Moreover, if, instead of the vectorial sum, an N-words sum is required, the PUs are provided with an interconnections network which allows performing that sum in a reduction tree-like mode, saving time. For this purpose, each PU can access values in the other eight neighbours, in both left and right directions, and are identified by the PUs at 1, 2, 4, and 8 positions of distances referred to the PU at hand.

CLiMA: Configurable Logic-in-Memory Architecture

The CLiMA [14] is a PiM architectural model that is proposed as a baseline framework aimed at building a LiM device customizable for any algorithm. For this reason, the CLiMA merges several macroblocks, each embodying one of the listed PiM branches, so that all benefits brought by the different approaches can be conveyed and emphasized in a single architecture. Moreover, the coexistence of multiple heterogeneous parts assures the model as much flexibility as possible in mapping a wide range of applications. This is because, even within the same application, there may be routines marked by different processing requirements, so each procedure may need to be mapped on a computing block matching its specifications. Therefore, the

CLiMA is split into two main communicating sections, one implementing the CnM principle and the other carrying out the CiM and LiM approaches together, as illustrated in Figure 2.11. In particular, the former is based on a standard memory block accompanied by a dedicated logic block, aimed at performing all the instructions which do not fit an in-memory-like implementation. Instead, the latter is composed of a set of LiM layers, each, in turn, surrounded by blocks of extra logic elements (which behave like the peripheral circuitry in CiM systems), composing what in Figure 2.11 is called CLiM Array. More in deep, each layer of this array is composed of CLiM cells which are LiM cells that embrace the configurability property, to make this section suited for running different types of data-intensive applications, which are the ones that best suit a LiM implementation. Besides, to fulfill the specific data exchange required by the algorithm to be mapped on the CLiM Array, cells are enabled to communicate, and their interconnection can be configured too. Then, the CiM part is thought of for handling all the remaining operations on top of the LiM processing.

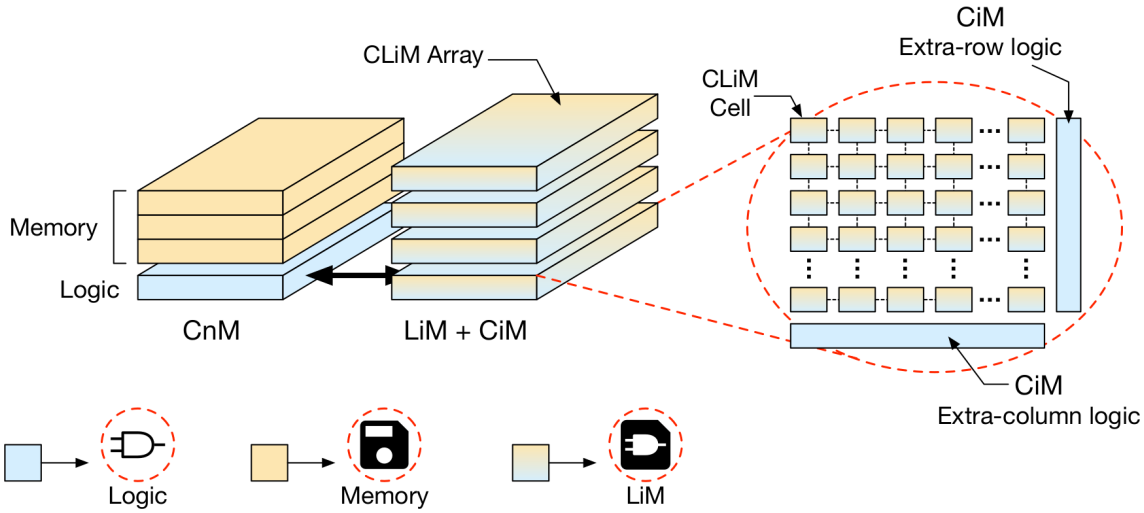


Figure 2.11: High-level view of the CLiMA main composing parts [9].

Going deeper into the core of the CLiMA framework, i.e. the CLiM Array, this was designed thinking of all the possible data exchange and manipulations that may be required by most of the algorithms. Thus, its basic block, the CLiM cell working on 1-bit data, is organized as depicted in Figure 2.12. It is divided into two areas:

the storage one, represented by the actual memory cell, and the processing one, composed, in turn, by two other blocks, one, i.e. the config block, which can be configured to implement standard boolean logic operations and another, i.e. the full adder, aimed at performing the sum. Besides these components, the cell is provided with further connections and input signals of primary importance for lending maximum programming flexibility to the array. Specifically, the memory element is forwarded to both the processing part and a multiplexer that, in addition, takes the outputs from the computing blocks. Then, the returned signal is input to all the other cells in the same column and the same row of the array. These interconnections are used to implement different kinds of data manipulation inside the array. Intra-row and intra-column data exchange are enabled that allow executing operations on the data in the same row and the same column, respectively. Similarly, inter-row and inter-column operations can be carried out, which fit bitwise-like functions between two rows or columns, respectively. Moreover, another multiplexer is inserted, which drives the input of the storage element, choosing between the result elaborated by the same cell and a value coming from the bit line, used to initialize the memory content from the outside in the standard memory mode. Concerning the processing part, the second operand is provided by another multiplexer which chooses, in the most complex case (which depends on the cell placement inside the array), among an external data signal, the output of the above cell, the output of the left cell and the neighbour one in the south-west diagonal. The last two connections together with the carry chains (points 4 a 5 in Figure 2.12) enable the composition of more complex computing blocks like a Ripple Carry Adder (RCA), implementable along the row, and an Array Multiplier (AM), which can be composed exploiting the whole array. In this last case, the RCAs on the rows can be used to evaluate the partial products, which can then be added all together through the column interconnections.

Thus, it follows that the CLiMA structure allows implementing sums and products directly inside the memory, which, as thoroughly explained, leads to sharp improvements in both time and energy expense. Nevertheless, the components that can be assembled on the CLiM array to perform those computations (i.e. the RCA and the MA) do not represent the fastest hardware in those fields. Therefore, according to the algorithm, it may be worth thinking of other optimized implementations to be

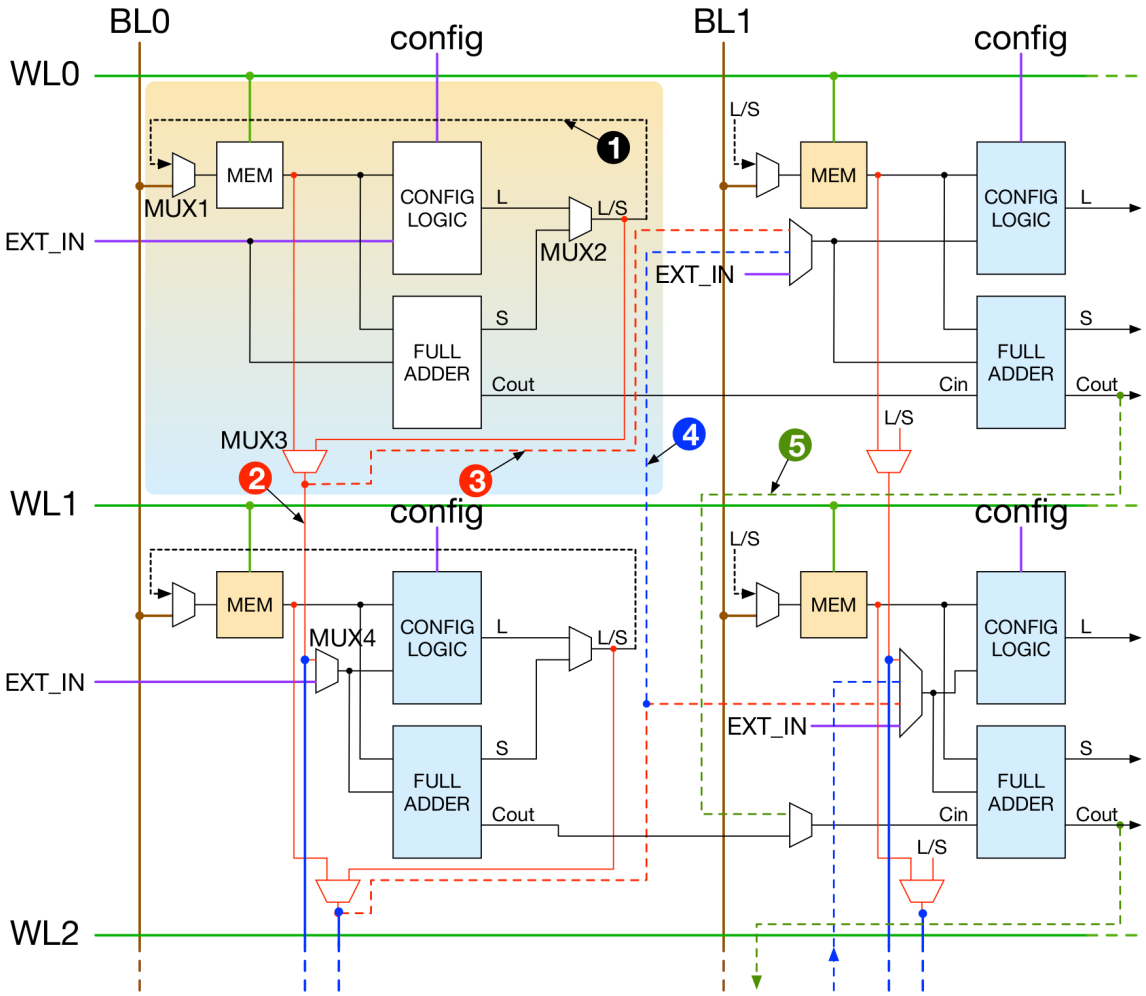


Figure 2.12: Detail of the CLiM cells internal structure and their interconnections [9].

embedded in the dedicated logic units outside the LiM block. In this way, some of the available interconnections in the CLiM array (provided to implement the RCA and the MA) may be cut so that the overall system performance may be further enhanced.

It follows that although CLiMA well suits a wide set of algorithms, they can still be identified some features that unite the algorithms for which the CLiMA implementation wins over the most common architectural solutions (as GP processors); which are high level of data processing parallelism for simple operations and a great

demand for the processing of vast data set. In [14] a platform running a Convolutional Neural Network (CNN), specifically a ShiftCNN, is developed starting from the CLiMA model. Through this benchmark, the CLiMA proved to be compliant with the expected enhancements in terms of reduced data movement, speed-up of data-intensive applications, and high adaptability to different algorithms. However, a significant limitation is pointed out, which is linked to the control of data movement. The availability of different paths that the data are allowed to travel requires considerable efforts for the control part design, which has to be customized each time for the specific application at hand and may even reach a high level of complexity to prevent errors in the data exchange.

Chapter 3

Programmable LIM

Inside this chapter, it is deeply investigated another proposal of a PiM system taken from the literature, i.e. the Programmable LiM (PLiM). It was examined to prepare the basis for the development of this thesis subject. The PLiM marries the LiM concept of processing elements integration inside the memory cells but moves away from its ASIC feature to go toward a new LiM branch that winks programmable systems. It means that, once the resulting device is fabricated, it can change at run time the performed function (among the ones made available by the hardware structure) depending on how the control signals going into the LiM array are driven. Specifically, the PLiM is a LiM modular template that steals the reconfigurability feature from the CLiMA model while trying to remedy its main limit. As stated in chapter 2, the major drawback in the use of the CLiMA framework lies in the request for the design of a specialized control unit (CU) from scratch, each time a LiM device must be created for a new algorithm. This results in lots of efforts, time, and resources wasted on building a system capable of only running a single specific task that is decided upstream of the fabrication process. The PLiM tackles this issue by providing a design starting model for both the LiM array and the control part that drives the data processing and exchange inside the array during the application execution. The request for both a CU starting model and programming capabilities convey in the same solution. Following the basic organization of the GP-SIMD (chapter 2), the PLiM is provided with a microprogrammed machine (uCU), which is in charge of handling the instructions flow the LiM has to run. The

uCU picks the program instructions from the LiM Instruction Memory which can be written at compile-time, providing programming flexibility. As long as the required instructions set is made executable by the specific LiM array composition, there is no need to physically fabricate a different LiM architecture for each application to be run, but the system can be initialized at compile-time with a different program each time. Moreover, the CU instantiation drastically reduces the LiM design time if the PLiM model is used as a canvas. Even if the LiM array composition changes, the uCU does not need to be redesigned or adapted.
The PLiM remains a modular template to build customizable LiM devices that follow the SIMD computing model and retain some programming flexibility. The basic LiM array is easy to be adapted. Thanks to the modularity granted by the PLiM model, the user can easily insert inside the array combinational or sequential blocks aimed at implementing new functionalities. Apart from the design of the new blocks to be integrated, the rest of the architecture requires only few changes to be compliant with the new ISA.

From the GP-LiMA design standpoint, the PLiM represents the basis for its definition. During this thesis work, the PLiM was first explored and then slightly changed to enhance its programmability. Afterward, a set of benchmarks, selected to point out the PLiM strengths and the weaknesses to be addressed during the GP-LiMA design, was run on the architecture. The study is reported below according to the following outline:

- ***Section 3.1*** **- PLiM Model** presents the PLiM template, outlining the goal, the general structure, together with the blocks that compose it, and how the generated LiM can be integrated inside a bigger system.
- ***Section 3.2*** **- PLiM Performance on Benchmarks** reviews all the benchmarks chosen to test the LiM together with all the modifications made each time on the starting PLiM template to generate the related specific architectures. Then, the performances (both post-synthesis and post-place&route) achieved by the generated LiM devices are shown and compared for the related benchmarks.
- ***Section 3.3*** **- Conclusions** takes stock of the retrieved results, pointing

out for which kind of algorithms the PLiM model guarantees an efficient implementation and for which it does not. Then, in preparation for the next chapter, all the weak points the GP-LiMA seeks to strengthen are listed and discussed.

3.1 PLiM Model

The PLiM approach [37] is proposed as a replacement for the standard design flow aimed at developing LiM architectures specialized for a target algorithm. It aspires to reduce the LiM design complexity and the ASIC feature of the accomplished LiM devices carved in stone. Thus, the PLiM architecture comes as a baseline template allowing a hardware designer or a specialized tool to build LiM systems that can suit new applications without requiring to set up from scratch brand-new LiM architectures.
This model provides an already structured control part (which also gives programmability to the generated LiM device) sided by a highly modular computing block, the actual LiM array, which can be modified with a pretty high degree of freedom. The memory block equipped with processing capabilities (LiM array) is organized so that each location is composed of a compulsory storage unit connected to logic blocks that can be easily inserted and removed (during the design phase) to make the resulting LiM architecture compliant with the algorithm to be mapped. While the user defines the memory location complexity, conversely, the data exchange among them is fixed by the PLiM framework interconnections, which are already established. Notably, this designing approach relies on a technology-independent library of RTL components from which the user can draw and adequately combine the blocks he needs to compose the target LiM device. Then, according to the created LiM skeleton, the specific ISA is derived so that the final user (the LiM system programmer) can write and load the current program to be run. Therefore, adaptability and user-friendliness are the keywords for this designing approach. Moreover, the PLiM paradigm leverages the intrinsic parallelism of the memory structure to implement LiM devices working following the SIMD computing model. Consequently, the generated architectures are particularly suited for executing data-intensive algorithms

with a high degree of parallelization.

3.1.1 PLiM Overview

Since the LiM concept is born to tackle the *memory wall* issue in CPU-centric systems, the specific devices generated from the PLiM template do not work as stand-alone systems. Despite they are characterized by a considerable autonomy in the processing of some code portions (thanks to the uCU), they still remain smart memories that bring actual advantages to the applications executions when substitute, or at least accompany, the standard memory in CPU-centric systems (that are the ones they were designed for). For this reason, apart from the PLiM Unit itself, that is the device gathered from the PLiM template, the whole PLiM framework comprises at least a CPU, a scheduler, and an instruction memory more, as shown in Figure 3.1.
This environment organization makes CPU and PLiM Unit interact as if this last was a standard data memory. The scheduler unit insertion eases the CPU task, which can work as a standard processor, completely neglecting the processing feature of the memory it works with. Indeed, the scheduler heads the overall PLiM platform behaviour, reading the application program and decomposing it into code portions to be executed by the CPU and portions to be entrusted to the PLiM Unit. Besides choosing which unit is more suitable for processing each instruction, the scheduler also takes care of the related time scheduling. It decides when certain instructions can be executed and whether they can run in parallel to others by exploiting the intrinsic hardware parallelism of the structure. In practice, the PLiM Unit is handled like a SIMD co-processor dedicated to the more parallel parts of the code, while, simultaneously, the CPU is concerned with the sequential instructions. Specifically, the scheduler interacts with the PLiM Unit through two signals: the `Start LiM` and the `LiM Program Address`. The first one is used to start the PLiM Unit in processing mode, while the second signal carries information about the related program the PLiM Unit has to run. Inside the PLiM platform, instruction memory (IMem) is inserted, which is loaded at compile-time with all the possible programs the PLiM Unit has to run. Each time the PLiM Unit has work in processing mode, the scheduler sends to it through the `LiM Program Address` signal

the address of the IMem location where the first instruction of the program to be performed is stored. Then the LiM Unit correctly handles this address to access the IMem and retrieve sequentially all the instructions composing the program that it will then execute.

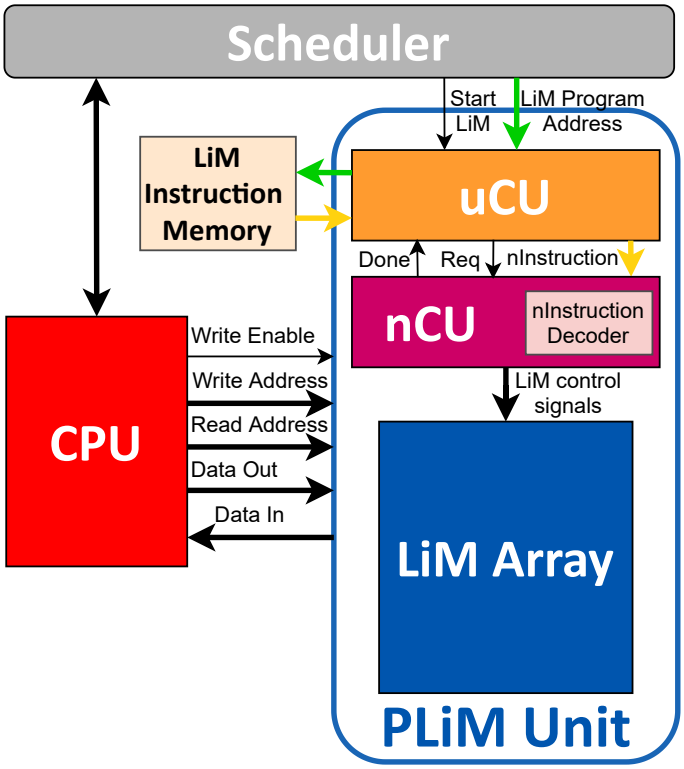


Figure 3.1: Overview of the ecosystem accommodating the PLiM and coarse insight of the PLiM structure.

To do this, the PLiM Unit is composed of three macro-blocks: the microprogrammed machine (uCU), the Nano Control Unit (nCU), and the LiM Array. The first two units implement the control logic enabling the PLiM Unit to behave like a coprocessor, while the last one embodies the LiM datapath section, namely, where the data are stored and processed at the same time. More precisely, the uCU takes care of the instruction flow. It takes the starting address coming from the scheduler and sends it to the IMem to fetch the first instruction. Once the instruction is obtained, the uCU extracts the bits sequence bringing information about the data processing

to be executed, and sends it to the nCU. The rest of the instruction is decoded by the uCU itself to retrieve the address of the next instruction. This operation is repeated till the program end, flagged by the value of a specific bit in the last gathered instruction. While the uCU prepares the fetching of the next instruction, the nCU works on the instruction portion (`nInstruction`) just received from it and, according to its value, properly drives all the control signals that manage the data elaboration inside the LiM array. Thus, this structure is organized in a 3-stages pipeline form, as each of the mentioned operations takes one clock cycle. The uCU is associated with the fetch stage, the nCU represents the decode one, while the LiM array the execution and the intrinsic write-back of the data.
From here, it can be already seen as the PLiM framework guarantees remarkable adaptability and user-friendliness. Even if the ISA implemented by the LiM Array changes, only small parts of the nCU need to be modified accordingly, while the uCU can remain unaltered, being independent of the array structure. Also, it results in a simplified debug process for the LiM hardware designer and more accessible programming for the PLiM user.
Moreover, concerning the interface between PLiM Unit and CPU, this reflects the one of a standard two ports memory (one asynchronous read port and one synchronous write port), and it is implemented inside the PLiM Unit leveraging on the standard decoder mechanism.

3.1.2 PLiM Datapath: LiM Array

The real element on which the PLiM paradigm poses its foundations is the LiM Array, which is the memory array composed of smart cells embedding both storing and logic items. Unlike the GP-SIMD and most common PiM devices that host more than one word in a single memory row, the LiM Array is logically organized in a long stack of one N-bit word per row. Moreover, not all the memory cells forming the LiM array are smart cells, as it is composed of a mix of standard and smart ones to grant a good data storage capability without running into an extreme overhead in terms of complexity and power consumption. As it can be seen from Figure 3.2 the memory is divided into two macro sections: the smart section and the standard section. The first section comprises an alternating sequence of standard

locations and smart ones, called standard rows and smart rows. The smart rows are the core of the LiM Array and are where the data elaboration inside the memory occurs, while the adjacent standard rows act as further storage elements from which the smart rows can draw other data or even insert some data when the demanded computation requires it. Then, the second section is thought of for standard storage purposes and usually is dedicated to holding constants involved in the execution of common algorithms. If, on the one hand, the combination of smart and standard rows make up the basic structure for the data processing, on the other hand, the memory interface constitutes the interconnections network that enables the data transfer inside that base, aiming at correctly fulfilling the algorithm at hand.

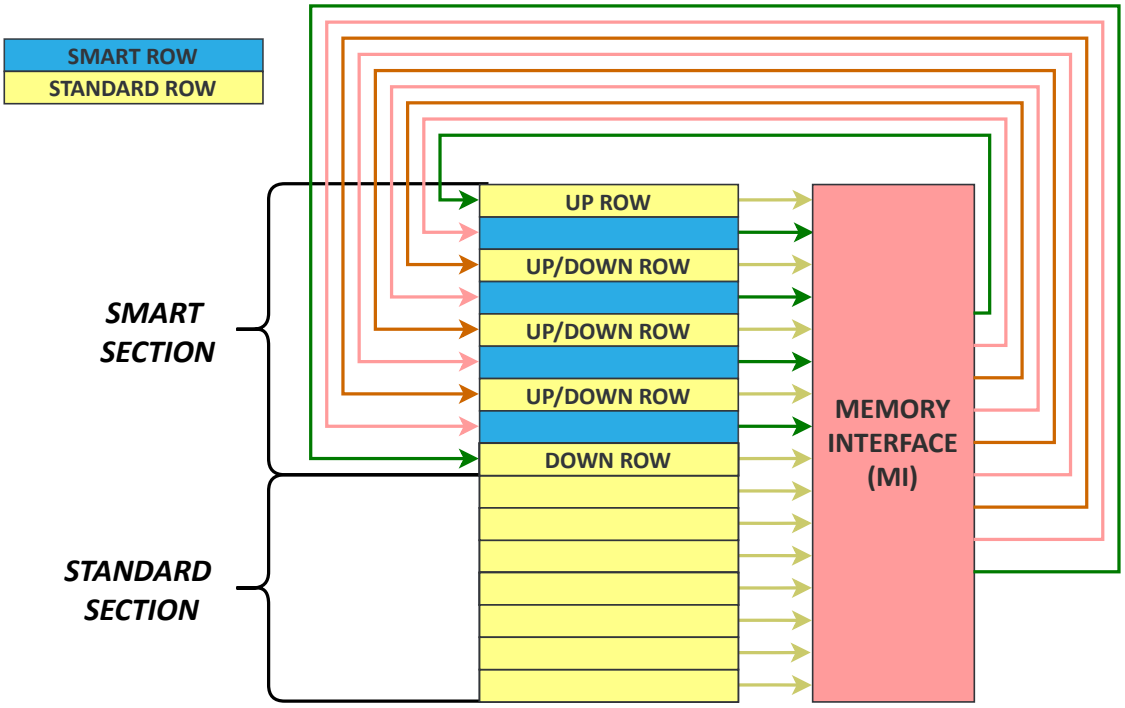


Figure 3.2: General arrangement of the LiM array. Example for a PLiM with a smart section on 8 smart rows and a standard section on 7 standard rows.

Going deeper into the LiM array philosophy, each smart row can perform single-operands or two-operands instructions, and the data it works on can be taken from either the storage elements inside the smart row itself or the standard rows. However, there is a difference in the possible usage of the standard row belonging to the

smart section compared with the one included in the standard one; that is how the smart row can retrieve the data from them. As already mentioned, the computing model implemented by the PLiM reflects the SIMD one, this is because, given the just detailed LiM array structure, it comes straightforward making all the smart rows perform the same function in parallel on different data. Specifically, from Figure 3.2 the regular structure of the smart section stands out, which sees each smart row surrounded by a standard up row and a standard down one. This organization enables the smart rows to run the same instruction, each on a different set of data. When the operand `Up row` or `Down row` is called, each smart row takes simultaneously the data from the standard row above or below it, respectively. On the other hand, when the smart rows access the standard section, they can only perform, in parallel, a function involving the same data taken from a pointed row in that section, specified as `Other row` together with the address of the concerned location. For this reason, when the PLiM Unit works in processing mode, the rows in the standard section are accessible in read-only mode to prevent data conflict caused by the writing of the same cell performed simultaneously by two different smart rows. However, the CPU can initialize their content through a simple write operation, seeing the LiM Array as standard data memory. Conversely, the smart section standard rows can be accessed by the smart rows in writing and reading mode. Thanks to the SIMD computing mode, the writing can be handled in such a way that, at the same time, the smart rows can all modified either their up row or their down row. Therefore, it will never happen that the same row is driven by two different smart rows simultaneously. However, two consecutive smart rows can write, at different times, in the same standard row, as the down row for a smart row corresponds with the up row of the following smart row in the array. This property can be used to implement a complex data exchange that has a key role in speeding up applications implying the execution of a given number of sequential operations, where each is independent of the others and can be sequentially performed in parallel with the other ones. An example is the Matrix-Vector Multiplication algorithm presented in subsection 3.2.2, where u sequential sums are run in parallel.

As already amply explained, the PLiM Unit follows the SIMD paradigm. Therefore all the smart rows composing the LiM Array have the exact same layout and work simultaneously. However, the hardware designer can set the final array such that the

LiM programmer can enable only a subset of smart rows during a certain computation. Specifically, the smart section can be divided into multiple macro-blocks (up to a maximum of 4) that can be enabled independently from each other. Moreover, the entire LiM array can be sized by the hardware designer, which can choose the memory parallelism, the number of smart rows comprising the smart sections, the number of standard rows making up the standard section, as well as the number of smart rows included in each of the enabling macro-block groups.

Indeed, referring to the PLiM paradigm described in [37], different arrangements for the LiM array are available. However, in this discussion, only a specific version is investigated since it represents the one that better suits most of the algorithms. It means that the general skeleton of the examined LiM array is the one just presented, while the specific organization considered for the smart row (Figure 3.3) is the one outlined in the following section 3.1.2. For further versions, refer to [37].

Smart Row

Starting diving deeper into the LiM array, the following relevant element to be addressed is the smart row. As shown in Figure 3.3, it is composed of a customizable chain of different blocks, which involves in order the row word, a sequence of generic row interfaces (RIs), and the input and output buffer. This layout reflects how the PLiM framework tackles the LiM approach. Here, the LiM idea is pursued on two levels represented by the row word and the RIs. The first is used to implement the LiM computation at the finer grain level, namely inside the single 1-bit memory cell, whereas each RI guarantees the data elaboration inside the memory at the word level. While the row word is the base around which the smart row is built, embodying the more straightforward implementation of the LiM concept (the user cannot modify cells), the RI is the novelty that characterizes this approach. It is what guarantees modularity and so adaptability to the proposed PLiM template. The RIs allow the introduction of further processing components inside the memory words, without requiring single memory cell modifications. The hardware designer can easily insert and remove RIs in the smart rows to adapt the processing capabilities of the array to the demanded algorithm without needing to care about compatibility with the

rest of the smart row components.

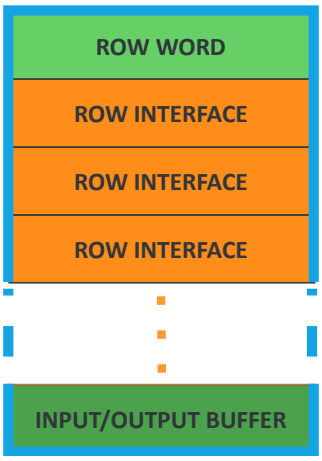


Figure 3.3: Insight of the internal layout of a generic smart row.

- **Row Word & Arithmetic Cell (ACell)**
  The row word is where the word data associated with the smart row is stored and where first simple data elaborations occur, like bitwise logic operations. It guarantees the finer level of LiM as it is composed of a set of single 1-bit LiM cells that represent the most elementary LiM items inside the array.
  In the PLiM version taken as a case of study in this thesis, the kind of LiM Cell making up the row word is the Arithmetic Cell (ACell). In Figure 3.4 the row word composition is depicted together with the inside of a single ACell. Each ACell is based on a 1-bit storage unit that constitutes the actual memory cell, whose output goes into a Full-Adder (FA), passing through two multiplexers whose purpose is to select the operands for each committed instruction properly. The FA acts as a programmable block for implementing simple logic functions, as, by appropriately setting the value of the input carry, it is possible to retrieve from the sum and output carry signals the result either of the `XOR`, the `XNOR`, `AND`, or the `OR` between the two input signals, respectively. Besides, all the embedded elements are properly driven by configuration signals coming from the nCU.

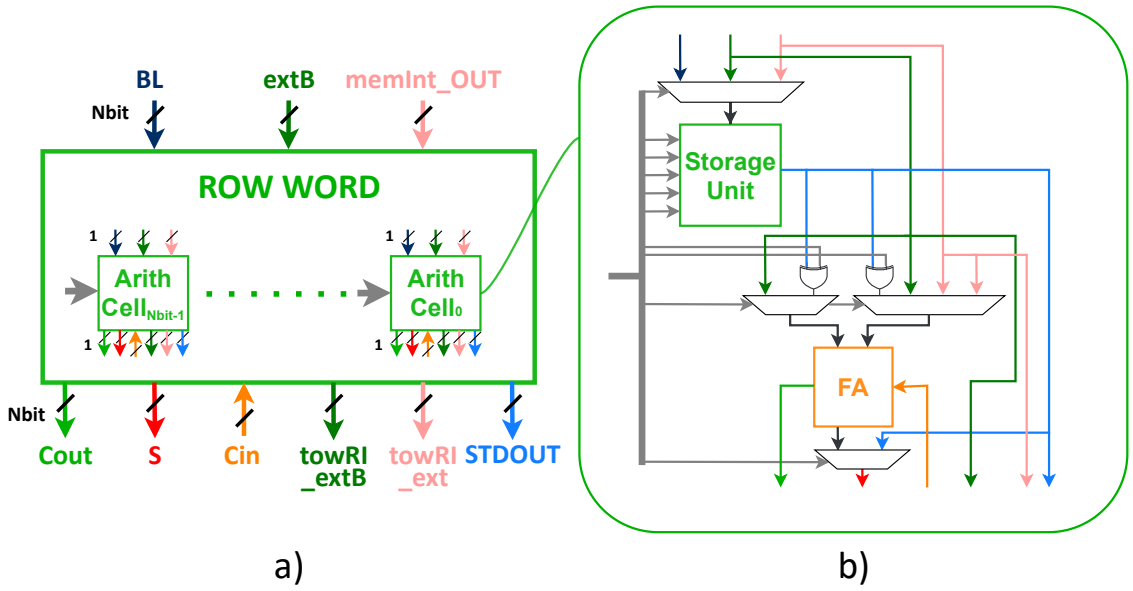


Figure 3.4: a) Row word composition. b) Detail of the internal structure of the arithmetic cell representing the 1-bit elementary LiM cell on which the row word is built.

For what concerns the overall row word, it is given by a number of ACell equal to the memory parallelism (Nbit). In Figure 3.4 the row word interface is shown, which provides information about the kind of connections and operands on which it can work. The `BL` signal is used to initialize the row content with the word value driven by the CPU when the LiM Array is used in data-memory mode. However, data coming from the input/output buffer or any of the rows belonging to the LiM array can also be saved inside the storage units, through the `extB` and the `memInt_OUT` signals, respectively. Then, the data on which the row word can perform the mentioned operations are again the ones brought by the `extB` and the `memInt_OUT` signals and the data stored in the row word itself that is also forwarded outside of the block through the `STDOUT` signal. Moreover, the ACells are always coupled with a specific row interface to which they are directly connected, namely the RCA&LOGIC RI. This block is the one that, taking the `Cout` output signal from the row word, returns the input carry signal (`Cin`) to all the FAs in the row word. In this way, a carry chain connecting all the FAs is instantiated so generating a real

RCA. Lastly, the value of the output signal `S` going inside the following RIs can be chosen among the sum signal from the RCA and the content of the storage units.

- **Input/Output Buffer**
  The Input/Output Buffer is the item closing the smart row chain of blocks. It serves as further storage space, mainly exploited for saving temporary values resulting from the application running. As illustrated in Figure 3.5, this block is composed of a set of two registers, called input buffer and output buffer, and three multiplexers.

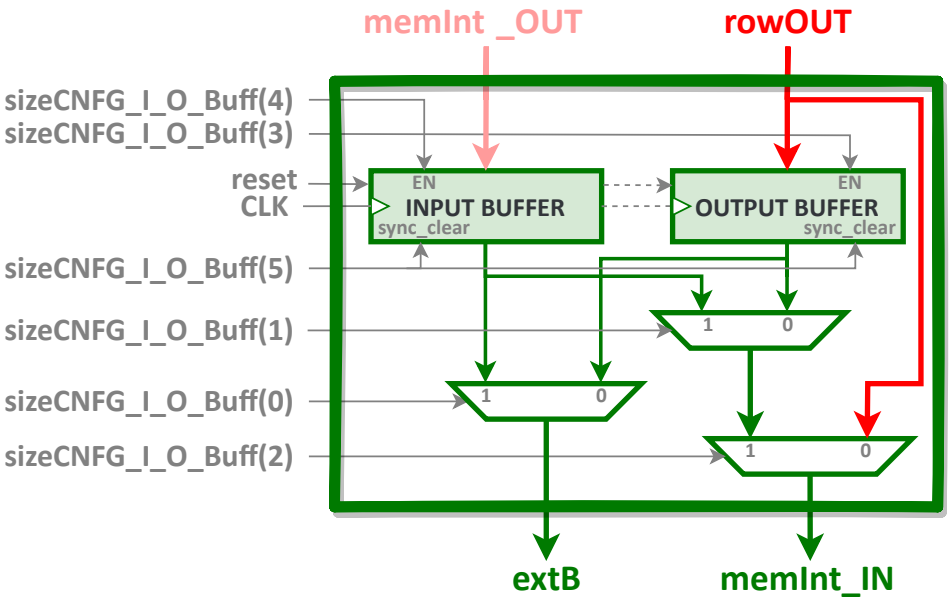


Figure 3.5: Composition of the I/O buffer.

As all the computing units included in the smart row, this block can be enabled only in the PLiM processing mode. It is designed so that the input buffer can only save values coming from the MI `memInt_OUT`, namely the content of either the standard rows or the row words in any of the smart rows composing the LiM Array. In contrast, the output buffer is dedicated to holding the value returned by the previous RI `row_OUT`, which can be either the content of its

related row word storage units or the result of the programmed computation currently carried out by the smart row. Then, two output signal come out from the block, one going inside the row word `ext_B` and another forwarding the data output of the whole smart row to the MI `memInt_IN`. Specifically, `ext_B` can carry the content of either the input buffer or the output one, while the MI can receive through `memInt_IN` either the value of the registers or the data brought by the `row_OUT` signal.

- **Row Interfaces (RIs)**
  While the row word and the input/output buffer are the compulsory extremes of the chain composing the smart row and cannot be modified (in this LiM Array version), the RIs are the blocks filling the inside of the queue. They can be of various nature and number and can be embedded independently on each other. The RI block typology contains processing logic customized for the application the generated PLiM has to run. Thus, it represents the focus of the PLiM paradigm and is what the modularity feature of the template consists of. Each time the PLiM model is used as the canvas for a customized LiM device design, the main user's task is to create the specific RIs he needs and enter them in a given order into the smart row skeleton. However, this task is made simple by the standardization of the external interface of the generic RI. It means that for each new designed RI, the input, the output and the control signals going inside and outside the block are always fixed as constraints that the user has to follow to make the PLiM Unit work properly. On the other hand, this limitation offers the user greater freedom in the internal layout of the specific RI. As pointed out in the scheme in Figure 3.6, each RI has 3 data input signals: `inRI` that carries the output coming from the previous interface (that can be either the row word content or the intermediate processing result), `fromMI_extB` that is connected to the `extB` signal output by the input/output buffer, and `fromMI_ext` that holds the data provided by the MI (that is any word of the memory array). In the case of a two operands function, the RI selects a specific combination of 2 of the possible inputs through 2 multiplexers driven by the 2-bit configuration signal `S_inLogic`. Concerning the fixed output signals, these are: `towRI_extB` that copies the `fromMI_extB` value,

`towMI_ext` that forwards to the output the `fromMI_ext` signal, and `outRI` that is driven by a multiplexer controlled by the `S_outRI_noOut` 1-bit configuration signal that chooses its value between the data carried by the `inRI` input signal and the result output by the computing logic comprising the RI. It follows that for each new RI added to the chain, only two new control signals have to be instantiated and driven, independently of its actual composition, which makes the few modifications to be performed on the control part easy to be implemented.

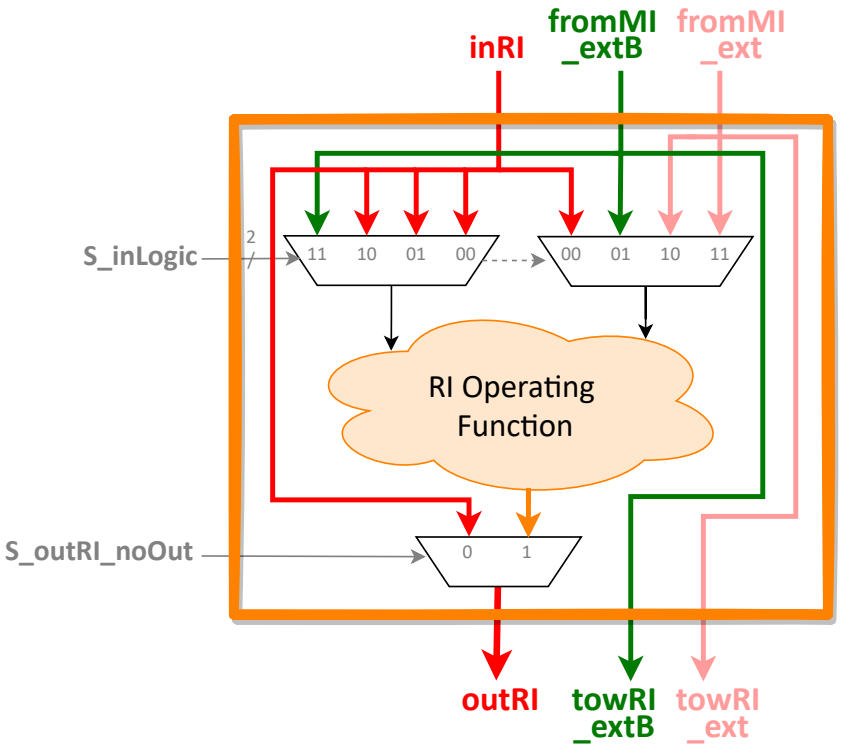


Figure 3.6: Internal structure of a generic row interface.

As it can be noticed, the number of data inputs matches the output one; this allows a direct connection of the RIs in cascade since each input signal has its own counterpart in the output signal of the previous block and vice-versa. Moreover, the output interface also matches the input one of the input/output buffer. However, there is no direct correlation between the input interface of

a generic RI and the output of the shown row word. This is why, in smart rows with row words composed of ACells, it is mandatory to always insert after the row word a specific RI called RCA&LOGIC. This block has twofold utility. As already anticipated, it is exploited to generate an RCA leveraging the already existing FAs, and, in addition, it acts as an adapter for the easy insertion of other RIs in cascade after it. Its input interface is complementary to the output one of the row word, while its output interface returns the usual 3-signal package to be connected to the following generic RI or the input/output buffer. Lastly, the RCA&LOGIC block combined with the ACells allow to implement most common and basic logic and arithmetic instructions that are listed in Table 3.1.

RCA&Logic Operations	
Instruction (FUNC)	**Instruction definition** (OpA OP OpB)
SUM	OpA + OpB
SUB1	OpA - OpB
SUB2	- OpA + OpB
SUB3	- OpA - OpB
ANDop	OpA AND OpB
ORop	OpA OR OpB
XORop	OpA XOR OpB
XNORop	OpA XNOR OpB
ORopV1	OpA OR NOT OpB
OR2opV2	NOT OpA OR OpB
OR3opV3	NOT OpA OR NOT OpB

Table 3.1: List of operations the RCA&Logic RI can implement. To correctly run one of these operations, in the instruction, the OPCODE = RCA&LOGIC_i must be combined with the related value for the FUNC field (for the meaning of FUNC and OPCODE see section 3.1.2).

Furthermore, it is worth examining a specific row interface, inserted in the RTL components library, that was used for the implementation of some of the algorithms tackled in section 3.2, that is the Temporary Storage RI depicted

in Figure 3.7.

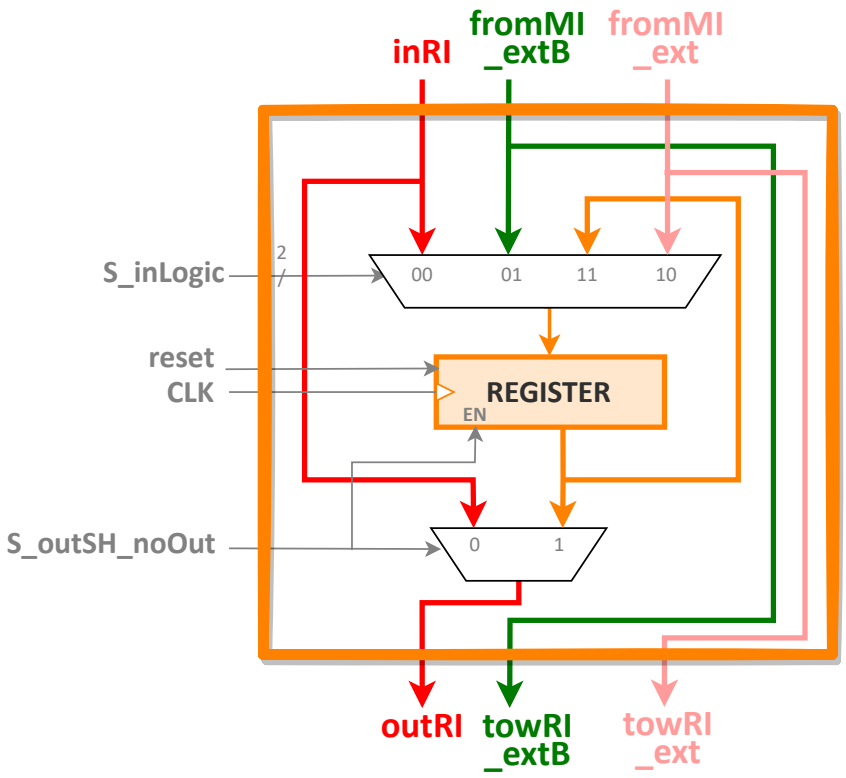


Figure 3.7: Design detail of the Temporary Storage RI.

The function of this RI is to provide the smart row with further storage capabilities for holding a larger number of temporary variables. The design of a new block, characterized by a functionality similar to the already existing input/output buffer, is preferred over the use of another same type block because of the design complexity that the insertion of another input/output buffer in the smart row would imply. The PLiM template is already prepared for the easy integration of new RIs, while it is not set up to include blocks with a different external interface. Thus, the designed temporary storage performs a task comparable to the input/output buffer one while having the user-friendly RI interface. This block can be controlled through only three 1-bit configuration signals instead of six, like required by the input/output buffer, and it is composed of a register that represents the RI operating function, an input

multiplexer that selects the data input to be loaded through the `S_inLogic` signal and the usual RI output multiplexer providing the `outRI` signal. However, some tricks are present in this block design to comply with the stated configuration signals interface. The same control signal `S_outSH_noOut` is connected to both the output mux and the enable signal of the register. It means that each time the content of the register is sent to the `outRI` signal, namely when the temporary storage data is read, at the next clock cycle, the content of the register will be updated with the value present at its input. So, to avoid losing the temporary storage content after each reading, the output of the register is brought back to the multiplexer that selects the input for the register itself. In this way, when the LiM programmer needs to access the temporary storage in reading mode, to prevent the register content loss, he has to drive the `S_inLogic` so that the register is overwritten with its same output data. This is done by specifying the special operand `write_back` in the `OPERAND` field of the instruction that enables the temporary storage (see section 3.1.2).

Memory Interface: MI

So far, the main components constituting the processing part of the LiM Array were reviewed. Howbeit, little was told about the path of the data along the array. The Memory Interface represents the routing network in charge of handling the data exchange between different smart rows and standard rows, following the SIMD paradigm. It is mainly implemented through a set of multiplexers, each driving the input data for a different row in the smart section. In Figure 3.8 two implementations for the MI connected to the array are depicted. The scheme on the left shows the first MI proposed in [37], while the right one details the final structure made up during this thesis. The issued modifications allow a more flexible data transfer which speeds up the execution of the algorithms. The designed MI can implement two kinds of data transfer: it can provide all the smart rows with either different data or the same data from a specified array row, depending on what is indicated by the performed instruction. Looking at the b) case in Figure 3.8, the first column of multiplexers on the right of the LiM Array selects the required input for the smart rows. When an instruction specifying the `up row` (or the `down row`) operand

in the `OPERAND` field is demanded, all multiplexers simultaneously select the signal output by the standard row above (or below) the smart row they drive ("different data case"). While, when the instruction calls the `other row` operand, all the multiplexers select the same signal between the two registers outputs on the left of the LiM Array ("same data case"). Each register, in turn, is driven by a multiplexer connected to a memory array different half, so that it is possible to retrieve the content of any location in the array.

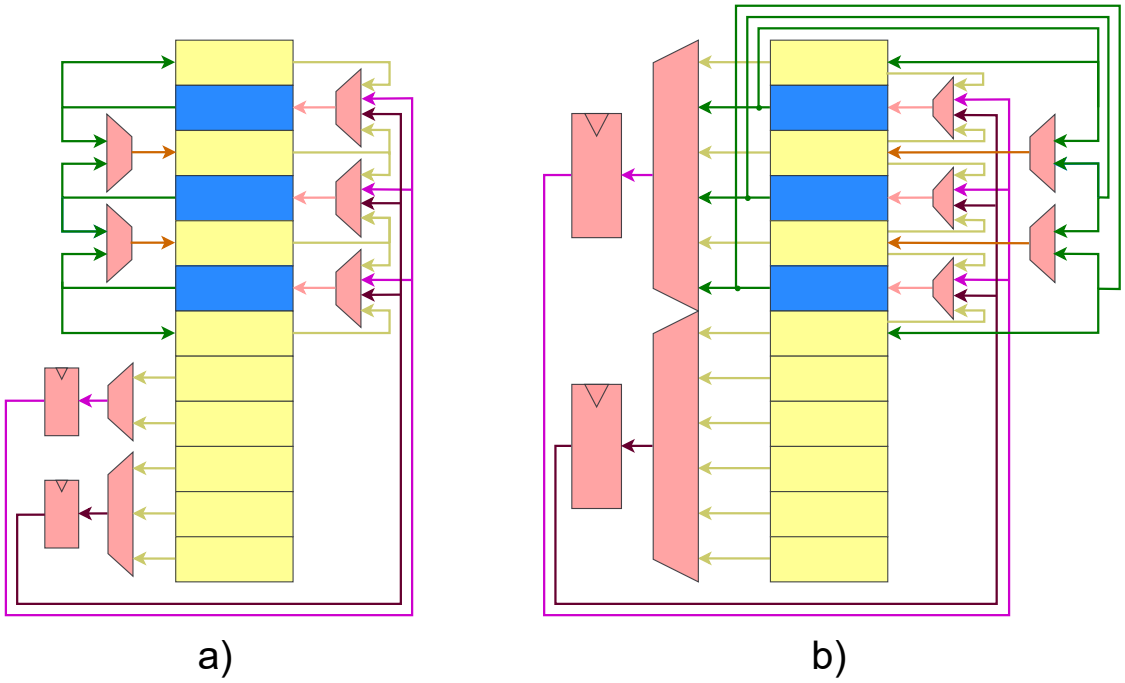


Figure 3.8: a) Detailed scheme of the initial MI block connected to the LiM array core. Example for a PLiM with a smart section on 8 smart rows and a standard section on 7 standard rows. b) Final structure of the modified MI.

In this case, the registers act as cache registers used to split the critical path. However, in the "same data case", they introduce a latency of one clock cycle more to gather the actual content of the pointed row, leading to possible data hazards due to the pipelined behaviour of the PLiM device processing. Specifically, the MI provides to the smart rows the wrong data value when the executed instruction takes

as `other row` source operand the content of a location that is written by the instruction that precedes it. To avoid this kind of hazard, the LiM programmer must insert a null instruction (textttnullOP) in the middle of the two to allow the correct updating of the cache registers before accessing them.
Lastly, the last column of multiplexers on the right of the LiM array (see Figure 3.8) is used during the writing of the standard rows in the smart section. When a store instruction kind is called, all multiplexers select the signal output by the smart row above or below the standard row they drive, depending on whether the value specified in the `OPERAND` field corresponds to `down row` or `up row`, respectively.

Instruction Set Architecture (ISA)

From the drafted description of the architecture, it is evident how the device can be handled to behave like SIMD coprocessors. Concerning the portion of the instruction directly related to data elaboration inside the LiM array (`nInstruction`, see Figure 3.1), like most processors, it is divided into fields, each associated with a different meaning for the instruction processing. In the following, the `nInstruction` composition is detailed together with the meaning of the fields in order of appearance:

- `OPCODE`: expresses the kind of instruction to be performed, typically by specifying the sequence of RIs in the smart blocks that must be activated;

- `OPERAND`: specifies from which storage locations the data to be elaborated must be taken or in which locations the data must be saved, depending on the instruction in the `OPCODE`. All the possible combinations of source operands are spelt out in Table 3.2;

- `OUT_BUFF`: tells whether the output register in the input/output buffer must load the data at its input. Indeed, the LiM Array is organized so that the computation results cannot be directly saved in the rows storage units, but first, they have to be temporarily stored in the output buffers. Afterwards, through the next instruction, the output buffers content can be loaded into either the standard rows or the row words;

- `ADDRESS`: when the `OPERAND` field equals to `other row`, it holds the address value of the involved LiM array row;
- `FUNC`: when the `OPCODE` field equals `RCA&LOGIC_i`, it specifies which is the particular operation to be performed. The set of functions among which it can choose is listed in Table 3.1.

Operands Combinations			
Operand A	***Operand B***	***Operand A***	***Operand B***
Row word	Row word	Row word	Up Row
Row word	Output buffer	Output buffer	Up Row
Row word	Input buffer	Input buffer	Up Row
Row word	Down row	Row word	Other Row
Output buffer	Down row	Output buffer	Other Row
Input buffer	Down row	Input buffer	Other Row
Special operand *in case of Temporary Storage RI*			
Write back			

Table 3.2: List of all the possible two operands combinations for a generic operation defined as Operand A `OP` Operand B.

Example `nInstruction`				
`RCA&LOGIC_i`	`RowWord_UpRow`	`StoreBuff`	`nullAdd`	`SUM`
Instruction description				
All the smart rows compute the sum between their row word content and the one of the standard row above them and save the result in their output buffer.				

Since the PLiM template can be exploited to generate programmable LiM devices customized for the application at hand, the ISA changes according to the specific LiM Array structure. The available ISA is defined by the smart row layout and, specifically, it is given by the particular RIs embedded in the smart row. Apart from the ever-present basis instructions (i.e. load, store etc...), all the other instructions are identified by all the possible combinations of sequences of RIs that need to be enabled. The only constrain is that, in a single instruction, the data path through the enabled RIs must follow the only allowed order, which means that, in the same

clock cycle, data can be elaborated and passed only from the first RIs to the physically next ones and not vice-versa. An example of the logic composition of a smart row is showed in Figure 3.9, while in Table 3.3 the whole instruction set related to this specific smart row is reported.

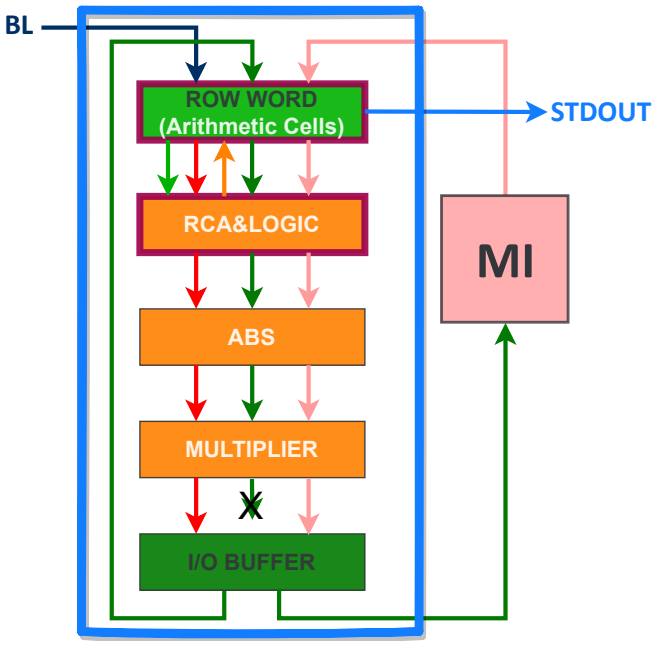


Figure 3.9: Example of the internal composition of a smart row associated with the ISA reported in Table 3.3.

In this example, there are no single instructions that can bring data coming from the multiplier to the input of the absolute value block. The only way to execute such an operation is to exploit two consequent instructions: first, the value at the output of the multiplier is stored in the output buffer, then, the data is brought back from the output buffer to the ABS block input. Therefore, to fully exploit the potentialities of this architecture, the RIs inclusion order inside the smart row assumes a fundamental role in the performance maximization for a target algorithm.

ISA Example	
Instruction (OPCODE)	**Instruction definition** (OpA OP OpB)
Basic instructions	
nullOP	Null instruction used to wait for one clock cycle to handle data hazard occurrences.
store	Copies the content of the output buffer into either the down row or the up row according to the OPERAND field value.
storeI	Copies the content of the input buffer into either the down row or the up row according to the OPERAND field value.
load	Copies into the input buffer the content of either the down row, the up row or a row in the standard section according to the OPERAND field value and the added address in case a standard section row is selected.
RCA&LOGIC_i	Execute one of the operations provided by the RCA&LOGIC RI according to the value of the associated FUNC: OpA RCA OpB.
RIs dependent instructions	
ABS_i	|OpA|
MULTIPLIER_i	OpA × OpB
RCA&LOGIC_ABS_i	|OpA RCA OpB|
RCA&LOGIC_MULTIPLIER_i	(OpA RCA OpB) × (OpA RCA OpB)
ABS_MULTIPLIER_i	|OpA| × |OpA|
RCA&LOGIC_ABS_MULTIPLIER_i	|(OpA RCA OpB)| × |(OpA RCA OpB)|

Table 3.3: ISA derived for the PLiM architecture identified by the specific smart row in Figure 3.9. Note: OpA and OpB are always derived from the OPERAND field.

3.1.3 PLiM Control

The control part represents the other novelty, besides the modular RIs mechanism, wrought by the PLiM approach, which strongly contributes to make up a flexible template for speeding up and facilitate the design of different customized LiM devices. The idea is to provide the user (the hardware designer) with an already

organized control part that needs only a few changes to be customized for a specific algorithm. Specifically, the control is subdivided into two different stages (fetch and decode) and, for each of them, a dedicated unit is instantiated, i.e. the micro-control unit (uCU) and the nano-control unit(nCU), respectively. When the template must be modified to map a specific application, the only block that must be adapted accordingly is the nCU. It is in charge of properly driving the configurations signals for the LiM Array after decoding the instruction coming from the uCU. For this reason, it needs to be compliant with the datapath it controls, so it must be modified to implement the new required ISA. On the other hand, the uCU, which is the unit that takes care of handling the instruction flow inside the PLiM Unit, can remain unaltered since, thanks to this 2-stage control implementation, it is independent of the LiM Array structure. In the PLiM Unit model, the uCU is given by a standard micro-programmed machine that takes care of the instruction fetching from the LiM instructions memory. In this way, the PLiM framework not only guarantees a predesigned base for the control part but also confers programming capabilities to the produced LiM device. Moreover, the use of a uCU allows the implementation of further instructions, besides the ones in the ISA, that handle the program progress rather than the direct data computation.
In general, an entire PLiM instruction is composed of 2 macro fields: one entrusted to the nCU (the `nInstruction`), that contains all the information about the computation to be performed on the specified data (as described in section 3.1.2), and one that is taken by the uCU to decide which is the next instruction to be executed, if there is one. Hereinafter, for the sake of simplicity, this last portion will be called `uCU_Instruction`. The `uCU_Instruction` is used to code programming flow-concerning instructions that involve explicit addressing, namely the address of the next instruction to be fetched is explicitly included in a dedicated field of the instruction. In the default functioning mode, the uCU always prepares to fetch the instruction stored in the IMem at the address specified in the just fetched instruction. In this way, `jump` instructions are straightforwardly implemented. Then, by setting some pre-defined bits in the `uCU_Instruction` also `jump&link` and `return` instructions can be performed. Specifically, the uCU contains a mechanism that allows saving the address of the instruction where to return after a `return` instruction is called. The textttreturn instruction is identified by looking at the values assumed

by the "return" bits of the textttuCU_Instruction.
Moreover, the uCU takes care of another major task aimed at the correct functioning of the whole PLiM Unit. This block handles the two functioning modes of the PLiM Unit: simple data memory mode and processing mode. Thus, it is composed of a set of components, mainly registers, that is used to start the PLiM in the working mode when the `Start LiM` signal, coming from the scheduler, is asserted. When `Start LiM` toggles to 1, the uCU takes 3 clock cycles to fetch the first instruction. Therefore, the total latency to retrieve the results for the first instruction is given by the sum among the 3 clock cycles employed by the uCU for the initialization, the 1 clock cycle required by the nCU for the decoding phase, the 2 clock cycles taken by the LiM Array to perform the data computation and store the result. It follows that the PLiM Unit overall latency equals 6 clock cycles. However, after the initialization phase, namely, after the pipeline constituting the architecture is full, a PLiM device completes one different instruction after each clock cycles. Therefore it shows a throughput of 1 instruction per clock cycle. Nevertheless, if the data throughput is considered, once the pipeline is full, this equals the number of smart rows integrated into the LiM Array.
Lastly, as already said, the instruction decoding task is performed by the nCU block that is mainly based on a set of small decoders, each dedicated to a different field of the `nInstruction`. The output signals by all of these blocks directly go inside the LiM Array, acting as control signals. Besides, the decoders are also connected to the handler of the MI, called Mem Int Unit, that is the component, located in the nCU, that controls the data exchange in the LiM Array based on the fetched instruction, also by adequately handling the cache registers inside the MI.
For further details on the specific uCU and nCU schemes refer to [37].

3.2 PLiM Performances on Benchmarks

So far, the PLiM framework was outlined, as it was conceived in [37]. This section illustrates the study conducted during this thesis about the applications for which the PLiM template can be suited. The investigation goal is to track down the strengths of this architecture and the cases where they are exploited to the full. However, this also leads to standing out the counterpart, namely when the generated

PLiM device fails to perform the tested algorithm efficiently. Hence, the features that make the architecture work poorly are derived and treated, too.
Five benchmarks are covered:

- K-Nearest Neighbour
- Matrix-Vector Multiplication
- K-means
- Mean & Variance
- Discrete Fourier Transform

The first three are examples of algorithms with a high level of parallelism, like the data-intensive applications, which can benefit from the combination of the SIMD computation arrangement and the LiM attribute of the generated devices. In particular, the first benchmark can be fully parallelized, while the other two are composed of both a code portion that fits well a SIMD implementation and a sequential part, less compliant with the PLiM Unit idea. Then, the last two benchmarks are picked to bring out the sore points of the PLiM model, so identifying which are the features the algorithms that are poorly mapped on it share. These algorithms are basically iterative routines characterized by data dependencies between different iterations.
The reported discussion is organized so that for each of the benchmarks, it is clear which is the algorithm portion actually implemented by the PLiM device, how the smart row is modified to customize the PLiM model for that application and how the LiM Array content is properly initialized, and the logical steps that compose the LiM program corresponding to the stated benchmark (for each step the number of LiM instructions employed to perform it is noted). The benchmarks are approached in a parametric way, meaning that the number of data samples they work on and all the other possible algorithm parameters are not explicitly fixed to a particular value but are kept in a generic form. It means that the number of instructions involved for a certain benchmark is not stated as an absolute value but in terms of the number of samples considered. From the execution time esteem standpoint, the number of clock cycles needed to perform a generic benchmark is expressed as the sum among the clock cycles taken to properly initialize the content of the LiM

Array with the data to be elaborated, the number of LiM instructions required to run that benchmark, and the overall latency of the PLiM architecture. The detail of the single contributions in the total number of clock cycles for each benchmark is specified in Table 3.9.

3.2.1 K-Nearest Neighbour (K-NN)

The K-NN belongs to the machine learning branch of algorithms, aiming at data classification. Its goal is to classify a new incoming sample starting from a predefined set of classes and a training dataset, where each sample is associated with one of the known classes. The K-NN assigns the class to the query sample through a majority vote, looking at the classes of the K nearest samples in the training dataset. Generally, the data are d-dimensional, and the metric for evaluating the distances between sample a and b is based on the L_p norm (or Minkowski distance):

$$L_p(a,b) = \left(\sum_{i=1}^{d} |a_i - b_i|^p \right)^{\frac{1}{p}}, \tag{3.1}$$

where p defines the particular chosen norm.
Here, the PLiM structure is modified to implement all the distances computations among each sample in the dataset and the new sample. Thanks to the SIMD processing mode, the evaluation of the distances can be performed in very few instructions independently of the dataset size.

Algorithm to be mapped

Given a bi-dimensional dataset (d = 2) composed of N samples (x_i,y_i) and the sample to be classified (x_s,y_s), compute all the N distances D_i between the dataset samples and (x_s,y_s), using the Manhattan norm (p = 1):

$$D_i = |x_s - x_i| + |y_s - y_i| \tag{3.2}$$

PLiM Structure & Algorithm mapping

The K-NN mapping on the PLiM Unit is thought of to associate with each couple made of smart row with the related standard up row a different 2-dimensional sample of the dataset, while the target sample is saved occupying two locations in the standard section (one for x_s and the other for y_s). Specifically, the LiM Array must be made of at least N smart rows and two standard section rows. Each of the N row words is initialized with the associated x_i value, while its up row contains the y_i value. Then, at the end of the benchmark execution, the results (i.e. all the distances) will be stored in the standard down rows.
It follows that the CPU will take $2 \times N + 2$ clock cycles to correctly load all the involved data in the memory array before the algorithm starts, using the standard write mechanism for a one single write port data memory.
Moreover, to perform the required operations, the structure of the smart row embedded in the LiM Array, beside the evergreen RCA&LOGIC RI, must include the ABS RI, aimed at computing the absolute value involved in the distance formula.
The information about the cells initial content and the smart row layout is summarized in Figure 3.10.

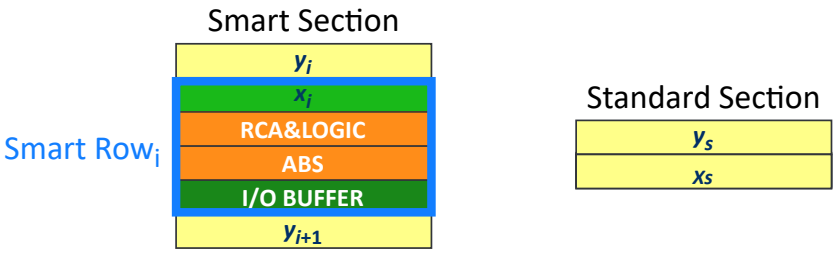


Figure 3.10: Smart row composition and LiM array content initialization for the K-NN benchmark.

Steps	***#Instructions***
1. Save into the input buffers all the y_i values to free the up/down rows.	1
2. Compute all the $\|x_s - x_i\|$ terms and save them into the down rows.	2
3. Compute all the $\|y_s - y_i\|$ terms and save them into the output buffers.	1
4. Compute the final sums and store them into the down rows.	2
# Instructions K-NN	6

3.2.2 Matrix-Vector Multiplication (MVM)

The MVM function falls within the set of applications that can be successfully mapped onto the PLiM since it is mainly based on a set of data multiplications and sums that can be executed simultaneously if the data inside the LiM Array are properly placed. The MVM operation is defined as:

$$\overline{Z} = \overline{\overline{X}} \times \overline{Y} \text{ where } \overline{\overline{X}} \in \mathbb{R}^{u \times v}, \overline{Y} \in \mathbb{R}^{v \times 1}, \overline{Z} \in \mathbb{R}^{u \times 1} \tag{3.3}$$

Even if the PLiM implementation of this benchmark is pretty much faster than the standard processors one, unlike the K-NN, the number of LiM instructions needed to perform it depends on the size of the elaborated data.

Algorithm to be mapped

Given N matrices $\overline{\overline{X}} \in \mathbb{R}^{u \times v}$ *and N vectors* $\overline{Y} \in \mathbb{R}^{v \times 1}$*, compute N matrix multiplications, each on a different couple of* $\overline{\overline{X}}$ *and* $\overline{Y}$*, evaluating for each product all the u elements* z_i *of the resulting vector* $\overline{Z} \in \mathbb{R}^{u \times 1}$*:*

$$z_i = \sum_{j=0}^{v-1} x_{i,j} y_j \text{ , with } i = 0, 1, ..., u\text{-}1 \tag{3.4}$$

PLiM Structure & Algorithm mapping

To fully exploit the SIMD elaboration capability of the PLiM Unit, the array has to be initialized so that all the products contributing to the final vector elements are computed in parallel, each through a different smart row. For this reason, the $\overline{\overline{X}}$ matrix elements are stored each in a different row word. Specifically, the first v locations of the array contain the elements of the first $\overline{\overline{X}}$ matrix row, then the following v row words hold the elements of the second $\overline{\overline{X}}$ row, and so on. Besides, the y_j elements are saved multiple times inside the array to enable the direct execution of all the products in a single LiM instruction. The $\overline{Y}$ is copied u times filling the up rows, starting with the first up row that is initialized with the y_0 element, then the following one is filled with the y_1 element, and so on. Once all the $\overline{Y}$ elements are over, the next up row is driven again with the y_0 element, going on with the filling of following up rows by repeating the same previous association. It follows that the LiM array must comprise at least $u \times v$ smart rows while no constraints are present for the sizing of the standard section. Once the algorithm has been run, the final values of the $\overline{Z}$ elements will be stored in the down rows, with the following correspondence: the z_i element will be contained in the $(2 \times i)th$ down row.
Concerning the number of clock cycles required for the LiM Array content initialization, this corresponds to $2 \times u \times v$.
From the smart row composition standpoint, apart from the RCA&LOGIC RI required for computing the final sums, a further RI aimed at performing the products must be added, i.e. the multiplier, as shown in Figure 3.11.

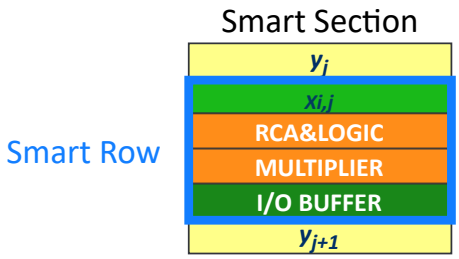


Figure 3.11: Smart row composition and LiM array content initialization for the MVM benchmark.

Steps	***#Instructions***
1. Save into the input buffers all the y_j values to free the up/down rows.	1
2. Compute all the $x_{i,j}y_j$ terms and save them into the row words and in the down rows.	3
3. To compute in parallel all the final sums that generate the $\overline{Z}$ elements: execute each sequential sum that returns the z_i element by propagating along the array the values of the related partial sums. Each partial sum result is obtained by taking the previous result stored in the up row, summing it to the row word content and then saving it again into the down row. This step is repeated for $v-1$ times to gain the final values of the $\overline{Z}$ elements.	$2 \times (v-1)$
# Instructions MVM	$4 + 2 \times (v-1)$

3.2.3 K-means

Another machine learning application selected as a testbench for the LiM Unit is the K-means. It is used for clustering a set of data into K groups based on similarities that are common to the samples included in the same cluster. So, the K-means performs a task that resembles the data classification one, but, differently from the K-NN, it works in an unsupervised way. It means that, to classify a sample, it does not rely on an already classified training dataset known at prior, but it relies on the information on the data features identifying a specific class (or cluster) gathered during its same execution. For this reason, it is an iterative algorithm, which converges to the final composition of the data clusters by repeating the same routine multiple times. The metric used to state if the samples are similar is again their distance. Data that result close in the d-dimensional samples space are considered similar and are assigned to the same group. K initial reference samples, called centroids, are provided before the algorithm starts to perform the data clustering.
The K-means iterative routine is divided, in turn, into three subroutines: first, all the distances between the dataset samples and all the centroids are evaluated, then an initial clustering is proposed by assigning each data sample to the nearest centroid. Once K groups of samples are gathered, the new reference point for each cluster is retrieved by computing the mean among the samples belonging to that specific cluster. These three steps are repeated until the composition of the clusters at the end of the macro-routine remains the same as the one at the beginning of the same step.
Here, the PLiM design is customized to map the first two subroutines of the macro iterative step. The distances evaluation task can be massively sped up thanks to the device processing parallelism. Moreover, the samples assignment task is carried out in parallel for all the data. Therefore it can equally benefit from the PLiM structure parallelism. However, it still remains a part to be run sequentially. All the distances between the centroids and a specific dataset sample are sequentially compared and, for each sample, the smallest distance is saved with the information about the centroid (or the cluster) to which that distance refers.
It follows that, although most of the operations can be executed simultaneously, the duration of the K-means still depends on the required clustering, and, specifically,

it is proportional to the number of clusters in which the dataset must be grouped.

Algorithm to be mapped

Given a 2-dimentional dataset composed of N samples (x_i,y_i) and K centroids (x_{cj},y_{cj}), find for each of the sample the closest centroid and save both the related distance value and the number of the centroid to which that distance refers. Implement the requested task following the procedure below.

```
for j in K centroids (x_cj, y_cj) :
    for i in N samples (x_i, y_i) :
       D_i = (x_i - x_cj)^2 + (y_i - y_cj)^2;
       if j = 0 :
          ClusterID_i = j;
          SmallerD_i = D_i;
       else :
          if D_i < SmallerD_i-1 :
             ClusterID_i = j;
             SmallerD_i = D_i;
```

PLiM Structure & Algorithm mapping

To accomplish the K-means algorithm parallelizing the mentioned steps, like in the K-NN case, each smart row of the LiM Array is dedicated to a different dataset sample. In particular, the row word contains the y_i sample value while the related up row the x_i one. Then, all the centroids are saved in the standard section with their identification (ID) number. Therefore, the K-NN needs the LiM Array to involve a standard section composed of at least $3 \times K$ locations and a smart section characterized by N smart rows.
It follows that the array content initialization lasts for $2 \times N + 3 \times K$ clock cycles. Concerning the centroid ID use, this is exploited to encode the final association between the samples and the centroids. More in deep, the data inside the algorithm are handled to keep the most significant bits as a dedicated field that, once a distance is computed, stores the centroid ID to which that distance refers. In this way, it is

possible to keep track of the association between the smaller distances and the associated centroids until the algorithm's end. It means that the single data is composed of two different fields: the one involving the MSBs is assigned to the ID information, while the LSBs hold the distance value information. Therefore, for instance, if the memory parallelism is of 16 bits and K equals 3, the first 2 MSBs will contain an ID number, while the lower 14 bits will express a distance value. Moreover, the generation of this kind of data is performed through or-masking operations; therefore, in the example, the initial ID values stored in the standard locations will be given by the ID value shifted left by 14 positions.
Furthermore, at the end of the algorithm execution, the final results (that provide for each sample the cluster it belongs to and the distance from the cluster centroid) are stored each in the down row of smart row it was assigned originally.
Lastly, the smart row structure needs to be considerably extended by inserting a series of further RI besides the RCA&LOGIC one, as illustrated in Figure 3.12.

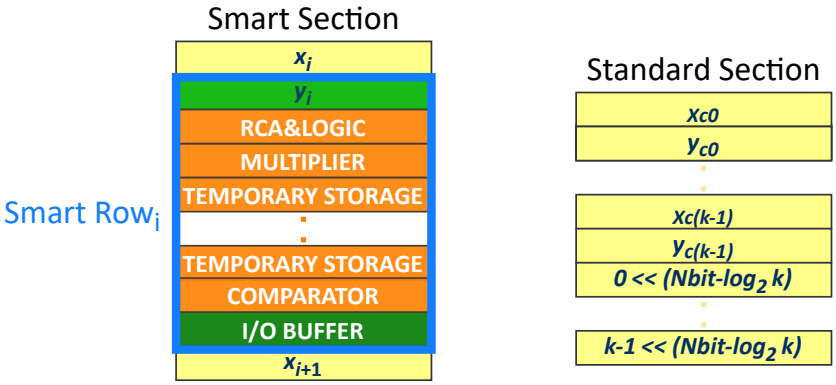


Figure 3.12: Smart row composition and LiM array content initialization for the K-means benchmark.

It must include a multiplier to perform the square function, a series of temporary storages and a customized comparator. Since the K-means execution is organized first to evaluate all the distances between all the dataset samples and the centroids and then all the associations, further storage space is required inside the single smart row to store all the distances between the sample smart rows stores and all the centroids. Thus, inside each smart block, K temporary storages are instantiated.

The comparator block is, instead, exploited during the association of the samples to the closest centroid. It is a simple combinational block that takes two data input, compares the two data portions holding the distance information and returns a data containing in the LSBs the value of the smaller distance and in the MSBs the ID centroid number referred to it.

Steps	***#Instructions***
1. Save into the input buffers all the x_i values to free the up/down rows.	1
2. Compute all the $(x_i - x_{cj})^2$ terms and save them into the down rows.	2
3. Compute all the $(y_i - y_{cj})^2$ terms and save them into the output buffers.	1
4. Compute the final sums (distances) and save them into the output buffers.	1
5. Apply a mask on the computed data to force their MSBs to be equal to the centroid number they refer to and save the results in the first Temporary Storages.	1
6. Repeat steps 2., 3., 4. and 5. for all the k centroids, changing each time the Temporary Storages where the final results are stored.	$5 \times (K - 1)$

7.	Compare the content of the first two Temporary Storages (excluding the MSBs identifying the centroid) and save the lowest value, together with the ID it refers, in the down row.	5
8.	Repeat step 7 for other K - 2 times, taking each time the data in the down row and the one in the next Temporary Storage.	$3 \times (K-2)$
	# Instructions K-means	$8 \times K - 1$

3.2.4 Mean & Variance (μ&σ^2)

Calculating the mean and the variance on a set of data represents the first faced application that does not perfectly fit a PLiM implementation. The reason is that the parallel processing potentiality of the PLiM architecture cannot be fully exploited since this benchmark involves several interdependent sums, each on a number of elements equal to the dataset size, which can only be performed one after the other in a defined order. Moreover, it follows that the number of instructions employed to perform both computations grows accordingly with the number of considered samples.

Algorithm to be mapped

Given a set of N data, compute the mean μ and the variance σ^2:

$$\mu = \sum_{i=0}^{N-1} \frac{x_i}{N} \qquad ; \qquad \sigma^2 = \frac{\sum_{i=0}^{N-1}(x_i-\mu)^2 - \frac{\left[\sum_{i=0}^{N-1}(x_i-\mu)\right]^2}{N}}{N} \tag{3.5}$$

PLiM Structure & Algorithm mapping

Like for all the benchmarks treated previously, the LiM array is initialized to keep the usual association smart row - sample. Thus, each row word has to hold a different data x_i of the dataset. While, from the standard section standpoint, only two

other values must be stored (0 and $log_2 N$) representing constants used during the algorithm execution. Then, in the end, the resulting μ and σ^2 values will be stored in the down row and in the row word of the last smart row, respectively.
Hence, the LiM Array must be sized to include a standard section of at least 2 locations and a smart section on N smart rows, whereby the LiM Array initialization will take $N + 2$ clock cycles.
Here, the smart row needs to be carefully designed to avoid an extreme overhead in terms of complexity and power consumption. As outlined in Figure 3.13, four other RIs are embedded, in the order: an AR Shifter, 2 Temporary Storages and a Multiplier.

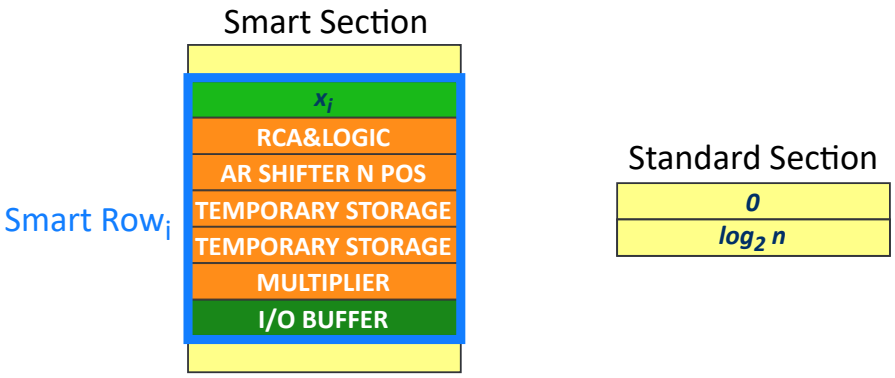


Figure 3.13: Smart row composition and LiM array content initialization for the $\mu\&\sigma^2$ benchmark.

It is worth noting that, unlike previous benchmarks, the multiplier is put at the tail. This is done to take full advantage of the smart row structure and use the last Temporary Storage as if it were a standard source operand. Looking at the algorithm steps listed below, in step 9, the temporary storage content can be directly taken and passed to the multiplier. Then the produced result can be stored in the output buffer, all during a single instruction. While, if the multiplier were inserted above that temporary storage, two instructions would be required to perform the same operations. Moreover, to perform the divisions by N, a programmable $Nbit$ positions arithmetical right shifter is inserted inside the smart row, meaning that the generated PLiM cannot be used to perform this benchmark for any value of N,

but the size of the target dataset needs to be a power of 2. This constraint cannot be avoided since, otherwise, one divisor block should be inserted into each smart row, leading to a prohibitive area and power overhead.

Steps		***#Instructions***
1.	Compute $\sum_{i=0}^{N-1} x_i$, by repeating for N times the addition between the content of the row word x_i and the up row and the storing of the result in the down row.	$2 \times N - 1$
2.	Compute the μ by shifting the result (contained in the output buffer of the last smart row) of $log_2 N$ position towards right and save it into the down row.	2
3.	Save into the first Temporary Storages all the x_i values to free the row word locations.	1
4.	Save in all the input buffers the computed μ.	1
5.	Compute all the $x_i - \mu$ terms and save them into the rows and down rows.	3
6.	Compute $\sum_{i=0}^{N-1}(x_i - \mu)$, by repeating for $N-1$ times the addition between the content of the row word $(x_i - \mu)$ and the up row followed by the storing of the result in the down row. Save the result of the last iteration in the other Temporary Storage (the second one).	$2 \times N - 3$
7.	Compute all the $(x_i - \mu)^2$ terms and save them into the rows and down rows.	3

8.	Compute $\sum_{i=0}^{N-1}(x_i-\mu)^2$, by repeating for $N-1$ times the addition between the content of the row $(x_i-\mu)^2$ and the up row and the result storage into the down row.	$2 \times (N-1)$
9.	Compute $\frac{sum3^2}{N}$, by multiplying by itself the content of the second Temporary Storage and then shifting the result of $log_2 N$ positions towards right. Save the outcome into the output buffer of the last smart row.	2
10.	Compute the final variance, by taking the content of the last down row $\left(\sum_{i=0}^{N-1}(x_i-\mu)^2\right)$ and subtracting the content of the output buffer $\left(\frac{sum3^2}{N}\right)$, then again shift right the result of $log_2 N$ positions and store the final result into the last row word location.	3
11.	Take the μ value contained in all input buffers and copy it in the related down row.	1
	# Instructions μ & σ^2	$6 \times N + 10$

3.2.5 Discrete Fourier Transform (DFT)

The last tested benchmark is the DFT, whose PLiM implementation shows a performance trend similar to one of the mean and variance algorithm. Here, the PLiM Unit carries out the evaluation of a single frequency component, which is characterized by a first parallelized phase, followed by a more serial one.
It derives that, again, the instruction amount deployed in this benchmark grows linearly with the number of timing samples. Although more efficient algorithms are proposed to perform this function in the literature, here the straightforward expression is implemented to pop the criticalities of the PLiM framework.

Algorithm to be mapped

Given a set of N timing samples x_i, compute the k-th frequency component X_k:

$$X_k = \sum_{i=0}^{N-1} x_i \times \left[\cos\left(\frac{2\pi ik}{N}\right) - j\sin\left(\frac{2\pi ik}{N}\right)\right] \tag{3.6}$$

PLiM Structure & Algorithm mapping

The LiM Array is organized so that the first half of the smart section is dedicated to the computation of the real part of the frequency component, while the second half processes the imaginary contribution, so that both terms can be elaborated in parallel in the same array. For this reason, during the memory initialization phase, the set of timing samples is entered twice in the array, one for preparing the processing of the real value and one for setting up the imaginary part elaboration. For each of the two macro-blocks in which the smart section is split, the row words are filled with the i index value, namely 0 for the first row word, 1 for the second and so on till the macro-block end. Then, in each up row, the value of the timing sample, associated with the index i stored in the related row word, x_i is stored. As for the previous benchmark, the standard section is used to store 5 values of constants that are: k, which identifies the required frequency component, 2*pi that is a known value called in Equation 3.6, and $log_2 N$, 0, and 1 that are used by the smart rows RIs to accomplish some specific functions, like the cosine, the sine and the division by N functions. Concerning where the real and imaginary part results will be available at the algorithm end, these will be retrieved from the last down rows of both macro-blocks.
Summing up, the LiM Array must contain at least $2 \times N$ smart rows and 6 standard section rows. Hence, initializing the LiM Array content $4 \times N + 5$ clock cycles is needed.
Furthermore, Figure 3.14 portrays how the PLiM smart row has to be customized for this benchmark.

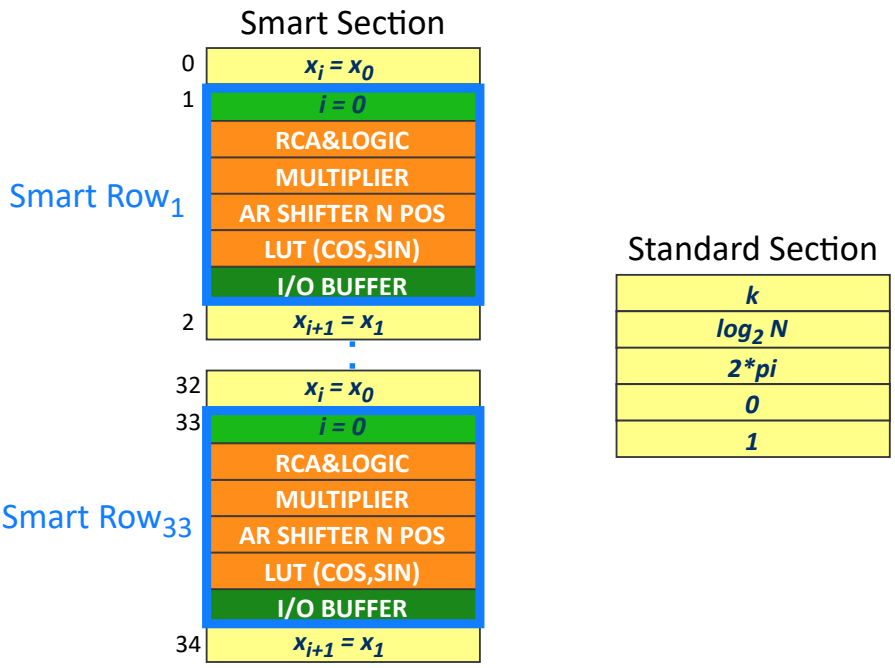


Figure 3.14: Smart row composition and LiM array content initialization for the DFT benchmark.

The RIs chain is composed of the sequence of the standard RCA&LOGIC, exploited to perform the final sums, a multiplier, for all the product operations, an arithmetical *Nbit* positions right shifter, to perform the division by N, a final customized 32x6 bits LUT block handled to implement the cosine and the sine functions. Once more, there is a constraint on the N value that can be only equal to a power of two due to the same reason as the previous benchmark. For the cosine and sine functions implementation, the CwM technique is exploited. Sine and cosine are performed in a discretized way. It means that several sampled values from the trigonometric functions (16 for each of them) are saved in the LUT. In this way, when the LUT is addressed with the data on which the function must be applied, the closest approximated sine or cosine value is returned. Despite the smart section is thought of so that the first half has to perform only the cosine function while the second the sine one, there is no way to diversify the smart row composition so that the smart rows belonging to the first macro-block include a LUT only providing the cosine function while the ones in the second macro-block embed a LUT implementing only

the sine function. It means that the LUT block has to include both functions and, to distinguish between the two, it must take two input data, one forwarding the value on which the trigonometric function must be evaluated and one expressing if a sine or a cosine processing is required. Note that a configuration signal cannot carry this choice because the interface of the generic RI is fixed, as already explained. Moreover, being the computing model a SIMD one, to correctly achieve the real and the imaginary values, the enable per macro-blocks mechanism must be used. When the first half of the LiM array has to evaluate the cosine, the second half must be disabled, and vice-versa to avoid the overwriting of the other half results.

Steps	***#Instructions***
1. Compute all the $\frac{2\pi ik}{N}$ terms and save them into the output buffers.	3
2. Enable only the upper half of the smart section and compute all the $cos(\frac{2\pi ik}{N})$ terms, by giving to the LUT the content of the output buffer and the value 0, taken from the standard section, and save them into the output buffers.	1
3. Enable only the lower half of the smart section and compute all the $sin(\frac{2\pi ik}{N})$ terms, by giving to the LUT the content of the output buffer and the value 1, taken from the standard section, and save them into the output buffers.	1
4. Compute all the $x_i \times cos(\frac{2\pi ik}{N})$ and $x_i \times sin(\frac{2\pi ik}{N})$ terms, by multiplying the output buffers content with the row word locations, and save the results in the row words themselves.	2

5.	Save into the input buffers all the x_i values to free the up/down rows.	1
6.	Store the value of the output buffers into the down rows.	1
7.	Compute the final sum, by repeating for $N-1$ times the addition between the content of the row word and the up row, followed by the storing of the result in the down row. The final value of X_k will be stored in the down rows of the last smart row of both smart section macro-blocks.	$2 \times (N-1)$
	# Instructions DFT	$2 \times N + 7$

Algorithms	**# LiM initialization clock cycles**	**# Algorithm instructions**	**# Algorithm clock cycles**
K-NN	$2 \times N + 2$	6	$2 \times N + 14$
MVM	$2 \times u \times v$	$4 + 2 \times (v-1)$	$(2 + 2 \times u) \times v + 8$
K-means	$2 \times N + 3 \times K$	$8 \times K - 1$	$2 \times N + 11 \times K + 5$
µ&σ^2	$N + 2$	$6 \times N + 10$	$7 \times N + 18$
DFT	$4 \times N + 5$	$2 \times N + 7$	$8 \times N + 18$

Table 3.9: Detail on the number of clock cycles required by each tested benchmark to complete its execution.

3.2.6 PLiM Performance Results

To complete the study and gather more realistic details about the performances achieved by the customized PLiM devices, the generated architectures for all the benchmarks were synthesized and then passed through the place&route process. The results coming out from these analysis steps are reported in Table 3.10 and Table 3.11, respectively. Note that the LiM Array was implemented through standard cells using the Nangate45 library.

Algorithms	K-NN	MVM	K-means	μ&σ^2	DFT
Parameters	N = 256	u = 16, v = 16 t = 1, N = 1	N = 256 K = 3	N = 256	N = 128
Area **[mm²]**	0.46	0.59	0.8	0.74	1.07
Critical Path **[ns]**	4.2	4.68	5.28	6.43	5.77
Max Clock Frequency **[MHz]**	238.09	212.76	188.67	153.84	172.41
Power **[mW]**	60.56	60.99	79.78	61.68	119.99
Execution Time **[#Clock Cycles]**	526	552	550	1810	1042
Execution Time **[μs]**	2.2	2.59	2.91	11.76	6.04
Energy **[μJ]**	0.13	0.15	0.23	0.72	0.72

Table 3.10: Worst case performance achieved after the synthesis process for all the benchmarks-customized architectures.

To better understand and compare the performance among the different benchmarks, apart from the smart row composition that, as already said, is customized for each application, the LiM Array size chosen is the same for all the synthesis.

LiM Array Structure

```
Standard Section:   #Standard Rows = 159
Smart Section:      #Standard Rows = 257 , #Smart Rows = 256
                    #Enabling Blocks = 4 , #Smart Rows ∀ Block = 64
-------------------------------------------------------------------
# Total Rows = 672 , Memory Parallelism (Rows width) = 16 bits
⟹ Addressable Space = 1344 bytes
```

Hence, the number of samples N on which the benchmarks were run was chosen to maximize the use of all the smart rows. In this way, the throughput under the same (worst case) power consumption is enhanced, causing, in turn, a drop in the energy per sample expense. Therefore, each benchmark was run on the maximum number of samples allowed by the combination of the LiM Array structure with the initialization constraints for that benchmark (as described in the previous sections

3.2.1, 3.2.2, 3.2.3, 3.2.4, 3.2.5). In the `Parameters` row of both Table 3.10 and Table 3.11 the specific values set for all the parameters proper of each algorithm are spelt out. As it can be noticed, most of the benchmarks work on a dataset on 256 samples, while the DFT processes a halved dataset due to the splitting of the result in imaginary and real contributions. Lastly, the MVM represents a special case since it runs on a different data typology, i.e. matrices and vectors. Nevertheless, all of them succeed in exploiting all the available smart rows to parallelize and speed up the execution of all the required tasks as much as possible.

Algorithms	**K-NN**	**MVM**	**K-means**	**μ&σ²**	**DFT**
Parameters	N = 256	u = 16, v = 16 t = 1, N = 1	N = 256 K = 3	N = 256	N = 128
Area **[mm²]**	0.43	0.56	0.78	0.74	1.03
Critical Path **[ns]**	4.01	3.92	4.08	4.11	4.21
Max Clock Frequency **[MHz]**	243.9	250	243.9	238.09	232.55
Power **[mW]**	246.13	430.05	604.4	578.74	614.85
Execution Time **[#Clock Cycles]**	526	552	550	1810	1042
Execution Time **[μs]**	2.15	2.2	2.25	7.6	4.48
Energy **[μJ]**	0.52	0.94	1.35	4.39	2.75

Table 3.11: Worst case performance achieved after the place&route process for all the benchmarks-customized architectures.

The discussion here presented directly relates to the post place&route analysis since it is the most faithful to the actual performance the devices could achieve due to the more detailed technology models on which it works. Thus, taking a look at Table 3.11, different information on each of the PLiM architectures, specifically customized for each algorithm, can be retrieved, such as on-chip area occupation, maximum allowed clock frequency, and worst-case power and energy consumption. First of all, it must be highlighted that the values shown for the metrics in the last

three tables rows were obtained with a clock rate equal to the maximum clock frequency allowed. Then, it is worth first focus on the critical path and power metrics that are tightly correlated to the smart row composition but disjointed from the size of the considered dataset.
For all the benchmarks, the critical path covers both the MI, which, due to the initial LiM Array setting, always has the same structure, and the entire smart row, since it is mostly made up of chained combinational blocks, as the modular RI insertion mechanism requires. So, the slowest PLiM architecture is the one performing the DFT (Critical Path = 4.21 ns). Recalling the RIs series constituting the DFT smart row, the critical path passes through a 16 bits RCA block, followed by a multiplier on 16 bits input data, a programmable 16 positions shifter (in practice given by a multiplexer with 16 inputs each on 16 bits), and a series of multiplexers that bypasses the LUT (implemented through registers) and it ends into the MI. On the contrary, the fastest device is the one returning the MVM results (Critical Path = 3.92 ns) because the RIs inside the smart row are only given by the 16-bit RCA&LOGIC block and a successive 16-bit multiplier. However, the speed difference between these two architectures is relatively small (0.29 ns). This happens because, for all the benchmarks, the main contributions to the total critical path are given by blocks belonging to the compulsory starting base of the PLiM Unit, namely, by the collection of MI, row word and input/output buffer that take altogether about 1.5 ns and by the RCA&LOGIC RI that employs about 1.3 ns. Moreover, another remarkable contribution comes from the commonly used multiplier RI, which runs for about 0.6 ns. From here, the first weakness of the PLiM model emerges, as, differently to what typically happens, the processing of sums operations takes double time compared with the products one. The reason is that, while the multiplier block is instantiated in its optimized form, the sums are performed through an RCA component, which is the least efficient adder in terms of timing due to the carry chain which propagates along all the FAs in the row word. It follows that the bigger the memory parallelism, the longer the critical path. Moreover, the cascade-like development of the smart row seems to be the leading cause for the significant length of the critical path, which would result in a shorter version if the modular structure of the RIs were designed to have all the RIs placed side by side instead of in sequence. Through this arrangement, the critical path would be dictated by the slower RI component

instead of the sum of all the embedded RIs. Estimating the DFT critical path case, it would be reduced by about 1 ns, reaching a maximum clock frequency of 333 MHz instead of 232 Mz. It must be noticed that, in this way, there would be no more "composed" instructions in the ISA (the ones given by the enabling of multiple RIs in sequence in the same clock cycle). Whereby, some of the operations which, in the original version, require a single clock cycle to be processed, with this new smart row composition, would need to be computed through a sequence of simpler instructions and employ more than one clock cycle. However, analyzing the LiM codes associated with all the mentioned benchmarks, this kind of complex instructions are very rarely used. Therefore, the benefit in terms of total algorithm execution time, which would be gained because of the improved operating frequency, would extensively compensate for the negligible increase in the total clock cycles required to complete the application.
Summing up, the maximum clock frequency among all the customized devices ranges from about 232 MHz to a maximum of 250 MHz, reached in the MVM case.
Then, from the worst-case power standpoint, the architecture performing worst is once again the DFT one, even if it reaches a value similar to the ones of the k-means and $\mu\&\sigma^2$ devices (600 mW in the average). However, unlike the critical path metric, the power range between the most and the least consuming systems is significant and equals 370 mW. The reason for this remarkable difference lies again in the structure of the single smart row, which, in the most efficient case (K-NN with power = 246 mW), is only given by the RCA&LOGIC block and the absolute value evaluator, while, in the least one (DFT with power = 614 mW), it is composed of the RCA&LOGIC block, the multiplier, the programmable shifter and the LUT (composed by 32 registers on 6 bits each). Support for this argument is get by taking a look at the algorithms characterized by similar power consumptions (K-means, $\mu\&\sigma^2$, and DFT), as their related PLiM platforms all integrate smart rows with similar structures, both concerning the number of included RIs and their functionality. Moreover, it is worth mentioning that the power consumption directly grows with the clock frequency used to retrieve the circuit power estimation. However, the set frequencies do not vary much from one benchmark to the other, and, therefore, they do not affect the visible differences in the power consumptions.
Lastly, from Table 3.11 it can be seen that the area occupation among the different

benchmarks follows the same trend of the power consumption, going from 0.43 mm^2 in the best case (K-NN) to a doubled area occupation in the worst case (DFT, area = 1.03 mm^2). That is because, as the power, also the area is strictly related to the structure of the smart rows, assumed that the number of rows forming the LiM Array is the same for all the cases.
So far, all the results for the metrics uncorrelated with the number of processed samples were commented. However, for the remaining metrics, namely, the execution time and the energy, the samples number plays a major role. How the execution time depends on the dataset size changes according to the particular algorithm considered and, in Table 3.9, the expressions describing these relations are detailed for all the benchmarks. Referring once again to Table 3.11, the execution time strongly depends on the number of clock cycles required to execute the benchmarks rather than the clock period. This could be expected since the difference in the benchmarks clock period is not so relevant, as already highlighted. The slowest algorithm is the $\mu\&\sigma^2$ (7.6 µs) followed by the DFT (4.48 µs), while the remaining three benchmarks show about the same time value. Both the slowest algorithms are characterized by sums operations involving a number of elements equal to the dataset size and that, as already explained, do not fit well the PLiM computing paradigm. While, for the other benchmarks, the correlation between the instructions number and the samples number is slight and, in particular, the total execution time is driven by the memory content initialization phase. Indeed, for the fastest benchmark (K-NN, execution time = 2.15 µs) the instructions number is totally disjointed from the number of elaborated data. Furthermore, as summarizing metric to evaluate the goodness of architectures derived through the PLiM model, the worst-case energy consumption is computed, multiplying the execution time by the worst-case power. The outlined trend replicates the execution time one, meaning that the execution time has a leading role in the total energy consumption, representing the actual issue to tackle when trying to improve the system performance. In conclusion, in terms of energy costs, the least efficient system is the one generated for the $\mu\&\sigma^2$ benchmark (4.39 µJ), while the most performing architecture is the one linked to the K-NN algorithm (0.52 µJ).
Ultimately, since, as already stated, the number of considered samples differs for each benchmark, to provide a further and fairer analysis on the suitability of the

PLiM model for different algorithms, in Figure 3.2.6 the values for the execution time, power and energy evaluated on a per-sample basis are reported, so gaining a picture of the performance accomplished by the PLiM architectures that abstracts from the specific dataset considered.

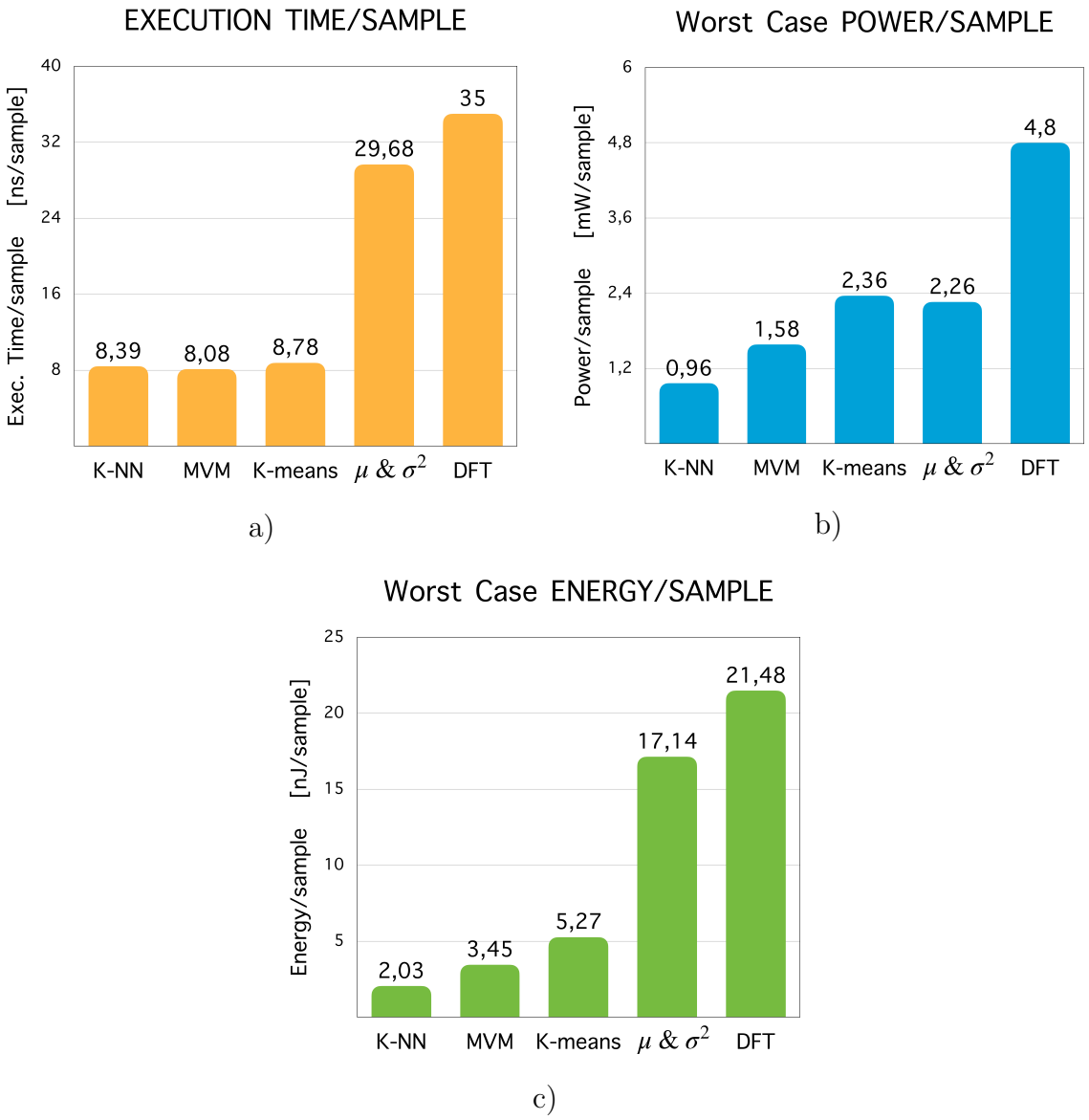


Figure 3.15: The PLiM devices performance trends across all the benchmarks, concerning a) the execution time, b) the worst-case power, and c) the worst-case energy consumption, are outlined on a per-sample basis.

Starting from the power/sample, the trend across the different benchmarks reflects the total power one since, as already said, the worst-case power has no direct dependence on the number of samples. So, the best power consumption is achieved by the structure that implements the K-NN (0.96 mW/sample) while the worst by the DFT architecture one (4.8 mW/sample), returning a difference of 80% on the power expense. Then, dealing with the metrics which are actively affected by the dataset size, a turnaround is felt concerning the achievements of the last two benchmarks. Both for the execution time and the energy metrics, the values on a per-sample basis show performances that worsen in the DFT architecture compared with the $\mu\&\sigma^2$ case. The reason is that, while the DFT system processes a samples number that is half the number of data elaborated by the $\mu\&\sigma^2$ platform, the difference both in the total execution time and in the energy consumption is less than half. Anyway, the execution time/sample and the energy/sample graphs clearly point out for which kind of algorithms the PLiM model acts as a perfect canvas to develop customized solutions and for which it does not. The algorithms that profitably exploit the PLiM Unit processing parallelism (K-NN, MVM, K-means) perform about 70% - 80% better than the ones who do not ($\mu\&\sigma^2$, DFT), reaching, in the best case, about 8 ns/sample and 2 nJ/sample for the execution time and the energy, respectively, while touching in the worst one 35 ns/sample and 21.48 nJ/sample values.

3.3 Conclusions

The PLiM framework represents a template for designing programmable LiM custom solutions that follow the SIMD computing paradigm.
From the analysis carried out on the performance results, it derives that, as expected, algorithms requiring a considerable set of RIs converge to devices characterized by higher clock periods and significant power consumptions. However, this does not necessarily result in long execution times and heavy power consumptions per sample, as long as the algorithms processing can be fully parallelized (high throughput) and the inserted RIs can be used to implement very customized instructions which benefit from a hardware implementation. For this reason, the architectures derived for running heavy data crunching applications, whose processing can be proficiently

parallelized, are very efficient, especially in terms of execution time and energy consumption.
Summing up, the PLiM approach strengths are tied back to the multiple-level of provided computing parallelism, given by the SIMD PLiM Unit sided by a standard CPU implementing a sequential processing paradigm, and to the easiness of customizability of the starting template, which translates into reduced efforts and design time.
Nevertheless, it still suffers from some criticalities that can be tackled to develop an improved paradigm with a next-level efficiency. As anticipated in the previous subsection 3.2.6, if the energy consumption is picked as the metric assessing the goodness of a device to enhance the performances, it is helpful to shorten the execution time. The timing reduction can be accomplished by addressing two aspects: the critical path (meaning increasing the working frequency) and the number of instructions required to perform the algorithms. Concerning the first one, some ideas to cut down the critical path length might be the parallel arrangement of the RIs inside the smart row, replacing the sequential one, and the insertion of a time-optimized adder component, different from the RCA one, as already explained previously. While, the reduction of the instructions number per program could be very useful, especially in the case of applications involving more sequential operations, like iterative routines characterized by data dependencies among different iterations (examples are sums or maximum/minimum search functions performed on a massive amount of data). In this case, it could be helpful to streamline the data exchange mechanism inside the LiM Array, which, for instance, could be done by implementing the direct writing inside the row words or the standard rows after a data elaboration (instead of the compulsory passing through the output buffer) or incrementing the number of temporary storage elements which can act as direct providers for source operands.
However, these are only a few hints on how the PLiM framework can be optimized to achieve better performance while adapting to a broader set of algorithms. Another final hint could be implementing a finer row enabling mechanism to compensate for the SIMD behaviour when algorithms with a lower level of parallelism need to be mapped on an architecture generated starting from this model.

Chapter 4

GP-LiMA Paradigm

Here, the novelty brought by this thesis work is presented. A new model for developing programmable LiM architectures is outlined, called General Purpose Logic in Memory Architecture (GP-LiMA), which takes a cue from both the PLiM approach (deeply investigated in chapter 3) and the CGRAs platforms (see subsection 2.1.2). The aim is to develop a general-purpose framework that merges the two approaches to take advantage of the strengths coming from both worlds while compensating for their weaknesses. Starting from the PLiM idea of providing a template for speeding up the design process of LiM architectures, the proposed model is intended to minimize as much as possible the effort required to develop customized LiM devices for new applications, providing an already made LiM structure characterized by a high level of programming generality. If required, the proposed architecture can be still easily customized by the hardware designer before the fabrication phase, as it happens for the PLiM approach. However, usually, only a few changes are involved mostly related to the size of the memory array rather than the specific processing capabilities. Thus, the task of mapping new algorithms on this architecture is almost completely entrusted to the LiM programmer, which, after fabrication, at compile-time, can count on a wider LiM ISA, designed ad hoc to be as general as possible and suitable for most algorithms. The programming generality enhancement is obtained by modifying the organization of the LiM Array in the PLiM solution by making it resemble the arrangement of the RCs immersed in a routing network typical of CGRAs. The merging of the PLiM architecture with the CGRAs

characteristic structure comes quite straight since RCs and Smart Rows are both essentially processing units that need to be interconnected in some way to enable the data transfer required by the algorithm they run. Briefly, this integration gathers benefits from both these solutions. From the CGRA it borrows the post-fabrication reconfigurability features, which means flexibility in adapting to efficiently performing a more heterogeneous set of algorithms. Then, it also inherits the general LiM approach advantages, which are the drastic cut of the execution time and the energy expense, due to the sharp drop in the memory accesses number and the full utilization of the memory data parallelism to speed up the processing of data-intensive applications. Finally, it keeps the co-processor asset, so proving a further level of processing parallelism between CPU and LiM Unit that contributes to lower even more the total computing time, leading, in turn, to an additional decrease of the power consumption.
The GP-LiMA paradigm is illustrated in this chapter throughout three sections, following an organization similar to the one of the PLiM Model description (see section 3.1):

- ***Section 4.1* - GP-LiMA Model** outlines the general idea at the basis of the proposed architecture, dealing with aspects such as the working mode, the system environment in which it can be embedded and the macro units that comprise it.

- ***Section 4.2* - GP-LiMA Datapath: LiM Matrix** goes deeper into the LiM Array composition and the data routing network, which altogether form the LiM Matrix. Here, all the details about the processing capabilities of the memory block are extensively treated, highlighting which are the structural aspects of the GP-LiMA that can be easily customized by the hardware designer, when needed.

- ***Section 4.3* - GP-LiMA Control** focuses on the control part of the GP-LiMA model, deepening the blocks that comprise it and how the LiM programmer can actually map an application on the GP-LiMA through the development of a LiM code, composed of a sequence of structured LiM instructions.

4.1 GP-LiMA Model

The GP-LiMA is a model of a general-purpose programmable LiM architecture, which makes programming flexibility and multiple-level processing parallelism its strong points. As the overall basic structure, it traces the PLiM one, namely memory array composed of cells embedding processing elements (LiM array) driven by a 2-steps control unit (uCU and nCU) and which can be started in two different modes: the standard data memory mode in which the CPU interacts with the device as if it were a bare data storage, and the processing mode in which the architecture is enabled to actively process a real algorithm while the CPU is busy performing conditional or pure sequential portions of the overall application committed to the system where the GP-LiMA is embedded. However, the architecture can be handled so that the processing mode can be implemented following not only the SIMD computing model but also the Multiple SIMD one (typical of CGRAs). It means that the LiM array is enabled to simultaneously run a different instruction on each different set of data. Thus, the architecture can be configured to run up to K different concurrent instructions, where K is the level of instructions processing multiplicity and depends on both how the architecture structure is configured during the pre-fabrication phase and how it is programmed after the fabrication. The M-SIMD feature is what brings the processing parallelism of this architecture to the next level, returning a system with boosted algorithm adaptability and reduced execution times.
Indeed, the original idea (at the beginning of this thesis work) was to design an adaptable and very user-friendly pre-fabrication model for producing LiM solutions that could be customized for a wider set of algorithms thanks to the design and integration of an enhanced routing network. However, the LiM array dimension setting and the customization of the internal structure of the processing units were still left up to the hardware designer, as for the PLiM template. Nevertheless, despite this initial objective, the final GP-LiMA paradigm, here proposed, is conceived so as to avoid requiring as much as possible changes to the processing units composition. Indeed, the model steals from the CGRA philosophy not only the density and flexibility of the interconnections but also the generality of the RC. In particular, the GP-LiMA mainly draws inspiration from the MorphoSys structure (see section 2.1.2)

for the processing blocks layout, the M-SIMD computing model implementation and the array arrangement, which involves multiple types of interconnections. Thus, the GP-LiMA framework is provided with a default layout for the processing units which well suits the implementation of most algorithms. This is the reason why the proposed architecture is closer to a general-purpose LiM device solution rather than a pre-fabrication template for the speeding up of customized LiM architectures design. Moreover, the default structure of the processing units exploits also the CwM technique, integrating a LUT component to enable the hardware implementation of some customized specific functions even after the fabrication process, as it happens for FPGAs and CGRAs solutions. This technique confers to the GP-LiMA even hardware reconfigurability properties, up to some extent. It follows that, if the LiM array size, the memory parallelism and the M-SIMD mechanism are properly configured, during the pre-fabrication phase, to be as general as possible, the gathered architecture can efficiently map a wide set of applications, so the function it runs can be changed theoretically infinite times by only re-writing the content of the LiM instruction memory, as it occurs in general-purpose CPU-centric systems.
Nonetheless, it is worth mentioning that, if strictly required, the GP-LiMA processing elements can still be customized by the hardware designer (before fabrication), which can insert logic blocks designed ad hoc to map a specific application. From this standpoint, the GP-LiMA is provided with a mechanism for facilitating the insertion of further computing components inside the processing units (like the modular RIs mechanism does in the PLiM model).
However, hereinafter, the pre-fabrication model with the default configuration for the processing units layout will be thoroughly addressed.

4.1.1 GP-LiMA Overview

As broadly stated, the GP-LiMA is mainly intended as an architectural solution to cope with the memory wall issue, so it must be inserted in standard systems in place of the data memory with which the CPU interacts to retrieve and save data. However, thanks to its internal management of the processing mode, the GP-LiMA well suits to be used as a real co-processor, too. In Figure 4.1 the system environment hosting the GP-LiMA is shown. Although this system is reported in the basilar

form, i.e. it only contains the compulsory blocks, it can be as complex as needed, like real-world systems.

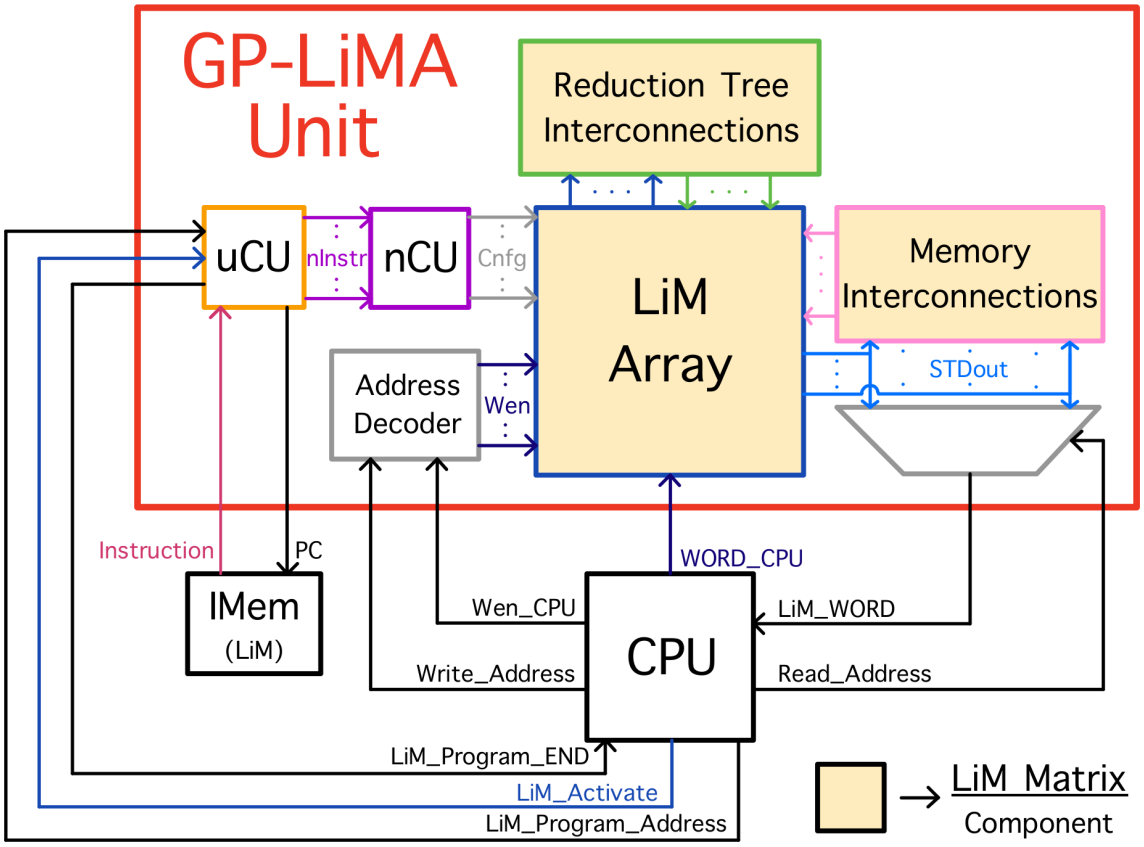
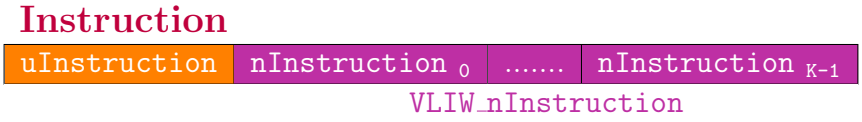


Figure 4.1: Outline of the GP-LiMA framework. The GP-LiMA internal macro-arrangement is shown together with the connections with the external CPU and the LiM instruction memory (IMem).

The main component interfacing with the GP-LiMA Unit is the CPU. Unlike the PLiM framework, here the GP-LiMA computing capabilities are not hidden to the CPU, which, instead, is in charge of starting the GP-LiMA in the processing mode when required. For this reason, the CPU ISA must be customized, inserting a few specific instructions aimed at correctly handle the GP-LiMA processing, as done in the Morphoing System which includes a customized TinyRISC (refer to section 2.1.2). Thus, here, the CPU external interface used to communicate with the

GP-LiMA Unit is composed of 2 different sets of signals: one used to interact following the standard double port data memory protocol (1 asynchronous reading port and 1 synchronous writing port) and one exploited to control the GP-LiMA working mode. The first set includes the following signals: the `Wen_CPU`, to enable the writing inside the GP-LiMA memory, the `Write_Address`, to specify the address of the memory word where the new data must be store, the `WORD_CPU`, which forwards to the GP-LiMA the data to be written, the `Read_Address`, to indicate the address from where the read data must be taken, and the `LiM_WORD`, which carries the read data from the memory. While the other signals set is composed of the `LiM_Activate` signal, sent by the CPU to start the GP-LiMA Unit in the processing mode, the `LiM_Program_Address` signal, output again by the CPU to tell the GP-LiMA which is the next program it is demanded to execute, and the `LiM_Program_END` signal that is asserted by the GP-LiMA Unit to flag the CPU that the results produced by the previously committed program are ready and available inside the memory array.
Moreover, the overall system includes a LiM instruction memory (IMem), which is asynchronously accessed in reading by the GP-LiMA Unit and which must be initialized with all the possible programs the GP-LiMA might be required to execute. Thus, other two signals are present at the GP-LiMA interface: the `PC` signal that is sent by the control part of the GP-LiMA itself to fetch the next instruction it has to execute, by pointing to the address of the IMem where the instruction is stored, and the `Instruction` signal that enters the GP-LiMA to provide the demanded LiM instruction. Besides, to specify which program the GP-LiMA has to run, the CPU drives the `LiM_Program_Address` so that it holds the address of the IMem where the first instruction of the involved program is contained.
From the internal GP-LiMA overview perspective, the architecture is composed of 3 macro sections: the control section, the datapath, and the section dedicated to carrying out the data exchange between the memory array and the CPU.
Concerning the datapath, it is the core of the GP-LiMA paradigm and is identified by the cooperation between the LiM Array, namely the mesh of storage cells filled with logic elements, and the set of different interconnections enabling the flexible and programmable data exchange across the whole array. In particular, as illustrated in Figure 4.1, the routing network comprises two different kinds of interconnections: the Memory ones and the Reduction Tree ones. Then, the combination of these three

components, i.e. the LiM Array, the Reduction Tree Interconnections (Rdt Int) and the Memory Interconnections (MI) constitute what is called the LiM Matrix.
Furthermore, the control part was designed along the lines of the PLiM model. It is given by 2 control units connected in sequence: the uCU, which regulates the instruction flow, and the nCU, which translates the LiM instructions into configuration signals values that drive both the LiM Array and the routing network to properly carry out the data elaboration required. It follows that, as the PLiM template, also the GP-LiMA is a pipelined architecture, that splits the processing into 4 stages. Again the fetch and the decode stage are linked with the uCU and the nCU, respectively, while the execution and the results saving are committed to the LiM Matrix. In particular, the uCU is the control section part that handles the communication with the IMem, so it is the unit that first receives the fetched LiM instruction (from the `Instruction` signal). This `Instruction` is mainly composed of 2 macro fields called `uInstruction` and `VLIW_nInstruction`. The bits composing the `uInstruction` are directly taken by the uCU to prepare the fetching of the new instruction from the IMem, while the remaining bits, which encode the specific operation to be performed, are sent from the uCU to the nCU that adequately sets the LiM Matrix for the processing scheduled for the next clock cycle. Moreover, due to the M-SIMD computing model the GP-LiMA has to implement, the instruction portion forwarded to the nCU is actually a Very Long Instruction Word (VLIW) and so, in turn, it is divided into K fields, each containing one of the K instructions (called `nInstructions`) the LiM Array has to run simultaneously. For this reason, the nCU is, in practice, given by a set of K instruction decoders, each dedicated to the translation of a different `nInstruction` driving a specific set of configuration signals. The nCU decoders organization is depicted in Figure 4.3, while below the referred `Instruction` partition is shown.



Lastly, the section enabling the GP-LiMA Unit to be used as a data memory involves an address decoder and a simple huge multiplexer. The address decoder is used to implement the memory write operation when the GP-LiMA is addressed by the

CPU in the standard data memory mode. This component outputs all the enable (`Wen`) signals for all the LiM Array locations. Then, when a write operation request comes from the CPU (`Wen_CPU` = '1'), it actives only the enable signal connected to the memory word associated with the address sent on the `Write_Address` signal, so that, at the next clock, the addressed location will be initialized with the value forwarded by the `WORD_CPU`. On the other hand, the multiplexer is used to implement the asynchronous memory read operation. It takes as input all the signals holding the content of each of the LiM Array words (`STDout` signals) and assigns to the output signal `LiM_WORD` the content value of the location pointed by the address forwarded by the `Read_Address` signal.

4.2 GP-LiMA Datapath: LiM Matrix

Diving into the GP-LiMA Unit, the first item to be investigated is the LiM Matrix, i.e. where the data elaboration and storage take place thanks to the effective interaction between the LiM Array and the routing network. The Matrix LiM scheme can be observed looking at Figure 4.2.
As for the PLiM template, the LiM Array is the memory array that elaborates the data it holds thanks to the processing elements that are integrated inside each memory cell. Yet, unlike the PLiM Array, it is organized following the real layout of standard memories; that is, in the same memory row more than one word are placed one next to the other. Therefore, the resulting LiM Array is given by a matrix of memory words, which, besides, reflects the arrangement of the RCs inside the MorphoSys framework (see section 2.1.2). The reason for this design choice is traceable to the easiness and flexibility this kind of processing units placement implies in terms of data transfer. The interconnections that can be instantiated inside this grid are denser and more punctually programmable, therefore, during the algorithm execution, the data can be moved along a greater number of possible paths across the entire array. This allows having more direct data transfers between different processing elements, implying a reduction in the number of instructions employed for the data preparation in sight of the next operation performing. As a consequence, the total number of instructions composing the programs is decreased, leading to a total lowering of the execution times as well as the energy spent in the

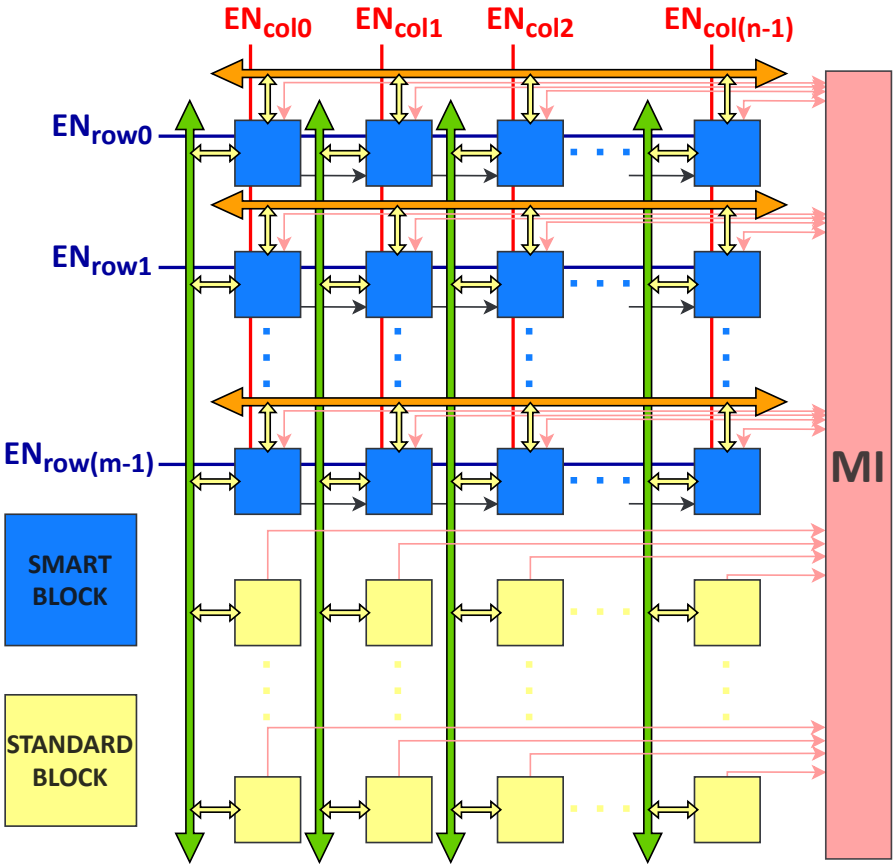


Figure 4.2: LiM Matrix scheme showing a generic LiM Array immersed in the routing network. The orange and green arrows identify the rows and column interconnections, respectively, comprising the Reduction Tree Interconnections, while the pink arrows represent the LiM Array connections to the MI.

process. Referring to Figure 4.2, the mesh-like topology layout of the memory words pops out (each of the squared blocks is associated with one of the words composing the memory array). Moreover, from here, it can be clearly seen the division of the LiM Array into two sections. The upper one, called the smart section, encloses all the memory word locations embedding both computing and storage elements, i.e. the Smart Blocks. They are the key at the foundations of the GP-LiMA paradigm since they are provided with a default scheme that makes them working like small general-purpose processing units. While the other section, called the standard section, comprises all the memory word locations only made of storage elements, which from now on will be referred to as Standard Blocks. They are used to enlarge the

data storage capacity of the LiM array while bounding the consequent increase in the array complexity. Thus, they act as possible sources for data operands on which the Smart Blocks can work. In particular, the elaboration inside the Smart Block can involve both data hold in the block itself and data coming from any of the blocks composing the LiM Array that are accessible thanks to the mentioned data communication network. Moreover, while the word contained in the smart blocks can be read and modified during the LiM processing, the memory locations constituting the standard section can be read by the Smart Blocks, while their writing can be performed only by the CPU accessing the GP-LiMA in the standard data memory mode, to initialize the LiM Array content before the algorithm starts.
Dealing with the routing network, it involves two kinds of interconnections: the Reduction Tree Interconnections and the Memory Interconnections. Looking at Figure 4.2, the orange and green arrows identify the row and column interconnections, respectively, that belong to the first cited type, while the pink arrows outline the MI that belongs to the other category. Each row interconnection enables the data transfer among the smart blocks belonging to the same row, while the single column interconnection stretches over the standard section too, allowing the smart blocks to take data from any of the blocks placed along the same column. Instead, the MI extends all over the LiM Array so that, through it, each Smart Block can retrieve the data it wants from any block. However, the two interconnection kinds do not stand out for the set of blocks they make communicating, but rather differ for how they distribute the data among the different blocks. The Rdt Int kind allows all the smart blocks attached to the same connection to take simultaneously a different data even if they are driven by the same instruction, while the MI type is used when blocks controlled by the same instruction all need to pick the same word coming from a specific block in the LiM Array. More details about the routing network functioning will be provided in subsection 4.2.2.
Furthermore, the new mesh-like arrangement of the blocks composing the LiM Array has a twofold utility. Besides accommodating a more dense and flexible interconnection network, it also allows to easily design a more precise enabling system for the Smart Blocks of the array. Unlike the PLiM Array where the smart section is divided at most into 4 macro sections that, during the execution of a given instruction, can be enabled one independently of the others, the GP-LiMA smart section is

provided with an enabling system that allows specifying more punctually the Smart Blocks that need to be enabled at a time without asking for the LiM programmer to explicitly list for each instruction of the program which are the blocks to be activated and which are not. This leads also to a reduced overhead in terms of wires needed to carry the information about the Smart Blocks enabling, and IMem size. In particular, this system is organized leveraging the grid shape of the LiM Array. It provides a number of enable signals equal to the sum between the columns number and the smart rows number (namely the number of the rows inside the smart section composed of Smart Blocks) and it associates each enable signal with a different Smart Blocks column or row, as shown in Figure 4.2. It follows that each Smart Block is connected to 2 enable signals, so, if one of them is not active, that Smart Block will be disabled. Thus, the Smart Blocks enabling is performed following a battleship game-like mechanism. In this way, the LiM programmer, for each instruction, has to specify the enabling state only for each row and column not for all the single Smart Blocks composing the array. Moreover, it is worth noting that this enabling system allows to activate even a single Smart Block at a time and its management is designed to be compatible with the M-SIMD computing model proper of the GP-LiMA paradigm.

So, concerning the implementation of the M-SIMD working mode, as already anticipated, this provides for integrating into the GP-LiMA Unit a number of instruction decoders equal to the number of different concurrent running instructions involved in the M-SIMD paradigm to be implemented on the GP-LiMA at hand. Then, the LiM Array is organized so that Smart Blocks belonging to the same row are always driven by the same instruction, while Smart Blocks placed in different smart rows can be controlled by different instructions. This is carried out by connecting to the same instruction decoder all the smart rows which always have to run simultaneously the same instruction, as illustrated in Figure 4.3.

Finally, since the GP-LiMA framework still remains a pre-fabrication model to produce different LiM devices, it grants some degrees of freedom for the overall LiM Matrix layout. Indeed, the hardware designer can choose the size of the LiM Array, fixing the number of columns, smart rows and standard rows (namely rows inside the standard section composed of Standard Blocks), and the memory parallelism. He can decide the maximum level of M-SIMD processing, i.e. the maximum number

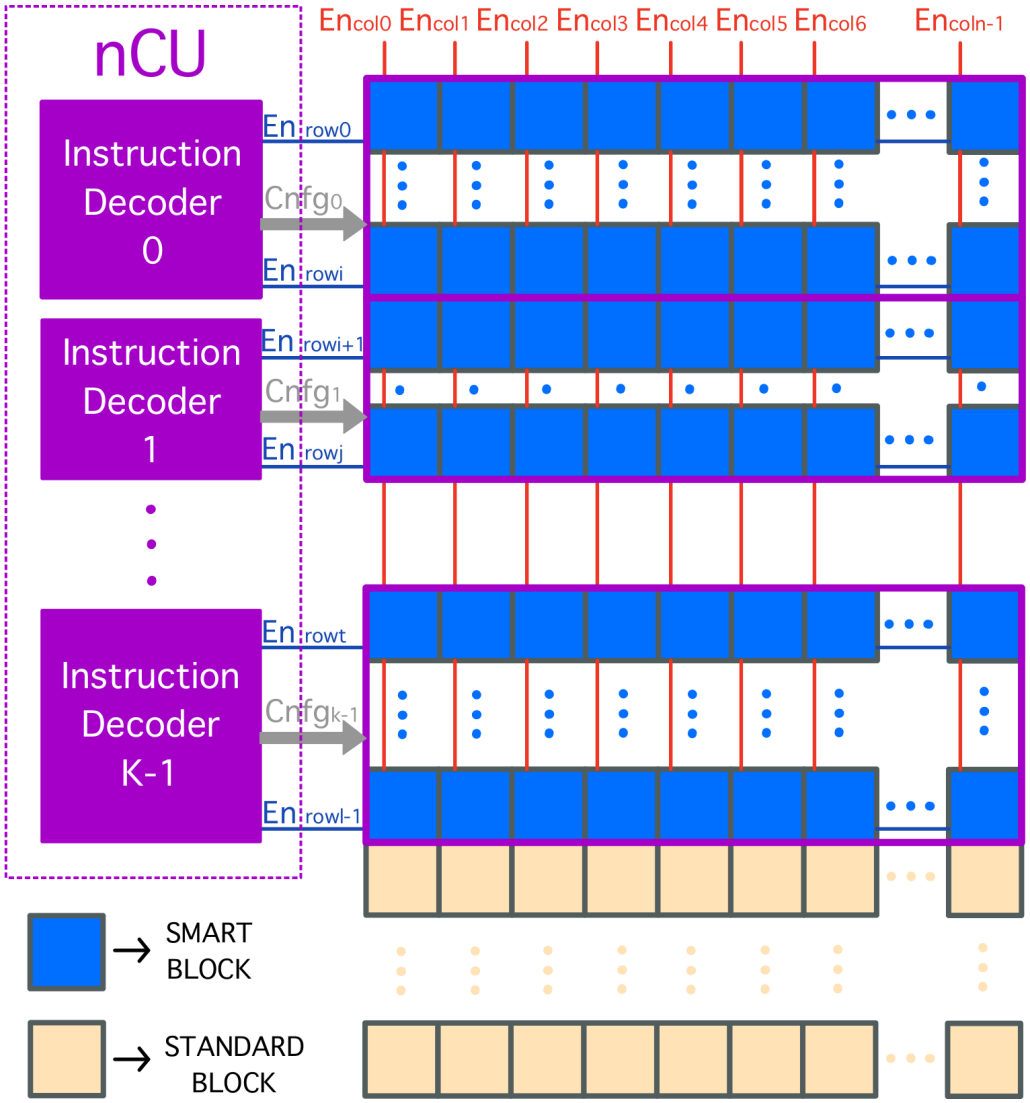


Figure 4.3: Enable signals arrangement for a generic M-SIMD LiM Array processing up to K different instructions simultaneously.

(K) of different instructions that can be run simultaneously, and how it is performed on the architecture. For each instruction decoder, he has to set the number of smart rows it drives and specify which are these. Note that the number of driven smart rows can change from one instruction decoder to the other, however, each of them can only control consecutive smart rows. It follows that, by connecting all the smart rows to the same instruction decoder before the device fabrication or specifying the same instruction in all the fields of the `VLIW_nInstruction` during the GP-LiMA

programming, the architecture computing mode can be easily reduced to a SIMD model. Similarly, the GP-LiMA can carry out a MIMD processing, if the hardware designer sets the GP-LiMA model parameters so that each instruction decoder controls a single row and all the rows are composed by one block only. However, in general, because of the structure of the instantiated interconnections, the optimal layouts for the LiM Array, in terms of the limited latency and power consumption, are the ones that try to equal the rows number with the columns one. Therefore, if a considerable set of Smart Blocks needs to be driven by the same instruction, it is worth distributing the blocks over multiple rows handled by the same instruction (same instruction decoder) rather than keeping all of them on the same line.
To know which are in practice the parameters the hardware designer has to set on the GP-LiMA model to generate the target GP-LiMA system, refer to section A.1 of the User's Manual.

4.2.1 Smart Block

Going down into the LiM Array, it stands out the basic block starting from which the whole GP-LiMA paradigm was engineered, i.e. the Smart Block detailed in Figure 4.4. The Smart Block identifies the unit component of the LiM Array that performs operations on the data it holds or on the data forwarded by the routing network taken from the other blocks of the array. Thus, it is a memory location holding a memory word, which, besides the standard storage capability, is provided with processing functionalities. However, the single Smart Block default composition is quite complex, involving lots of elements that go beyond the simple processing logic that is usually involved in standard LiM solutions. Indeed, the Smart Block emulates the behaviour of a small processor and it is what confers to the GP-LiMA the general-purpose-like processing capability.
Giving a quick overview of the Smart Block layout in Figure 4.4, the fundamental element is represented by the Block Word, that is where the word associated to that Smart Block is located and simple bitwise logic functions are computed. However, the Smart Block is also provided with further storage elements aimed at saving the temporary results produced during the LiM program execution. It includes a small Register File (RF), with two asynchronous read ports and one synchronous write

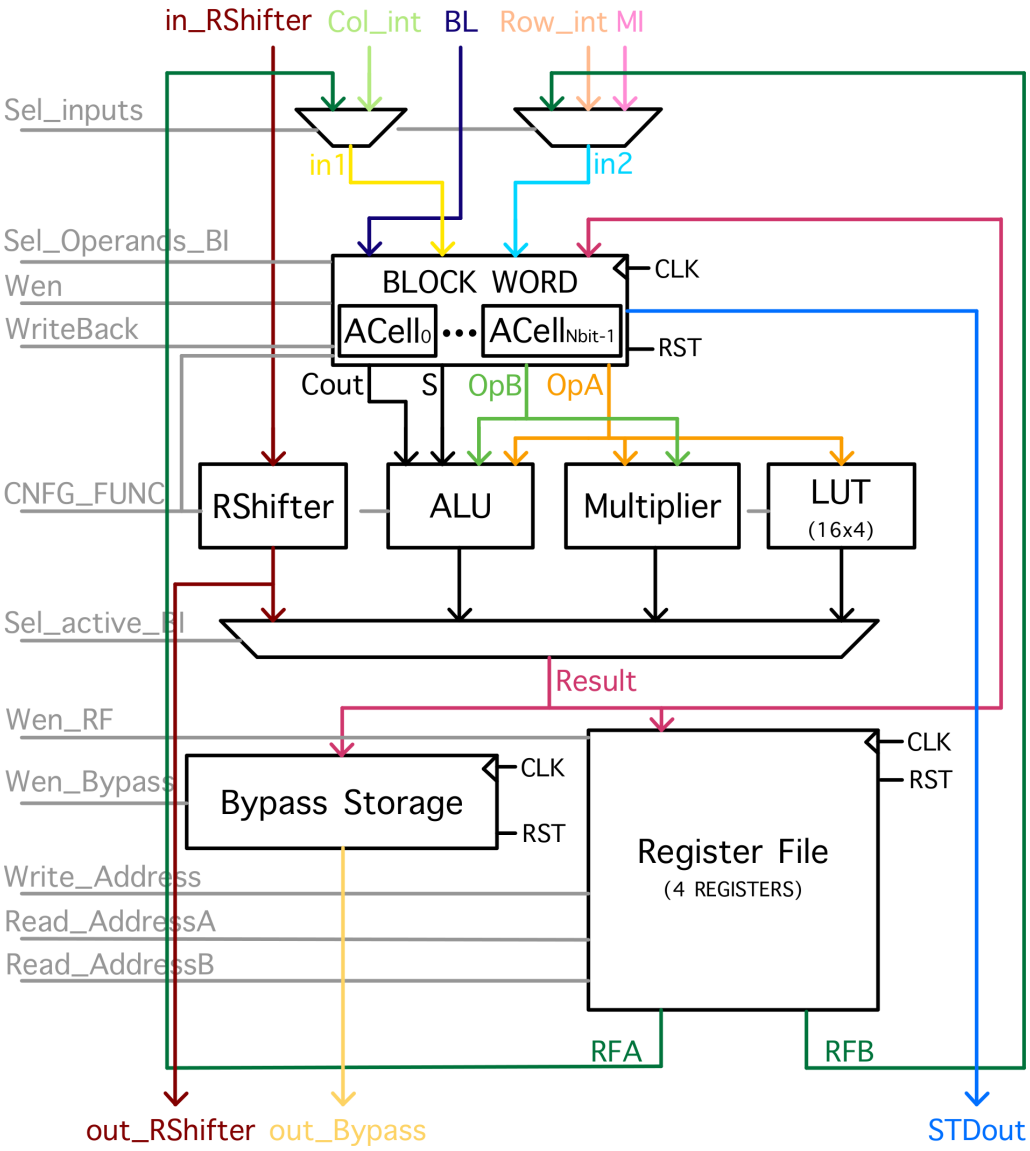


Figure 4.4: Smart Block scheme.

port and that comprises 4 registers in the default Smart Block version (refer to section A.1 to see how the RF size can be changed), and another single register called Bypass Storage. In particular, this last register, besides acting as a data saving component, is the provider for the data the Reduction Tree Interconnections have to pass to the other Smart Blocks. While, the more complex elaboration capabilities of the Smart Block are given by a set of blocks underneath the Block Word, which act

similarly to the Row Interfaces of the PLiM. All these blocks work in parallel taking as operands the data filtered by the Block Word, which, in turns, receives part of the data it handles from the output of two multiplexers connected to most of the input signals of the Smart Block. Therefore, in general, the Smart Block can operate on the data stored in the Block Word itself (`STDout`), on the ones coming from its RF (`RFA`, `RFB`), on the value forwarded by the Memory Interconnections (`MI`), and on the data brought by the row and the column interconnections (`Col_int`, `Row_int`). Also the content of its Bypass Storage can be elaborated. Nevertheless, to reduce the wires congestion inside the Smart Block, there is no a direct internal connection between the output of the Bypass Storage and the input multiplexers, so to retrieve that register value, it must be propagated through either the column or the row interconnections and then taken from the related signals `Col_int` or `Row_int`. Then, to know all the possible 2-operands combinations on which the Smart Block functions can run refer to **??**. While, summing up, all the possible destination components where the computation result can be sent are the Block Word itself, the RF and the Bypass Storage, and they are all driven by the same `Result` signal output by the middle multiplexer, which, for each instruction, properly select one of the signals coming out from the logic blocks implementing the more complex functions, placed underneath the Block Word.

Concerning this set of blocks, it represents the portion of the Smart Block that can be customized by the hardware designer. Apart from the ALU and the RShifter, that cannot be removed or modified, the designer can choose to insert the logic blocks he needs, or to remove the unused ones, to customize the generated GP-LiMA for the applications he wants. The insertion of a new component is rather an easy task, as, looking at Figure 4.4, it only requires connecting `OpB` and `OpA` signals to the block inputs and enlarging the multiplexer that returns the `Result` signal to make it receiving as further input the output of the new block. In the case the new block requires some configuration signals, it can directly exploit the ones already instantiated for the ALU. Similarly, also the removal of a component can be easily performed, since it only implies a few actions: erasing the component instance inside the Smart Block, reducing the `Result` multiplexer size of one signal less, and setting some constant values to adjust the new association between the multiplexer inputs and the configuration signal values. By the way, the details on how the user (HW

designer) can modify the GP-LiMA template to enter a new block, or take out an already instantiated block, are spelt out in the User's Manual at section A.2. However, as already anticipated, the GP-LiMA model comes with a default configuration for the logic units characterizing the Smart Block, which were chosen while keeping in mind the final goal of architecture programming generality. Figure 4.4 displays the predefined Smart Block layout which includes: the RShifter block, integrated to allow the execution of a small set of divisions, the ALU component, intended to perform basic arithmetic operations and comparisons functions, a 16x4 bits LUT, exploited to customize the GP-LiMA hardware platform even after fabrication, and the Multiplier, which performs the signed product between two data. In particular, in order to limit the Smart Block complexity, the multiplier instantiated takes as input signals only the lower half of `OpA` and `OpB` in order to provide a final result on a number of bits equal to the memory parallelism (that matches OpA and OpB width). However, this design choice requires the LiM programmer to comply with a constraint to prevent wrong results occurrences, i.e. the data values to be multiplied must be representable on a number of bits that is half the memory parallelism.

In the following, the Smart Block components which deserve to be addressed more carefully are investigated, such as: the Block Word, the ALU, the RShifter, and the LUT.

Block Word & Arithmetic Cell & ALU

The core of the Smart Block is represented by the Block Word that borrows the idea of the Row Word inside the PLiM Smart Row. It encloses a given number of the most elementary LiM units, namely the 1-bit memory cells embedding simple logic gates aimed at processing the lowest level operations. As for the PLiM model, these special cells are called Arithmetic Cells (ACells) and their number inside a single Block Word matches the LiM Array parallelism. The composition of a single ACell is outlined in Figure 4.5. The memory feature of the ACell is embodied by the Storage Unit, while the processing capability is implemented through a Full Adder followed by an `XOR` gate.
The Storage Unit can load either the value produced by the current instruction

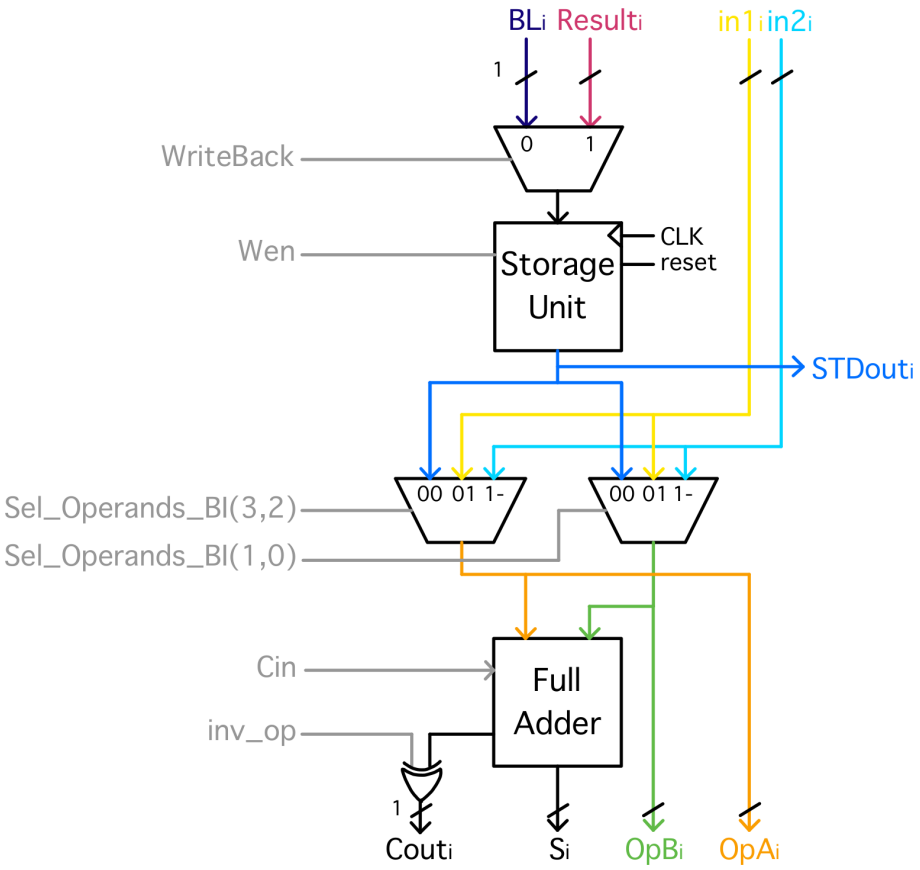


Figure 4.5: ACell scheme.

performed by the Smart Block, through the $\texttt{Result}_i$ signal (GP-LiMA activated in processing mode), or the data directly coming from the CPU carried by the $\texttt{BL}_i$ signal (standard data memory GP-LiMA usage). Then, the Storage Unit content is both forwarded to the output of the ACell through the $\texttt{STDout}_i$ signal, to allow the CPU to read the stored data whenever it wants, and propagated inside the ACell, to accomplish the data processing inside the memory.

The Full Adder can operate logical bitwise functions on the data selected by the two middle multiplexers (see Figure 4.5), which take the memory content ($\texttt{STDout}_i$) and two input signals ($\texttt{in1}_i$ and $\texttt{in2}_i$) that come from the previous selection at the input of the Smart Block. While the multiplexers at the Smart Block input are used to perform a preliminary selection of the data to be elaborated by the Smart Block, the two multiplexers inside the ACell are used to finalise this choice and to define

the order with which these operands must be considered, namely which is the first term ($\texttt{OpA}_\texttt{i}$) and which is the second one ($\texttt{OpB}_\texttt{i}$) of a generic operation (particularly useful for the subtraction where the order of the operands changes the operation outcome). Thanks to these multiplexers, lots of possible operands combinations can be specified through a LiM instruction (see **??**). Moreover, as already anticipated, the selected data are sent to the logic blocks below the Block Word through the `OpA` and `OpB` signals. Then, the ACell outputs other two signals direct entering the ALU block, i.e. $\texttt{Cout}_\texttt{i}$ and $\texttt{S}_\texttt{i}$. According to the values of the control signals `Cin` and `inv_op` and the data selection performed by the multiplexers, $\texttt{S}_\texttt{i}$ can carry the result of a `XOR` or a `NOR` operation, while $\texttt{Cout}_\texttt{i}$ can forward the outcome of the `AND`, the `NAND`, the `OR`, the `NOR` or the `NOT` operation. Moreover, the FA can also be handled so that at the $\texttt{Cout}_\texttt{i}$ output the value of one of the selected data is reported without changes (used in case of a Load operation, see Table 4.2). Specifically, in this case, both multiplexers select the same value for $\texttt{OpA}_\texttt{i}$ and $\texttt{OpB}_\texttt{i}$ and the selection signals are driven so that $\texttt{Cout}_\texttt{i}$ delivers the `AND` between the two signals.
Furthermore, the Block Word must be necessarily accompanied with the ALU block, which both completes the set of provided arithmetic and logic functions and forwards to the final Smart Block multiplexer the outcome of the Block Word when required. Looking at Figure 4.6 the schematic of the ALU can be viewed. It takes as inputs the *Nbit* `OpA` and `OpB` signals coming from the Block Word and it can be divided into 3 macro-sections. The one on the upper left, comprising one signed adder, one signed subtractor and a 2-inputs multiplexer, is used to compute either the sum, the subtraction or the absolute value function. While, the section on the upper right, composed of a set of logic gates and a 4-inputs multiplexer, implements all the comparison functions ($>$, $<$, $=$, $\neq$). Finally the lower section, made up of 3 2-input multiplexers, performs the selection of the ALU output signal, which can assumes either the outcome of one of the functions computed by the ALU logic or the value of one of the signals produced by the Block Word (`S` or `Cout`). In Table 4.1 there is summarized the set of all the executable functions, given by the combinations of the operations implemented by the Block Word and the ones achieved by the ALU, together with the correspondent values for the configuration signals.

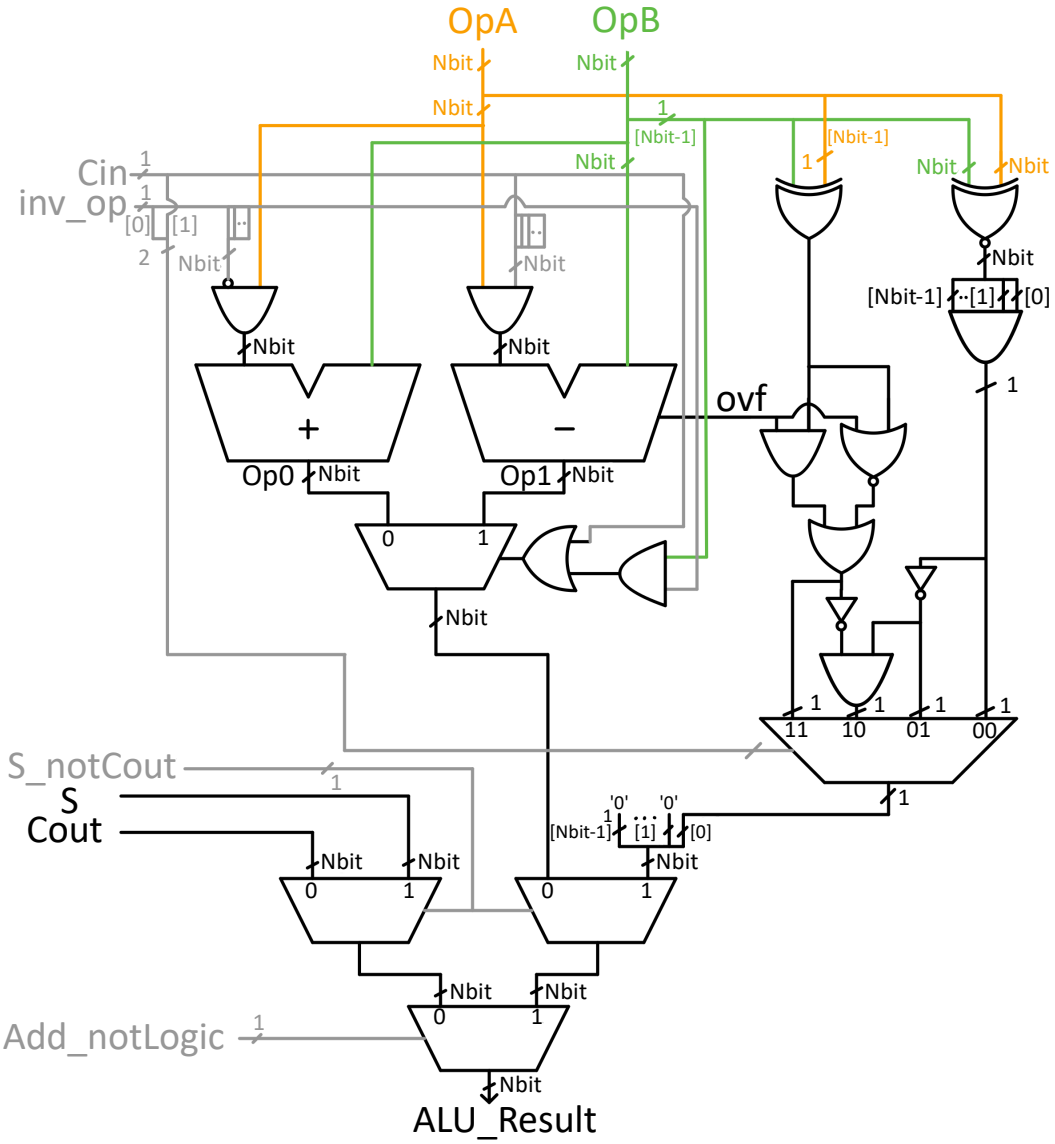


Figure 4.6: ALU scheme.

ACell & ALU Operations				
Configuration signals (CNFG_FUNC[3-0])				***Operation***
Add_notLogic	Cin	inv_op	S_notCout	Result = OpA OP OpB
0	0	0	0	Result = OpA AND OpB
0	0	0	1	Result = OpA XOR OpB
0	0	1	0	Result = OpA NAND OpB
0	0	1	1	Result = OpA XOR OpB
0	1	0	0	Result = OpA OR OpB
0	1	0	1	Result = OpA XNOR OpB
0	1	1	0	Result = OpA NOR OpB
0	1	1	1	Result = OpA XNOR OpB
1	0	0	0	Result = OpA + OpB
1	0	0	1	Result = 1 if OpA = OpB 0 if OpA ≠ OpB
1	0	1	0	Result = |OpB|
1	0	1	1	Result = 1 if OpA ≠ OpB 0 if OpA = OpB
1	1	0	0	Result = OpA - OpB
1	1	0	1	Result = 1 if OpA > OpB 0 if OpA ≤ OpB
1	1	1	0	Result = OpA - OpB
1	1	1	1	Result = 1 if OpA < OpB 0 if OpA ≥ OpB

Table 4.1: List of the executable functions, implemented by the ALU block and the ACells, together with the related values for the configuration signals. *Result* refers to the signal at the output of the ALU block (see Figure 4.4).

LUT

In the default layout of the Smart Block, one of the logic blocks inserted below the Block Word is the LUT block. It belongs to the kind of blocks that can be either removed or modified by the hardware designer before the fabrication phase (refer to section A.1 to see how the LUT size can be changed). It is essentially a memory composed of 16 locations of 4 bits each, which is addressed by a 4-bit input signal. This component allows the user to implement in hardware a customized function even after the whole GP-LiMA unit has been manufactured, by exploiting the CwM

approach. At compile-time, the content of all the LUTs inside the LiM Array Smart Blocks can be initialized to fulfil the implementation of the wanted function, as explained in chapter 2.
In particular, even if the entire Nbit `OpA` signal enters the LUT component, only the lower 4 bits of the `OpA` signal are connected to the four 1-bit input values of the LUT instance. Then, the returned 4-bit data is extended to reach the width of the data handled by the Smart Block and is sent to the Smart Block multiplexer as the LUT block outcome. Moreover, the LUT component takes also a 1-bit configuration signal that is used to choose the kind of extension for the output data, which can be signed or unsigned.
Furthermore, all the LUTs are provided with a mechanism for their initialization, which allows to specify the content value for each single LUT location, independently of the others. It means that different Smart Blocks can include LUTs holding different values and so performing different functions. In this way, when the instruction involving the LUT usage is called, even Blocks driven by the same control signals (same instruction decoder) can simultaneously execute different operations. The initialization mechanism leverages the daisy chain connection. It allows to write all the LUTs locations without incurring in complexity overhead. All the LUTs inside the whole LiM Array are connected sequentially into a chain so that the output of a LUT represents the input of the following one. An example for the chain organization is illustrated in Figure 4.7. Each LUT is implemented through a set of registers, each having the input signal connected to the output of the register below. Then, the LUTs writing occurs following a sequential procedure, where at each clock cycle a new data to be written in the LUTs is entered inside the chain. Thus, to initialize all the LUTs, only two input external pins for the GP-LiMA are needed: the `Input_LUT_Daisy_Chain` signal that is used to insert a new data at each clock cycle, and the `En_LUT_Daisy_Chain` signal that enables the LUTs writing. As long as the `En_LUT_Daisy_Chain` signal is at 1, each LUTs register updates its content with the value coming from the previous register. It follow that, to initialize the entire set of LUTs inside the LiM Array, the number of clock cycles needed is equal to the product between the number of Smart Blocks and the number of locations inside a single LUT. In the design, the chain starts from the last location of the LUT contained in the last Smart Block and ends in the first LUT register of the

first Smart Block. In this way the first data to be entered inside the chain must be the content of the first location of the first LUT, then, following the order, the last data input must be the one to be associated with the last register of the last Smart Block LUT.

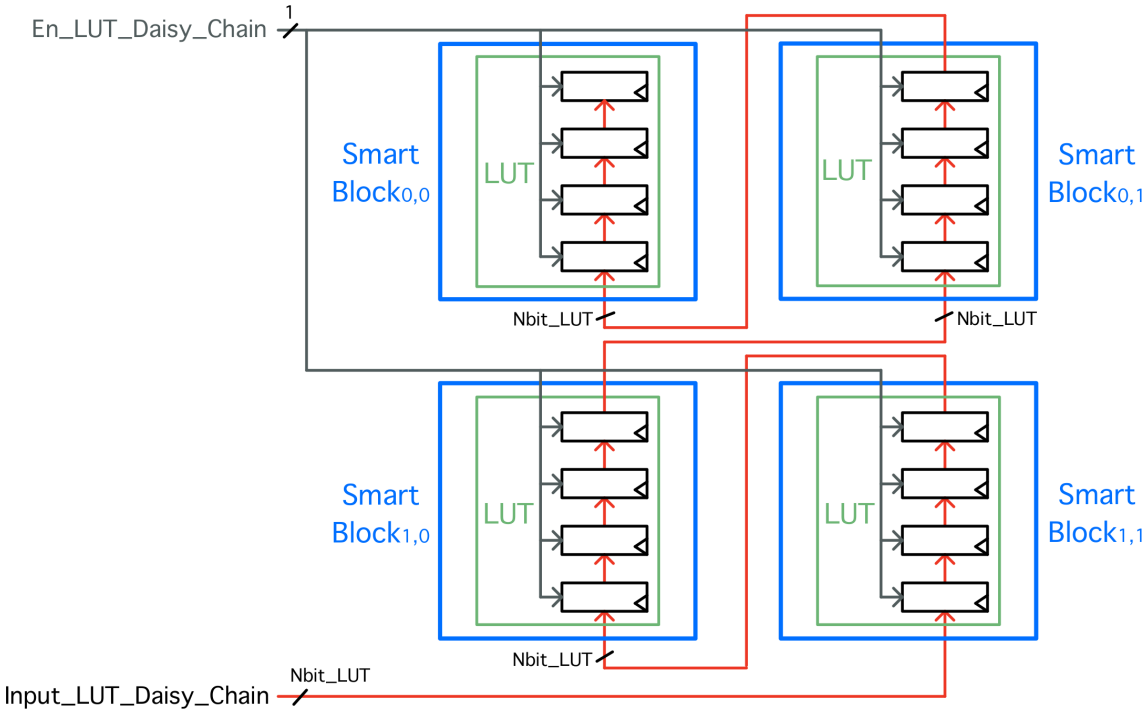


Figure 4.7: Example of the LUT daisy chain connection for a smart section composed of 2 columns and 2 rows. The chain passes through the LUTs in all the Smart Blocks.

RShifter

Although the RShifter is introduced as one of the logic blocks placed below the Block Word, it differs from the other processing components since the data it elaborates is not released by the Block Word but is directly gathered from the set of the Smart Block inputs. The aim of this component is to assure the LiM Array with some division processing capabilities without inserting a real divisor in each Smart Block which would certainly lead to a massive overhead in terms of area and power consumption. However, even including a programmable right shifter component, to implement divisions by numbers that are powers of 2, was considered an excessive

waste of resources for the default version of the Smart Block. So, the RShifter inside each Smart Block is a very slight component, made up of only a set of parallel wires, able to perform 1-bit right shift of the input data, programmable with the signed or unsigned extension. However, to implement a wider set of possible divisions all the single RShifters belonging to the Smart Blocks in the same row are connected in sequence, creating a chain of shifts as depicted in Figure 4.8.

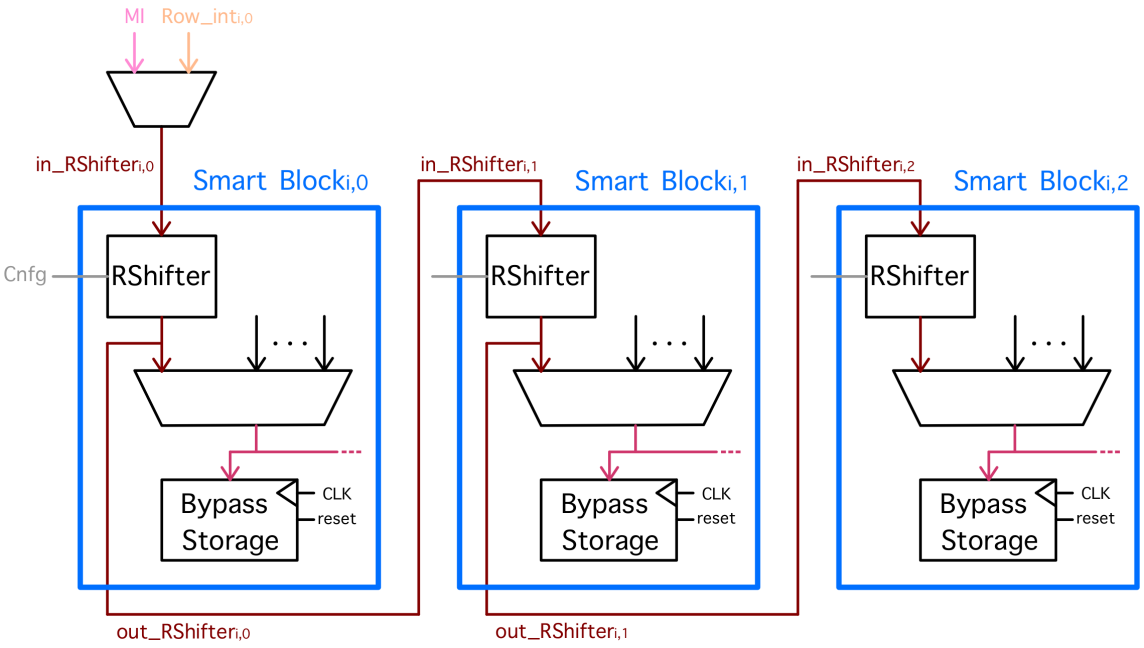


Figure 4.8: Chain Connection among the RShifter components belonging to the Smart Blocks of the same row. Example for a generic smart row composed of 3 Smart Block.

So, a part from the RShifters embedded in the Smart Blocks at the extremes of a row, all the other RShifters take as input the signal coming from the RShifter on their left and return the shifted version of it to the RShifter on their right. At the same time, for each Smart Block, the signal produced by the integrated RShifter goes in input to the multiplexer that, according to the current issued instruction, chooses the data to be stored in one of the possible destinations. It follows that, each time an instruction involving the RShifter is called, all the multiplexers in the same smart row select as value for the `Result` signal the outcome from their RShifter, so the first Smart Block will store the value of the specified source operand divided by 2, the second Smart

Block the same value divided by 4, the third Smart Block the value divided by 8 and so on till the end of the row. In this way the programmer can specify the kind of division he wants to operate by simply enabling the Smart Block associated to that division type. Then, the data that each smart row elaborates can derive from either the MI or the row interconnection, as the RShifter input of the first Smart Block is output by a multiplexer connected to these two routing networks. It means that the division operation is handled at the smart row granularity level instead of at the Smart Block one, namely, the Smart Blocks in the same row all work on the same data instead of on a different data each. Moreover, the possibility to select as data to be shifted the one coming from the row interconnections allows smart rows driven by the same instruction to elaborate simultaneously different data. Thus, when the same RShifter operation is sent to all the smart rows, coupled with the source operand coming from the row interconnections, the number of divisions on a different data performed in the same clock cycle is equal to the number of smart rows composing the LiM Array. In addition, for each of these data, a set of divisions is directly computed in parallel. The number of available divisions for each data equals the number of Smart Blocks included in a row and the set of provided divisors values is given by all the numbers that are a power of 2 in increasing order, starting from 2 up to $2^{\#\text{LiM Array Columns}}$. However, it is worth highlighting that the values of the divisors can only be constants known at programming time. It means that a given data cannot be divided by the value of a data computed during the previous LiM instructions. Nevertheless, when this kind of operation is strongly required by a specific application, the hardware designer can still modify the Smart Block composition, inserting a dedicated component before the GP-LiMA realization.

4.2.2 GP-LiMA Interconnections

The programmable routing network that branches throughout the whole LiM Array constitutes one of the turning points of the GP-LiMA model for its overall programming flexibility. It leverages two different kinds of interconnections which differ on the type of instructions that allow performing:

- the **Memory Interconnection** (MI), which provides to all the attached Smart Blocks the exact same data;

- the **Reduction Tree Interconnection** (Rdt Int), which forwards at the same time a different data for each of the Smart Blocks it is connected to.

In Figure 4.9 an example for the LiM Matrix arrangement is detailed. Here, there

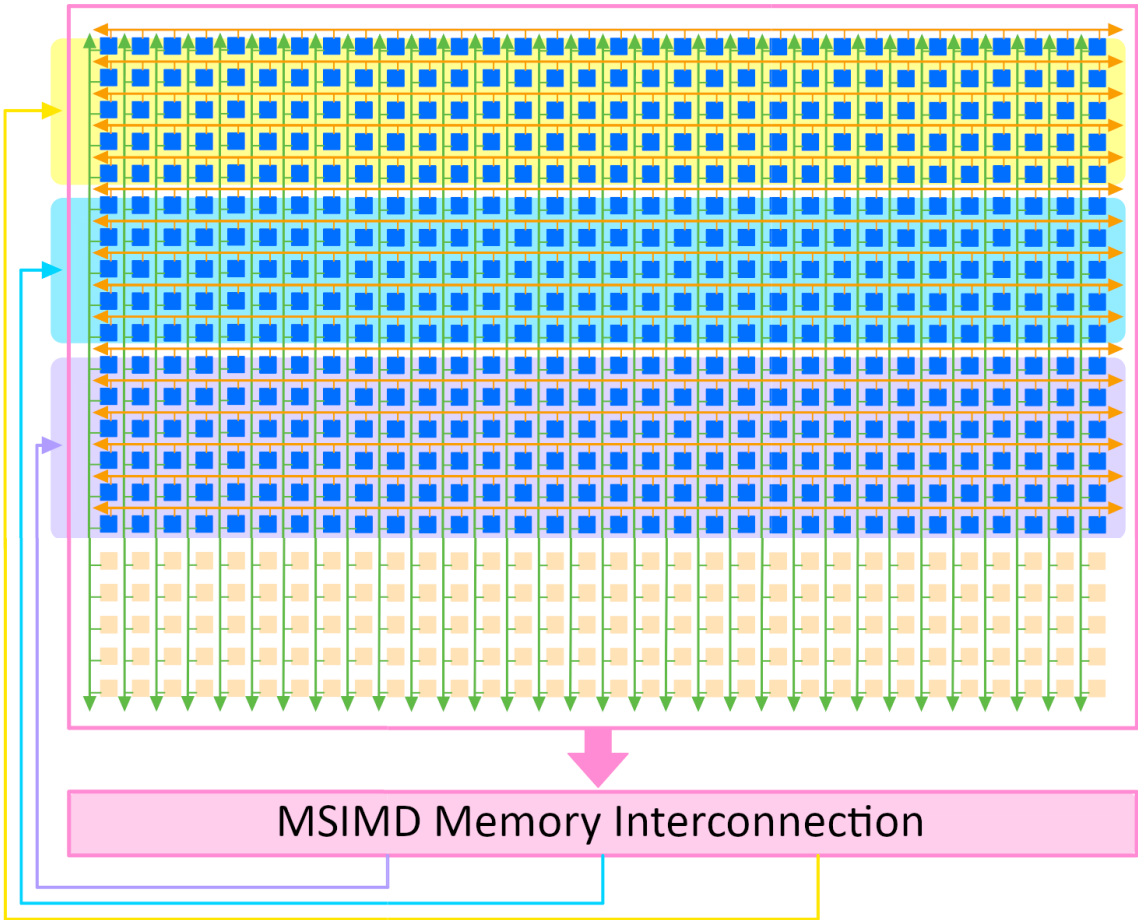


Figure 4.9: Interconnections filling a LiM Array composed of 32 columns, 16 smart rows and 5 standard rows, working in the M-SIMD mode with K = 3. The orange and green arrows represent the reduction tree interconnections, in particular, the row and the column ones, respectively. The M-SIMD MI outputs K signals, each going inside all the smart blocks of one of the 3 sub-sections that is driven by a different nInstruction.

are outlined all the interconnections filling a LiM Array that is composed of 32 columns, each enclosing 16 Smart Blocks and 5 Standard Blocks, and that can be programmed to perform up to 3 different instructions simultaneously. All the horizontal orange arrows and the green vertical ones represent the connections among

the blocks in the same row or in the same column, respectively, and are implemented through the reduction tree type. While, the pink, the yellow, the light blue and the violet arrows compose the M-SIMD Memory Interconnection which extents across the whole LiM Array. From Figure 4.9, it can be already seen that there is a correlation between the degree of the involved M-SIMD computing mode and the interconnections structure, as it directly determines the number of signals output by the Memory Interconnection. The Smart Blocks driven by the same instruction (instruction decoder) are all connected to the same data signal coming from the M-SIMD MI.

Memory Interconnections

The M-SIMD Memory Interconnections intent is to enable the Smart Blocks to take one data from any of the Blocks inside the LiM Array, without constraints on its placement. To do this, the generic memory interconnection has to take the content of all the Blocks, namely both the values inside the Block Words of all the Smart Blocks and all the data stored by the Standard Blocks, and select one single value. For this reason, the straightforward implementation for this entity is a massive multiplexer with a number of inputs equal to the number of Blocks composing the LiM Array, each on a width matching the memory parallelism. Nevertheless, since the GP-LiMA works following the M-SIMD model, different instructions could access the same MI specifying different block addresses, leading to conflicts. Therefore, to cope with this issue, the M-SIMD MI is composed of a set of multiplexers, each one representing a different SIMD MI and dedicated to one of the simultaneous instructions. In Figure 4.10 an example for the connection of the M-SIMD MI to a small LiM Array is shown together with the insight of the internal interconnection composition. The depicted LiM Matrix can perform up to 2 different instructions at the same time, each carried out by one different smart row composed of 2 Smart Blocks. Then the LiM Array includes also one standard row of 2 blocks. As expected, the MSIMD MI comprises two SIMD MIs each driven by a different instruction decoder. The SIMD MIs is identified by a multiplexer taking 6 Nbit inputs and returning one Nbit signal that is then brought at the input of all the Smart Blocks driven by the same instruction that controls that multiplexer.

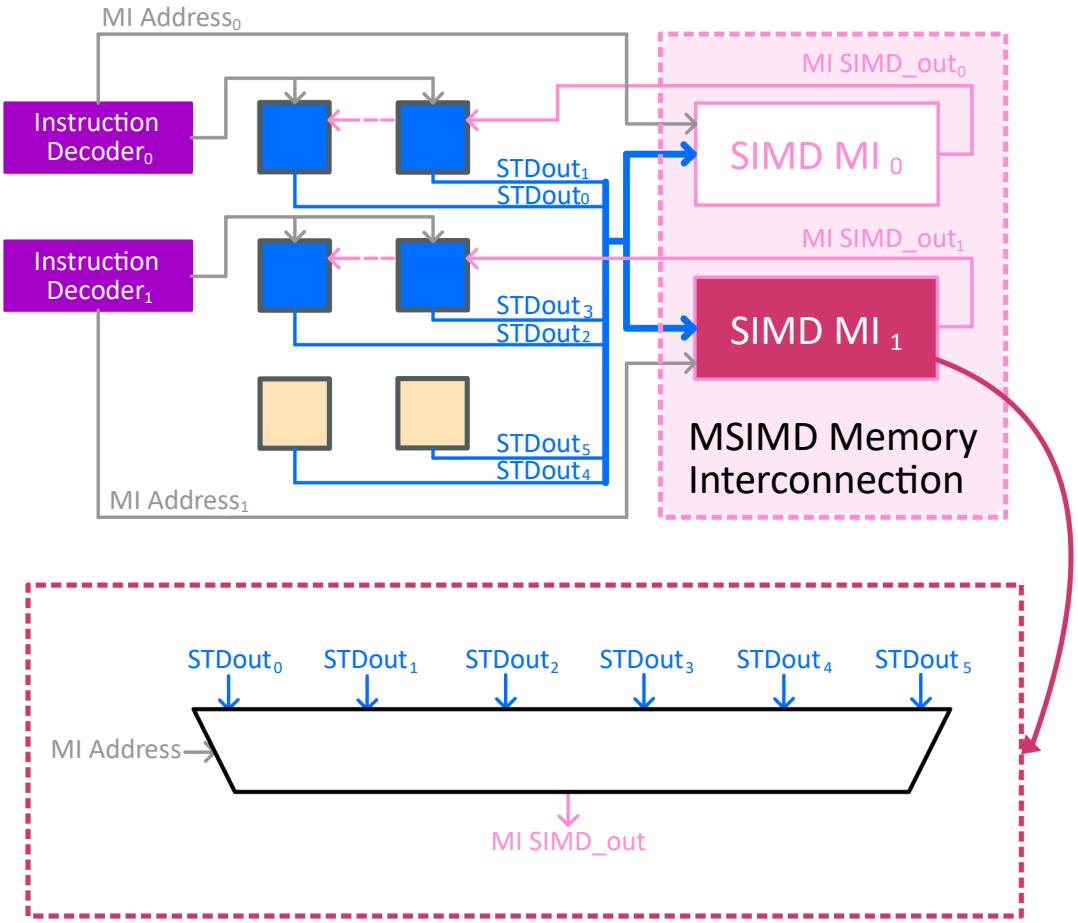


Figure 4.10: On the left, it is reported an example for the M-SIMD MI on a LiM Array composed of 2 columns, 2 smart rows and 1 standard row, working in the M-SIMD mode with K = 2. On the right, it is shown the basic implementation for the single SIMD MI block making up the M-SIMD MI in the example.

Summarizing, the M-SIMD MI is used each time the Smart Blocks driven by the same instruction have to elaborate simultaneously the same data, which can be specified inserting the address of the LiM Array location that stores it.
However since usually the LiM Array is composed of a significant number of blocks, the resulting multiplexers could become very complex components, determining the

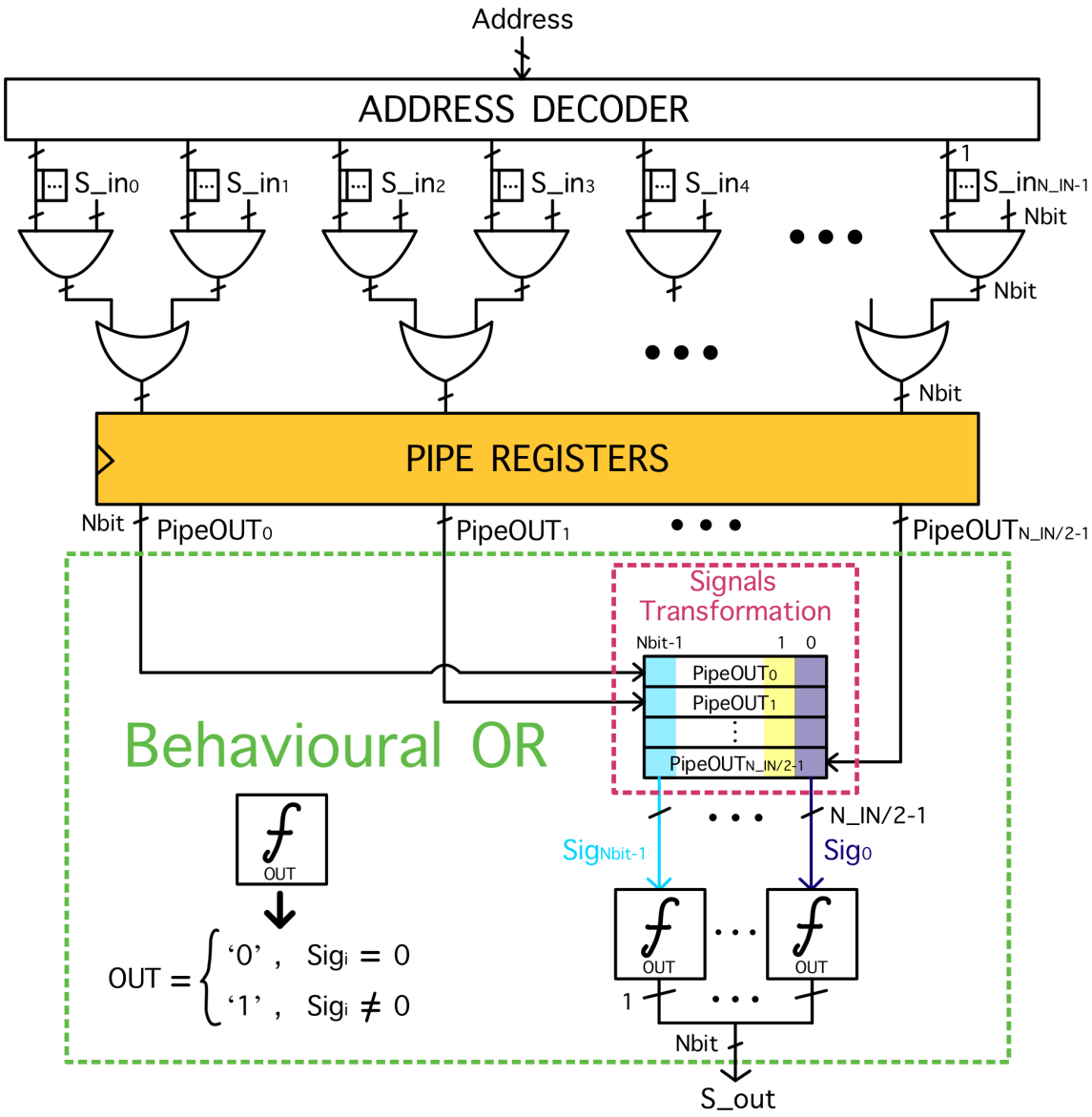


Figure 4.11: Pipelined implementation of a multiplexer on N_IN input signals each on Nbit. This kind of implementation matches the one of both the SIMD MI block and the GP-LiMA Unit output multiplexer that returns the `LiM_WORD` signal to the CPU.

GP-LiMA performance limiting factor, especially in terms of maximum allowed operating frequency. For this reason, the final implementation for each N_Blocks inputs multiplexer instantiated in the GP-LiMA model consists of a pipelined version, where the critical path is split in two. Figure 4.11 outlines the new multiplexer

structure embedding pipe registers in the middle. Note that this pipelined structure is exploited to implement also the GP-LiMA output multiplexer providing the read data signal (`LiM_WORD`) to the CPU.
Lastly, it is worth mentioning a possible issue that can arise while dealing with this pipelined version of the MI. Because of the registers insertion, a data hazard can occur when an instruction uses this interconnection to access a data that was updated in the previous instruction. When this happens, the updated value of the data is firstly saved inside the interconnections pipe registers, meaning that it is not present at the output of the multiplexer when the Smart Blocks need to fetch that value for the required elaboration. Thus, to prevent wrong data elaborations, another instruction or a null operation must be inserted between the instruction that updates the data value and the one that wants to elaborate the updated value.

Reduction-Tree like Interconnections

The programmable Reduction Tree Interconnections kind represents the item of the GP-LIMA paradigm which strongly impacts on its programming generality attribute. The employment of these kind of interconnections is particularly thought of for both facilitate the moving of large data blocks inside the LiM Array and speeding up the sequential portions of common algorithms, such as sequential sums or maximum/minimum searching functions to be performed on high number of data. For the second goal, the intent is to exploit the high level of the LiM Matrix processing parallelism to implement the sequential functions following a reduction tree-like organization that can reduce the total number of LiM instructions encoding that type of computations.
In Figure 4.12 a black box scheme of the Rdt Int connecting 8 Smart Blocks is shown. The main requirement for this kind of interconnections, to help carrying out the reduction tree like processing mechanism, is to be able to take N data, one from each of the attached blocks, and to distribute them at the inputs of the same blocks in a different order, forwarding to each block a different data. To explain the kind of data transfer implemented among the Smart Blocks linked to the same Rdt Int, in Figure 4.13 an example of a sequential sum computation on 8 data accomplished following a reduction tree-like execution is reported.

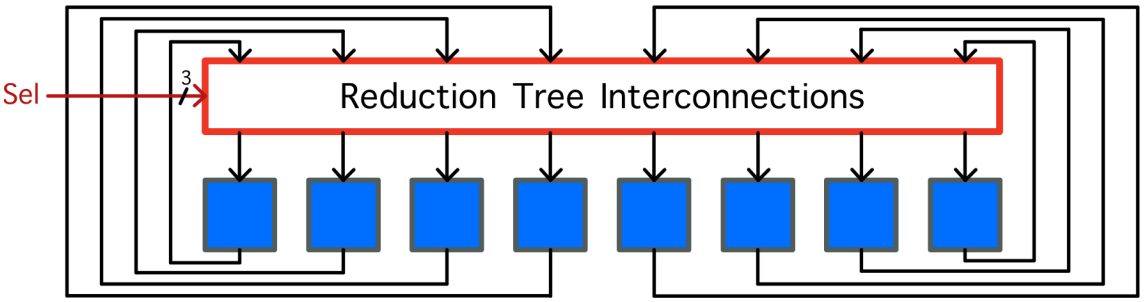


Figure 4.12: Black box scheme of a reduction tree interconnections among 8 blocks.

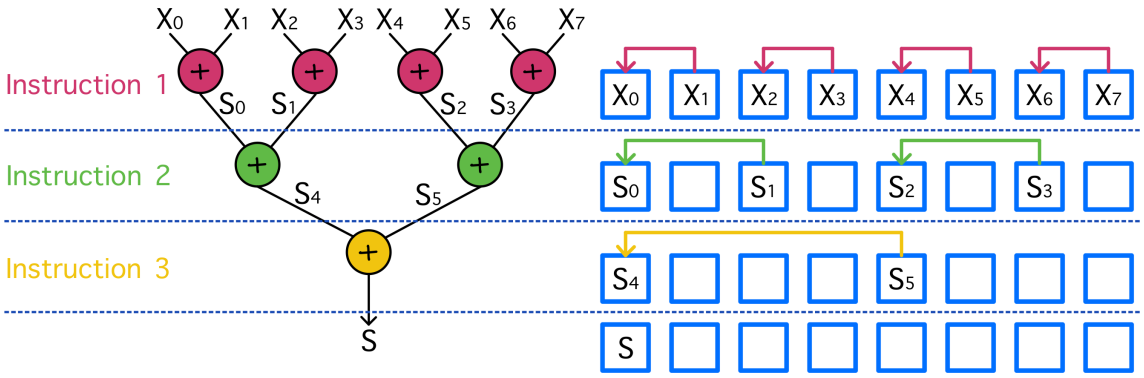


Figure 4.13: Usage example of the reduction tree interconnection for computing a sum among 8 data, each contained in one of the 8 smart blocks attached to the same interconnection.

Before the sum starts, all the 8 data must be saved inside the Bypass Storages of the 8 Smart Blocks. During the first instruction, 4 sums are computed in parallel each on a different couple of the starting 8 data. To do this, all the Smart Blocks at the even positions (namely the one containing the X_0 data, the one storing the X_2, the one holding the X_4 and the one containing X_6) are enabled to take the content of the Smart Block that is on their right, sum it to the content of their Bypass Storage and updating the same Bypass Storage with the sum result. Then, during the second instruction two parallel sums are evaluated, by enabling only the first and the fifth Smart Blocks to sum their Bypass Storage content with the value stored in the Smart Block that is two positions right. Again the operation results are used to overwrite the linked Bypass Storages content. Lastly, the third instruction activates only the first Smart Block that sums its value with the one coming from the fifth Smart Block. In this way the whole 8 data sum operation is performed into

3 instructions, each taking one clock cycles. It follows that for a generic sum on N data the total number of clock cycles involved is equal to $\log_2$N instead of N-1 as it would be following the standard execution of a sequential algorithm, namely one sum on 2 data for each clock cycle.
As already anticipated, to guarantee this kind of data elaboration, the reduction tree interconnection kind is used to implement all the row and column interconnections instantiated inside the LiM Matrix. Each of them retrieves the data from the Bypass Storages inside the Smart Blocks to which it is connected and brings back them in a shifted order to the inputs of the same Smart Blocks.

- **Basic implementation for the Column interconnections**

  The most straightforward way to implement the Rdt Int accomplishing the mentioned data transfer is through the use of multiplexers. As shown in Figure 4.14, the single Rdt Int component is made up of a set of multiplexers, each driving the input of a different Smart Block. Then, all the embedded

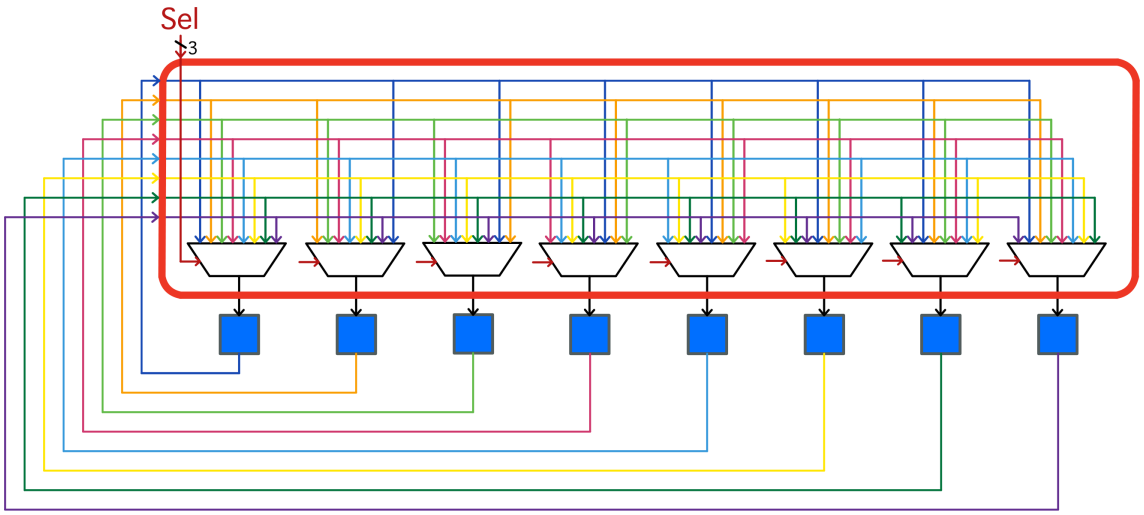


Figure 4.14: Example for the basic implementation of a reduction tree interconnection among 8 blocks. This kind of implementation matches the column interconnections one. Note that, for each multiplexer, the first input signal from the left is selected when `Sel` = 0, the second when `Sel` = 1, and so on up to the last signal that is selected when `Sel` = 7.

  multiplexers receive in input the signals output by all the Smart Blocks that

the same Rdt Int feeds. It follows that a generic Rdt Int interconnecting a line of N Smart Blocks is composed of N multiplexers each on N inputs of Nbit each, where, in this case, Nbit is the LiM array parallelism. In general, each multiplexer could be potentially driven by a different selection signal, since, for each multiplexer, the control signal is generated by the same instruction decoder that drives the Smart Block to which that multiplexer is attached. This organization takes into account the case in which the Smart Blocks attached to the same Rdt Int are not driven by the same instruction, i.e. for the column interconnections involved in the GP-LiMA Unit with a M-SIMD processing mode degree greater than 1. The association between each of the N Rdt Int input signals and the position of the related multiplexer entry is different for all the multiplexers. In particular, for each multiplexer, the value of the selection signal matches the number of distance positions towards the right at which the Smart Block from where the data must be taken is placed with respect to the Smart Block that the multiplexer drives. It means that when the selection signal equals 0 the multiplexer outputs the value coming from the same Smart Block it feeds, while if it equals 1 the output of the Smart Block on the right of the Smart Block that multiplexer drives is selected and so on. In the case in which there are no more Smart Blocks left on the right of the driven Smart Block the next output chosen is the one of the first Smart Block of the line and so on in a circular way. Note that for the column interconnections the same organization is carried on addressing the outputs of the Smart Blocks below the considered one rather then the ones on the right.
However, this Rdt Int implementation results in a significant complexity overhead that grows linearly with the memory parallelism and quadratically with the number of Blocks that are attached to the interconnection.
Thus, to try limiting the complexity introduced in the design only for the row interconnections an optimized implementation for the Rdt Int is used, illustrated in Figure 4.15.

- **Optimized implementation for the Row interconnections**

  A special case for the Rdt Int is represented by the row interconnections,

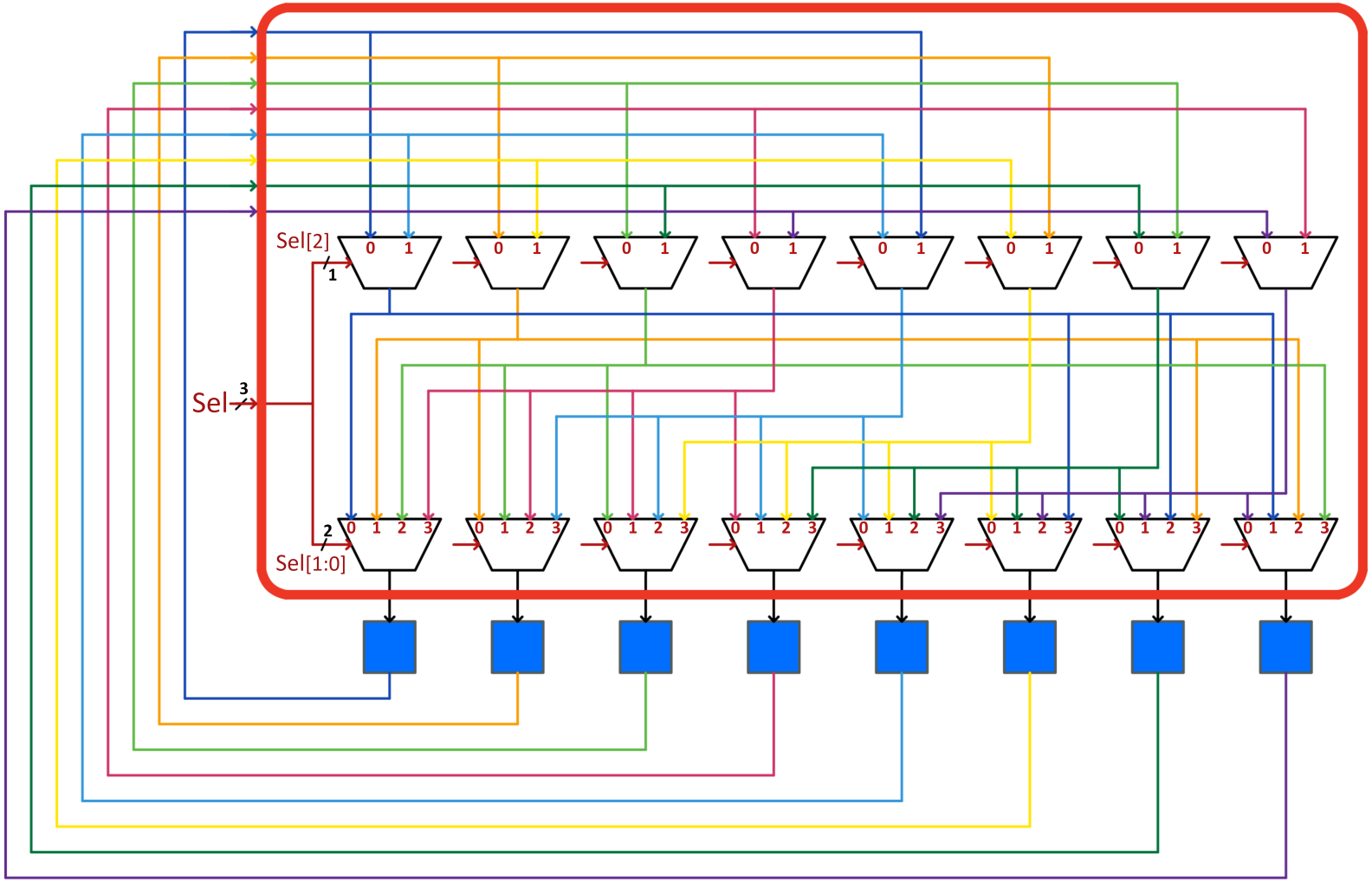


Figure 4.15: Example for the optimized implementation of a reduction tree interconnection among 8 blocks. This kind of implementation matches the row interconnections one.

since the Smart Blocks attached to each Rdt Int are always driven by the same instruction. A special case for the Rdt Int is represented by the row interconnections, since the Smart Blocks attached to each Rdt Int are always driven by the same instruction. In this case the Rdt Int acts as a programmable circular left shifter that takes a big data composed of the concatenation of the outputs of all the Smart Blocks to be interconnected and returns this data shifted of a number of positions equal to the value assumed by the configuration signal. This behaviour allows to develop the single row interconnection on two level of multiplexers. Each level includes a number of multiplexers equal to the number of Smart Blocks to be interconnected, but each characterized by a lower number of inputs. The first level carries out a coarse grained shift of the composed data input, while the second level refines the final data making it matching the actual number of shifted demanded through the interconnections configuration signal. Looking at the row interconnection example reported in Figure 4.15, the first level of interconnections shifts the data either of 0 or 4 positions, while the second level can shift the data output by the first level either of 0, 1, 2 or 3 positions. Thus, for example, if the selection signal equals 6, the first level of multiplexers performs a shift of 4 positions, while the second level a shift of 2.

This kind of Rdt Int implementation allows to reduce the overall introduced complexity that results to be half the complexity of the previously shown basic implementation. However, the more the dimension of the multiplexers belonging to the different levels is similar the more the reduction in terms of complexity is. Note that the GP-LiMA design model is organized so that the hardware design can tune the division among the number of shifts performed by both levels.

Ultimately, it is worth to highlight that, due to the excessive length of the involved critical path, the outputs of all the row and column interconnections immersed in the LiM Array are followed by pipeline registers. This means that, to avoid data hazard issues like the ones encountered for the MI, between the instruction that writes in the Bypass Storages of a group of Smart Rows and the one that wants to read the content of the same set of Bypass Storages another generic instruction or a null one

must be inserted. It follows that, for the execution of the sequential operations that exploit the reduction tree-like mechanism, as the sum among N data, the actual number of instructions (or clock cycles) needed is equal to $2*\log_2 N-1$.

4.3 GP-LiMA Control

Up to now, the internal composition of the LiM Matrix was thoroughly investigated, highlighting its processing potentialities and how they can be exploited to efficiently run a generic algorithm. However, almost nothing was told about how the GP-LiMA manages to perform a whole LiM program. Thus, this section overviews all the main points of the GP-LiMA control part, specifically stating the LiM `Instruction` structure and how the LiM programmer can develop code allowing the mapping of a target application onto the GP-LiMA Unit.
As hinted in subsection 4.1.1, the control part is divided into two different units each representing one stage of the GP-LiMA pipeline. At first, there is the uCU that takes care of both initializing the system in the desired mode, allowing to choose between the processing or the standard data memory mode, and handling the instruction flow. Therefore, it interfaces with both the CPU and the IMem and is dedicated to the implementation of the instruction fetch stage. Then, the uCU is followed by the nCU that is the unit in charge of the instruction decoding, interacting with the uCU from one side and with the LiM Matrix on the other. It takes a subsection of the instruction, properly cropped by the uCU, and elaborates it to gather the values to be assigned to the configuration signals. These signals are the ones that, in the next clock cycle, enter the LiM Matrix to make it performing the computation demanded through the analysed LiM instruction on the addressed data.
So, the entire LiM `Instruction` aimed at programming the GP-LiMA, is split into multiple nested fields. From an high level view, there are 2 macro slices: the first one, called `uInstruction` that is acquired by the uCU, containing informations such as the next instruction to be fetched or when the end of the program is reached, and the second one, i.e. the `VLIW_nInstruction`, that is directly forwarded to the nCU to handle the algorithm computations in the M-SIMD mode. Then the `VLIW_nInstruction` is in turn composed of K fields, each called `nInstruction` and containing one of the simultaneous instructions that the GP-LiMA can perform.

Lastly, the single `nInstruction` is made up of multiple fields, each associated with a subset of the control signals driving a fixed group of Smart Blocks that are dedicated to the execution of that specific instruction.

In the following, the description of the GP-LiMA control part is organized into two sections: one for the uCU and the other for the nCU. Then, both sections provides an insight on the composition of the instruction slice the related control unit processes. Thus, in the uCU section all the fields comprising the `uInstruction` are detailed, deepening how the LiM Programmer has to initialize them to define the specific LiM program instructions flow. While, in the nCU section the single `nInstruction` structure is exhaustively illustrated, providing a guide for the LiM Programmer on how to set all the `nInstruction` fields to make the GP-LiMA performing the wanted computation. In particular, all the possible values the `nInstruction` fields can assume are explicitly listed together with their meaning in terms of involved LiM Matrix actions. However, to see a complete and detailed example of a real LiM program refer to subsection 5.2.1 where the LiM code for mapping the K-NN algorithm on the GP-LiMA is reported.

4.3.1 Micro Control Unit (uCU) & uInstruction

As for the PLiM model, the uCU is implemented through a microprogrammed machine, outlined in Figure 4.16. It exchanges information both with the CPU and the nCU to determine the actions that the GP-LiMA Unit has to perform.
Concerning the interface with the CPU, apart from the signals already mentioned in subsection 4.1.1, as the `LiM_Activate`, for starting the LiM in the processing mode, the `LiM_Program_Address`, to specify the exact LiM program to execute, and the `LiM_Program_END`, which flags the end of the LiM algorithm execution, the uCU takes three others input signals more, used to properly set the architecture before the processing: the `waitADD`, the `waitLoadEn` and the `queueWen`. These signals are the ones that allow the CPU to compose at run-time a bigger LiM program made up of a sequence of smaller pieces of code saved in different parts of the IMem. In this way, it is not strictly necessary to write the IMem for each new LiM application

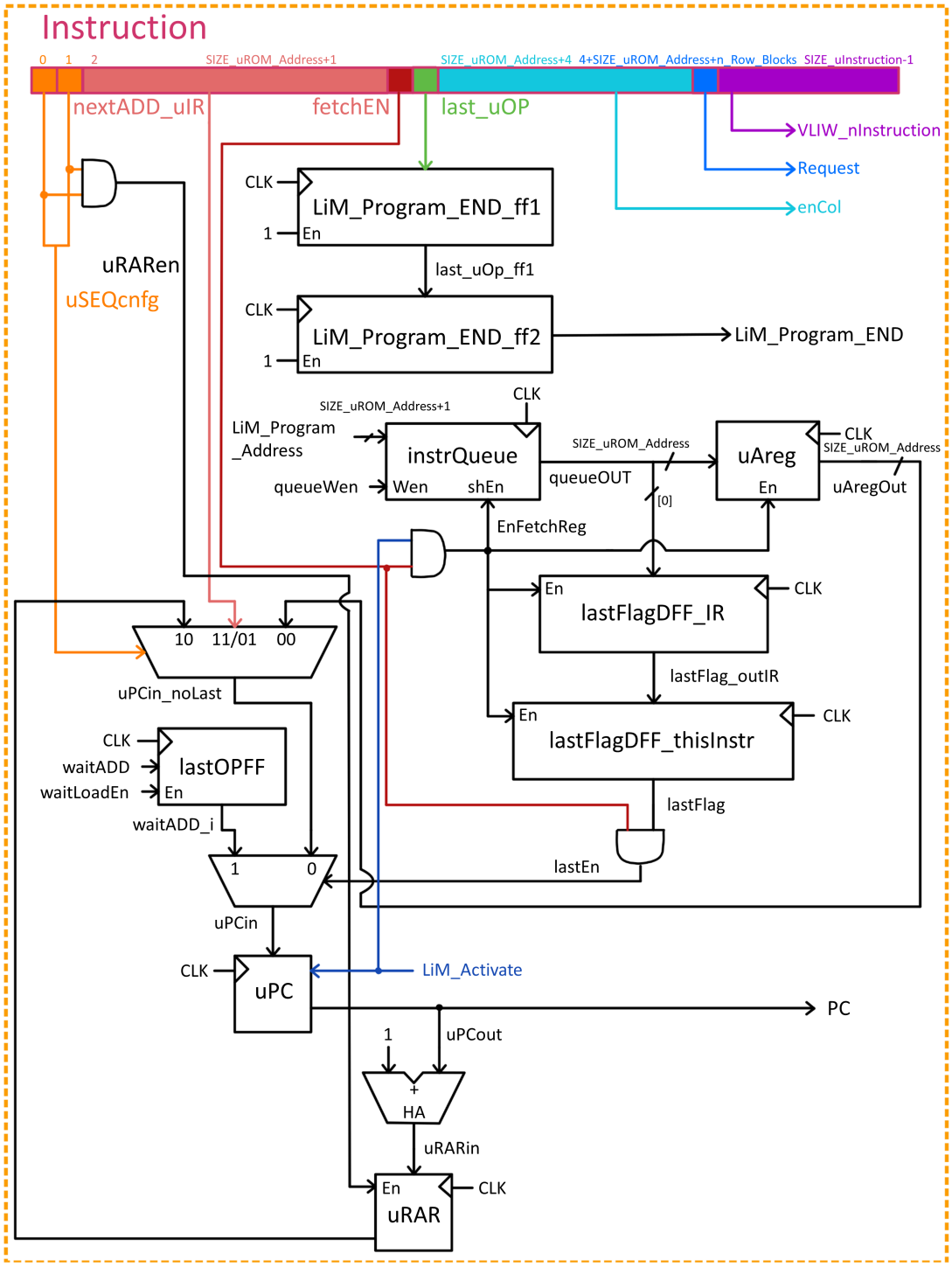


Figure 4.16: uCU scheme.

to be run, but it can be initialized at compile-time with all the possible code pieces that may be used by the addressed applications.
To do this, the uCU embeds a set of registers, which are the `lastOPFF` and the `istrueQueue` that is actually composed of a chain of registers (5 in the GP-LiMA default structure). All of them must be properly initialized before the GP-LiMA starts working in the processing mode. The `instrQueue` is used to keep the information about the list of the single small LiM programs that have to be run one after the other, creating the required bigger application. During the GP-LiMA initialization phase, the CPU has to insert inside the `instrQueue`, in the desired order, the address of the first instruction for each of the LiM programs to be run in sequence. Thus, the size of the `instrQueue` defines the number of LiM programs that can be consequently run without the intervention of the CPU. Also in this case, if needed, the hardware designer can choose to change this parameter before the architecture realization (see section A.1). So, to initialize the `instrQueue` content the CPU has to set to '1' the enable signal `queueWen` and assign to the `LiM_Program_Address` one address value for each clock cycle. Note that even if the final LiM Program is composed of a number of subprograms lower than the size of the `instrQueue` the CPU has to force the remaining registers with the address 0. It means that in any case the `queueWen` signal must be set to 1 for a number of clock cycles equal to the number of registers inside the queue minus 1. Then, once the `instrQueue` is filled, the CPU has to drive the `LiM_Program_Address` with the 0 value even during the GP-LiMA processing mode. Moreover, to inform the uCU about which is the last program that it has to execute, all the addresses inserted in the queue are complemented with a further bit that equals always '0' except for the address of the last program, where it is set to '1'. In addition, to allow the GP-LiMA to properly conclude the macro-program execution, the CPU has to initialize the `lastOPFF` register by setting for one clock cycle the `waitLoadEn` signal to '1' and driving the `waitADD` signal with the address value of the IMem location where a special instruction, called `wait`, is stored. This instruction is a compulsory one that must be always called as last instruction before the end of the LiM processing mode.
Furthermore, to accomplish the LiM macro-program execution, the uCU is provided with further registers such as `uAreg`, `lastFlagDFF_IR` and `lastFlagDFF_thisInstr`. The last two store the value of the last bit of the address at the output of the

instrQueue and are used to properly synchronize the end of the LiM processing mode when the address of the last program to be run is shifted out by the registers queue. While, when the GP-LiMA is run in the execution mode, the uAreg register always holds the address of the new LiM program to be run, if any. Thus, during the GP-LiMA processing, each time the last instruction of a middle sub-program is fetched, the EnFetchReg signal goes to one for one clock cycle, so enabling, at the next clock cycle, the shifting of the instrQueue and the update of the uAreg with the old value output by the registers queue.
However, the core of the uCU is given by a sequencer, that is actually a cascade of two multiplexers, the uPC register, the uRAR register and a simplified adder. All of these components are used to control the instruction flow inside the GP-LiMA during the actual LiM program execution. The uPC is the register holding the address of the LiM instruction to be fetched from the IMem, while the sequencer is the component that selects the next address to be provided to the uPC input. In general, the uCU together with the uInstruction exploit the explicit addressing for dictating the evolution of the current program. It means that, in each uInstruction there is a dedicated field reporting the address value for the next instruction to be fetched. In this way also the jump instruction is implicitly implemented since it is enough writing in that field the address of the instruction where to jump. Moreover, the uCU is also supplied with the mechanism for performing the call to a function, by means of the jump&link and the return instructions. They are implemented leveraging the mentioned uRAR register and simplified adder. When a jump&link instruction is called, the input of the uPC is updated with the next address value specified by that instruction, while the uRAR register loads the address of the instruction inside the IMem that is subsequent to the jump&link, taking it from the output of the adder that sums 1 to the address of the currently fetched instruction. Then, when the return instruction is fetched, at the next clock cycle, the content of the uPC is updated with the value stored inside the uRAR. Note that this kind of organization does not allow to perform nested calls to subroutines, meaning that, after a jump&link instruction has been called, before issuing a further jump&link, the return from the previous function must have been already called necessarily.
Summing up all the steps required for a whole LiM macro-program execution, at compile-time the LiM programmer has to initialize the IMem with all the useful

LiM sub-programs and load a special instruction at the address 0 that allows the GP-LiMA to start correctly after the reset. Then, before the GP-LiMA is started in the processing mode, the CPU has to initialize the `instrQueue` with the sequence of the LiM subprograms to implement and the `lastOPFF` with the address of the `wait` instruction, and, if needed, it can fill the LiM Array with the data to be processed. After the initialization phase is completed, the CPU can assert the `LiM_Activate` signal to flag the GP-LiMA that it can start performing the addressed programs. Note that this signal must be kept at '1' for the entire duration of the macro-program execution. Then, as soon as the `LiM_Activate` switches to '1', at the first active clock edge, the `instrQueue` shifts out the address of the first program to run. At the next clock cycle, the `uAreg` takes the data previously output by the instructions queue while the sequencer selects the uAreg content as the next value to be loaded inside the `uPC` register. One clock cycle after, the `uAreg` loads the value of the next program address, the `instrQueue` outputs the value of the third program address, and the `uPC` is updated with the address of the first program. In this way, the first instruction is fetched through the `PC` signal, which sends the address to the IMem, and the `Instruction` signal that returns to the uCU the entire fetched LiM instruction. Then, this instruction is split in two sub-portions: the `uInstruction`, that is used to drive the sequencer to fetch the next instruction, and the remaining slice composed of the `VLIW_nInstruction`, the `Request` and the `enCol` that are sent to the nCU. At the next clock cycle the second instruction is fetched and so on till when the last sub-program instruction is reached. At the next clock cycle after the fetching of the subprogram last instruction, the `uPC` is updated with the value of the second sub-program address stored in the `uAreg`, while both the `uAreg` and the `instrQueue` are updated with the next addresses values. Then this process is repeated till when the last instruction of the last subprogram is reached. In this case, at the next clock cycle the uPC is updated with the value of the `wait` address stored inside the `lastOPFF` register. The clock cycle after, the uPC returns to the 0 address, namely at the resest configuration. Then, after two clock cycles, the `LiM_Program_END` signal is set to 1 for one clock cycle to flag to the CPU that the GP-LiMA terminated the demanded applications. Note that only at this point the `LiM_Activate` signal can be disabled.

To make the uCU able to implement the desired instruction flow, the LiM programmer has to properly set the bits belonging to the `uInstruction` portion inside all the LiM instructions saved in the IMem. Thus, in the following, it is described how to initialize the fields composing the `uInstruction` in order to obtain the wanted GP-LiMA behaviour.

uInstruction

- `uSEQcnfg`:
  - `"00"` → for the `wait` and the last instruction of a sub-program;
  - `"11"` → for the `jump&link` instruction;
  - `"10"` → for the `return` instruction;
  - `"01"` → for all the other instructions (`jump` or sequential).
- `nextADD_uIR`: contains the address of the next instruction to be fetched. Note that in the case of a `return` instruction it can assume any value.
- `fetchEN`:
  - `"1"` → for the last instruction of a sub-program and for the one saved at address 0;
  - `"0"` → for all the other instructions.
- `last_uOP`:
  - `"1"` → for the `wait` instruction;
  - `"0"` → for all the other instructions.
- `enCol`: contains the enable signals for the Block columns inside the LiM Array. Each bit of the `enCol` field is associated with a specific column, starting with the first bit that drives the first column on the left and so on in an increasing order.

- `Request`:
  - `"0"` → for the `wait` instruction and for the one saved at address 0;
  - `"1"` → for all the other instructions.

4.3.2 Nano Control Unit (nCU) & nInstruction

As already anticipated, the nCU is the linking unit between the uCU and the LiM Matrix that is dedicated to the handling of the data transfer and processing inside the GP-LiMA following the M-SIMD computing paradigm. As it can be seen from Figure 4.17, it is a quite simple block that mainly encloses a number of instruction decoders equal to the maximum number of different operations that the LiM Matrix has to perform simultaneously on different data.

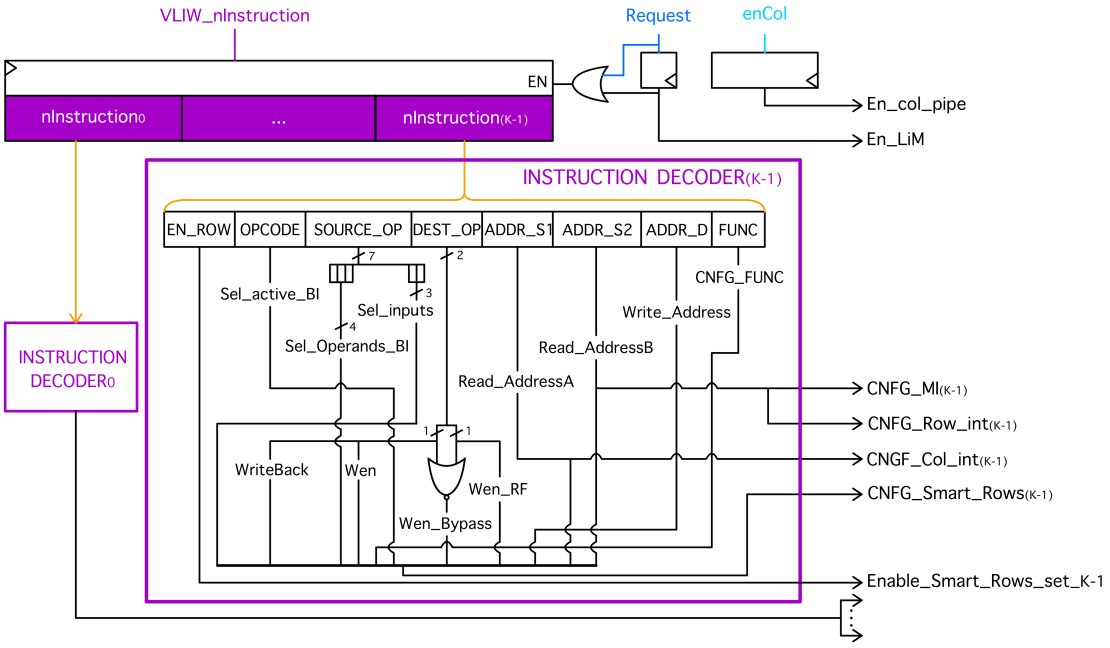


Figure 4.17: Overview of the internal structure of the nCU.

The nCU input interface is given by 3 input signals coming from the uCU, which are: the `Request` signal, that informs the nCU when it can start the decoding process, the `enCol`, containing the enable signals for the Block Columns of the LiM Array, and the `VLIW_nInstruction`, holding the LiM instruction portion that the nCU has to

decode. All of these signals are sampled at the beginning of the block by the decode stage pipe registers and the returned `Request` and `enCol` signals are both directly sent to the LiM Matrix, while the resulting `VLIW_nInstruction` is portioned into smaller sub-parts, each forwarding a different `nInstruction` to its dedicated instruction decoder. Then, each instruction decoder outputs a set of configurations signals that is sent to the LiM Matrix and that comprises the `CNFG_Smart_Rows_i`, which controls the processing inside a generic Smart Block, the `CNFG_MI_i`, which drives one specific SIMD MI block inside the MSIMD MI, the `Enable_Smart_Rows_set_i`, for enabling some smart rows, the `CNFG_Row_int_i` and the `CNFG_Col_int_i`, which control a subset of row or column interconnections, respectively. In particular, each `CNFG_Smart_Rows_i` drives all the Smart Blocks belonging to the smart rows which are associated with the instruction decoder that outputs it. In the same way, the `CNFG_Row_int_i` and the `CNFG_Col_int_i` signals act as the selection signal for the column or the row interconnections that must be controlled by the same instruction processed by the instruction decoder that generates them.
Furthermore, looking at Figure 4.17, it can be noticed that the single instruction decoder is a very simple block, mainly implementing signals associations. It takes the `nInstruction` and properly generates the control signals, by directly connecting, in most of the cases, the bits associated to a specific field to the correspondent control signal. This straightforward association is feasible thanks to a proper code assignment organized for the values that each field can assume. It means that, in most of the cases, the code value requiring a specific operation directly corresponds to the values that the configuration signals must assume to drive that operation execution inside the Smart Block.

To understand which fields of the `nInstruction` drive which components of the Smart Block it is enough to match the control signals names between Figure 4.17 and Figure 4.4. Moreover, to know how the LiM Programmer can set the `nInstruction` fields to properly dictate the wanted data processing inside the LiM Matrix, in the following a thorough insight on the `nInstruction` structure is proposed.

nInstruction

EN ROW	OPCODE	SOURCE OP	DEST OP	ADDR S1	ADDR S2	ADDR D	FUNC

- EN ROW: defines which are the smart rows that have to be enabled for the execution of that specific `nInstruction`, among the rows driven by the instruction decoder in which this `nInstruction` enters. To each row a different bit of the `EN ROW` field is dedicated in an increasing order, starting from the first bit that is linked to the row with the lowest identification number (namely, looking at Figure 4.3, the closest smart row to the upper part of the LiM Array).

- OPCODE: specifies the kind of instruction to be performed. A part from the null and load instructions, all the other values for the `OPCODE` field are identified by the names of the logic blocks under the Block Word inside the Smart Block. So, to perform a target operation, the `OPCODE` of the `nInstruction` must be initialized with the name of the logic block that can perform it. In the case the enabled logic block implements multiple functions, to select the wanted specific operation the `FUNC` field must be set, too.
  In Table 4.2 the ISA associated with the default configuration of the GP-LiMA is reported, detailing all the possible combinations of values for the `OPCODE` and `FUNC` fields that identifies all the programmable executable operations.

- SOURCE OP: indicates from where the operands for the demanded computation have to be taken. In the case of an `nInstruction` involving two operands, their insertion order inside the `SOURCE OP` field defines the association with the first and the second operand, namely the values assumed by the Smart Block signals `OpA` and `OpB`, respectively. However, in some cases other complementary informations must be added to precisely identify the data that has to be elaborated; e.g. when the MI is specified as source operand, the address of the block from which the data must be taken has to be specified too. For this reason, the `SOURCE OP` field is coupled with other 2 fields, i.e. `ADDR S1` and `ADDR S2`, used to specify these additional informations for both the operands. In Table 4.3 all the possible combinations of operands, i.e. all the feasible values for the `SOURCE OP` field and related settings for the `ADDR S1` and `ADDR`

S2 fields, are described.
Note that all the listed operands can be used for any of the available operations (OPCODE values) expect for the right shift operation (OPCODE = OP_RShifter) which accepts as source operand only a data coming from either the row interconnections (Row_int) or the MI(MI).

- DEST OP: specifies the destination storage component inside the Smart Block where the outcome of the demanded operation has to be saved. Also the DEST OP is complemented with another nInstruction field, i.e. the ADDR D, which provides, when needed, further information about the particular addressed storage element.
  Table 4.4 illustrates all the possible values for the DEST OP and its correct usage together with the ADDR D field.

- ADDR S1: is associated with the SOURCE OP field and, when needed, contains the address for one of the two specified operands. It reports a valid address only in case either RFA or Col_int is indicated as a source operand, regardless of whether it is specified as the first or the second operand. In particular, RFA indicates a generic register inside the RF, while Col_int tells that the data to be processed must be retrieved from the Smart Block input connected to the column interconnection.
  The details on the ADDR S1 usage are provided in Table 4.3.

- ADDR S2: is associated with the SOURCE OP field and, when needed, contains the address for one of the two specified operands. It reports a valid address only in case either RFB, Row_int or MI is indicated as a source operand, regardless of whether it is specified as the first or the second operand. In particular, RFB indicates a generic register inside the RF, while Row_int and MI tell that the data to be processed must be retrieved from the Smart Block input connected to the row interconnection or to the MSIMD MI, respectively.
  The details on the ADDR S2 usage are provided in Table 4.3.

- ADDR D: used only when the destination storage component specified in the DEST OP field corresponds to the RF. In this case, ADDR D must be initialized with the address identifying the particular register inside the RF that must be

written.

The details on the ADDR D usage are provided in Table 4.4.

- FUNC: is used, coupled with the OPCODE value, to specify, only when needed, the particular function the logic block set in the OPCODE field has to perform. In Table 4.2 all the possible FUNC values are listed for each of the available OPCODE values.

ISA		
OPCODE	FUNC	**Instruction definition** Res = OpA OP OpB Res = OP(Op)
nullOP	nullFunc	Null instruction used to wait for one clock cycle to handle data hazard occurrences.
OP_Load	nullFunc	Copy the value of any source operand into the specified destination (Block Word, Bypass Storage or Register File).
OP_ALU	AND_OP	Res = OpA AND OpB
OP_ALU	XOR_OP	Res = OpA XOR OpB
OP_ALU	SUM_OP	Res = OpA + OpB
OP_ALU	SUB_OP	Res = OpA - OpB
OP_ALU	ABS_OP	Res = |Op|
OP_ALU	OR_OP	Res = OpA OR OpB
OP_ALU	NAND_OP	Res = OpA NAND OpB
OP_ALU	NOT_OP	Res = NOT Op
OP_ALU	NOR_OP	Res = OpA NOR OpB
OP_ALU	XNOR_OP	Res = OpA XNOR OpB
OP_ALU	EQ_OP	Res = 1 if OpA = OpB 0 if OpA ≠ OpB
OP_ALU	NOT_EQ_OP	Res = 1 if OpA ≠ OpB 0 if OpA = OpB
OP_ALU	GT_OP	Res = 1 if OpA > OpB

		0 if OpA ≤ OpB
OP_ALU	LT_OP	Res = 1 if OpA < OpB 0 if OpA ≥ OpB
OP_Multiplier	nullFunc	Res = OpA × OpB
OP_LUT	nullFunc	Res = "0...0"&LUT(Op)
OP_LUT	Sign_ext_LUT_OP	Res = "LUT(Op)(MSB)...LUT(Op)(MSB)"&LUT(Op)
OP_RShifter	Logic_RShift_OP	Res = '0'& Op(MSB)& ... &Op(1)
OP_RShifter	Arith_RShift_OP	Res = Op(MSB)& Op(MSB)& ... &Op(1)

Table 4.2: ISA associated with the default Smart Block structure. All the operations executable by the single Smart Block are listed and described.

Source Operands Mechanism			
SOURCE OP	ADDR S1	ADDR S2	**Instruction description**
Mem_Mem, Mem	nullADDR_S1	nullADDR_S2	Both OpA and OpB take the value stored in the Block Word.
Mem_RFA	RF Address	nullADDR_S2	OpA forwards the content of the Block Word while OpB carries the value held by the register inside the RF pointed by the address specified in the ADDR S1 field of the instruction.

Mem_Row_int	nullADDR_S1	Row Int Shifts Number	OpA forwards the content of the Block Word while, for each Smart Block, OpB carries the content of the Bypass Storage inside the Smart Block placed at the right of the current Smart Block of a number of positions equal to the value specified in the ADDR S2 field of the instruction.
Mem_Col_int	Col Int Shifts Number	nullADDR_S2	OpA forwards the content of the Block Word while, for each Smart Block, OpB carries the content of the Bypass Storage inside the Smart Block placed below the current Smart Block of a number of positions equal to the value specified in the ADDR S1 field of the instruction.
Mem_MI_int	nullADDR_S1	LiM Array Address	OpA forwards the content of the Block Word while OpB takes the value output by the MI that fetches the Block Word data pointed by the LiM Array address specified in the ADDR S2 field of the instruction.

RFA_Mem	RF Address	nullADDR_S2	OpA carries the value held by the register inside the RF pointed by the address specified in the ADDR S1 field of the instruction while OpB forwards the content of the Block Word.
RFA_RFA RFA	RF Address	nullADDR_S2	Both OpA and OpB assume the value stored in the register inside the RF pointed by the address specified in the ADDR S1 field of the instruction.
RFA_RFB	RF Address	RF Address	OpA takes the value from the register inside the RF pointed by the address specified in the ADDR S1 field of the instruction while OpB assumes the value stored in the RF register pointed by the address specified in the ADDR S2 field of the instruction.

RFA_Row_int	RF Address	Row Int Shifts Number	OpA takes the value from the register inside the RF pointed by the address specified in the ADDR S1. For each Smart Block, OpB carries the content of the Bypass Storage inside the Smart Block placed at the right of the current Smart Block of a number of positions equal to the value specified in the ADDR S2 field of the instruction.
RFA_MI_int	RF Address	LiM Array Address	OpA takes the value from the register inside the RF pointed by the address specified in the ADDR S1. OpB takes the value output by the MI that fetches the Block Word data pointed by the LiM Array address specified in the ADDR S2 field of the instruction.

RFB_Col_int	Col Int Shifts Number	RF Address	OpA assumes the value stored in the RF register pointed by the address specified in the ADDR S2 field of the instruction, while, for each Smart Block, OpB carries the content of the Bypass Storage inside the Smart Block placed below the current Smart Block of a number of positions equal to the value specified in the ADDR S1 field of the instruction.
Row_int_Mem	nullADDR_S1	Row Int Shifts Number	For each Smart Block, OpA carries the content of the Bypass Storage inside the Smart Block placed at the right of the current Smart Block of a number of positions equal to the value specified in the ADDR S2 field of the instruction, while OpB forwards the Block Word content.

Row_int_RFA	RF Address	Row Int Shifts Number	For each Smart Block, OpA carries the content of the Bypass Storage inside the Smart Block placed at the right of the current Smart Block of a number of positions equal to the value specified in the ADDR S2 field of the instruction. OpB takes the value from the register inside the RF pointed by the address specified in the ADDR S1.
Row_int_Row_int Row_int	nullADDR_S1	Row Int Shifts Number	For each Smart Block, both OpA and OpB carry the content of the Bypass Storage inside the Smart Block placed at the right of the current Smart Block of a number of positions equal to the value specified in the ADDR S2 field of the instruction.

Row_int_Col_int	Col Int Shifts Number	Row Int Shifts Number	For each Smart Block, OpA carries the content of the Bypass Storage inside the Smart Block placed at the right of the current Smart Block of a number of positions equal to the value specified in the ADDR S2 field of the instruction, while OpB carries the content of the Bypass Storage inside the Smart Block placed below the current Smart Block of a number of positions equal to the value specified in the ADDR S1 field of the instruction.
Col_int_Mem	Col Int Shifts Number	nullADDR_S2	For each Smart Block, OpA carries the content of the Bypass Storage inside the Smart Block placed below the current Smart Block of a number of positions equal to the value specified in the ADDR S1 field of the instruction, while OpB forwards the content of the Block Word.

Col_int_RFB	Col Int Shifts Number	RF Address	For each Smart Block, OpA carries the content of the Bypass Storage inside the Smart Block placed below the current Smart Block of a number of positions equal to the value specified in the ADDR S1 field of the instruction. OpB assumes the value stored in the RF register pointed by the address specified in the ADDR S2 field of the instruction.
Col_int_Row_int	Col Int Shifts Number	Row Int Shifts Number	For each Smart Block, OpA carries the content of the Bypass Storage inside the Smart Block placed below the current Smart Block of a number of positions equal to the value specified in the ADDR S1 field of the instruction, while OpB carries the content of the Bypass Storage inside the Smart Block placed at the right of the current Smart Block of a number of positions equal to the value specified in the ADDR S2 field of the instruction.

Col_int_Col_int Col_int	Col Int Shifts Number	nullADDR_S2	For each Smart Block, both OpA and OpB carry the content of the Bypass Storage inside the Smart Block placed below the current Smart Block of a number of positions equal to the value specified in the ADDR S1 field of the instruction.
Col_int_MI_int	Col Int Shifts Number	LiM Array Address	For each Smart Block, OpA carries the content of the Bypass Storage inside the Smart Block placed below the current Smart Block of a number of positions equal to the value specified in the ADDR S1 field of the instruction. OpB takes the value output by the MI that fetches the Block Word data pointed by the LiM Array address specified in the ADDR S2 field of the instruction.
MI_int_Mem	nullADDR_S1	LiM Array Address	OpA takes the value output by the MI that fetches the Block Word data pointed by the LiM Array address specified in the ADDR S2 field of the instruction, while OpB forwards the content of the Block Word.

MI_int_RFA	RF Address	LiM Array Address	OpA takes the value output by the MI that fetches the Block Word data pointed by the LiM Array address specified in the ADDR S2 field of the instruction. OpB takes the value from the register inside the RF pointed by the address specified in the ADDR S1.
MI_int_Col_int	Col Int Shifts Number	LiM Array Address	OpA takes the value output by the MI that fetches the Block Word data pointed by the LiM Array address specified in the ADDR S2 field of the instruction. OpB carries the content of the Bypass Storage inside the Smart Block placed below the current Smart Block of a number of positions equal to the value specified in the ADDR S1 field of the instruction.
MI_int_MI_int MI_int	nullADDR_S1	LiM Array Address	Both OpA and OpB assume the value output by the MI that fetches the Block Word data pointed by the LiM Array address specified in the ADDR S2 field of the instruction.

Table 4.3: All possible combinations of source operand kinds are listed together with their usage in case of addresses to be specified. Note that OpA and OpB refer to the couple of data on which each Smart Block performs the required operations.

Destination Operand Mechanism		
DEST OP	ADDR D	**Instruction description**
DEST_Mem	nullADDR_D	The value assumed by the Result signal inside the Smart block, i.e. the outcome of the demanded operation, is stored inside the Block Word itself.
DEST_Bypass	nullADDR_D	The value assumed by the Result signal inside the Smart block, i.e. the outcome of the demanded operation, is stored inside the Bypass Storage.
DEST_RF	RF Address	The value assumed by the Result signal inside the Smart block, i.e. the outcome of the demanded operation, is stored inside the register inside the RF pointed by the address specified in the ADDR D field of the instruction.

Table 4.4: All possible destination operand kinds are listed together with their usage in case of addresses to be specified.

Chapter 5

GP-LiMA Performance

Finally, in this chapter the performance achieved by the proposed GP-LiMA are estimated to verify whether the expected improvements for the applications executions are accomplished. In particular, being the GP-LiMA essentially an architectural model, here, the synthesis for a specific configuration of it is analysed, which has the values set for the LiM Matrix size and the M-SIMD degree parameters that lead to a GP-LiMA version that is as general as possible for the mapping of different kinds of algorithms. Afterwards, the gathered performance achievements are compared with the ones of already existing LiM and standard CPU-centric solutions.
Specifically, the chapter outline develops into two sections:

- ***Section 5.1*** **- GP-LiMA Layout & Performance Evaluation** illustrates the composition of the specific investigated GP-LiMA device and reports the post-synthesis and post-place&route results for the performance parameters that do not depends on the specific algorithm run on the architecture, such as the maximum operating frequency, the area occupation and the worst case power consumption.

- ***Section 5.2*** **- Benchmarks Mapping on the GP-LiMA & Comparisons with other Architectures** is divided into multiple subsections. The first one deals with all the benchmarks used to test the GP-LiMA in terms of programming generality and execution efficiency. In particular,

here all the benchmarks used to estimate the PLiM performance are considered and, for each of them, the way they can be mapped onto the synthesized GP-LiMA is explained. Besides, for the most simple benchmark, i.e. the K-NN, the actual LiM code is provided as a guided example to understand how a generic algorithm can be translated into a sequence of GP-LiMA instructions. Then, the following subsection, focus on the post-place&route performances associated with each tested algorithm, i.e. the benchmark execution time, the back-annotated power consumption and both the worst case and the back-annotated energy expenses. Then, the execution time, the worst case power and energy consumptions achievements are compared on a per-sample basis with the ones reached by the PLiM customized architectures retrieved for each benchmark. Finally, the last subsection presents a further comparison that sees the GP-LiMA back-annotated energy attainments juxtaposed with the energy values spent by a standard CPU-centric system, i.e. the RISC-V connected to its standard memory hierarchy. Here, the benefits gained thanks to the employment of the LiM approach in terms of algorithms execution efficiency are highlighted, validating the worthiness for future investigations of this new computing paradigm.

5.1 GP-LiMA Layout & Performance Evaluation

To assess the goodness of the designed architectural model, an analysis of the accessed performance must be carried out. However, being the GP-LiMA a model for speeding up the design time of LiM architectural solutions, to gather estimations, at first a specific architecture must be produced by setting the tunable parameters of the GP-LiMA model. Since the GP-LiMA is intended to lead to programmable LiM devices with a high level of programming generality, just to validate this feature, the chosen values for these parameters aim at identifying a LiM Matrix structure that further maximizes the adaptability to different algorithms. In Figure 5.1 the final arrangement of the specific version of the GP-LiMA considered for the performance evaluation is depicted. In particular, the image details the LiM Array layout together with the connections to the nCU. While, in the box below, all the architectural features that characterize the specific structure of the GP-LiMA under

study are summarized.

LiM Array Structure

```
# Columns = 32
Standard Section: #Rows = 5  ⟹ #Standard Blocks = 160
Smart Section:    #Rows = 16 ⟹ #Smart Blocks = 512
                  #MSIMD Instructions = 3
                  Instruction_0 drives smart rows:  0,1,2,3,4
                  Instruction_1 drives smart rows:  5,6,7,8,9
                  Instruction_2 drives smart rows:  10,11,12,13,14,15
# Total Rows = 672 , Memory Parallelism (Blocks width) = 16 bits
⟹ Addressable Space = 1344 bytes
```

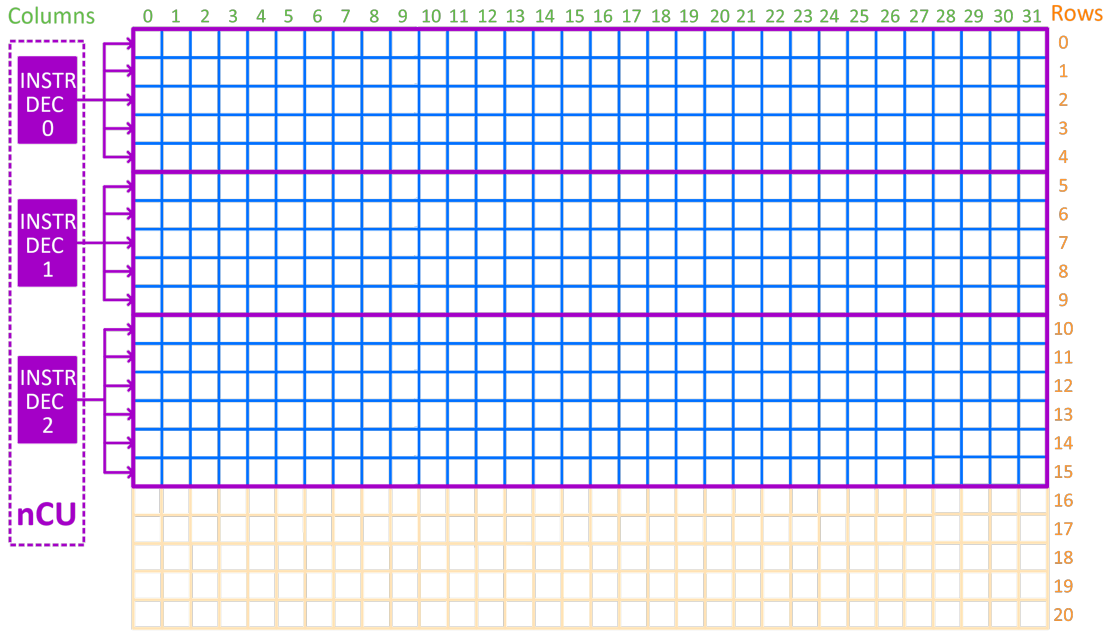


Figure 5.1: LiM Matrix arrangement set for the GP-LiMA device to be synthesized. The blu squares identify the Smart Blocks, while the cream-colored ones represent the Standard Blocks.

For most of the algorithms, the number of Smart Blocks composing the LiM Array determines the total number of samples on which the application can run. For this

reason, in the identified GP-LiMA, the number of Block columns and smart rows is sized trying to find the right trade-off between memory array storage capacity and device efficiency while making the maximum number of processable samples be a power of 2, i.e. 512. This is also thought of for algorithms involving division operations on the stored data by the number of total samples, which in this way can be performed by leveraging the RShifter mechanism across the smart rows already provided with the Smart Block default version. Moreover, to maximize the architecture efficiency for the chosen number of total LiM Array Blocks, the number of Blocks included in a single row is higher than the one composing an array column. This because, as already explained in section 4.2.2, the row interconnections implementation is optimized compared with the column one, so the row interconnections can host more blocks than the column ones under the same complexity. Then, to promote mapping generality the architecture is structured so that it can perform up to 3 different instructions simultaneously, each on a different set of data stored inside the LiM Array. Besides, the M-SIMD processing mode is organized inside the LiM Array so that the number of data that can be processed in parallel by the same instruction is quite balanced among the three different simultaneous instructions. In particular, the first 2 instructions drive 5 smart rows each, while the last instruction controls the remaining 6. Furthermore, it is worth highlighting that, for most algorithms, the number of considered samples directly impacts the computing precision. The bigger the processed dataset is, meaning the higher the number of LiM Array Smart Blocks is, the higher the chance to incur data overflow during the algorithm's execution is. This is one of the reasons why, in the GP-LiMA at hand with memory parallelism of 16 bits, the number of smart rows and columns is sized so that the total number of Smart Blocks equals 512, rather than 1024. Besides, this design choice also points to limit the complexity of the overall structure as well as the energy and the power consumption.

5.1.1 Synthesis

After having set the model parameters looking for a trade-off among storage capacity, programming generality and energy efficiency, the resulting GP-LiMA device

was synthesized to retrieve a first estimation for the values of the performance metrics that are independent of the run application.
The Synopsys tool was used to perform multiple synthesis based on standard cells, one for a different technology node: 45 nm, 28 nm, and 15 nm. Moreover, to further minimize the GP-LiMA power consumption, the tool was set to obtain the equivalent of the GP-LiMA netlist embedding the clock gating mechanism. The clock gating is a technique exploited in low power systems which aims at reducing the power consumption by disabling the clock signal individually for each of the registers in the design when that specific element does not have to be updated during that generic clock cycle. To do so, the clock entry for each register is fed by a control block which disables the signals according to the values of the registers enable signals and the general clock signal. In this way, the power consumed by each flip flop due to the clock signal switching is strongly cut resulting in significant power savings. The commands used in the Synopsys script to force the clock gating insertion are reported below in the colour blue.

```
set_clock_gating_style
analyze ...
set compile_clock_gating_through_hierarchy true
elaborate ...
compile -gate_clock
report_clock_gating -ver -gating -gated -multi_stage >
     clock_gating_DUT.txt
ungroup -all -flatten
write ...
```

In Table 5.1 the outcomes of the performance analysis on the post-synthesis netlists generated using all the mentioned technologies are summarized. Indeed, for each technology node, two syntheses of the GP-LiMA version previously declared were carried out, each with different memory parallelism, i.e. 16 bits and 32 bits, to have a rough hint on the performance scaling according to the word precision.
As expected, the metrics values improve with the scaling of the technology node, while worsen with the increasing of the memory parallelism.

Technology	45 nm	28 nm	15 nm	Nbit
Critical path	3.89 ns	1.78 ns	0.53 ns	16
	3.88 ns	1.85 ns	0.53 ns	32
Area	1.78 mm^2	1.27 mm^2	0.47 mm^2	16
	3.57 mm^2	2.52 mm^2	0.96 mm^2	32
% Gated registers	93.32%	93.32%	93.32%	16
	92.28%	92.28%	92.28%	32
Maximum clock frequency	256.41 MHz	555.55 MHz	1.66 GHz	16
	256.41 MHz	526.31 MHz	1.66 GHz	32
Worst case power	158.06 mW	132.03 mW	255.58 mW	16
	259.4 mW	196.79 mW	380.43 mW	32
% Interconnections area / GP-LiMA area	22.97 %	23.49 %	25.77 %	16
	23.12 %	23.27 %	26.04 %	32
% Interconnections power / GP-LiMA power	51.19 %	51.19 %	56.68 %	16
	62.2 %	65.54 %	74.41 %	32

Table 5.1: GP-LiMA synthesis results for different technologies and for memory parallelism equal to 16 and 32 bits. Note that the power values refer to the GP-LiMA maximum clock frequencies.

In general, the maximum operating frequency allowed ranges from about 256 MHz, with the 45 nm technology, to 1.66 GHz with the 15 nm, showing an improvement of 548%, while the area occupation decreases of 73%. Only the worst case power seems to worsen passing from 158 mW to 255.58 mW for the 16 bits versions. However, it is not a real deterioration since the estimations for the power consumptions are all retrieved fixing the clock frequency to the maximum value allowed. It can be seen that the enhancements reached in terms of maximum operating frequency (548%) are way more significant than the worsening of the power results (61% for 16 bits and 45% for 32 bits). Thus, another metric merging the impact of the two contributions should be considered, such as the energy/clock cycle. In the case of 16 bits, the energy/clock cycle for the 45 nm technology node equals 616.43 pJ, for the 28 nm it equals 237.65 pJ, while for the 15 nm it equals 153.96 pJ, pointing out

a total improvement of 75% (from the 45 nm to the 15 nm).
Concerning the performance trend according to the memory parallelism, when the number of bits composing the word doubles, the critical path remains almost the same, while, as expected, the area doubles and the power consumption increases of about 64%.
Furthermore, since, as already saw in subsection 4.2.2, the interconnections represent a key point of the GP-LiMA paradigm for the programming flexibility but also for the complexity overhead introduced in the LiM Matrix, a procedure for estimating their actual impact on both the total area occupation and power expense of the architecture was conducted. In particular, an aside entity was created and then synthesized embedding all the possible interconnections branching in the LiM Array of the synthesized GP-LiMA. Specifically, it included 16 row interconnections, each interfacing 32 Smart Blocks, 32 column interconnections on 21 blocks each and one MSIMD MI composed of 3 SIMD MI, each connected to all the 672 Blocks in the LiM Array. Then, the results on the interconnections impact are reported at the bottom of Table 5.1 and are computed as: (Interconnections Area/GP-LiMA Area)×100 and (Interconnections Power/GP-LiMA Power)×100. It derives that, for all the technologies and the memory parallelisms, in the average the interconnections represent about 24% of the GP-LIMA total area, while they consume a power amount of the total GP-LiMA power that ranges from about 51% in the best case (16 bits with the 45 nm) to 74% in the worst one (32 bits with the 15 nm). Thus, these results clearly confirm the massive role that the interconnections play in terms of added complexity to the whole design.

5.1.2 Place & Route

After the synthesis, a place&route procedure was performed only on the netlist synthesized with the 45 nm technology and representing the 16 bit version of the indicated GP-LiMA. As for the PLiM performance evaluation (subsection 3.2.6), this further step aimed at gather more precise estimations of the considered performance metrics, especially for the actual impact of the interconnections on the whole GP-LiMA. The results of the analysis both post-place&route and post-synthesis are reported in Table 5.2. Moreover, to verify the gain in terms of power consumption

obtained thanks to the clock gating insertion, the same 16 bit GP-LiMA version was at first synthesized using the 45 nm technology without forcing the clock gating and then passed through a place&route process. The final metrics outcomes are shown always in Table 5.2.
While, in Figure 5.2 it is highlighted the critical path after the place&route process for the clock gated GP-LiMA.

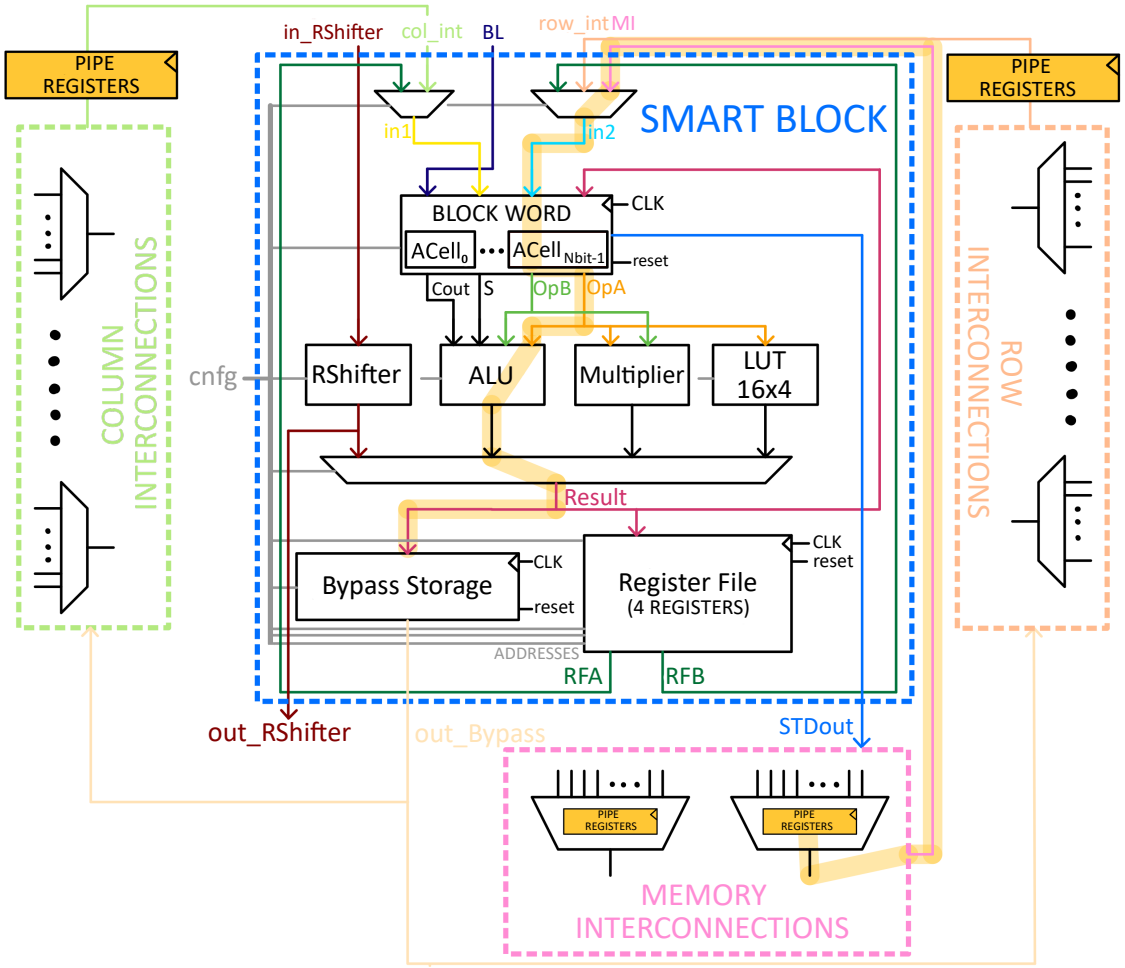


Figure 5.2: Post-Place&Route GP-LiMA critical path.

As expected, the critical path passes through the single Smart Block. It starts after the pipe registers of a single SIMD MI inside the M-SIMD MI, then it enters the Smart Block and passes through the right input multiplexer, the multiplexer

driving the `OpA` signal inside a single arithmetic cell and the ALU block, crossing the subtractor and 3 multiplexers, and finally it ends at the Bypass Storage input passing through the multiplexer returning the `Result` value. Note that the critical path passes through the ALU rather than the multiplier. This happens just because while the substractor is on 16 bits the multiplier instantiated takes as inputs signals on 8 bits (returning an output on 16 bits).

	Synthesis		**Place & Route**	
Clock gating insertion	YES	NO	YES	NO
Critical path	3.89 ns	3.89 ns	4.29 ns	4.38 ns
Area	1.78 mm^2	1.83 mm^2	1.85 mm^2	1.87 mm^2
Clock frequency	232.55 MHz	227.27 MHz	232.55 GHz	227.27 MHz
Worst case power	146.82 mW	264.7 mW	690.79 mW	1201.77 mW
% $\frac{\textbf{Interconnections area}}{\textbf{GP-LiMA area}}$	22.97 %	22.38 %	25.11 %	24.78 %
% $\frac{\textbf{Interconnections power}}{\textbf{GP-LiMA power}}$	50.58 %	29.5 %	29.25 %	16.45 %

Table 5.2: GP-LiMA synthesis and place&route results using the Nangate45 library and for memory parallelism equal to 16 bits. The table reports the achieved performance for both the architecture exploiting the clock gating technique and the one which does not. Note that the power values refer to the specified clock frequency values.

Looking at Table 5.2, first of all, it can be noticed that, for the clock gated GP-LiMA version, the metrics values after the place&route worsen as the minimum clock period changes from 3.89 ns to 4.29 ns, showing a worsening of 10%, while the area occupation grows from 1.78 mm^2 to 1.85 mm^2. However, the greatest difference is found for the worst case power consumption that after the place&route process, under the same clock frequency value, (232.55 MHz) reaches 690.79 mW, pointing out a deterioration of 370%.
Furthermore, always looking at the power values, the contribute of the clock gating insertion to the overall GP-LiMA power saving stands out. After the place&route the GP-LiMA netlist exploiting the clock gating technique consumes around 42% fewer than the one without clock gating.
Lastly, as already anticipated, the post-place&route estimation of how the interconnections affect the overall area occupation and power consumption of the final

architecture can be seen in the lower part of Table 5.2. While the result about the area percentage changes slightly from the post-synthesis value (the interconnections occupy about 25% of the total area), the power outcomes undergo a significant reduction, passing from 50% to 29%. However, although the post place&route results provide a better prediction about the interconnections power consumption percentage on the overall GP-LiMA expense, the value remains still quite high.

5.2 Benchmarks Mapping on the GP-LiMA & Comparisons with other Architectures

To gather informations about the average values for the execution times and the energy expenses characterizing the synthesized GP-LiMA and to eventually compare the attained performances with the ones of other architectural solutions, a set of benchmarks to be mapped on the GP-LiMA was selected. Specifically, the same benchmarks used for the PLiM validation were run onto the GP-LiMA to both verify its programming flexibility and have a more direct comparison to make. As for the PLiM, the size of the involved dataset for each benchmark was chosen so as to maximize the number of Smart Blocks already instantiated inside the LiM Array that contribute to the elaboration of the algorithm results, with the aim of both speeding up the execution and improving the performance in terms of energy/sample.
In the following, for each benchmark, a dedicated subsection is reported explaining how the associated application can be successfully implemented by the GP-LiMA. Each subsection starts with the definition of the specific algorithm that must be mapped onto the architecture, then it includes two figures: one showing how the LiM Array content has to be initialized with the starting dataset for preparing the GP-LiMA to the execution of the demanded algorithm and another displaying in which memory array locations the final results will be stored at the end of the execution. Note that in the lower-left corner of each figure a legend is present indicating the association between each colour and the data it represents. Finally, each subsection ends with the explanation of the algorithm implementation that is organized in macro step, each providing information about the main operations the GP-LiMA performs and their goal together with the number of LiM instructions

needed to perform that step. Note that the instructions count is specified in a parametric way, namely keeping, in the final expression, the number of processed dataset samples as a variable.

5.2.1 K-NN: K Neirest Neighbour

Given a bi-dimensional dataset ($d = 2$) composed of N samples (x_i,y_i) and the sample to be classified (x_s,y_s), compute all the N distances D_i between the dataset samples and (x_s,y_s), using the Manhattan norm ($p = 1$):

$$D_i = |x_s - x_i| + |y_s - y_i| \tag{5.1}$$

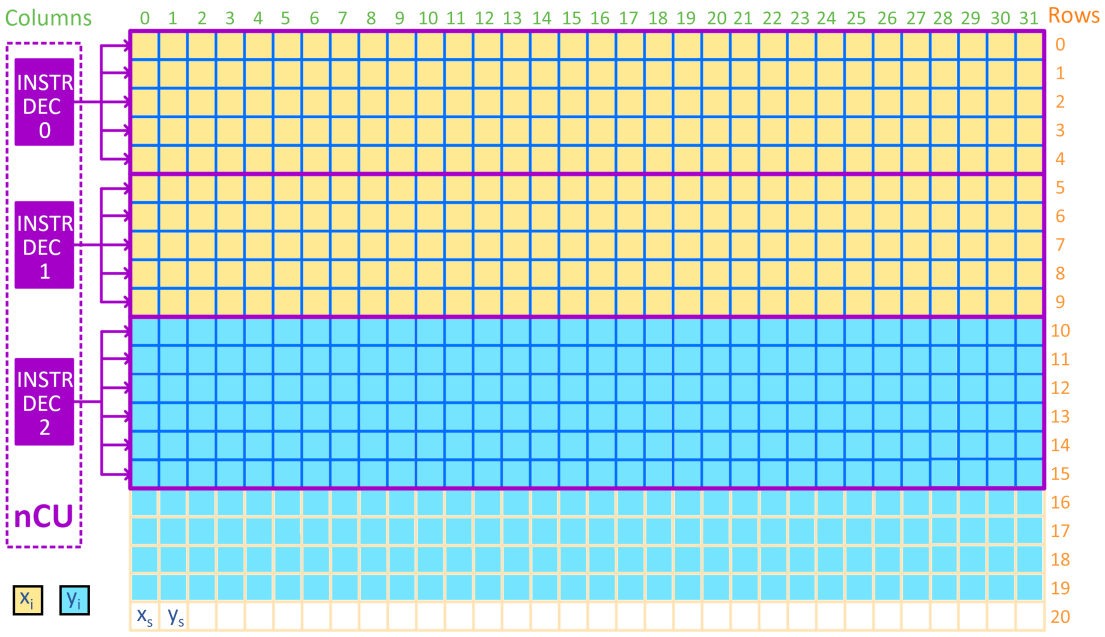


Figure 5.3: Synthesized LiM Array content for the K-NN benchmark.

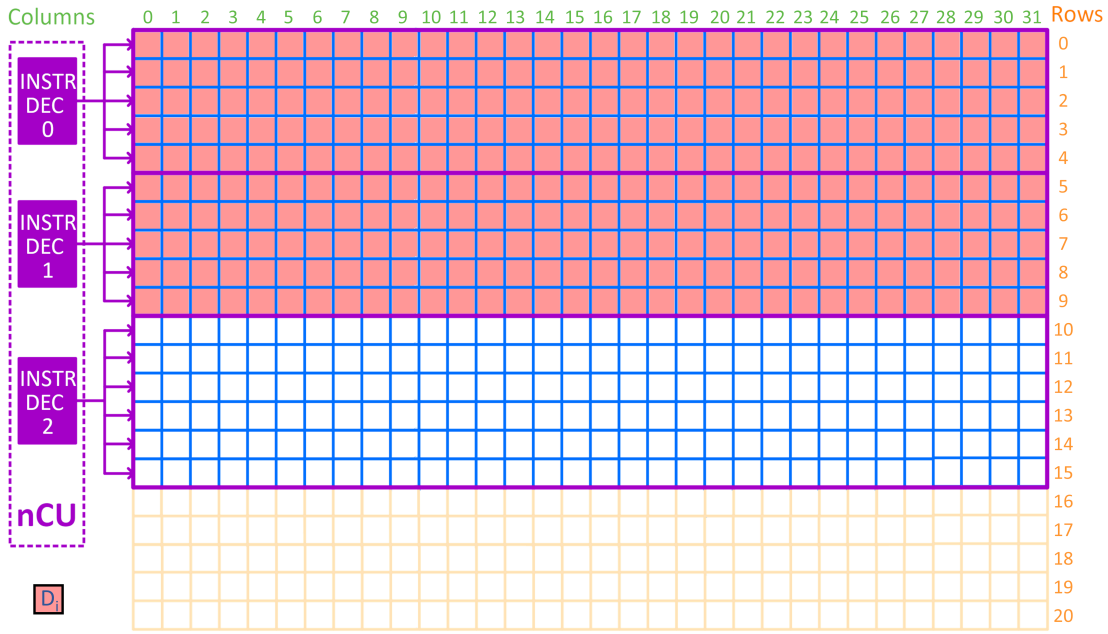


Figure 5.4: Results placement inside the LiM Array for the K-NN benchmark.

Here, the K-NN LiM program is explicitly reported as a quick guided example for pointing out the functioning of the GP-LiMA Unit and how an algorithm can be translated into an GP-LiMA program. The explanation is organized so that first the specific LiM instruction is declared and then it follows a comment illustrating the operations and data transfers implied inside the LiM Matrix. So, a list of all the program instructions is drafted, and for each of them the different `Instruction` macro-fields are highlighted according to this colour association: `uInstruction`, `nInstruction_0`, `nInstruction_1`, and `nInstruction_2`.

- ***Step 1***: Compute all the $x_s - x_i$ terms and save them into the RF1s (RF1: register at address 0 inside the RF).

```
init_instructions(1) <=
  "01"&std_logic_vector(to_unsigned(2,SIZE_uROM_Address))&
    '0'&'0'&All_Col_Enabled&'1'&
  "11111"&OP_ALU&MI_int_Mem&DEST_RF&nullADDR_S1&
    std_logic_vector(to_unsigned(640,SIZE_ADDR_S2))
      &std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&SUB_OP&
  "11111"&OP_ALU&MI_int_Mem&DEST_RF&nullADDR_S1&
    std_logic_vector(to_unsigned(640,SIZE_ADDR_S2))
      &std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&SUB_OP&
  Disable_instr_dec_last;
```

Since all the x_i values are contained in the first 10 smart rows, `nInstruction_0` and `nInstruction_1` enable them setting the bits in their `EN ROW` field to 1. To complete their enabling, all the bits in the `enCol` field of the `uInstruction` are also set to 1. While, the entire set of smart rows controlled by the last `n_Instruction`, i.e. the last 6 smart rows, is disabled (`nInstruction_2` = `Disable_instr_dec_last`). Then, all the Smart Blocks belonging to the first 10 smart rows have to perform the same action, that is why `nInstruction_0` and `nInstruction_1` are equal. Each enabled Smart Block selects as value for the `Result` signal the one at output of the ALU (`OPCODE = OP_ALU`), which, in turn, is the outcome of a subtraction (`FUNC = SUB_OP`) between the content of the Block Word, i.e. x_i, and the value forwarded by the MI (`SOURCE OP = MI_int_Mem`), i.e. x_s. In particular, the MI provides the content of the Block at address 640 (`ADDR S1 = std_logic_vector(to_unsigned (640,SIZE_ADDR_S2))`). Then, the `Result` value is stored into the register inside the RF (`DEST OP = DEST_RF`) pointed by the address written into the `ADDR D` field (specifically, into the register at address 1, being `ADDR D = std_logic_vector(to_unsigned(1,SIZE_ADDR_D))`).

Then, in the `uInstruction` the next instruction to be executed is coded through a combination of values of different fields. The `uSEQcnfg` is set to `"01"`, meaning that the address for accessing the IMem must be taken from the `nextADD_uIR` field, that in this case points to the location at address 2.

Besides, the `Request` field is set to `'1'` while the `last_uOP` to `'0'`, explicating that the current instruction is not a `wait` one. Lastly, the `fetchEN` field is fixed at `'1'` to indicate that the program will last at least for one instruction more.

- ***Step 2***: Compute all the $|x_s - x_i|$ terms and save them into the RF1s.

```
init_instructions(2) <=
  "01"&std_logic_vector(to_unsigned(3,SIZE_uROM_Address))&
    '0'&'0'&All_Col_Enabled&'1'&
  "11111"&OP_ALU&RFA&DEST_RF&
    std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&nullADDR_S2&
      std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&ABS_OP&
  "11111"&OP_ALU&RFA&DEST_RF&
    std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&nullADDR_S2&
      std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&ABS_OP&
  Disable_instr_dec_last;
```

Also during this instruction, only the Smart Blocks in the first 10 smart rows are enabled to take the outcomes of the ALUs and save them into the RF1s. However, in this case, the function performed is the absolute value (`FUNC = ABS_OP`) of the RF1 content (`SOURCE OP = RFA , ADDR S1 = std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))`) that is $x_s - x_i$.

- ***Step 3***: Load all the y_i values into the related Bypass Storages to make them accessible by the upper Smart blocks through the Rdt Int.

```
init_instructions(3) <=
  "01"&std_logic_vector(to_unsigned(4,SIZE_uROM_Address))&
    '0'&'0'&All_Col_Enabled&'1'&
  Disable_instr_dec&
  Disable_instr_dec&
  "111111"&OP_Load&Mem&DEST_Bypass&nullADDR_S1&nullADDR_S2&
    nullADDR_D&nullFunc;
```

To prepare the data for the parallel computation of all the $|ys - yi|$ terms, carried out by the Smart Blocks inside the first 10 smart rows, the data delivery mechanism of the reduction tree interconnections must be exploited and so initialized. In particular, in the next instructions, the column interconnections are used by the first 320 Smart Blocks to fetch in parallel all the values y_i stored in the 320 Blocks below them (each of the first Smart Blocks will take a different y_i value). For this reason, the last 6 smart rows Bypass Storages, that are the registers from which the Red Int take the input data, must be updated with the y_i values saved in the associated Block Words. If follows that during this LiM instruction the first 10 smart rows will be disabled (`nInstruction_0` = `nInstruction_1` = `Disable_instr_dec&`) , while the last 6 smart rows will copy the content of the Block Word into the Bypass Storage (`DEST OP = DEST_Bypass`).

- ***Step 4***: Preserve the LiM Array content of the Smart Blocks that will overwrite their Block Word with the values of the final results. Copy the content of these Block Words into the associated RF0.

```
init_instructions(4) <=
  "01"&std_logic_vector(to_unsigned(5,SIZE_uROM_Address))&
    '0'&'0'&All_Col_Enabled&'1'&
  "11111"&OP_Load&Mem&DEST_RF&nullADDR_S1&nullADDR_S2&
    std_logic_vector(to_unsigned(0,SIZE_ADDR_D))&nullFunc&
  "11111"&OP_Load&Mem&DEST_RF&nullADDR_S1&nullADDR_S2&
    std_logic_vector(to_unsigned(0,SIZE_ADDR_D))&nullFunc&
  Disable_instr_dec_last;
```

Here, again, only the Smart Blocks belonging to the first 10 smart rows are enabled. Since the `Op_load` instruction is called specifying `Mem` as source operand and `DEST_RF` as destination, all the enabled Smart Blocks copy the data stored in their Block Work into the RF0.

- ***Step 5***: Compute all the $y_s - y_i$ terms and save them into the Block Words.

```
init_instructions(5) <=
  "01"&std_logic_vector(to_unsigned(6,SIZE_uROM_Address))&
    '0'&'0'&All_Col_Enabled&'1'&
  "11111"&OP_ALU&MI_int_Col_int&DEST_Mem&
    std_logic_vector(to_unsigned(10,SIZE_ADDR_S1))&
      std_logic_vector(to_unsigned(641,SIZE_ADDR_S2))&
        nullADDR_D&SUB_OP&
  "11111"&OP_ALU&MI_int_Col_int&DEST_Mem&
    std_logic_vector(to_unsigned(10,SIZE_ADDR_S1))&
      std_logic_vector(to_unsigned(641,SIZE_ADDR_S2))&
        nullADDR_D&SUB_OP&
  Disable_instr_dec_last;
```

Each of the Smart Blocks in the first 10 rows takes a different y_i value from the column interconnections and subtract it to the y_s value forwarded by the MI (`OPCODE=OP_ALU, FUNC=SUB_OP, SOURCE OP=MI_int_Col_int`). Thanks to the column interconnections, simultaneously, each enabled smart row takes the

content of the row that is placed 10 position below, as specified through the ADDR S2 field of nInstruction_0 and nInstruction_1 that is equal to std_logic_vector (to_unsigned(10,SIZE_ADDR_S1)). While the same y_s term is forwarded to all the enabled Smart Blocks thanks to the MI that fetches it from the Block at address 641 ADDR S1 = std_logic_vector(to_unsigned (641,SIZE_ADDR_S2)). Then, all the results $y_s - y_i$ are saved in all the Block Words of the first N Smart Blocks (DEST OP = DEST_Mem).

- ***Step 6***: Compute all the $|y_s - y_i|$ terms and save them into the Block Words.

```
init_instructions(6) <=
  "01"&std_logic_vector(to_unsigned(7,SIZE_uROM_Address))&
    '0'&'0'&All_Col_Enabled&'1'&
  "11111"&OP_ALU&Mem&DEST_Mem&nullADDR_S1&nullADDR_S2&
    nullADDR_D&ABS_OP&
  "11111"&OP_ALU&Mem&DEST_Mem&nullADDR_S1&nullADDR_S2&
    nullADDR_D&ABS_OP&
  Disable_instr_dec_last;
```

All the first N Smart Blocks take the content in their Block Word (SOURCE OP = Mem) and pass it to the ALU that returns its absolute value (OPCODE = OP_ALU, FUNC = ABS_OP) that is then saved again into the Block Word (DEST OP = DEST_Mem).

- ***Step 7***: Compute all the $|x_s - x_i| + |y_s - y_i|$ terms saving the results into the Block Words.

```
init_instructions(7) <=
  "00"&std_logic_vector(to_unsigned(14,SIZE_uROM_Address))&
    '1'&'0'&All_Col_Enabled&'1'&
  "11111"&OP_ALU&Mem_RFA&DEST_Mem&
    std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
    nullADDR_S2&nullADDR_D&SUM_OP&
  "11111"&OP_ALU&Mem_RFA&DEST_Mem&
    std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
    nullADDR_S2&nullADDR_D&SUM_OP&
  Disable_instr_dec_last;
```

All the Smart Blocks in the first 10 rows take the associated $|y_s - y_i|$ value from their Block Word and sum it to the $|x_s - x_i|$ saved in the related RF1 and, finally, write back the result into their Block Word.
Then, in the `uInstruction`, to flag that this is the last instruction of the program, the `fetchEN` bit is set to `'1'` and the `uSEQcnfg` field to `"00"`, while the `nextADD_uIR` field points to the location where the `wait` instruction is stored.

- **Wait Instruction**

```
init_instructions(14) <=
  "00"&std_logic_vector(to_unsigned(14,SIZE_uROM_Address))&
    '0'&'1'&All_Col_Enabled&'0'&nullNinstr;
```

Above, the compulsory `wait` instruction that closes all the LiM programs is presented. It is specified through a specific setting for the fields of the `uInstruction` fields: the `Request` bit must be set to `'0'`, `last_uOP` to `'1'`, `uSEQcnfg` to `"00"` and `fetchEN` to `'0'`, while the `nextADD_uIR` must contain the address of the `wait` instruction itself.

5.2.2 MVM: Matrix-Vector Multiplication

Given N matrices $\overline{\overline{X}} \in \mathbb{R}^{u \times v}$ and N vectors $\overline{Y} \in \mathbb{R}^{v \times 1}$, compute N matrix multiplications, each on a different couple of $\overline{\overline{X}}$ and $\overline{Y}$, evaluating for each product all the u elements z_i of the resulting vector $\overline{Z} \in \mathbb{R}^{u \times 1}$:

$$z_i = \sum_{j=0}^{v-1} x_{i,j} y_j \ , \ with \ i = 0, \ 1, \ ..., \ u\text{-}1 \tag{5.2}$$

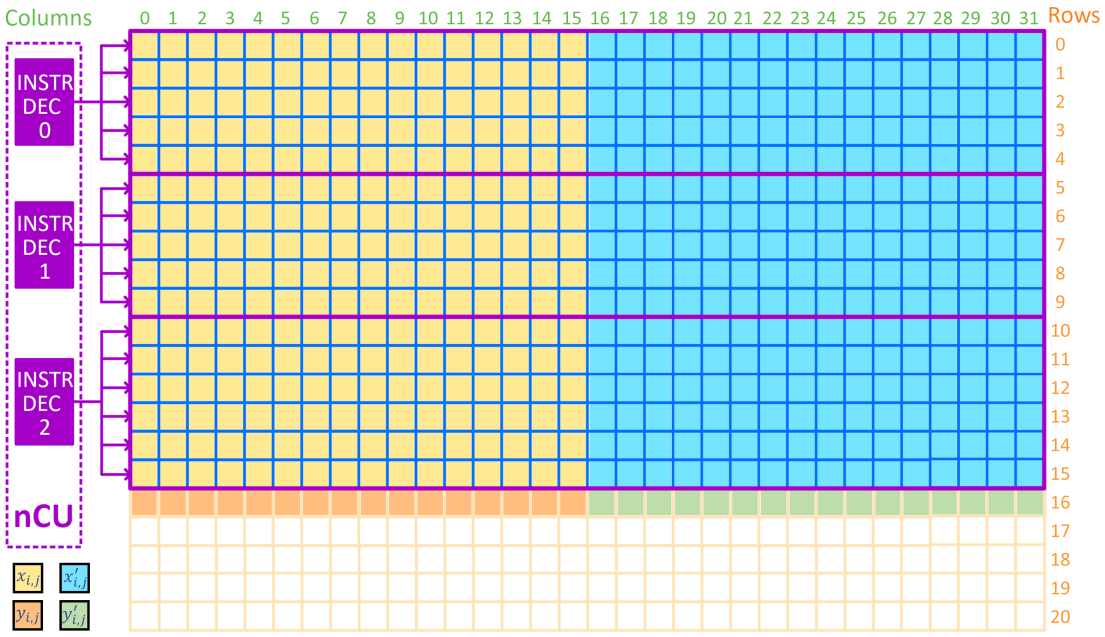


Figure 5.5: Synthesized LiM Array content for the MVM benchmark.

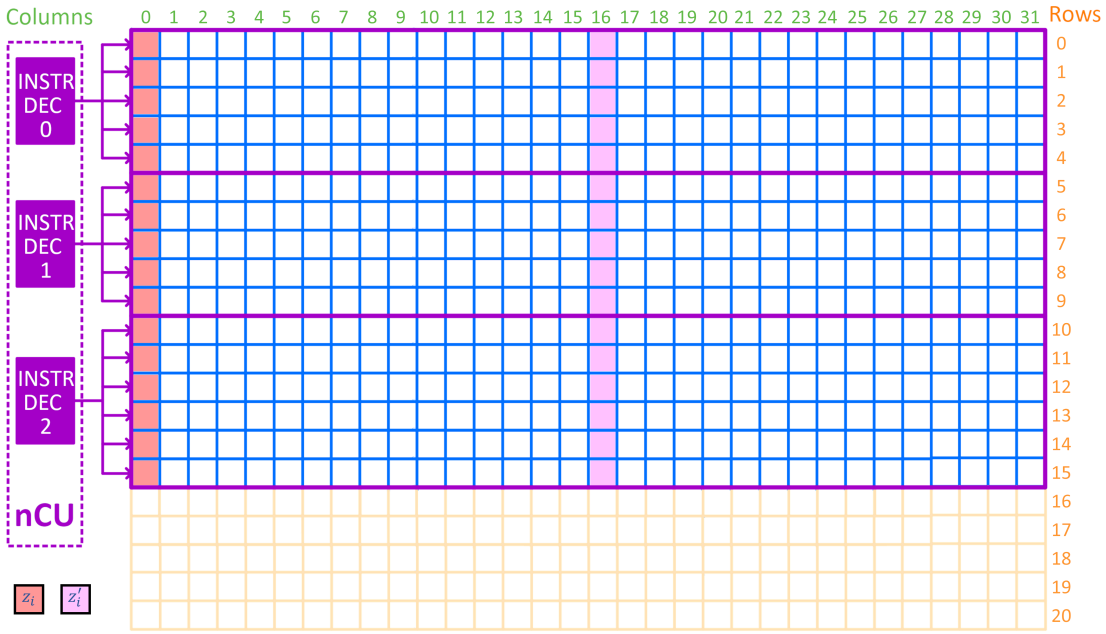


Figure 5.6: Results placement inside the LiM Array for the MVM benchmark.

Tabella con gli step.

Steps	***#Instructions***
1. Computation of all $x_{i,j} * y_j$ products, one smart row at a time. In the first `Instruction` the Smart Blocks belonging to the first smart rows multiply the content of their Block Word, i.e. $x_{0,j}$, by the value coming from the Column interconnections which forward the content of the first standard row, namely the y_j values. The outcomes are then stored in the Bypass Storages of the first smart row. The next `Instruction` drives the same operations that this time are performed only by the second smart row, so taking the $x_{0,j}$ values. The following `Instructions` repeat the same step for all the remaining $x_{i,j}$ values.	u

2.	LiM content preservation. The values contained in all the Block Words are copied into the related RF0s.	1
3.	Final sums computations to retrieve the z_i values. Each smart row performs in parallel to the others the sum on N values exploiting the row interconnections to implement the operation following the reduction tree organization. In the first `Instruction` each of the Smart Blocks in the even columns take the content of the Smart Block on their right and sum it to the one of their Bypass Storage. Then the result is stored again in their Bypass Storage. In the following instruction all the Smart Blocks in the columns that are multiples of 4 will sum the content of their Bypass Storage to the value of the Bypass Storage in the Smart Block placed at 2 positions right from them. This operations are repeated for $log_2 v$ times to reach the z_i final values. Note that all the instructions are interleaved with a `nullOP Instruction` to avoid the data hazard linked to the row interconnections usage.	$2 \times log_2 v - 1$
	# Instructions MVM	$u + 2 \times log_2 v$

5.2.3 K-means

Given a 2-dimentional dataset composed of N samples (x_i,y_i) and K centroids (x_{cj},y_{cj}), find for each of the sample the closest centroid and save both the related distance value and the number of the centroid to which that distance refers. Implement the requested task following the procedure below.

```
for j in K centroids (x_cj, y_cj) :
    for i in N samples (x_i, y_i) :
       D_i = (x_i - x_cj)^2 + (y_i - y_cj)^2;
       if j = 0 :
          ClusterID_i = j;
          SmallerD_i = D_i;
       else :
          if D_i < SmallerD_i-1 :
             ClusterID_i = j;
             SmallerD_i = D_i;
```

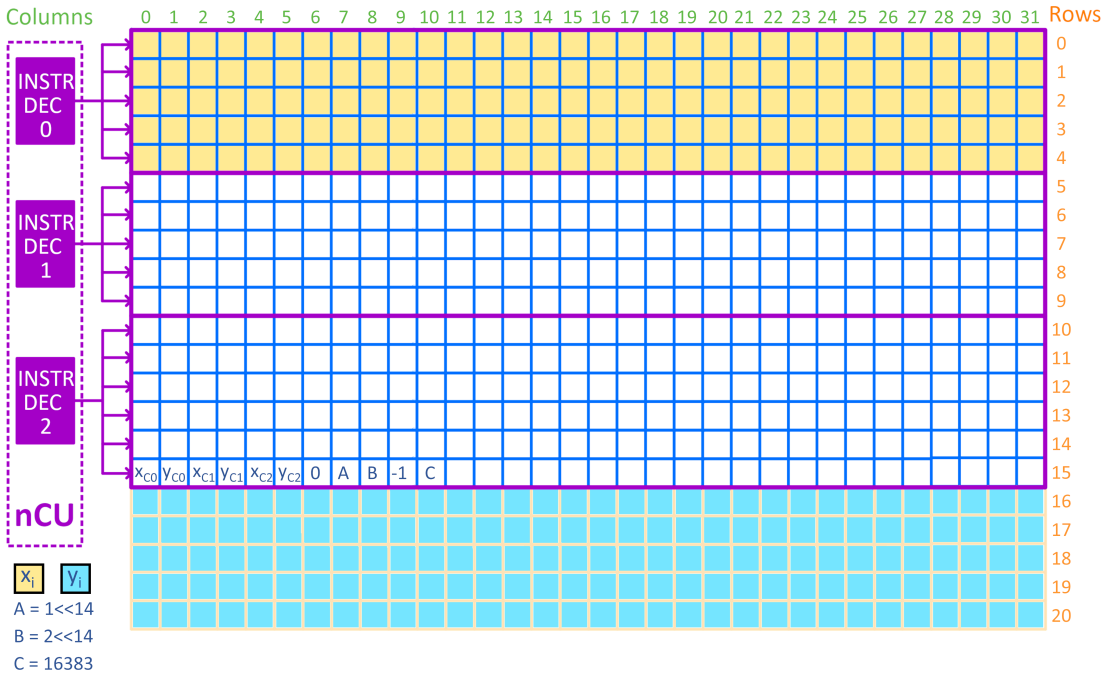


Figure 5.7: Synthesized LiM Array content for the K-means benchmark.

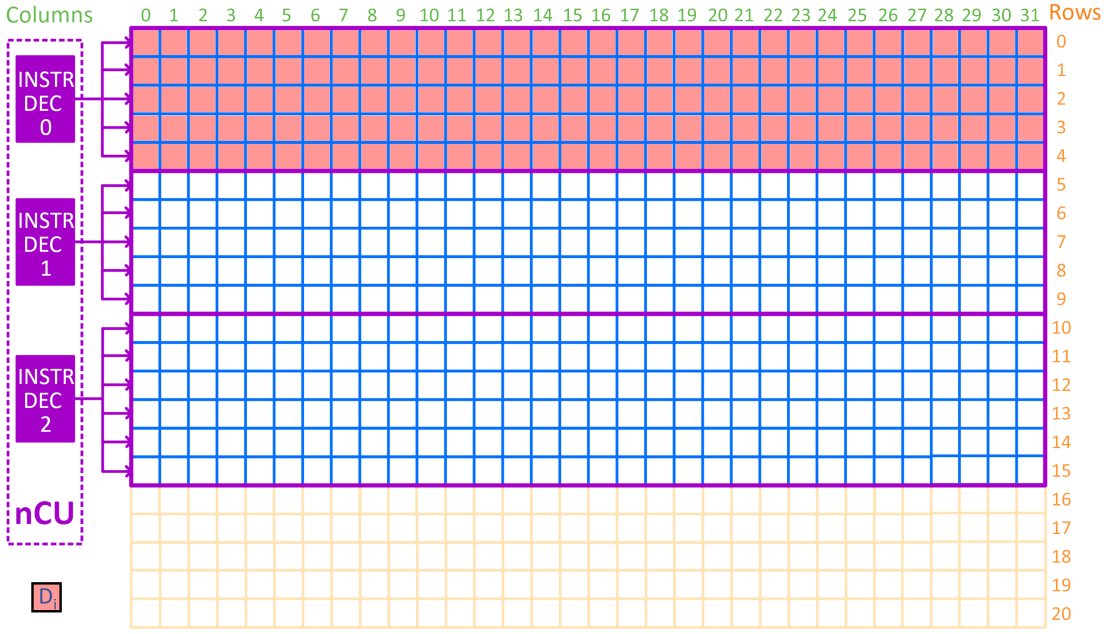


Figure 5.8: Results placement inside the LiM Array for the K-means benchmark.

Steps	***#Instructions***
1. LiM matrix initialization. The x_i values inside the Block Words of the first 5 smart rows are copied into the related Bypass Storages.	1
2. LiM content preservation. The data hold by the Block Words of the first 5 smart rows are saved inside the related RF0.	1
3. Simultaneous computation of all the $\|x_{cj} - x_i\|$ terms. In the first `Instruction`, each of the first N Smart Blocks will take the x_i value from its Block Word and subtract it to the x_{c0} value forwarded by the MI, saving then the result into the RF1. While, each of the following N Smart Blocks will take the x_i value from the column interconnections and subtract it to the x_{c1} value forwarded by the MI, saving the result into the RF1, and so on for all the remaining centroids. In the next `Instruction` all the $x_{cj} - x_i$ will be taken from the RF1 and their absolute value will be computed and stored in the Bypass Storages.	2
4. Computation of the $\|y_{cj} - y_i\|$ terms for all the centroids at the same time. This step is performed similarly to the previous one with the exception that for all the Smart Blocks the y_i values are retrieved from the Column interconnections. The final results will be stored in the RF1s.	2

5.	Simultaneous distances evaluation ($\|x_{cj} - x_i\| + \|y_{cj} - y_i\|$) for all the centroids. Each Smart Block takes the content of its Bypass Storage from the column interconnections and sums it to the data held by the RF1. Then all the smart rows connected to the first instruction decoder save the result into the RF1, while the others into the Bypass Storage.	1
6.	Association of each distance with the centroid it refers to. All the Smart Blocks insert in the MSB of the distance data they hold the ID value to which that distance is referred, by performing an OR-masking operation between the data and the specific ID value, stored in the last smart rows, properly forwarded by the MI. The final values are, then, saved inside the RF2s.	1
7.	Distances comparison to associate each sample to the nearest centroid. For each sample the distances comparisons are performed according to the reduction tree mechanism, exploiting the column interconnections, so that the total number of instructions involved are proportional to $log_2 K$. Then, all the comparisons are performed in parallel for each sample independently on the others.	$8 \times \lceil log_2 K \rceil - 1$
	# Instructions K-means	$8 \times \lceil log_2 K \rceil + 7$

5.2.4 Mean & Variance

Given a set of N data, compute the mean μ and the variance σ^2:

$$\mu = \sum_{i=0}^{N-1} \frac{x_i}{N} \quad ; \quad \sigma^2 = \frac{\sum_{i=0}^{N-1}(x_i - \mu)^2 - \frac{\left[\sum_{i=0}^{N-1}(x_i-\mu)\right]^2}{N}}{N} \tag{5.3}$$

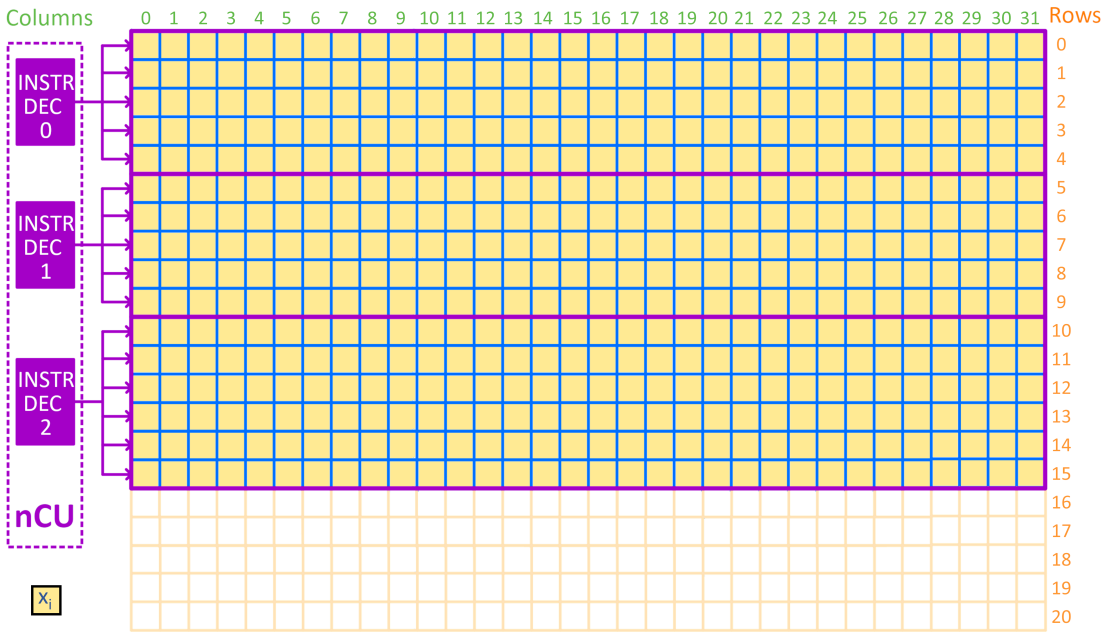


Figure 5.9: Synthesized LiM Array content for for the µ&σ^2 benchmark.

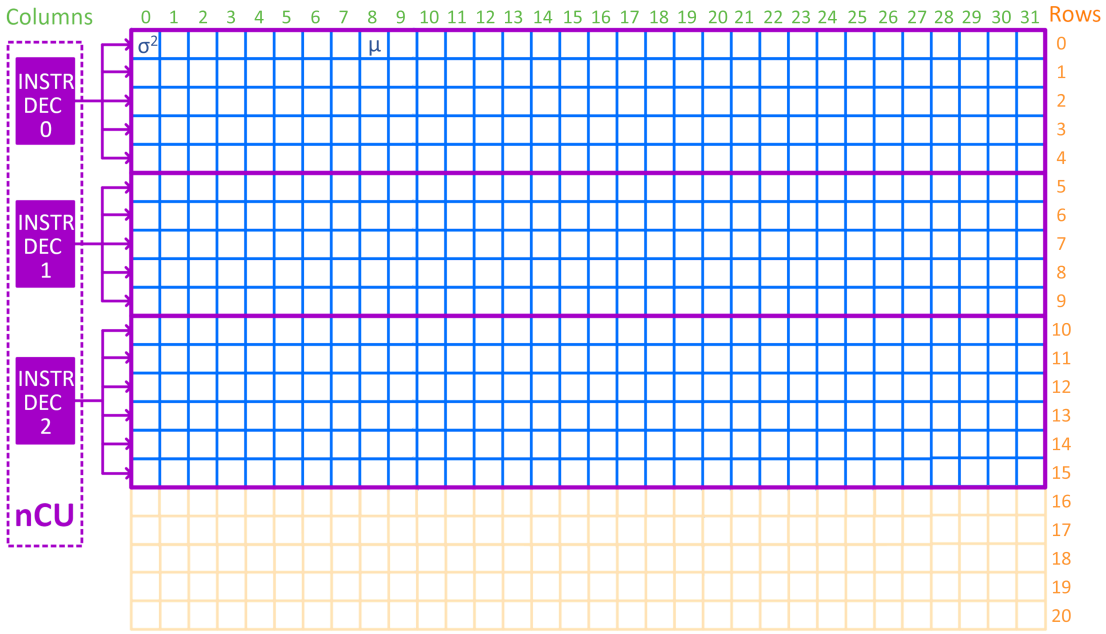


Figure 5.10: Results placement inside the LiM Array for the μ&σ^2 benchmark.

Steps	***#Instructions***
1. Data preparation for the $\sum_{i=0}^{N-1} x_i$ operation. All the x_i values contained in the Block Words are saved into the related Bypass Storages.	1
2. LiM content preservation. Block Word data are saved into the RF0s.	1

3.	Evaluation of $\sum_{i=0}^{N-1} x_i$ performed through the reduction tree computing mechanism, as done for the MVM benchmark. At first, all the smart rows compute in parallel the sum among the data in the Smart Blocks they include by using the row interconnections. In the end, the final results will be saved in the Bypass Storages of the first column. Then, all these values will be summed up leveraging the column interconnection, so that the final sum value will be stored in the Block Word of the first Smart Block. It follows that the entire step 3 is made up of $log_2 N$ `Instructions`, all interleaved with a `nullOP Instruction` to prevent the occurrence of data hazards.	$2 \times log_2 N - 1$
4.	μ evaluation through the RShifter mechanism. The `OP_RShifter` operation is called taking as source operand the content of the first Block Word through the MI. The outcome is then saved inside the Block Word. For $N = 512$, the actual value for μ will be contained in the Smart Block at address 8.	2
5.	Computation of all the $x_i - \mu$ terms. All the Smart Blocks take the μ value from the MI and subtract it to the content of the RF0s, i.e. x_i, saving the results into the Bypass Storages. Note that before performing the subtractions, a `nullOP` must be inserted to allow the MI to fetch the updated value from the Block Word.	2

6.	Storage of the $x_i - \mu$ terms inside the RF1s. All $x_i - \mu$ are recomputed and saved into the RF1s to keep their values ready for the computation of $\sum_{i=0}^{N-1}(x_i - \mu)^2$.	1
7.	Computation of $\sum_{i=0}^{N-1}(x_i - \mu)$ performed following the same procedure at step 3. The final result is stored in the Block Word of the first Smart Block.	$2 \times log_2 N - 1$
8.	Evaluation of $\left[\sum_{i=0}^{N-1}(x_i - \mu)\right]^2$. Only the first Smart Block is enabled, which takes the content of the Block Word and multiplies it by itself storing, then, the result into the Bypass Storage.	1
9.	Evaluation of $\frac{\left[\sum_{i=0}^{N-1}(x_i-\mu)\right]^2}{N}$ carried out following the same procedure of step 4. The outcome is saved into the RF2 of the Smart Block at address 8.	2
10.	Simultaneous computation of the $(x_i - \mu)^2$ terms. Each Smart Block takes the RF1s value and multiply it by itself and store the result into the related Bypass Storage.	2
11.	$\sum_{i=0}^{N-1}(x_i - \mu)^2$ is evaluated following the sum reduction tree procedure at step 3. The result is saved into the Block Word of the first Smart Block.	$2 \times log_2 N - 1$
12.	The $\frac{\left[\sum_{i=0}^{N-1}(x_i-\mu)\right]^2}{N}$ value contained in the RF2 of the 9th Smart Block is loaded the associated Bypass Storage.	1

13.	Evaluation of $\sum_{i=0}^{N-1}(x_i-\mu)^2-\frac{\left[\sum_{i=0}^{N-1}(x_i-\mu)\right]^2}{N}$. The first Smart Block takes the content of the Bypass Storage in the Smart Block placed at a distance of 8 positions to its right and subtracts it to the value contained in its RF2. The outcome is then saved into the Bypass Storage of the first Smart Block.	2
14.	Final σ^2 computation performed exploiting the RShifter mechanism in step 4. The final result will be available from the Block Word of the first Smart Block.	4
	# Instructions μ & σ^2	$6 \times log_2 N + 16$

5.2.5 DFT: Discrete Fourier Transform

Given a set of N timing samples x_i, compute the k-th frequency component X_k:

$$X_k = \sum_{i=0}^{N-1} x_i \times \left[\cos\left(\frac{2\pi ik}{N}\right) - j\sin\left(\frac{2\pi ik}{N}\right)\right] \tag{5.4}$$

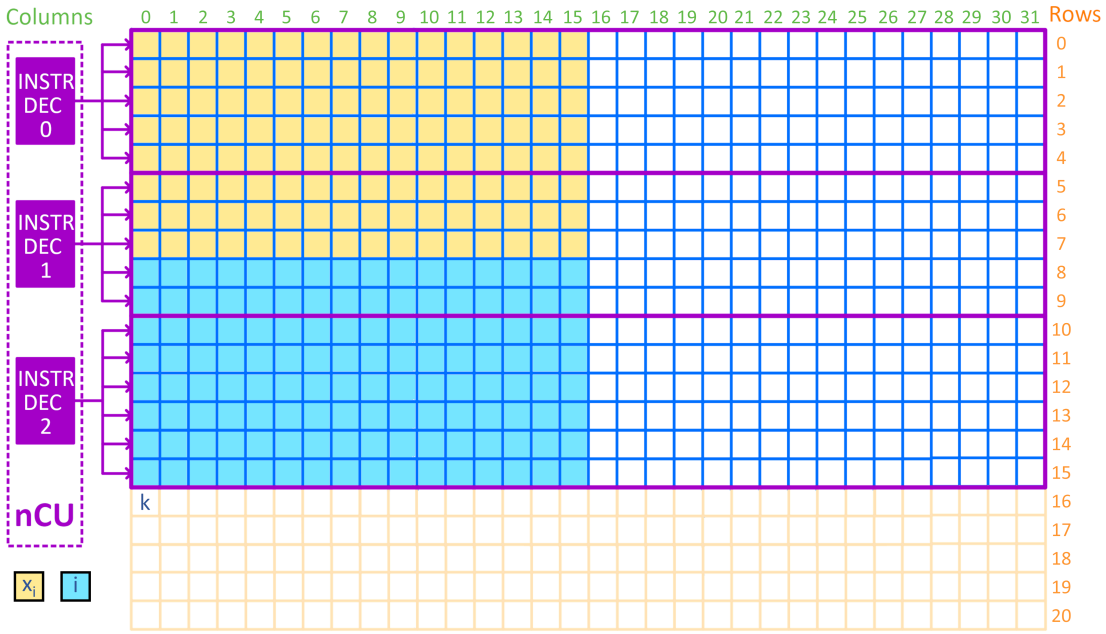


Figure 5.11: Synthesized LiM Array content for the DFT benchmark.

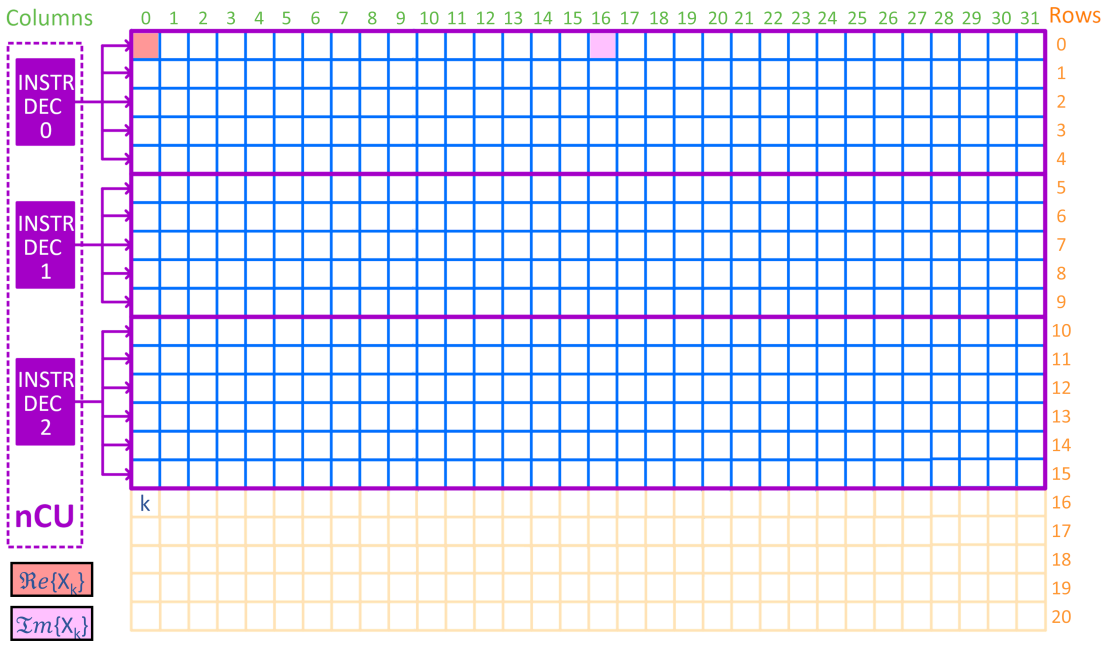


Figure 5.12: Results placement inside the LiM Array for the DFT benchmark.

Steps	***#Instructions***
1. Data preparation and LiM content preservation. All the x_i values present in the Smart Blocks belonging to the left half of the LiM Array are copied into the Block Words of the right half passing through the Bypass Storages connected to the row interconnections. In the first `Instruction` the content of the Block Words in the LiM Array left half is copied into the Bypass Storages. Then, a `nullOP` is instantiated before the `Instruction` that enables only the first N Smart Blocks of the LiM Array right half to take from the row interconnections the contents of the Bypass Storages on the left half and save them into their Block Words. The last `Instruction` preserves the content of the first Block Word, i.e. x_0, that will be overwritten at the end of the algorithm, by copying that value into the RF0.	4
2. Evaluation of the $i \times k$ terms. All the first N Smart Blocks in the LiM Array left half are enabled to take, through the column interconnections, the i values contained in the N Bypass Storages of the Smart Blocks below and multiply them by the k value forwarded by the MI. The results are then stored in the first N Bypass Storages belonging to the LiM Array left half.	1

3.	Computation of all the $\frac{2\pi ik}{N}$ terms. In this step only the left half of the smart section is enabled. All rows work in parallel and each of them computes a single $\frac{2\pi ik}{N}$ value each 3 *Instructions* exploiting the RShifter mechanism. At the end all the final results are contained in the Bypass Storages of the first N Smart Blocks in the left half of the LiM Array.	$\frac{3}{2}\times$#LiMCols
4.	Evaluation of the $\sin\left(\frac{2\pi ik}{N}\right)$ terms in parallel. These are computed by the first N Smart Blocks inside the right half of the smart section. The first `Instruction` is a `nullOP` one to allow the row interconnections to forward, in the next instruction, the updated values of the Bypass Storages in the first half of the LiM Array. Then, in the second `Instruction` all the enabled Smart Blocks save in their RF3 the outcomes from the LUTs, which implement the discretized sine function and which are addressed by the values $\frac{2\pi ik}{N}$ retrieved from the row interconnections.	2
5.	Evaluation of the $\cos\left(\frac{2\pi ik}{N}\right)$ terms in parallel. These are computed by the first N Smart Blocks inside the left half of the smart section. All the enabled Smart Blocks save in their RF3 the outcomes from the LUTs, which implement the discretized cosine function and which are addressed by the values $\frac{2\pi ik}{N}$ retrieved from the row interconnections.	1

6.	Evaluation of all the $x_i \times \cos\left(\frac{2\pi ik}{N}\right)$ and $x_i \times \sin\left(\frac{2\pi ik}{N}\right)$ terms in parallel. All the first $2 \times N$ Smart Blocks take the value stored in their Block Word, i.e. x_i, and multiply it by the RF3 content, saving the result in the Bypass Storage.	1
7.	Simultaneous computation of the real part and the imaginary part of X_k. The real term is obtained summing the content of all the first N Bypass Storages inside the left half of the smart section, while the imaginary term is given by the sum on all the values held by the first N Bypass Storages inside the smart section right half. Both sums are computed in parallel exploiting the reduction tree mechanism, as done for the sums on N data involved in the µ& σ^2 benchmark. At the end, the real part of X_k will be contained in the Block Word of the first Smart Block, while the imaginary part in the one of the first Smart Block inside the right half of the smart section.	$2 \times log_2 N$
	# Instructions DFT	$\frac{3}{2} \times$#LiMCols + $2 \times log_2 N + 9$

5.2.6 GP-LiMA Performances & Comparisons with the PLiM

Once all the benchmarks are defined and mapped onto the GP-LiMA it is possible to retrieve the total number of clock cycles taken for each execution so that other performance metrics can be taken into account. Table 5.7 summarizes the computation of the total number of clock cycles required to perform on the synthesized GP-LiMA each of the benchmarks previously investigated for a generic dataset. Besides, it also provides as reference the number of clock cycles taken for each benchmark to be run on the customized PLiM architectures. Note that for both the PLiM and the GP-LiMA the total number of clock cycles associated with an algorithm is computed summing the clock cycles taken by the memory initialization with the number of

instructions mapping that algorithm and the total latency of the pipelined architecture (6 clock cycles: 2 for the starting the GP-LiMA processing mode, 1 for the instruction fetch, 1 for the decoding, 1 for the data elaboration, and 1 for the results saving inside the LiM Array itself). Note that, in all the cases, the term which strongly contributes to the total amount of required time is given by the number of clock cycles employed to initialize the LiM array content, that is when the LiM Array is used as a standard data memory.

Algorithms	**LiM**	**# LiM init. clock cycles**	**# Algorithm instructions**	**# Algorithm clock cycles**
K-NN	PLiM	$2 \times N + 2$	6	$2 \times N + 14$
	GP-LiMA	$2 \times N + 2$	7	$2 \times N + 15$
MVM	PLiM	$2 \times u \times v$	$4 + 2 \times (v - 1)$	$(2+2 \times u) \times v + 8$
	GP-LiMA	$N \times v \times (u + 1)$	$u + 2 \times log_2 v$	$N \times v \times (u+1) + u + 2 \times log_2 v + 6$
K-means	PLiM	$2 \times N + 3 \times K$	$8 \times K - 1$	$2 \times N + 11 \times K + 5$
	GP-LiMA	$2 \times N + 3 \times K + 2$	$8 \times \lceil log_2 K \rceil + 7$	$2 \times N + 3 \times K + 8 \times \lceil log_2 K \rceil + 15$
$\mu\&\sigma^2$	PLiM	$N + 2$	$6 \times N + 10$	$7 \times N + 18$
	GP-LiMA	N	$6 \times log_2 N + 16$	$N + 6 \times log_2 N + 22$
DFT	PLiM	$4 \times N + 5$	$2 \times N + 7$	$8 \times N + 18$
	GP-LiMA	$2 \times N + 1$	$1.5\times$ #LiMCols $+2 \times log_2 N + 9$	$2 \times N + 1.5\times$ #LiMCols $+2 \times log_2 N + 16$

Table 5.7: Detail on the number of clock cycles required by each tested benchmark to complete its execution on the GP-LiMA and on the PLiM devices.

Then, the results for the post-place&route analysis concerning the algorithm-dependent metrics for the synthesized GP-LiMA (with clock gating and memory parallelism of 16 bits) are reported in Table 5.8. For each benchmark, the first row specifies the values for the parameters defining the specific dataset taken into account, chosen to make the most of the GP-LiMA processing potentialities. Then, the remaining rows specify the execution times, obtained fixing the clock frequency to the maximum value allowed (232.55 MHz), and the worst case energy values being the GP-LiMA worst case power equal to 690.79 mW. However, at a first glance nothing can be concluded for the GP-LiMA processing efficiency just by referring to the performance

Algorithms	**K-NN**	**MVM**	**K-means**	**μ&σ²**	**DFT**
Parameters	N = 320	u = 16, v = 16 t = 1, N = 2	N = 160 K = 3	N = 512	N = 128
Execution Time [#Clock Cycles]	655	574	360	588	334
Execution Time [μs]	2.81	2.46	1.54	2.52	1.43
Worst Case Energy [μJ]	1.94	1.69	1.06	1.74	0.99
Back Annotated Power [mW]	86.14	102.01	121.55	88.23	165.05
Back Annotated Energy [nJ]	242.05	250.94	187.18	222.33	236.02

Table 5.8: Performance achieved by the GP-LiMA after the place&route process for all the benchmarks, with: Minimum Clock Period = 4.3 ns and Worst Case Power = 690.79 mW.

values achieved for different benchmark. This because the number of elaborated samples strongly affects the execution time of the whole algorithm, especially due to the (at least) linear dependence of the memory initialization time with the number of samples composing the dataset to be stored, which, besides, has nothing to do with the actual processing capabilities of the GP-LiMA Unit. Thus, since the considered datasets size heavily changes depending on the run algorithm, to make a critical analysis on the actual GP-LiMA performance and its programming flexibility, the execution time and the worst-case energy values for each benchmark are next reported on a per-sample basis and compared with the ones accomplished by the PLiM customized architectures (see Figure 5.13 and Figure 5.15, respectively). However, from Table 5.8 they can still be retrieved some information about the expenses averages of the GP-LiMA architecture when it is exploited to the full. Concerning the execution times, between the fastest and the slowest algorithm, there is a difference of 1.38 μs on an average value of 2.15 μs, while for the energy consumption the values range from 0.99 μJ to 1.74 μJ showing a mean value of 1.48 μJ. Furthermore, to have more realistic estimations about the actual energy consumptions, the back annotated power values for each benchmark, together with the resulting back annotated energy values, are detailed at the bottom of Table 5.8. The power goes from a minimum of 86.14 mW to a maximum of 165.05 mW, with

an average value of about 112 mW that corresponds to 16% of the worst-case estimation. Similarly, also the back annotated energy average value (227.7 nJ) is about 15% of the worst-case value.

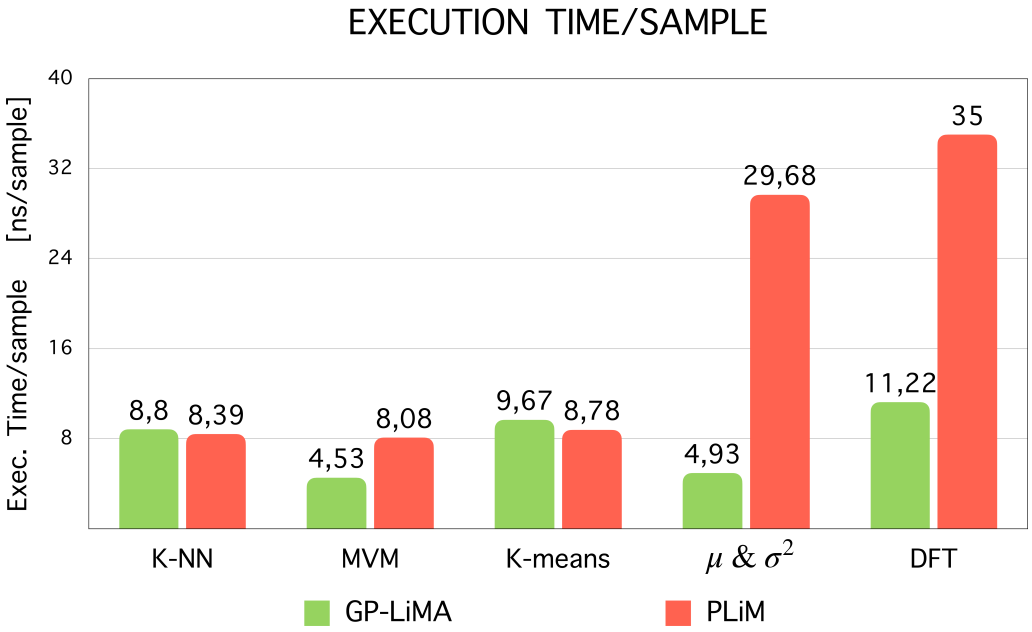


Figure 5.13: Comparison between the GP-LiMA and the PLiM architecture based on the execution time reached by all the run benchmarks on a per-sample basis. The results refer to a clock period of 4.3 ns in case of the GP-LiMA values, while in case of the PLiM architectures the clock period ranges from 3.92, in case of the MVM benchmark, to 4.21 in case of the DFT.

As previously anticipated, Figure 5.13 and Figure 5.14 illustrate the GP-LiMA execution time and the power consumption trends, respectively, according to the run benchmarks on a per-sample basis.
Concerning the timing values, they range from 4.53 ns/sample for the MVM algorithm to 11.22 ns/sample for the DFT. The benchmarks can be divided into two main groups: one including K-NN, K-means, and DFT, and another given by the MVM and the $\mu\&\sigma^2$. The first is the one characterized by execution times per sample that doubles the ones of the benchmarks enclosed in the second group. However, the main cause for this timing difference does not lie in the actual GP-LiMA processing adaptability, but again it is traceable to the memory initialization phase contribute. Indeed, the first benchmarks group collects algorithms dealing with a

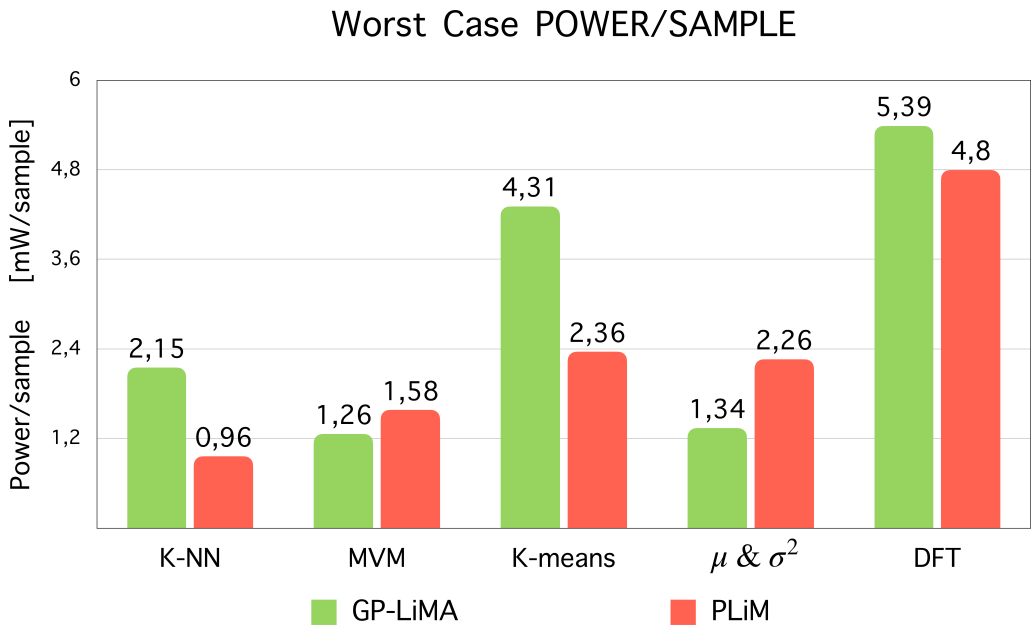


Figure 5.14: Comparison between the GP-LiMA and the PLiM architecture based on the worst case power consumed by all the run benchmarks on a per-sample basis.

two-dimensional dataset, meaning that for each sample two values have to be stored inside the LiM Array, while both the MVM and the μ&σ^2 save only one value for each sample. It follows that to compute the execution time/sample for the algorithms belonging to the first group, the total execution time is divided by a number that is half the one of the second group benchmarks, under the same number of samples. Thus, if only the number of LiM instructions employed to map each benchmark is taken into account, the actual benchmarks order organized according to the related GP-LiMA processing efficiency is given by: K-NN, MVM, K-means, μ&σ^2, and DFT. In fact, the K-NN requires only 7 instructions, independently of the elaborated dataset size, while the DFT requires a number of instructions equal to $1.5\times$ #LiMCols $+2\times log_2 N+9$, with N that is the number of analysed samples. It follows that, as it can be expected because of the M-SIMD computing mode, the GP-LiMA performs to its best for algorithms that can be fully parallelized, while it is still able to quite efficiently manage applications also involving sequential procedures.
Concerning the worst-case power consumptions per sample in Figure 5.14 they range from 1.26 mW/sample, for the MVM, to the 5.39 mW/sample, for the DFT. However, since the GP-LiMA used to map all the benchmark is the same, which in the

worst-case consumes always about 690 mW, the difference between these power/sample values is due only to the different dataset sizes considered for each benchmark. It follows that, in general, the same GP-LiMA structure can elaborate more samples when the MVM and the $\mu\&\sigma^2$ are run compared with the ones processable in case of the K-means and DFT.

Furthermore, to have a clearer idea of the actual GP-LiMA performances trends, the comparisons with the PLiM paradigm provided in Figure 5.13 and Figure 5.14 need to be taken into account. However, before starting the analysis some details must be stressed. While for testing the GP-LiMA paradigm the exact same structure was used to implement all the benchmarks, for the PLiM model, the achieved performances all refer to a different architectural structure customized ad-hoc for the specific benchmark at-run. Moreover, even if the addressable space of the synthesized GP-LiMA matches the one of all the PLiM architectures, the amount of smart locations composing the GP-LiMA doubles the one included in all the PLiM instances. Nevertheless, only the GP-LiMA Unit integrates the clock gating technique, which is why, looking at Figure 5.14, the GP-LiMA power values show in the average to keep up with the consumptions retrieved from the customized PLiM devices. Besides, always concerning the structural features of the synthesized GP-LiMA and all the specific PLiM architectures, it is worth mentioning that the GP-LiMA has a critical path that differs by only 0.24 ns with respect to the average of the PLiM solutions (4.3 ns vs 4.06 ns).

So, coming back to the timing analysis, looking at Figure 5.13, it can be derived that for the almost purely parallel benchmarks (K-NN and K-means) the GP-LiMA follows about the same behaviour of the PLiM, even if it is slightly worse, while for all the other algorithms including sequential procedures, in particular the sum of all the dataset samples, the GP-LiMA outperforms the customized PLiM solutions. From here, it stands out the importance of the reduction tree interconnections that make the GP-LiMA execution times be half the PLiM ones in the worst case (MVM) or even one-sixth in the best case ($\mu\&\sigma^2$). Besides, it must be noticed that the GP-LiMA succeeds in achieving lower execution times even if its maximum clock frequency does not reach exactly the values of the PLiM devices. This means that the number of LiM instructions needed to run a generic algorithm in the case of the GP-LiMA is usually much lower than the one required by the PLiM solutions.

Then, the gain reached by the GP-LiMA in terms of cut execution times directly reverses in the energy/sample values, which show performances considerably enhanced compared with the LiM case for the sequential algorithms, as Figure 5.15 depicts. For the GP-LiMA, the values for the worst-case energy/sample range from

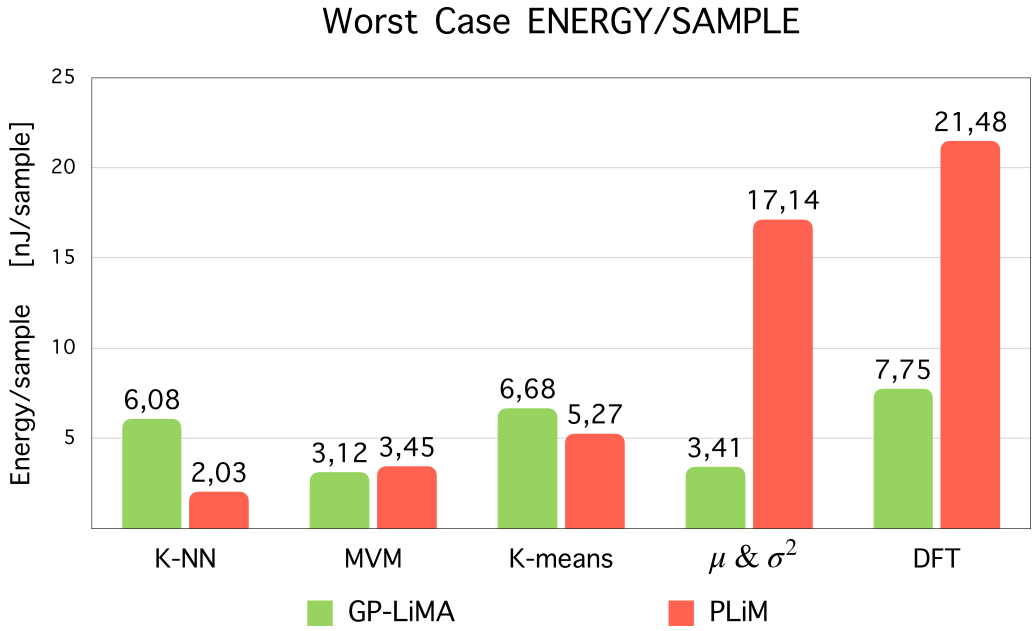


Figure 5.15: Comparison between the GP-LiMA and the PLiM architecture based on the worst case energy spent by all the run benchmarks on a per-sample basis.

3.12 nJ/sample, in the case of the MVM, to 7.75 nJ/sample for the DFT benchmark. Since the worst-case values are considered, which refer to the total worst-case power that is constant for all the benchmarks, the GP-LiMA energy/sample trend according to the algorithms traces the same of the execution time/sample. Moreover, although in the case of the PLiM model the power consumption changes depending on the algorithm, the GP-LiMA energy trend with respect to the PLiM solutions again follows the same behaviour of the execution time/sample one; namely for the K-NN and the K-means the PLiM architectures result to be more efficient than the GP-LiMA implementation, while for the MVM, but especially for the $\mu\&\sigma^2$ and the DFT the GP-LiMA energy values are way better than the PLiM ones. In particular, the DFT energy presents an improvement of 63% while in the case of the $\mu\&\sigma^2$ the GP-LiMA achieves a reduction of 80% with respect to the PLiM architecture energy. It follows that the enhancements achieved by the GP-LiMA, in terms of

strongly reduced execution times, succeed in counteracting the considerable power consumption amount leading to a cut in the total energy expense/sample. This applies specifically for the benchmarks involving sequential procedures (see DFT and $\mu\&\sigma^2$) where the reduction tree interconnections play a major role in the processing time shrinking. This means that, even if the interconnections massively impact the total power expense, consuming about 29% of the total GP-LiMA power, their integration is justified by the advantages they bring in terms of decreased processing times that, in turn, result in overall remarkable gains of energy savings.

Summarizing, the outcome of the performance comparison points out that the GP-LiMA outperforms the PLiM model in the case of more sequential algorithms, while it still retains the same PLiM advantages in terms of computing efficiency for the parallel algorithms. However, the main advantage brought by the GP-LiMA paradigm is that it succeeds in efficiently running a wider range of algorithms without requiring any hardware modification of the architectural structure, proving to hit the goal of programming flexibility combined with processing efficiency.

5.2.7 Energy comparison with a classical architecture: GP-LiMA vs RISC-V Memory Hierarchy

Finally, to demonstrate the benefits brought by the LiM paradigm usage to the GP-LiMA computing efficiency with respect to a standard CPU-centric system, in this section, the comparison with a classical von Neumann architecture based on a RISC-V ISA is discussed. The goal of a LiM architecture is to reduce the communication bottleneck between the CPU and the memory, because this unceasing motion of data implies a waste of both power and computational time. In this study, the same benchmarks tackled before are re-proposed in two different contexts: a classical von Neumann RISC-V architecture, communicating with its memory hierarchy, and a RISC-V based system embedding and using the GP-LiMA as both standard data memory and co-processor. Performance comparisons between these two solutions are reported. Since the intent is to highlight the effort required to access the memories, here, the energy values referring to the standard RISC-V system are only given by the memory accesses energy, while the processing part contribution is

completely neglected. While, for the hybrid RISC-V/GP-LiMA solution, the GP-LiMA total energy consumption is estimated considering both the energy involved for accessing the LiM Array in the standard memory mode and the energy spent during the data processing. It follows that the proposed comparison represents a best-case scenario for the RISC-V, since only the memories consumptions are considered, while it represents a worst-case scenario for the GP-LiMA, because both memory and computational units are taken into account.
To set up the addressed comparison, the classical RISC-V architecture was modelled with the Gem5 simulator [38], using a simple single-core In-Order CPU (TimingSimpleCPU) and 2 level of caches. The first cache level is made of two set-associative separated caches (instruction and data) of 1kB each and associativity equal to 2, while the second level is given by a single 8-way set associative shared cache of 256kB size (see Table 5.9).

Memory	**Size**	**Associativity**	**Access time [ns]**	**Technology**
L1DCache	1kB	2	0.207022	40nm
L1ICache				
L2SCache	256kB	8	2	40nm

Table 5.9: Parameters characterizing the memory hierarchy connected to the classical RISC-V system simulated.

Then, for all the benchmarks, the equivalent C codes were implemented and tested with Gem5, obtaining meaningful performance parameters of the modelled architecture: among them, the total number of memory accesses for each cache, as reported in Table 5.10.

Algorithm	**# L1 Cache Accesses**		**# L2 Cache Accesses**
	Data Cache	**Instruction Cache**	
K-NN	3890	24153	56
MVM	6488	35534	40
K-means	16766	98133	71
µ& σ^2	12000	56440	45
DFT	3674	19006	43

Table 5.10: Number of memories accesses required to perform each benchmark inside the classical RISC-V system.

Afterwards, a tool that thoroughly models caches and memories, i.e. Cacti by HP [39], was employed to estimate the memory consumption. Thus, the RISC-V caches were implemented on Cacti, which returned the values for both the read and write energy/access and the memory leakage (see Table 5.11).

Memory	**Energy/Access [pJ]**			**Leakage/Access [mW]**
	Read	**Write**	**Average**	
L1DCache	2.68525	3.48643	3.08584	3.3368
L1ICache				
L2SCache	365	407	386	313.698

Table 5.11: Energy values involved by the RISC-V memory hierarchy described in Table 5.9.

Then, as previously anticipated, the classical RISC-V system energy consumption for a given algorithm was evaluated as merely memory access energy: `# memory accesses` × `average energy/access`. However, to have a fairer comparison with the GP-LiMA, also the memory leakage effect was added in the final energy computation. In particular, its contribution was considered only for the time period in which the memory was used, namely during a memory access. For this reason, the added leakage contribution was evaluated as: `leakage` × `# memory accesses` × `clock period`.
Moreover, it is worth to point out that not only the data memory accesses were involved in the energy estimation, but also the energy spent for fetching the data from the instruction memory was considered. Thus, the final energy amount consumed by the classical RISC-V architecture to perform a certain benchmark was computed as (referring to the data reported in Table 5.11 and Table 5.10):

$$\#\text{ L1D accesses} \times (\text{L1 average energy/access} + \text{L1 leakage/access} \times T_{clock}) +$$

$$\#\text{ L1I accesses} \times (\text{L1 average energy/access} + \text{L1 leakage/access} \times T_{clock}) +$$

$$\#\text{ L2S accesses} \times (\text{L2S average energy/access} + \text{L2S leakage/access} \times T_{clock})$$

Then, concerning the energy consumption estimated for the GP-LiMA, the back annotated energy values reported in the previous section were taken into account.

These values already involve the contributions of both the data memory access energy and the data elaboration. However, they miss the instruction memory accesses contribution. Thus, to complete the GP-LiMA energy expense estimation, the energy spent for fetching the instructions from the memory was evaluated modelling the GP-LiMA IMem through Cacti. For an IMem of 20352 bytes, the value retrieved for the average energy/access was 9.079835 pJ, while the expense in terms of leakage energy/access was 4.316641 pJ for a clock period of 4.3 ns (value matching the one used to gather the GP-LiMA back annotated power values). So, the final energy amount consumed by the hybrid RISC-V/GP-LiMA solution to perform a certain benchmark was computed as:

Algorithm LiM Instructions×(average energy/access+L1 leakage energy/access)

Figure 5.16 and Figure 5.17 present all the energy values related to the tested benchmarks and gathered for both the classical RISC-V memory hierarchy and the GP-LiMA. In particular, Figure 5.16 illustrates the performance achieved by fixing the

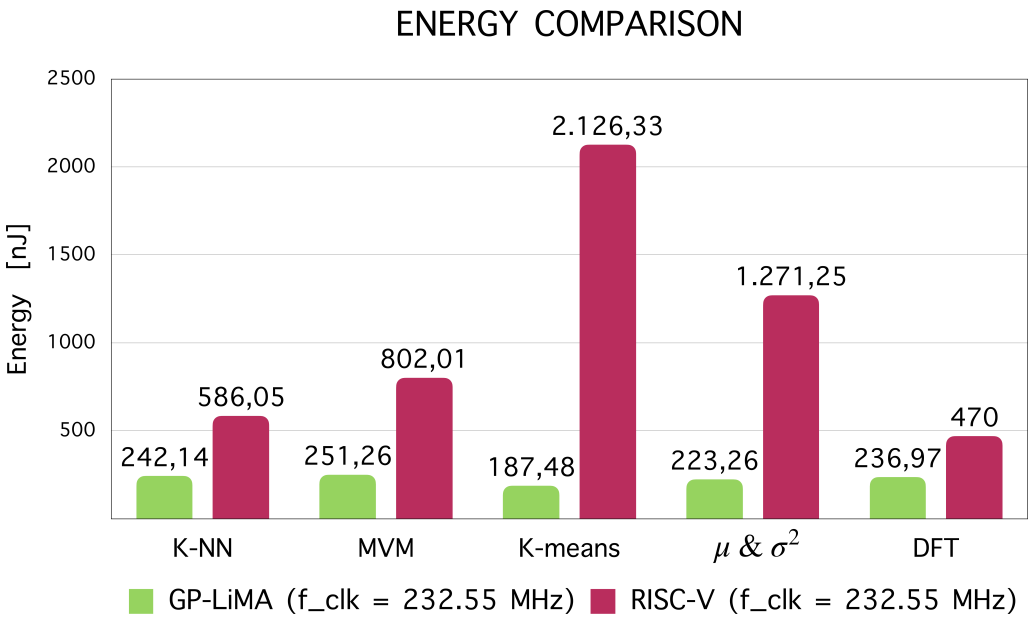


Figure 5.16: Energy comparison between the GP-LiMA and the RISC-V memory hierarchy for all the run benchmarks, under the same clock period (4.3 ns).

clock frequency to 232.55 MHz for both the architectures, while Figure 5.17 reports

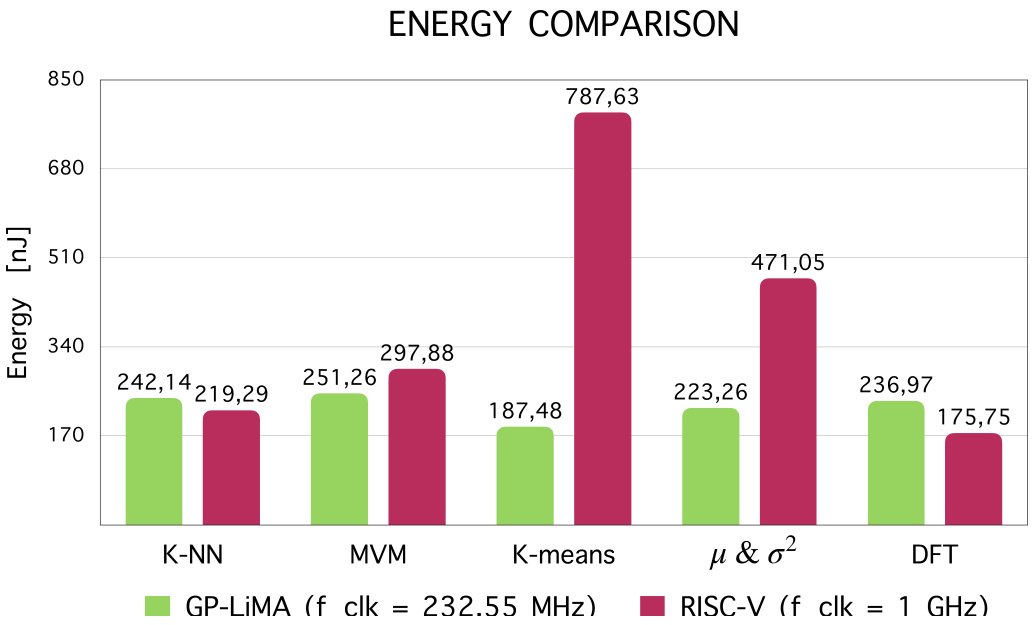


Figure 5.17: Energy comparison between the GP-LiMA and the RISC-V memory hierarchy for all the run benchmarks. The GP-LiMA values refer to a clock period equal to 4.3 ns, while the RISC-V memory hierarchy energies are obtained with a clock period equal to 1 ns.

the values obtained setting the GP-LiMA clock frequency to 232.55 MHz and the RISC-V system clock frequency to 1 GHz. Moreover, note that, unlike the GP-LiMA vs PLiM comparison, here there is no need of providing the energy values on a per-sample basis since the RISC-V system was programmed to perform each benchmark on a dataset matching the one used during the mapping of that benchmark on the GP-LiMA system.

Already at a first glance, it is evident how the GP-LiMA outperforms the RISC-V system under the same clock frequency. The hybrid RISC-V/GP-LiMA system shows power savings with respect to the classical RISC-V memory hierarchy ranging from about 49% in the worst case (DFT benchmark) to 91% in the best case (K-means algorithm). While, if the comparison is made between the energy consumptions of the RISC-V architecture run at 1 GHZ and the GP-LiMA run at 232.55 MHz, the results balance changes, as shown in Figure 5.17. Although in this new scenario the GP-LiMA improvements are reduced, for three out of five benchmarks

the GP-LiMA framework still demonstrates to work better than the classical CPU-centric architecture, bringing to a power saving of about 76% in the best case and to a worsening of 34% in the worst case. However, it must be highlighted that GP-LiMA energy achievements are obtained with a clock frequency 4 times lower than the one set for the RISC solution one and that they correspond to a physical implementation exploiting a quite old technology node (45 nm). It means that if more advanced technology are used to implement the same GP-LiMA structure, even better performance can be achieved both in terms of maximum allowed clock frequency and power consumption, as already seen from the post-synthesis comparison made among different technology nodes outcomes in subsection 5.1.1, where between the 15 nm and the 45 nm there is an energy/clock cycle reduction of 75%.
However, from the performance comparison it follows that the GP-LiMA wins over the RISC-V systems for the benchmarks that allow to fully exploit its massive processing parallelism, including both algorithm that can be easily parallelised (K-means, MVM) and algorithms involving sequential operations that can be performed in a reduction tree-like fashion (μ&σ^2, MVM).
Finally, it must be noticed that among the various terms composing the memory hierarchy RISC-V energy, the one linked to the memory instructions fetch is the one contributing to the most to the total energy consumption. Indeed, the number of memory accesses performed to the L1 Instruction Cache is about one order of magnitude higher than the data memory access one. On the other hand, the GP-LiMA that was initially thought of for cutting the number of data memory accesses, here demonstrates to be even more efficient in making the energy linked to the instruction memory accesses negligible. Figure 5.18 shows the difference in the number of instruction memory accesses performed by the classical RISC-V system and the GP-LiMA solution, while Figure 5.19 compares both architectures in terms of number of data memory accesses.

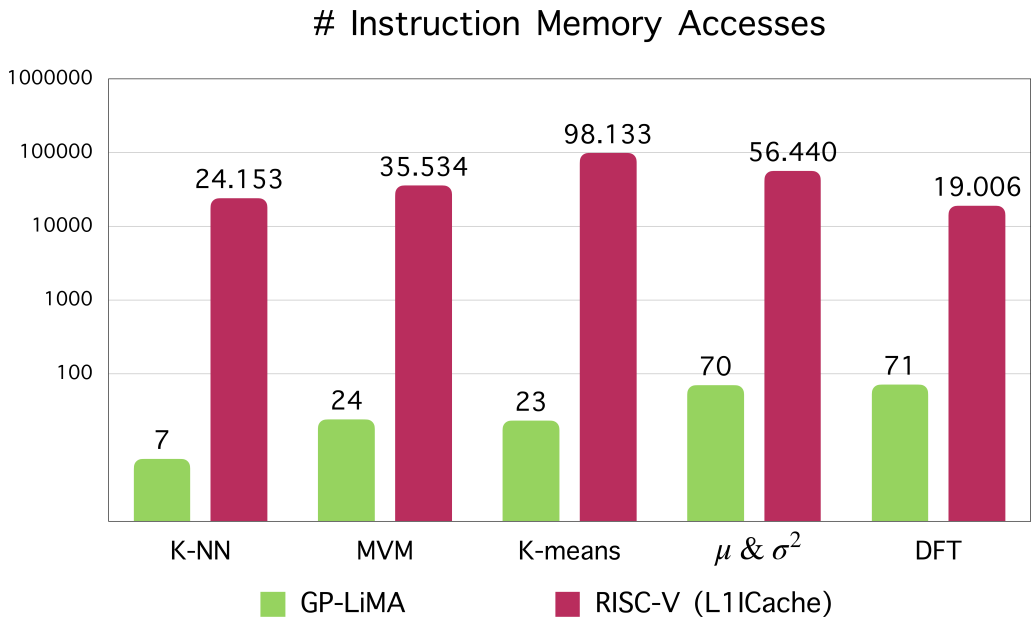


Figure 5.18: Comparison between the GP-LiMA and the RISC-V systems in terms of number of instruction memory accesses for each of all the tested benchmarks.

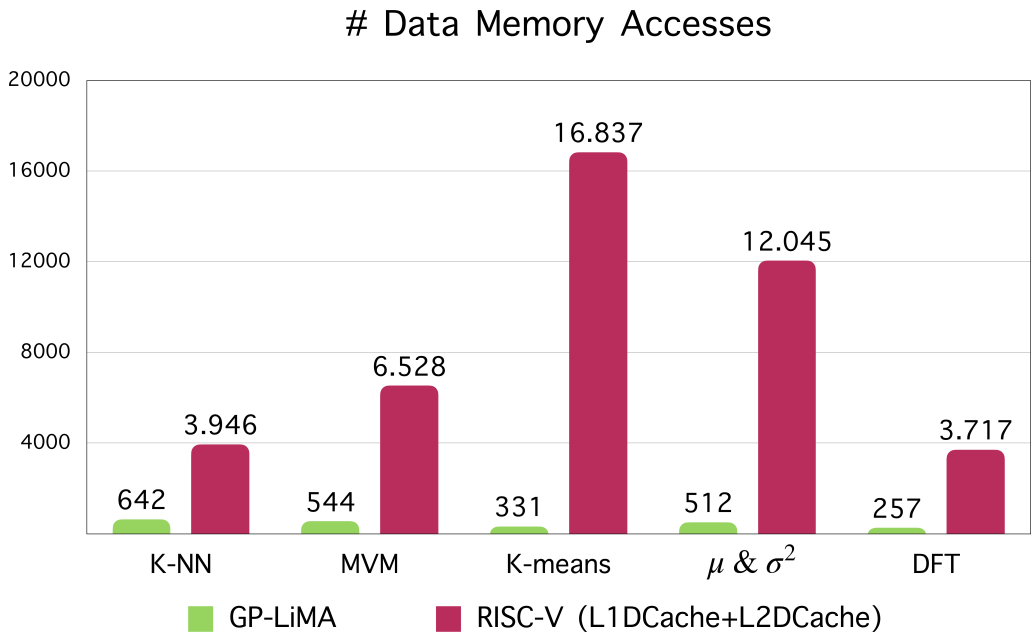


Figure 5.19: Comparison between the GP-LiMA and the RISC-V systems in terms of number of data memory accesses for each of all the tested benchmarks.

Chapter 6

Conclusions

The predicted stop of the growth curve of the processing systems performance, nowadays, requires finding new ways to keep retaining the usual desired exponential trend. The causes for this slowdown can be summarized into two main conceptual walls: the *utilization wall* and the *memory wall*. The last issue is the one that this thesis design proposes to tackle. The *memory wall* is strongly related to the architectural side of standard systems and, in particular, is caused by the high data traffic involved between the CPU and the memory, which, indeed, are two separate entities. This unceasing data movement entails both timing and power wastes, which are further exacerbated by the performance gap between the highly efficient processing units and the available memory units. Moreover, the systems performances are particularly undermined by lately demanded data-intensive applications, which heavily stress the communication between CPU and memory.
To cope with this problem, here, a new architectural model was engineered relying on the novel Logic-in-Memory computing paradigm, whose basic principle lays in providing the data memory with processing capabilities so that part of the required computation can be performed directly in memory, allowing to:

- avoid the massive data movement towards and from the CPU,
- exploit the full memory bandwidth to elaborate in parallel all the stored data,
- leverage the memory processing capability to further parallelize the application execution between the CPU and the memory itself.

However, the weakness of the already existing LiM solutions is that they are all customized for the application the system is intended to run, thus long design times are required to create new LiM components for each new application. Therefore, the design presented in this thesis is proposed as a model to develop LiM solutions, extremely reducing hardware designer's time and efforts. Taking a cue from the already exiting model called PLiM, the GP-LiMA was intended to provide a more general template allowing to produce LiM architectures characterized by high programming generality and improved processing efficiency for data-intensive applications and even for algorithms including more sequential procedures.
The resulting GP-LiMA Unit comprises two macro sections: the LiM Matrix, representing the configurable data memory integrating logic blocks and the set of uCU and nCU dedicated to driving the LiM algorithm execution inside the LiM Matrix. Then, the final GP-LiMA instance can be inserted into a standard system, working alongside the CPU that can start it either in the standard data memory or in the LiM processing mode, making it act as a co-processor. In this last mode, the GP-LiMA component works following the M-SIMD computing model, meaning that, when needed, it can execute up to K different instructions at the same time on different data sets, where K is the degree of M-SIMD set by the hardware designer. This working mode coupled with the LiM Matrix structure is the feature that bestows the programming flexibility to the GP-LiMA. In particular, the LiM Matrix is composed of general-purpose Smart Blocks working on a word level and placed following a grid-like fashion. This arrangement further accommodates a dense routing network that allows performing various and complex data exchanges among the blocks, heavily contributing to speed up the algorithms execution. Two kinds of interconnections are actually embedded inside the LiM Matrix constituting, on the one hand, the turning point for maximizing the programming generality and, on the other hand, the main cause for the GP-LiMA complexity and power overheads. Nevertheless, the final GP-LiMA design still represents an architectural model that needs to be properly configured before fabrication by a hardware designer. However, it is very easy to modify and indeed requires very few changes that are generally linked to the size of the dataset to be processed rather than to the algorithm processing requests, as instead, it happens for the PLiM model. Indeed, if the GP-LiMA is configured to be wide enough, it can successfully run different

kinds of algorithms without needing to be re-synthesized for each new application, as confirmed by the final tests performed on a specific synthesized configuration of the GP-LiMA framework.
At the end of this thesis work, the same GP-LiMA structure was used to map 5 different benchmarks (K-NN, MVM, K-means, $\mu\&\sigma^2$, DFT), so retrieving information about the processing efficiency and the algorithm adaptability achieved. The specific GP-LiMA Unit analysed was implemented through the 45 nm technology node and included 1344 bytes of memory addressable space. It could elaborate up to 512 16-bits data in parallel, working at a maximum frequency of 232.55 MHz. For each of the mapped benchmarks, the results in terms of execution time and energy consumptions were compared on a per-sample basis with the ones achieved by customized devices all retrieved starting from the PLiM model (one for each benchmark). Then, the comparison outcome confirmed the achievements of the objectives set out at the beginning of the GP-LiMA design. The GP-LiMA confirmed to reach almost the same performance of the customized PLiM architectures for the more parallel algorithms (K-NN and K-means), while for the ones involving sequential procedures (especially for the $\mu\&\sigma^2$ and the DFT) it demonstrated to outperform the PLiM solutions leading to a reduction of the execution times of about 67% and 37%, respectively, and to energy savings of about 80% and 63%, respectively. These trends allowed to justify the integration of the complex routing network (consuming about 29% of the total GP-LiMA power), being the main reason for the sequential algorithms speeding up and resulting energy savings.
Finally, to highlight the benefits in terms of improved processing efficiency brought by the insertion of the GP-LiMA paradigm inside a standard system, the energy results achieved by the GP-LiMA were compared with the ones retrieved through a classical CPU-centric system for the same 5 benchmarks. In particular, the energy values taken into account were the ones consumed only for accessing the memory hierarchy connected to the RISC-V architecture during the benchmarks execution (neglecting the energy spent by the RISC-V for the algorithm computation). From the results comparison, a clear gap between the two systems stood out, if run at the same clock frequency value (232.55 MHz). The GP-LiMA showed massive power savings with respect to the classical RISC-V memory hierarchy ranging from about

49% in the worst case (DFT benchmark) to 91% in the best case (K-means algorithm). While, running the standard RISC-V architecture at a frequency 4 times higher than the GP-LiMA one (1 GHz vs 232.55 GHz), the results still pointed out competitive values concerning the GP-LiMA energy expenses. In particular, the GP-LiMA demonstrated to still win over the RISC-V memory hierarchy in case of benchmarks that allow to fully exploit its massive processing parallelism, including both algorithms that can be easily parallelised (K-means), showing energy savings of 76%, and algorithms involving sequential operations that can be performed in a reduction tree-like fashion ($\mu\&\sigma^2$), providing energy savings of about 52%.
Summing up, it was proved that the GP-LiMA paradigm succeeds in efficiently mapping a wide range of algorithms, without requiring in most cases any hardware modification of the architectural structure, demonstrating to hit the goal of programming flexibility combined with a remarkable processing efficiency.

6.1 Future perspectives

To further test the programming generality of the proposed GP-LiMA model, other more complex algorithms could be mapped onto the architecture, such as the convolutional neural network LeNet-5, the advanced encryption algorithm (AES128) and the approximate message passing (AMP), belonging to the compressed sensing algorithms. In this way, the performance achieved by the GP-LiMA could be even compared with the ASIC solutions present in the literature implementing the proposed algorithms.
Moreover, place&route processes exploiting smaller technology nodes could be run on the GP-LiMA netlist to gather more realistic and up to date estimations of the accomplished performances.
Ultimately, as a further future perspective, the actual insertion of the GP-LiMA Unit inside a standard system could be addressed, specifically making the CPU and the GP-LiMA interact to run a whole application requiring both units to cooperate for reaching the final algorithm result.

Appendix A

GP-LiMA Model User's Manual

Here, there are set out all the modifications the hardware designer can make on the VHDL code describing the GP-LiMA model to gather the customized LiM device he desires. The GP-LiMA features customization is organized into two different sections:

- ***Section A.1*** **- Overall LiM Matrix Structure Configuration** addresses all the straightforward settings, such as the LiM Array size, the specs about the M-SIMD mode implementation, and the size of components inside the Smart Block.

- ***Section A.2*** **- Smart Block Deeper Customization** instructs the user on how to make deep-rooted changes, such as variations to the default composition of the Smart Block entity.

A.1 Overall LiM Matrix Structure Configuration

Since the available GP-LiMA model is designed in a parametric way, changing the general features, such as the LiM Matrix size or the instructions concurrency degree (K), is quite easy. The user has only to properly set the values of some predefined constants that are listed in the dedicated VHDL file: **myTypes.vhd**.

A.1.1 LiM Matrix Size Settings

This section provides a list of all the possible generic LiM Matrix features which can be tuned. For each feature a table is reported explicating all the constants that need to be set to correctly carry out its implementation. The first column of each table contains the parameters real name, the second one specifically reports the usage of that constant inside the GP-LiMA design, while the last column shows the values assumed by these parameters in the case of the synthesized GP-LiMA discussed in chapter 5.

Set the memory parallelism

Parameter	Meaning	GP-LiMA Example
`nbit`	LiM Array parallelism.	16
`log_nbit`	It must be equal to $\lceil log_2$`nbit`$\rceil$.	4

Table A.1: List of constants used to define the memory parallelism.

Set the number of LiM Matrix columns

Parameter	Meaning	GP-LiMA Example
`n_Row_Blocks`	Number of Blocks for each row of the LiM Array.	32
`SIZE_CNFG_Row_int`	Number of bits constituting the selection signal for the row interconnections. It must be set equal to $\lceil log_2$`n_Row_Blocks`$\rceil$.	5
`nInputBlockMux`	Inputs number of the single multiplexer composing the second muxes layer of the row interconnection. Refer to section 4.2.2 to know how to set the proper value.	4

SIZE_sel_CrossBarMux	Number of bits constituting the selection signal for the first muxes layer of the row interconnection. Refer to section 4.2.2 to know how to set the proper value.	3
SIZE_LiM_ADDR	Number of bits composing the address signal for the LiM Array. It must be set equal to $\lceil log_2$(n_Row_Blocks$\times$ (n_Smart_Rows+n_Standard_Rows))$\rceil$.	10
SIZE_ADDR_S1	Number of bits composing the ADDR S1 field of the nInstruction. It must be set equal to $\lceil log_2$MAX_BIT_ADDR_S1$\rceil$, with MAX_BIT_ADDR_S1 = MAX{n_Smart_Rows + n_Standard_Rows, SIZE_RF}.	5
SIZE_ADDR_S2	Number of bits composing the ADDR S2 field of the nInstruction. It must be set equal to $\lceil log_2$MAX_BIT_ADDR_S2$\rceil$, with MAX_BIT_ADDR_S2 = MAX{n_Row_Blocks $\times$ (n_Smart_Rows + n_Standard_Rows), SIZE_RF}.	10

Table A.2: List of constants used to define the LiM Matrix columns number.

Set the number of LiM Matrix smart rows

Parameter	Meaning	GP-LiMA Example
n_Smart_Rows	Number of rows composing the LiM Array smart section.	16

SIZE_CNFG_Col_int	Number of bits constituting the selection signal for the column interconnections. It must be set equal to $\lceil log_2$(n_Smart_Rows+n_Standard_Rows)$\rceil$.	5
SIZE_LiM_ADDR	Number of bits composing the address signal for the LiM Array. It must be set equal to $\lceil log_2$(n_Row_Blocks$\times$ (n_Smart_Rows+n_Standard_Rows))$\rceil$.	10
SIZE_ADDR_S1	Number of bits composing the ADDR S1 field of the nInstruction. It must be set equal to $\lceil log_2$MAX_BIT_ADDR_S1$\rceil$, with MAX_BIT_ADDR_S1 = MAX{n_Smart_Rows + n_Standard_Rows, SIZE_RF}.	5
SIZE_ADDR_S2	Number of bits composing the ADDR S2 field of the nInstruction. It must be set equal to $\lceil log_2$MAX_BIT_ADDR_S2$\rceil$, with MAX_BIT_ADDR_S2 = MAX{n_Row_Blocks $\times$ (n_Smart_Rows + n_Standard_Rows), SIZE_RF}.	10

Table A.3: List of constants used to define the rows number inside the LiM Matrix smart section.

Note that each time the number of smart rows changes, besides the re-definition of the set of constants in Table A.3, also the procedure for the M-SIMD mode configuration, illustrated in subsection A.1.2, must be performed to make sure that its organization is consistent with the new smart section one.

Set the number of LiM Matrix standard rows

Parameter	**Meaning**	**GP-LiMA Example**

n_Standard_Rows	Number of rows composing the LiM Array standard section.	5
SIZE_CNFG_Col_int	Number of bits constituting the selection signal for the column interconnections. It must be set equal to $\lceil log_2$(n_Smart_Rows+n_Standard_Rows)$\rceil$.	5
SIZE_LiM_ADDR	Number of bits composing the address signal for the LiM Array. It must be set equal to $\lceil log_2$(n_Row_Blocks$\times$ (n_Smart_Rows+n_Standard_Rows))$\rceil$.	10
SIZE_ADDR_S1	Number of bits composing the ADDR S1 field of the nInstruction. It must be set equal to $\lceil log_2$MAX_BIT_ADDR_S1$\rceil$, with MAX_BIT_ADDR_S1 = MAX{n_Smart_Rows + n_Standard_Rows, SIZE_RF}.	5
SIZE_ADDR_S2	Number of bits composing the ADDR S2 field of the nInstruction. It must be set equal to $\lceil log_2$MAX_BIT_ADDR_S2$\rceil$, with MAX_BIT_ADDR_S2 = MAX{n_Row_Blocks $\times$ (n_Smart_Rows + n_Standard_Rows), SIZE_RF}.	10

Table A.4: List of constants used to define the rows number inside the LiM Matrix standard section.

Set the size of the RF inside the Smart Block

Parameter	Meaning	GP-LiMA Example
SIZE_RF	Number of registers composing the RF.	4
SIZE_Address_RF	Number of bits of the address signals for the RF. It must be equal to $\lceil log_2$SIZE_RF$\rceil$.	2

SIZE_ADDR_S1	Number of bits composing the ADDR S1 field of the nInstruction. It must be set equal to $\lceil log_2$MAX_BIT_ADDR_S1$\rceil$, with MAX_BIT_ADDR_S1 = MAX{n_Smart_Rows + n_Standard_Rows, SIZE_RF}.	5
SIZE_ADDR_S2	Number of bits composing the ADDR S2 field of the nInstruction. It must be set equal to $\lceil log_2$MAX_BIT_ADDR_S2$\rceil$, with MAX_BIT_ADDR_S2 = MAX{n_Row_Blocks × (n_Smart_Rows + n_Standard_Rows), SIZE_RF}.	10

Table A.5: List of constants used to define the Smart Block RF size.

Set the size of the LUT inside the Smart Block

Parameter	Meaning	GP-LiMA Example
Nbit_LUT	Number of bits of the data in each LUT row, namely, the number of LUT columns.	4
N_inputs_LUT	Number 1-bit LUT inputs.	4

Table A.6: List of constants used to define the Smart Block LUT size.

Adapt the GP-LiMA structure to interact with a bigger external IMem

Parameter	Meaning	GP-LiMA Example
SIZE_uROM	Number of rows, i.e. maximum number of LiM instructions, composing the external LiM Instruction Memory.	1024

SIZE_uROM_Address	Number of bits composing the address signal for the IMem. It must be set equal to $\lceil log_2$SIZE_uROM_Address$\rceil$.	10

Table A.7: List of constants used to make the GP-LiMA Unit compatible with the external IMem size.

Set the size of the uCU LiM instructions queue

Parameter	Meaning	GP-LiMA Example
size_Queue	Number of registers composing the LiM instructions queue inserted in the uCU.	5

Table A.8: Constant used to define the uCU LiM instructions queue size.

A.1.2 Configuration of the M-SIMD computing mode

To fix the M-SIMD computing mode degree K, i.e. the maximum number of different concurrent instructions which can be properly run by the GP-LiMA Unit, and to determine for each row which is the instruction that drives it, four constants sets must be initialized. In the following, the constants meanings and how to correctly handle them is explained and the values set for the GP-LiMA Unit synthesized in chapter 5 are reported as example.

- n_INSTR_DEC: constant that identifies the value for K (M-SIMD computing mode degree). The GP-LiMA Unit is organized so that K instruction decoders are instantiated, each working on a different instruction to be performed by the LiM Matrix. Each instruction decoder drives a subset of smart rows and is associated with a unique ID number going from 0 to K-1. The instruction decoder with ID = 0 drives the first set of smart rows, the one with ID = 1 controls the second subset, and so on till the last set with ID = K-1.

  GP-LiMA Example

```
constant n_INSTR_DEC: integer:=3;
```

The smart section of the resulting LiM Matrix is composed of 3 subsets of smart rows each driven by a different instruction decoder. It follows that the generated nCU is made up of 3 instruction decoders with ID equal to 0, 1, 2, respectively.

- `MSIMD_array_init`: array of constants used to determine the association between the K instructions and the smart rows. Each array element refers to a different smart row and must be initialized with the ID of the instruction decoder that has to drive that smart row.

  GP-LiMA Example

```
type MSIMD_array is ARRAY(0 to n_Smart_Rows-1) of
        integer range 0 to n_INSTR_DEC-1;
constant MSIMD_array_init :
        MSIMD_array := (0,0,0,0,0,  1,1,1,1,1,  2,2,2,2,2,2);
```

  In this example, the smart section is composed of 16 rows which are driven according to the following association: the first 5 smart rows are connected to the first instruction decoder (ID = 0), the next 5 rows are driven by the second instruction decoder (ID = 1), while the remaining 6 smart rows are attached to the last instrustion decoder (ID = 2).

- `MSIMD_N_INSTR_DEC_array_init`: array of constants used to implement the connection between each of the K instruction decoders and the related smart rows subset. Each array element refers to a different instruction decoder and must be initialized with the number of smart rows it has to drive.

  GP-LiMA Example

```
type MSIMD_N_INSTR_DEC_array is ARRAY(0 to n_INSTR_DEC-1) of
        integer range 0 to n_Smart_Rows;
constant MSIMD_N_INSTR_DEC_array_init :
        MSIMD_N_INSTR_DEC_array := (5,5,6);
```

It follows that, as expected, the instruction decoders with ID equal to 0 and 1 drive 5 smart rows each, while the remaining one (ID = 2) controls 6 smart rows.

- `SUM_MSIMD_N_INSTR_DEC_array_init`: array of constants used to complete the connection between each of the K instruction decoders and the related smart rows subset. This array contains K elements (one for each instruction decoder) that need to be initialized in this way: the first one must be equal to the number of smart rows driven by the first instruction decoder, while each of all the other elements must be set equal to the sum between the previous element content and the number of smart rows handled by the specific instruction decoder that element refers to.

  GP-LiMA Example

```
type SUM_MSIMD_N_INSTR_DEC_array is ARRAY(0 to n_INSTR_DEC-1) of
        integer range 0 to n_Smart_Rows;
constant SUM_MSIMD_N_INSTR_DEC_array_init :
        MSIMD_N_INSTR_DEC_array := (5,10,16);
```

  In this example, the first instruction decoder drives 5 smart rows, thus the first array element is equal to 5, then the second element is given by the sum between 5 and the number of smart rows controlled by the second instruction decoder, that is again 5, and, finally, the last element is set equal to the sum between 10 and 6 that is the number smart rows connected to the last instruction decoder.

A.2 Smart Block Deeper Customization

Although the composition of the Smart Block inside the LiM Matrix was engineered to be as more general as possible to efficiently suit the mapping of most the algorithms, the GP-LiMA model still provides the user with the possibility to insert or remove the logic blocks under the Block Word, which are conceived to the elaboration of the more complex or customized operations.
In this section, both the procedures for entering a new logic block and taking out

an optional logic block are illustrated complemented by an example. Note that the RShifter and the ALU blocks cannot be touched.

A.2.1 New Logic Block Insertion Procedure

Before starting the insertion procedure the user has to implement on a separate VHDL file the logic block he wants to enter. This new block can have a maximum of two input operands and one single output signal, each on a number of bits equal to the LiM array parallelism. Then, it can have an arbitrarily long input configuration signal.
Suppose that a logic block (called `Example_Block`) having the following interface (entity) must be inserted as further computational unit in parallel with the default ones.

```
entity Example_Block is
  port(-- Two input data signals
       operand_A: in std_logic_vector(nbit-1 downto 0);
       operand_B: in std_logic_vector(nbit-1 downto 0);
       -- Configuration signal
       cnfg_Example_Block:
            in std_logic_Vector(SIZE_cnfg_Example_Block-1 downto 0);
       -- Output data signal
       result: out std_logic_vector(nbit-1 downto 0)
      );
end Example_Block;
```

The procedure to successfully introduce this new block inside all the LiM Matrix Smart Blocks is made up of the following steps:

1. Open the **SMART_BLOCK.vhd** file and insert the `Example_Block` in the list of the components inside the `Architecture` section of the `SMART_BLOCK`.

```
architecture structural of SMART_BLOCK is

  component Example_Block
      port(-- Two input data signals
           operand_A: in std_logic_vector(nbit-1 downto 0);
           operand_B: in std_logic_vector(nbit-1 downto 0);
           -- Configuration signal
           cnfg_Example_Block: in
               std_logic_Vector(SIZE_cnfg_Example_Block-1 downto 0);
           -- Output data signal
           result: out std_logic_vector(nbit-1 downto 0)
          );
  end component;
  [...]
  begin
```

2. Instantiate the `Example_Block` connecting the 2 input signals to `OpA` and `OpB`, the configuration signal to the correct `SIZE_cnfg_Example_Block` bits-slice of the `CNFG_FUNC`, and the output signal to the next available input signal of the multiplexer returning the `Result` signal, that is `BI_MUX_inputs(4)`.

```
  begin
  [...]
  -------------------------------------------------------------------
  -- INSTATIATION of the Example_Block logic block
  -------------------------------------------------------------------
    Example_Block_comp: LUT_BI
        port map(operand_A=>OpA, operand_B=>OpA,
                 cnfg_Example_Block=>
                     CNFG_FUNC(SIZE_cnfg_Example_Block-1 downto 0),
                 result=>BI_MUX_inputs(4));
  [...]
end architecture structural;
```

3. Open the **myTypes.vhd** file to adjust the constant values accordingly to the

changes made to the Smart Block.

4. Find the constant section regarding the Smart Block and increment by one the value of the `n_SMART_BLOCK_BI` constant representing the number of logic blocks underneath the Block Word.

5. Set the value of the `SIZE_Sel_active_BI` constant to $\lceil log_2$`n_SMART_BLOCK_BI`$\rceil$, which specifies the number of bits of the selection signal for the Smart Block multiplexer returning the `Result` signal.

6. Update the value of `SIZE_CNFG_FUNC` constant which must be equal to the number of bits of the widest configuration signal among the ones of the logic blocks under the Block Word.

```
--------------------------- SMART BLOCK ---------------------------
--
-- Number of logic blocks composing the SMART BLOCK:
-- RShifter, Adder, Multiplier, LUT, Example_Block
   constant n_SMART_BLOCK_BI: integer:= 5;

-- Size of the control signal of the SMART BLOCK multiplexer
-- selecting the output of one of the logic blocks
   constant SIZE_Sel_active_BI: integer:= 3;

-- Size of the instruction field defining the specific function to
-- the executed by the involved logic block, specified by the
-- OPCODE field value:
-- SIZE_CNFG_FUNC = max {SIZE_CNFG_FUNC_ADDER_LOGIC = 4,
--                       SIZE_CNFG_FUNC_RSHIFTER    = 1,
--                       SIZE_CNFG_FUNC_MULTIPLER   = 0,
--                       SIZE_CNFG_FUNC_LUT         = 1,
--                       SIZE_cnfg_Example_Block    = 5}
   constant SIZE_CNFG_FUNC: integer:= 5;
```

7. Find the constants section regarding the list of values for the `OPCODE` field of the `nInstruction` and add a new constant representing the `OPCODE` value

associated to the new `Example_Block`. Note that the value associated to its `OPCODE` must be equal to the value that the control signal for the multiplexer inside the Smart Block has to assume for connecting the `Result` signal to the output of that logic block.

```
------------------------------------------------------------------
-- OPCODE VALUES
------------------------------------------------------------------
constant OP_RShifter: std_logic_vector(SIZE_OPCODE-1 downto 0) :=
    std_logic_vector(to_unsigned(0, SIZE_OPCODE));

constant OP_Load, OP_Logic_Adder, nullOP:
    std_logic_vector(SIZE_OPCODE-1 downto 0) :=
        std_logic_vector(to_unsigned(1, SIZE_OPCODE));

constant OP_Multiplier: std_logic_vector(SIZE_OPCODE-1 downto 0) :=
    std_logic_vector(to_unsigned(2, SIZE_OPCODE));

constant OP_LUT: std_logic_vector(SIZE_OPCODE-1 downto 0) :=
    std_logic_vector(to_unsigned(3, SIZE_OPCODE));

constant OP_Example_Block: std_logic_vector(SIZE_OPCODE-1 downto 0)
    := std_logic_vector(to_unsigned(4, SIZE_OPCODE));
```

8. In the case the new block needs a control signal to select one of the functions it can perform, as for the example shown, add in the constants section dedicated to the values for the `FUNC` field of the `nInstruction` the list of all the new `FUNC` constants for the `Example_Block`.

```
------------------------------------------------------------------------
-- FUNC VALUES
------------------------------------------------------------------------
constant AND_OP, nullFunc, Logic_RShift_OP :
    std_logic_vector(SIZE_FUNC-1 downto 0) :=
        std_logic_vector(to_unsigned(0, SIZE_FUNC));
constant XOR_OP, Arith_RShift_OP, Sign_ext_LUT_OP :
    std_logic_vector(SIZE_FUNC-1 downto 0) :=
        std_logic_vector(to_unsigned(1, SIZE_FUNC));
constant SUM_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(8, SIZE_FUNC));
constant SUB_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(12, SIZE_FUNC));
constant ABS_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(10, SIZE_FUNC));
constant OR_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(4, SIZE_FUNC));
constant NAND_OP, NOT_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(2, SIZE_FUNC));
constant NOR_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(6, SIZE_FUNC));
constant XNOR_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(5, SIZE_FUNC));
constant EQ_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(9, SIZE_FUNC));
constant NOT_EQ_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(11, SIZE_FUNC));
constant GT_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(13, SIZE_FUNC));
constant LT_OP : std_logic_vector(SIZE_FUNC-1 downto 0) :=
    std_logic_vector(to_unsigned(15, SIZE_FUNC));
```

```
-- Add here the list of the FUNC values for the Example_Block

constant Example_Block_FUNC_0_OP :
    std_logic_vector(SIZE_FUNC-1 downto 0) :=
        std_logic_vector(to_unsigned(0, SIZE_FUNC));
constant Example_Block_FUNC_1_OP :
    std_logic_vector(SIZE_FUNC-1 downto 0) :=
        std_logic_vector(to_unsigned(1, SIZE_FUNC));
[...]
constant Example_Block_FUNC_31_OP :
    std_logic_vector(SIZE_FUNC-1 downto 0) :=
        std_logic_vector(to_unsigned(31, SIZE_FUNC));
```

A.2.2 Logic Block Removal Procedure

The removal of a logic block is a quite simple routine that mainly consists of erasing the component at issue and adjusting the signals connection to the multiplexer inside the Smart Block generating the `Result` signal. This last step is required only when the logic block output is connected to one of the multiplexer input signals that is located in an intermediate position.
Suppose to have to take out the Multiplier block from the Smart Block in the default configuration. The procedure to be carried out is made up of the following steps:

1. Open the **SMART_BLOCK.vhd** and remove the logic block by commenting the logic block instance inside the `Architecture` section of the `SMART_BLOCK`.

```
begin
[...]
-------------------------------------------------
-- INSTATIATION of Multiplier logic block
-------------------------------------------------
--  Multiplier_BI_comp: Multiplier_BI
--      port map(operand_A=>OpA, operand_B=>OpB,
--               result=>BI_MUX_inputs(2));
```

2. Adjust the `Result` multiplexer inputs connections shifting all the signals making them occupy at first all the lowest positions. In this specific case, only the output of the LUT logic block must be shifted to a lower position of the multiplexer input signals, that is the one just released from the Multiplier block: *BI_MUX_inputs(2)*.

```
architecture structural of SMART_BLOCK is
  [...]
  begin
  [...]
  ------------------------------------------------------------------
  -- INSTATIATION of the LUT logic block
  ------------------------------------------------------------------
    LUT_BI_comp: LUT_BI
      port map(operand_LUT=>OpA, CLEAR=>reset, CLK=>CLK,
               Input_LUT_Daisy_Chain=>Input_LUT_Daisy_Chain,
               Output_LUT_Daisy_Chain=>Output_LUT_Daisy_Chain,
               En_LUT_Daisy_Chain=>En_LUT_Daisy_Chain,
               Ext_cnfg=>CNFG_FUNC(0), Result=>BI_MUX_inputs(2));
  [...]
end architecture structural;
```

3. Open the **myTypes.vhd** file to adjust the constants value as done for the previous logic block insertion procedure.

4. Find the constant section regarding the Smart Block and decrement by one the value of the `n_SMART_BLOCK_BI` constant representing the number of logic blocks underneath the Block Word.

5. Set the value of the `SIZE_Sel_active_BI` constant to $\lceil log_2 \texttt{n\_SMART\_BLOCK\_BI} \rceil$, which specifies the number of bits of the selection signal for the Smart Block multiplexer returning the `Result` signal.

6. Update the value of `SIZE_CNFG_FUNC` constant which must be equal to the number of bits of the widest configuration signal among the ones of the remaining logic blocks under the Block Word.

```
--------------------------- SMART BLOCK ---------------------------
--
-- Number of logic blocks composing the SMART BLOCK:
-- RShifter, Adder, LUT
   constant n_SMART_BLOCK_BI: integer:= 3;

-- Size of the control signal of the SMART BLOCK multiplexer
-- selecting the output of one of the logic blocks
   constant SIZE_Sel_active_BI: integer:= 2;

-- Size of the instruction field defining the specific function to
-- the executed by the involved logic block, specified by the
-- OPCODE field value:
-- SIZE_CNFG_FUNC = max {SIZE_CNFG_FUNC_ADDER_LOGIC = 4,
--                       SIZE_CNFG_FUNC_RSHIFTER    = 1,
--                       SIZE_CNFG_FUNC_LUT         = 1}
   constant SIZE_CNFG_FUNC: integer:= 4;
```

7. Find the constants section regarding the list of values for the `OPCODE` field of the `nInstruction` and adjust the values related to all the logic blocks. Note that, for each logic block, the value associated to its `OPCODE` must be equal to the value that the control signal for the multiplexer inside the Smart Block has to assume for connecting the `Result` signal to the output of that logic block. In this case, only the LUT `OPCODE` needs to be modified.

```
------------------------- OPCODE VALUES -------------------------
constant OP_RShifter: std_logic_vector(SIZE_OPCODE-1 downto 0) :=
    std_logic_vector(to_unsigned(0, SIZE_OPCODE));
constant OP_Load, OP_Logic_Adder, nullOP:
    std_logic_vector(SIZE_OPCODE-1 downto 0) :=
        std_logic_vector(to_unsigned(1, SIZE_OPCODE));
constant OP_LUT: std_logic_vector(SIZE_OPCODE-1 downto 0) :=
        std_logic_vector(to_unsigned(2, SIZE_OPCODE));
```

Bibliography

[1] T. N. Theis and H. . P. Wong. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50, 2017.

[2] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[3] R. H. Dennard, F. H. Gaensslen, Hwa-Nien Yu, V. L. Rideout, E. Bassous, and A. R. Leblanc. Design of ion-implanted mosfet's with very small physical dimensions. *Proceedings of the IEEE*, 87(4):668–678, 1999.

[4] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, 2011.

[5] M. B. Taylor. Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse. In *DAC Design Automation Conference 2012*, pages 1131–1136, 2012.

[6] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores: reducing the energy of mature computations. *ACM Sigplan Notices*, 45(3):205–218, 2010.

[7] John Von Neumann and Ray Kurzweil. *The computer and the brain*. Yale University Press, 2012.

[8] Sally A McKee. Reflections on the memory wall. In *Proceedings of the 1st conference on Computing frontiers*, page 162, 2004.

[9] G. Santoro. Exploring new computing paradigms for data-intensive applications. *Politecnico di Torino: Torino, Italy*, 2019.

[10] A. Pedram, S. Richardson, M. Horowitz, S. Galal, and S. Kvatinsky. Dark memory and accelerator-rich system optimization in the dark silicon era. *IEEE*

Design Test, 34(2):39–50, 2017.

[11] 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, 2012.

[12] J. T. Pawlowski. Hybrid memory cube (hmc). In *2011 IEEE Hot Chips 23 Symposium (HCS)*, pages 1–24, 2011.

[13] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A survey on coarse-grained reconfigurable architectures from a performance perspective. *IEEE Access*, 8:146719–146743, 2020.

[14] Giulia Santoro, Giovanna Turvani, and Mariagrazia Graziano. New logic-in-memory paradigms: An architectural and technological perspective. *Micromachines*, 10(6):368, 2019.

[15] Abu Sebastian, Manuel Le Gallo, Riduan Khaddam-Aljameh, and Evangelos Eleftheriou. Memory devices and applications for in-memory computing. *Nature nanotechnology*, 15(7):529–544, 2020.

[16] Ian Kuon, Russell Tessier, and Jonathan Rose. *FPGA architecture: Survey and challenges*. Now Publishers Inc, 2008.

[17] Stamatis Vassiliadis and Dimitrios Soudris. *Fine-and coarse-grain reconfigurable computing*, volume 16. Springer, 2007.

[18] H. Singh, Ming-Hau Lee, Guangming Lu, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *IEEE Transactions on Computers*, 49(5):465–481, 2000.

[19] Edward Tau, Derrick Chen, Ian Eslick, and Jeremy Brown. A first generation dpga implementation. In *In Proceedings of the Third Canadian Workshop on Field-Programmable Devices*. Citeseer, 1995.

[20] Andre Dehon, Thomas F Knight Jr, Edward Tau, Michael Bolotski, Ian Eslick, Derrick Chen, and Jeremy Brown. Dynamically programmable gate array with multiple contexts, April 21 1998. US Patent 5,742,180.

[21] D. C. Chen and J. M. Rabaey. A reconfigurable multiprocessor ic for rapid prototyping of algorithmic-specific high-speed dsp data paths. *IEEE Journal of Solid-State Circuits*, 27(12):1895–1904, 1992.

[22] Arthur Abnous, Christopher Christensen, Jeffrey Gray, John Lenell, Andrew

Naylor, and Nader Bagherzadeh. Design and implementation of the "tiny risc" microprocessor. *Microprocessors and Microsystems*, 16(4):187–193, 1992.

[23] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, et al. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th international conference on Supercomputing*, pages 14–25, 2002.

[24] Thomas L Sterling and Hans P Zima. Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing. In *SC'02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 48–48. IEEE, 2002.

[25] T. Finkbeiner, G. Hush, T. Larsen, P. Lea, J. Leidel, and T. Manning. In-memory intelligence. *IEEE Micro*, 37(4):30–38, 2017.

[26] Ravi Nair, Samuel F Antao, Carlo Bertolli, Pradip Bose, Jose R Brunheroto, Tong Chen, C-Y Cher, Carlos HA Costa, Jun Doi, Constantinos Evangelinos, et al. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17–1, 2015.

[27] Shuangchen Li, Dimin Niu, Krishna T Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. Drisa: A dram-based reconfigurable in-situ accelerator. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 288–301. IEEE, 2017.

[28] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A Kozuch, Onur Mutlu, Phillip B Gibbons, and Todd C Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 273–287. IEEE, 2017.

[29] Shihui Yin, Zhewei Jiang, Jae-Sun Seo, and Mingoo Seok. Xnor-sram: In-memory computing sram macro for binary/ternary deep neural networks. *IEEE Journal of Solid-State Circuits*, 55(6):1733–1743, 2020.

[30] Avishek Biswas and Anantha P Chandrakasan. Conv-sram: An energy-efficient sram with in-memory dot-product computation for low-power convolutional neural networks. *IEEE Journal of Solid-State Circuits*, 54(1):217–230, 2018.

[31] 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.

[32] Jason K Eshraghian, Sung-Mo Kang, Seungbum Baek, Garrick Orchard, Herbert Ho-Ching Iu, and Wen Lei. Analog weights in reram dnn accelerators. In *2019 IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, pages 267–271. IEEE, 2019.

[33] I Giannopoulos, A Sebastian, M Le Gallo, VP Jonnalagadda, M Sousa, MN Boon, and E Eleftheriou. 8-bit precision in-memory multiplication with projected phase-change memory. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 27–7. IEEE, 2018.

[34] AV Khvalkovskiy, Dmytro Apalkov, S Watts, Roman Chepulskii, RS Beach, Adrian Ong, X Tang, A Driskill-Smith, WH Butler, PB Visscher, et al. Basic principles of stt-mram cell operation in memory arrays. *Journal of Physics D: Applied Physics*, 46(7):074001, 2013.

[35] Andrew D Kent and Daniel C Worledge. A new spin on magnetic memories. *Nature nanotechnology*, 10(3):187–191, 2015.

[36] Amir Morad, Leonid Yavits, and Ran Ginosar. Gp-simd processing-in-memory. *ACM Transactions on Architecture and Code Optimization (TACO)*, 11(4):1–26, 2015.

[37] U. Casale. Programmable lim: a modular and reconfigurable approach to the logic in memory. *Politecnico di Torino: Torino, Italy*, 2020.

[38] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[39] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP laboratories*, 27:28, 2009.