POLITECNICO DI TORINO

Electronic Engineering

Master degree course in Electronic Systems

Master Degree Thesis

Characterization and Performance Evaluation of Programmable Logic in Memory Architectures

Supervisors
prof. Maurizio Zamboni

Correlators:
prof. Marco Vacca
prof.ssa Giovanna Turvani

Candidate
Giorgio PERRONE
matricola: 252969

ACADEMIC YEAR 2020-2021

Summary

Nowadays, one task to fulfill is to overcome the bottleneck of the Von Neumann architecture. The most critical aspect of this kind of structure is that the operating unit has greatly improved its performance in recent years, while memories couldn't follow this trend. For this reason, especially in data-intensive applications, the memory is not able to provide data as fast as the operating unit can compute them, leading to a worsening of performance. A possible solution to this bottleneck is the Logic-in-Memory (LiM) approach. It consists of merging processing elements and storage together to get a hybrid memory capable of both storing and computing data. The state-of-the-art surrounding the Von Neumann Architecture and the LiM idea is analyzed, trying to find out a set of possible solutions adopted in the past to overcome this bottleneck. The Programmable Logic-in-Memory (PLiM), an architecture developed at the polytechnic university of Turin, is taken into account since it demonstrated to be very efficient to lighten the Von Neumann limitations. In this approach, the processing unit and the memory are no longer seen as two different entities, since the memory becomes also capable of performing some operations directly inside its array. This is accomplished by revolutionizing the common memory structure. According to PLiM approach, the memory is subdivided into a smart section and a standard section, where the former performs operations within it, while the latter retains the known classic memory configuration. In particular, the smart section can be split in different clusters, composed of three memory rows: one smart row, which is the heart of the cluster, its corresponding UpRow, and its corresponding DownRow. The real peculiarity of PLiM approach is that its operating functionalities are provided by hardware blocks, the row interfaces, instantiated in the smart section, that can make computations retrieving data from the previously discussed clusters and from the standard section. In this way, such a structure is ideally suitable for every application since every function can be implemented through the row interfaces, providing PLiM the feature of being a general-purpose architecture. The examined

structure is a micro-programmed machine where the micro-code written in a micro-ROM allows to perform the desired operations at each clock cycle. The smart section has a homogeneous structure in which the same chain of row interfaces has to be implemented in correspondence of each cluster. As a consequence, every smart row environment, within a clock cycle, is able to perform one operation among the ones available thanks to the row interfaces, inside the memory space. This allows PLiM to exploit the Single-Instruction-Multiple-Data (SIMD) computing mode; every micro-instruction written in the micro-ROM is applied to all the smart row environments. If n-smart rows are instantiated in the smart section, n-parallel operations can be performed as described by the corresponding micro-instruction.
After PLiM structure is presented, it is tested with different algorithms in order to check its performance and consumption estimates. The first tested algorithm is the Advanced-Encryption-Standard 128 (AES128), which is an encryption algorithm requiring special computations. Since its operations cannot be parallelized within PLiM environment, the whole encryption procedure is processed within one smart row environment with some special-purpose row interfaces. The second examined algorithm is the Approximate-Message-Passing (AMP), an iterative algorithm belonging to the compressed sensing world. Here the involved operations are simpler with respect to the AES case (sums and multiplications are required), therefore the chain of row interfaces is lighter compared to the AES case, bringing to a faster structure. For both algorithms, the SIMD feature of PLiM has been exploited, in particular, for the AES case, it is possible to parallelize the whole algorithm on the available smart rows. In the AMP, instead, it is possible to parallelize the execution of one or more operations required by the algorithm, such as matrix multiplication.
For each benchmark, performance values are evaluated in terms of timing, power, and energy dissipation. In the end, a comparison between the PLiM approach and a standard architecture, where memory and processor are kept separated, is carried out. In particular, considering the implementation of the same algorithm, PLiM leads to notable advantages compared to a RISC-V architecture, both in terms of the application execution time and energy consumption.
Taking into account the AES implementation, the most realistic one, an extra comparison is made also considering some encryption accelerators found in the related works. It has been found that the PLiM implementation for the AES128 can reach a throughput value of 358 Mbs, which is three orders of magnitude more than the one achieved by the RISC-V processor (264 Kbs)

and two or three orders of magnitude less than the highest throughput rates (up to 275 Gbs) of the deeply pipelined most modern accelerators. Considering the energy efficiency, which is the ratio between the throughput and the dissipated power, for the AES specific application, the result obtained for the PLiM implementation (21 Gbs/W) is just about one order of magnitude smaller than the one found for modern AES accelerators (about 400-500 Gbs/W) and three orders of magnitude greater than the one achieved by RISC-V architecture of 63 Mbs/W. PLiM architecture is not built to be an accelerator, in fact, its performances values are lower compared to the modern accelerators. However, it has been found that the PLiM approach leads to great benefits with respect to a classic approach and can be considered as a general-purpose solution with good performance and efficiency features.

Contents

List of Tables

List of Figures

Chapter 1

State of the art

1.1 Introduction to the Von Neumann Architecture and its bottleneck

Historically the Von Neumann architecture has been always compared to the Harvard one, the main difference between the two approaches stands in the different structure of the storage section. In a Harvard architecture data and instructions are stored in two different entities while in a Harvard structure there is a unique memory that manages both data and instructions as shown in Figure 1.1.
This aspect implies that the Von Neumann structure is cheaper, occupies less area, and its control unit is simpler with respect to the Harvard structure since it has to manage just one memory and for these reasons, modern computing is based on this architecture. On the other hand, an instruction fetch and a data operation cannot occur at the same time since they share a common bus, they must be scheduled and this is a reason that leads to a limitation of performance. The second aspect which influences most the performance of the architecture is the different operating frequency achievable by the storage unit and the operating unit. To conclude and clarify this introduction, the Von Neumann Architecture represents the best compromise between cost and performance. The analysis which is being carried out aims at improving the speed of the architecture controlling the related growth of its cost in terms of area and dissipated power. Only in this way, the Von Neumann architecture can get better remaining the unchallenged solution in the field of modern computing.

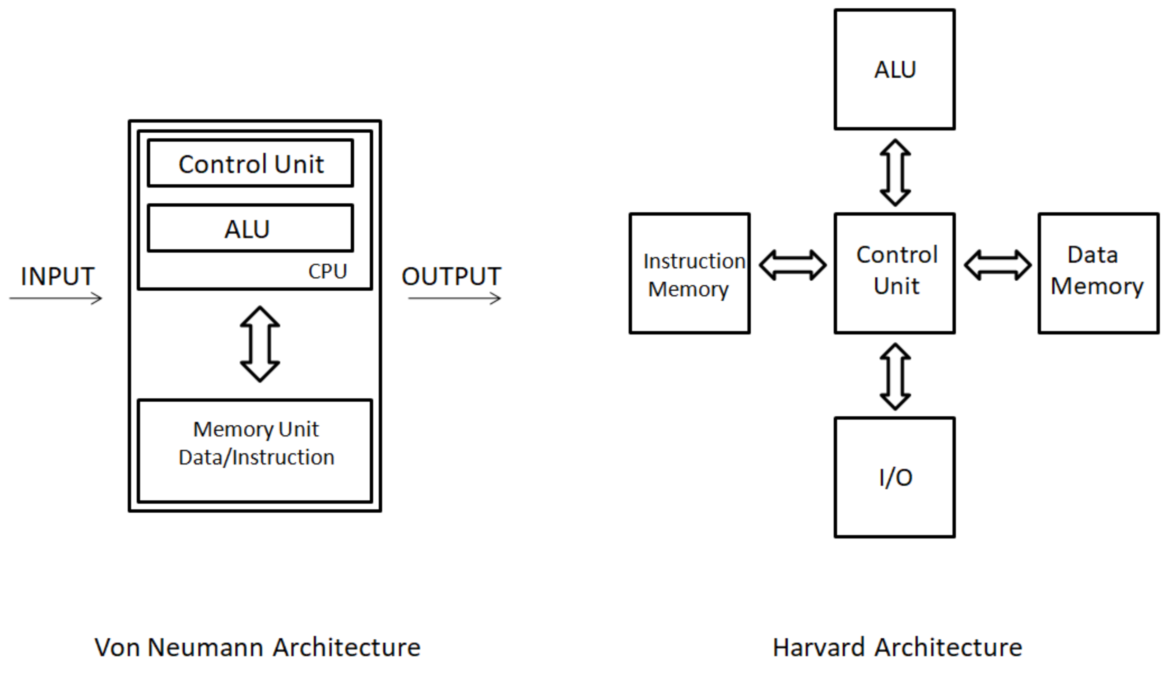


Figure 1.1. Von Neumann Architecture vs Harvard Architecture

1.2 Improvements adopted in the past

Different solutions have been proposed and adopted in order to overcome the bottleneck, some of these ideas have a long history, others belong to a more recent past.
Complementary Metal–Oxide–Semiconductor (CMOS) technology has spread since the early '70s and according to Moore's predictions, transistors have been scaled in the following years and their number has grown accordingly. As a consequence, Central Processing Units (CPUs) have become faster while memories did not, this is due to the need for more storage space, applications require larger memories and as a conclusion, they have turned into very expensive entities in terms of energy and time. If memories get huger their access time increases and also the energy required to perform this kind of

operation.

1.2.1 Memory Hierarchy

The goal of the Memory Hierarchy technique is to build a memory that is able to work almost at the same speed of the CPU and the information that the processor needs in a precise moment must already be saved inside it. This purpose has been achieved thanks to cache memories that exploit the concepts of spatial and temporal locality.

- **Temporal locality** When the processor accesses a location, it is likely that it will access it shortly after.

- **Spatial locality** When the processor accesses a location, it is very likely that shortly thereafter it will need to access nearby locations.

When the CPU has to run a program, all the information required is copied from the secondary memory (Hard Disks) to the main memory (RAM) and then provided to the processor. The idea, according to these two concepts, is to copy blocks of information in smaller and faster memories called cache memories in order to let the processor work just with this performing storage element. A hierarchy is therefore built as shown in Figure 1.2; here elements at the top of the pyramid are smaller and faster while the ones at the bottom are bigger and slower. The processor will look for data first in registers, if there is a miss, it will consult caches and so on until the bottom of the hierarchy is reached.

1.2.2 Prefetching

Prefetching consists of getting data from the main memory before it's actually needed and storing it in cache memory. Every x86 CPU has a bit of circuitry called prefetch unit which runs in the background, scanning both the CPU's internal registers and any cached instructions to determine what the execution unit is likely to need from memory next.

1.2.3 Branch Prediction

The environment which surrounds the Branch predictions technique is very vast, a lot of different methods of prediction have been developed. Basically,

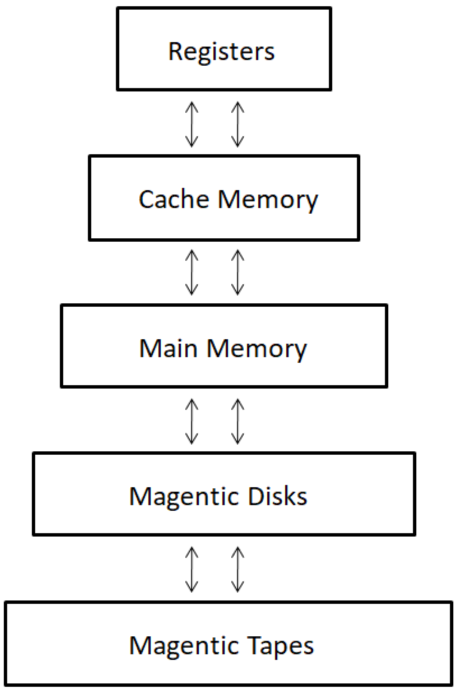


Figure 1.2. Memory Hierarchy

A CPU tends to execute instructions one after another, it goes through a section of memory in sequence, picking out instructions as it goes along until it has to choose between two paths of execution; a branch predictor is a digital circuit that tries to guess which way a branch will go before this is known definitively in order to improve the flow in the pipelined structure of modern processors. Branch prediction and prefetching are strictly linked to each other.

1.2.4 Loop nest optimization

Loop nest optimization is a technique that applies a set of loop transformations in order to improve cache performance and make effective use of parallel processing capabilities.

1.3 Most recent developments

Programs have grown in their complexity bringing to the demand for more capacious memories, in addition to this, technology scaling exploited to the limit has enabled the Arithmetic Logic Unit (ALU) to reach important peaks of speed. Therefore, the previously listed approaches have led to some improvements which are not sufficient to guarantee the same performance between memory and operating unit. Thus the modern concept of computation in memory has been taken into analysis.
In this review just the idea behind Computation In Memory is explained, the state of the art surrounding this topic is extremely wide, and extremely detailed architectural and technological aspects are presented. Four branches of the Computation In Memory subject are proposed according to the literature:

- **Computing Near Memory**
- **Computing With Memory**
- **Computing In Memory**
- **Logic In Memory**

This digression is meaningful since it clarifies the starting point and what this work of thesis aims at. The first three approaches are still comparable to a Von Neumann type architecture while the last one, the Logic-in-Memory approach, is the most modern one, it goes beyond the classical Von Neumann architecture in which CPU and memory are kept separate. It is the skeleton of the Programmable LiM architecture examined and tested in this work.

1.3.1 Computing-Near-Memory

Today's memory hierarchy is usually composed of multiple levels of cache, the main memory, and a non volatile storage element. The traditional approach consists of moving data from the storage to the cache levels and then processing them. Near memory computing, instead, aims at processing close to where data resides. An illustrative scheme is shown in Figure 1.2 to underline the difference with the traditional approach. The Computing Near Memory approach has spread starting from the first decade of the 21st century as a consequence of the appearance of heterogeneous 3-D integration of logic dies and memory dies based on Through-Silicon-Vias (TSV). This technique has

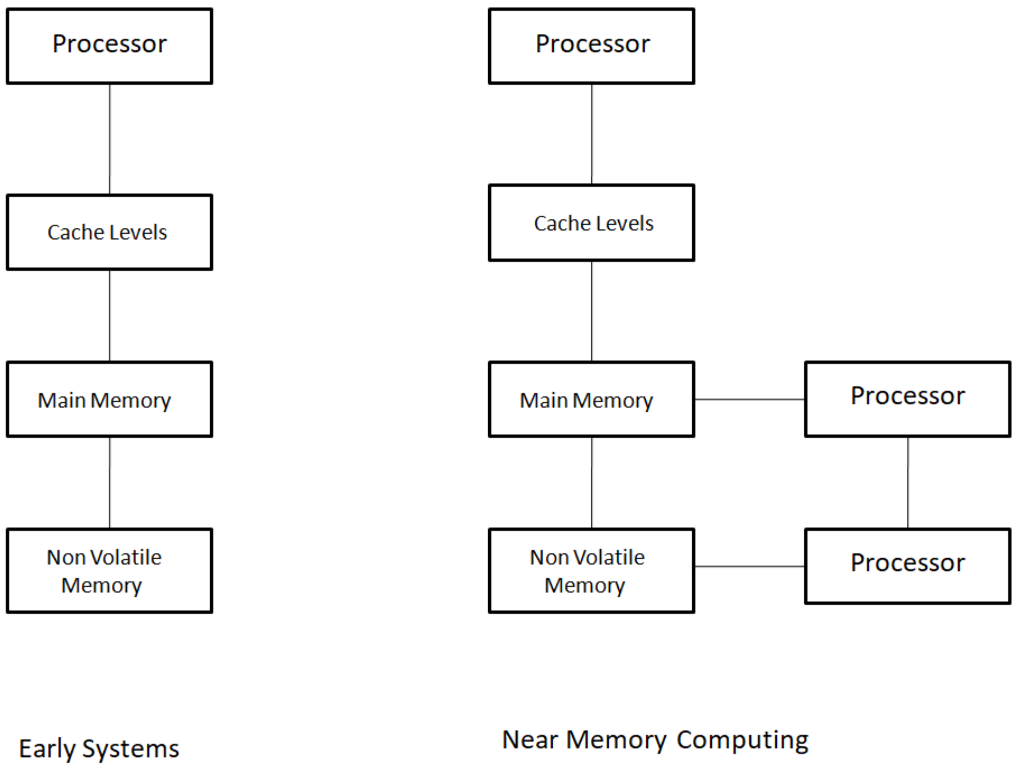


Figure 1.3. Early Systems vs Near Memory Computing

permitted to build memory cubes made of vertically stacked thinned memory dies. An accomplished project that has represented an important turning point is the Hybrid-Memory-Cube (HMC) [1] whose structure allows the integration of a memory cube stacked on top of a logic die called logic base (LoB). A simplified scheme of this is reported in Figure 1.4 The benefits of performing such operations closer to the disk are manifold: energy is saved by not having to move data through the memory hierarchy, interconnections get shorter, functionality over occupied area ratio increases, and the memory bandwidth requirements are reduced.
Other two related works found in the literature are the High-Bandwidth-Memory, [3] whose structure is composed of four layers of Hi-core DRAM stacked over a base logic die at the bottom and the Samsung Mobile Wide-I/O DRAM project [2], where high data bandwidth is achieved by adopting

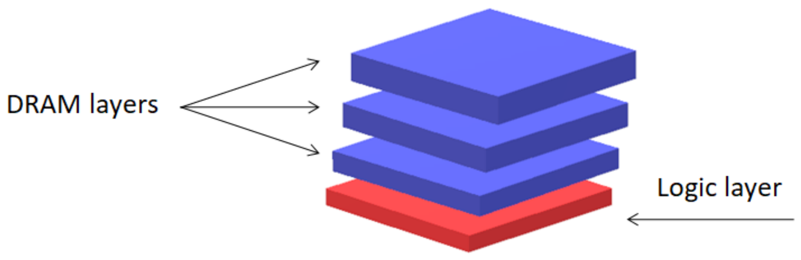


Figure 1.4. 3D stacked memory structure

a large number of I/O pins, memory density expansion is realized by stacking multiple chips using TSV technique. In this environment, Low-Power Double Data Rate memories have emerged to satisfy both demands for speed and power consumption.

1.3.2 Computing-In-Memory

Computing-Near-Memory (CNM) allows to move computing logic near memory by integrating DRAM with a logic die using 3D stacking; this approach has recently opened new opportunities in this area since memory can easily interact with logic. However this technique requires massive through-silicon-vias to connect logic to multi-layer memory stack and this could affect costs. Since data-intensive applications, such as artificial intelligence or real-time video streaming, require big computational capabilities, In-Memory-Computing (IMC) solutions have been proposed as a method for substantially increasing computational capabilities while simultaneously reducing energy consumption. In [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24] different methods to perform bitwise and more complex operations completely inside DRAM are presented; these techniques exploit the sense amplifiers, which is the peripheral circuitry used to extract data from the DRAM cells. Resistive RAMs (ReRAM) are taken into account to accomplish this project, they are based on resistive switching nanodevices (memristors or memristive devices) and represent one of the most promising candidates for next-generation nonvolatile memory (NVM) applications. These memories are also compatible with the current Complementary Metal-Oxide-Semiconductor (CMOS) fabrication process that enables their easy integration with the existing technology. It is noticeable that the memory structure has not been altered, it just exploits its analog features to process and sense data within it, processing is not performed outside memory hence the In-Memory-Computing name.

1.3.3 Computing-With-Memory

The Computing-With-Memory approach exploits the associative memory processing, here computations are transformed into Look up table operations employing a non-volatile memory. If a function has to be executed on two n-bit data, all the 2^{2n} possible patterns of the inputs can be mapped in a truth table. The different related works focus their attention on how to realize this idea in the best way. It is common to read among the related works that the two building blocks of this kind of processing are a Ternary Content-Addressable Memory (TCAM) and a memory, where a set of input patterns and their corresponding outputs are respectively stored. When a computation has to be performed, the input pattern is searched for in the TCAM, if there is a match, the corresponding result is returned from the memory. This process can be seen as a cooperation between the two storage elements from which the name Computing-With-Memory derives as conceptually depicted in figure 1.5.
The dense structure of CAMs allows to store a great number of patterns in the same unit of silicon. Memory-based computing has been shown significant energy efficiency for emerging non-CMOS memories, particularly well suited for dense non-volatile memory design. Several TCAM and CAM cells based on new technologies have been recently suggested to improve the density of pattern matching circuitry. In [9], [10] Resistive RAM (ReRAM) and Spin-Transfer-Torque RAM (STT-RAM) are presented as high speed and reliable non-volatile memories based on memristive devices and magnetic tunneling junctions, included in interesting applications such as real-time network intrusion detection, network packet routing, DNA sequence alignment. In [12] and [13] a 2T2R (two transistors, two resistors) TCAM memory is taken into account as the best possible solution to build a non-volatile memory that accomplishes the associative memristive computing.

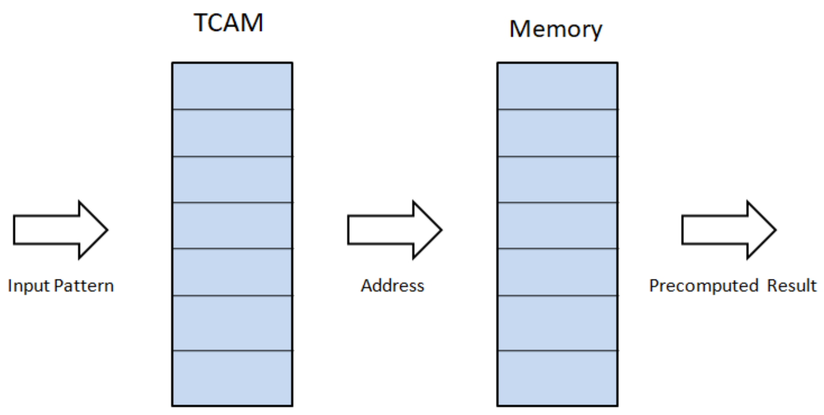


Figure 1.5. Computing-With-Memory

1.3.4 Logic-In-Memory

This is the newest and most revolutionary concept. The Von-Neumann paradigm is broken since memory and logic are no longer separated but merged into a single entity. Simple logic is integrated into the memory cell itself, these operations will not be performed outside the structure or in peripheral circuits but directly inside the memory array. In [25] CMOS logic for in-place data processing is integrated into the typical 6T SRAM array, memory structure is enriched with XOR and Look-Up-Tables (LUT) based units. In [26] a comparison between a Nano-Magnet-Logic (NML) LIM architecture and a CMOS one is presented to evaluate these two possible ways to realize the Logic-In-Memory concept. In [27] a hybrid MJT-CMOS architecture is shown, here magnetism and electronics are combined to develop next-generation Non-Volatile-memories, exploiting spintronic devices to overcome the issues of static power dissipation and volatility suffered by the complementary metal-oxide-semiconductor (CMOS) industry. To complete the historical picture in [28] the focus shifts completely to Complementary-MAgnetic-Tunnel junction logic (CMAT) which is meant to start the development of novel non-von Neumann architectures that may replace CMOS for general purpose computing.

Chapter 2

CLiMA as the starting point for PLiM

Configurable-Logic-in-Memory-Architecture (CLiMA) is an approach in which the cells of a memory array can be configured in order to perform different operations thanks to the logic integrated within them.
In Figure 2.1 a view of the cell filled with logic is portrayed, in this example, the basic cell can be seen as a box that no longer contains only a storage section but also units capable of performing some simple operations. The blocks are connected to each other, in Figure 2.1 this is not shown since it is simply explanatory and does not refer to any specific implementation of an architecture. Besides the introduction of extra logic, another novelty lies in the possibility to take as inputs data coming from other cells and therefore performing operations with them. The operations that can be performed and data exchange between cells are handled and determined by a proper pre-configuration of every single cell. In this way, according to the Logic-in-Memory (LiM) principle, the cell becomes capable of performing logic and arithmetic operations without moving data outside the memory array, reducing the number of accesses that are expensive in terms of performances and energy. In Figure 2.2 an illustrative possible layout of the memory array is reported; it shows how cells can be configured and interconnected in order to exploit more complex functions to satisfy the demand of different algorithms. Flexibility is another positive feature of CLiMA where intra-row, intra-column, inter-row, and inter-column communication can be achieved through interconnections. The weak point of this structure is that cells configuration is extremely application dependent, in other words, a specific configuration may be chosen to optimize one algorithm but it is

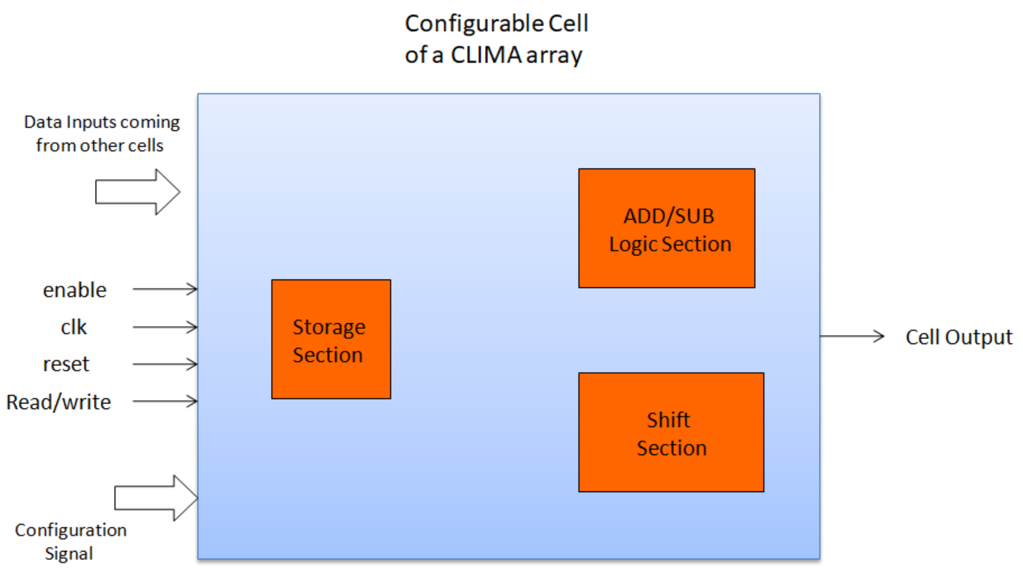


Figure 2.1. Illustrative overview of a CLiMA cell enriched with some logic

likely that it may make the execution of another algorithm slower with respect to a conventional approach. In particular, in [7] a configuration that optimizes a Convolutional Neural Network (CNN) network is analyzed and taken into account. Programmable Logic-in-Memory (PLiM) architecture aims at overcoming this weakness of the CLiMA idea by providing the architecture maximum flexibility in order to ideally make it suitable for each kind of algorithm. The key point is that memory cells, memory rows are not pre-configured to satisfy the execution of a specific algorithm but some of them are provided the feature to be smart. Being smart means to be capable of performing operations directly inside the array, not just one kind of operation depending on a predetermined configuration but ideally all of them. This is achieved by adding to these smart rows some building blocks, called Lego-LiM or Row Interfaces which can be seen as extensions of the smart rows capable of performing an operation. For instance, three Lego-Lim blocks could be: a full adder, a multiplier, and a shifter. Now the difference between CLiMA and PLiM approaches becomes clearer: with CLiMA two

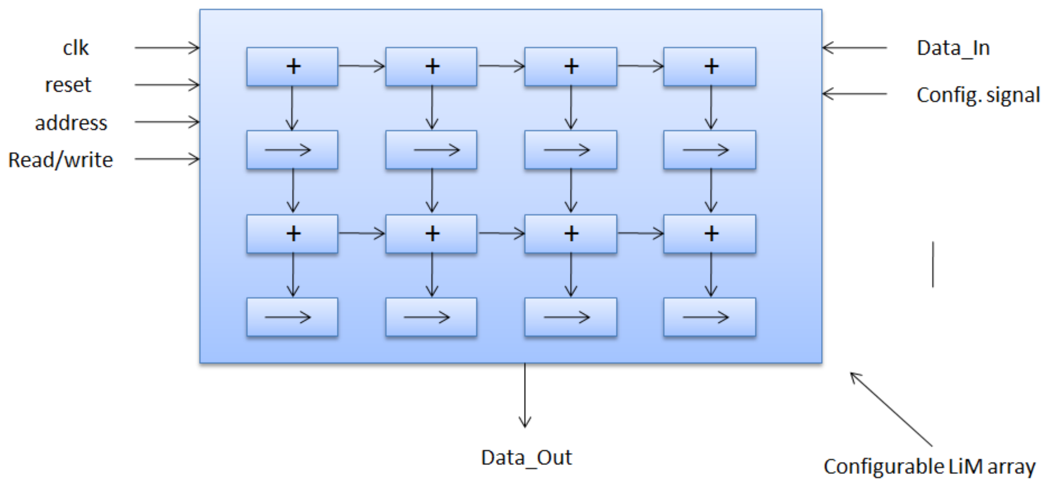


Figure 2.2. Overview of one possible configuration of a CLiMA array

different algorithms involving sums, multiplications and shifts may require a totally different configuration of the memory array, with PLiM instead, each algorithm can exploit the instruction set established by the type of Row Interfaces chosen to perform the available operations directly inside the memory array. It is clear that PLiM approach is heavier in terms of extra-hardware with respect to the CLiMA one, in fact, each smart row is enriched with one or more computational blocks with the same parallelism of memory. The goal of this work of thesis is to understand if the impact due to the extra hardware is balanced by a speedup of the application made possible by reducing the number of memory accesses. In chapter 3 PLiM is described, its structure, its features, and its functionalities are taken into account to make clear what executing an algorithm on this architecture means and to discuss the positive and negative aspects of this kind of solution.

Chapter 3

PLiM: how it works and its funcionalities

The key idea of Programmable Logic-in-Memory (PLiM) approach is to provide the system maximum flexibility, to make it able to execute several families of algorithms thanks to a technological independent and open-source library of components (Lego-Lim or Row Interfaces). In [8] a detailed explanation of PLiM following step by step each design evaluation is given. This work aims at testing its functionalities and for this reason schemes and figures sometimes will be illustrative, they will not report each signal or component of a specific section of the architecture but they will explain its working principle. In this chapter the PLiM memory structure is taken into account first, then the control part with the task of managing the flow of instructions is presented.

3.1 PLiM structure

In this section, the innovative memory structure that accomplishes the targets aimed by PLiM is presented and explained step by step to make clear what are the potentialities of the memory array to be exploited for the implementation of an algorithm.
The basic principle of PLiM is to have the main memory subdivided into a Smart section and a standard section as shown in Figure 3.1 The smart section is capable of managing both storing and computational functions thanks to Row Interfaces, logic or arithmetic blocks added in this special part of the main memory, the reason why the number of smart rows must necessarily be

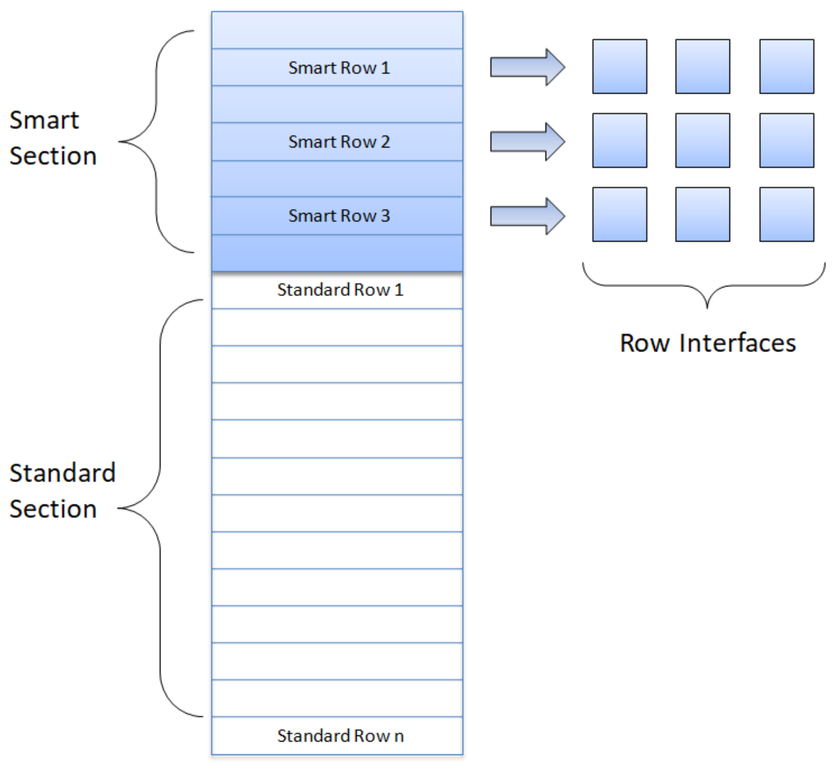


Figure 3.1. Structure of the PLiM architecture

restrained. On the other hand, the standard section almost retains its classic conformation to perform only a store function.

3.1.1 Smart Section

The smart section, the way it is managed, and the connection implemented between neighboring rows represent the novelty of this architecture. The number of smart rows chosen by the designer defines the smart section layout, each of them is surrounded by the corresponding UpRow and DownRow and then can communicate with them as shown in Figure 3.2. For sake of clarity every DownRow, except for the last one of the smart section, is also the UpRow of the next smart row but they are not different entities; while implementing an algorithm the same row will be labeled as UpRow or DownRow depending on the operation required at that specific step. Only

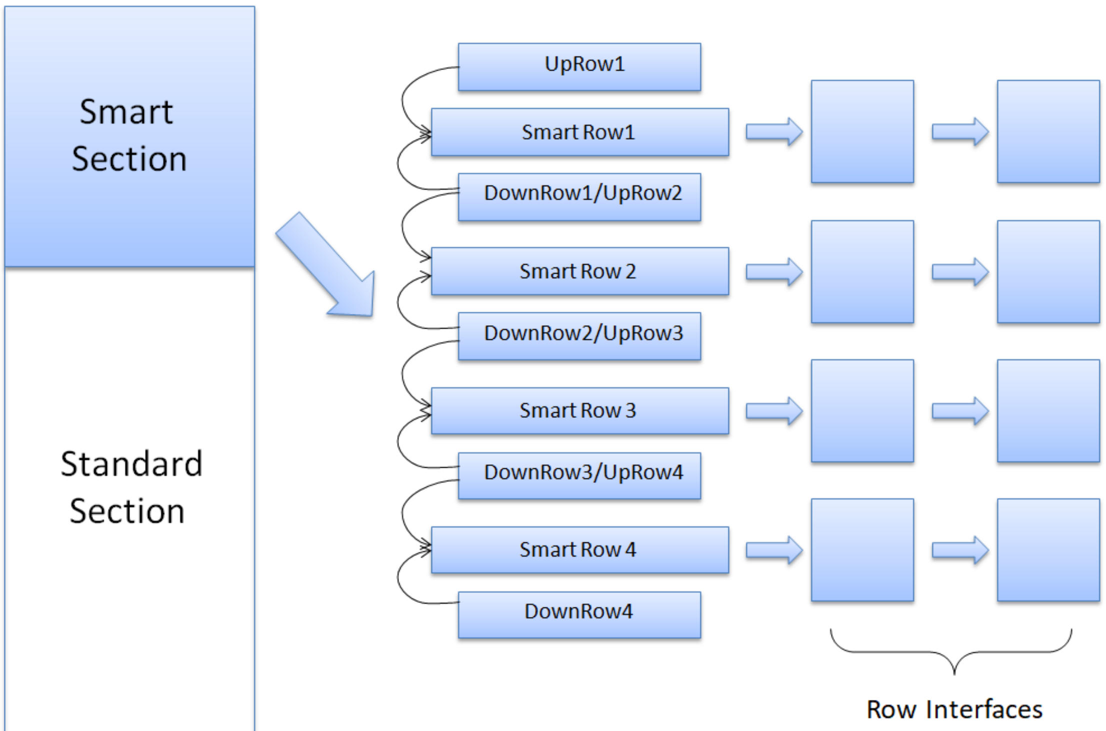


Figure 3.2. PLiM smart section organization

the smart rows are built with special cells called arithmetic cells (A-cells) as conceptually reported in Figure 3.3, with this configuration and plugged with the RCA row interface they become capable of managing many logic and arithmetic operations. It is noticeable how the cell, apart from the storage unit, is filled with a full adder a logic unit that filters the cell output. UpRows and DownRows , as standard sections rows are built with M cells.

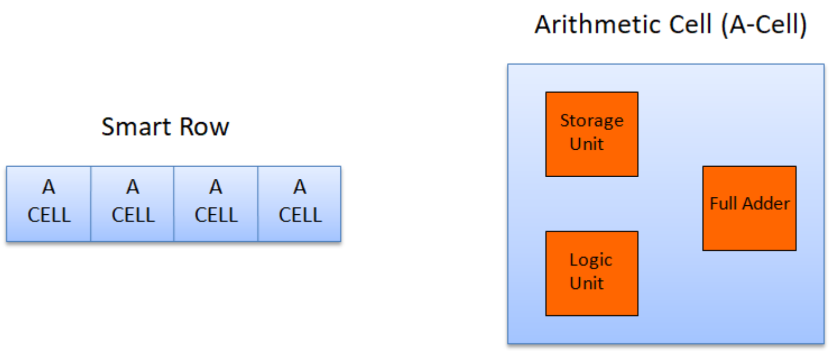


Figure 3.3. Smart Row and its building block

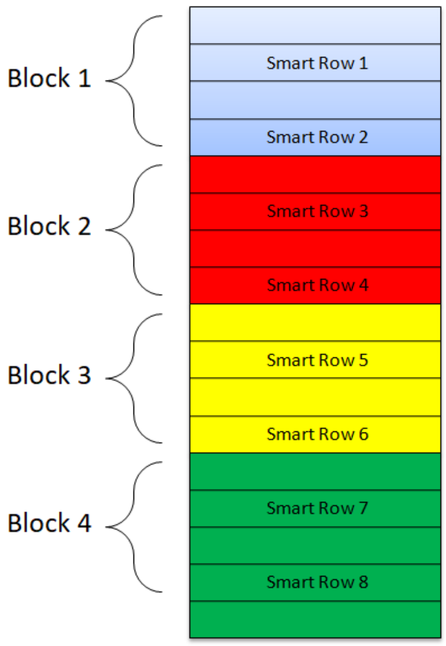


Figure 3.4. Division into blocks of the PLiM array

One other feature of PLiM is that smart rows can be grouped in blocks, the designer can choose how many blocks the smart section has to be divided into and as a consequence the number of smart rows contained in each block. In Figure 3.4 a PLiM structure with four blocks each containing two smart rows is shown. This extra feature in some cases could be useful since makes it possible to deactivate some portions of PLiM if some computations are not required by the algorithm in a specific point of its run.

3.1.2 Standard Section

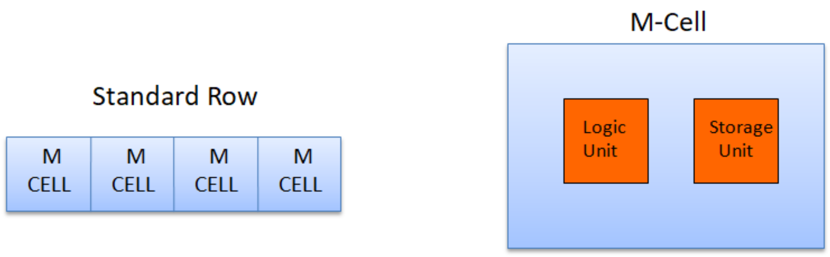


Figure 3.5. Standard Rows and its building block

The structure of the standard section is similar to that of normal memory, here standard rows, as UpRows and DownRows of the standard section, are built with M-Cells as conceptually portrayed in Figure 3.5; it is noticeable that the only difference with respect to a common memory cell stands in the integration of the logic unit that manages the output filtering. The standard section can communicate and provide data to the smart section thanks to the memory interface.

3.1.3 Memory Interface

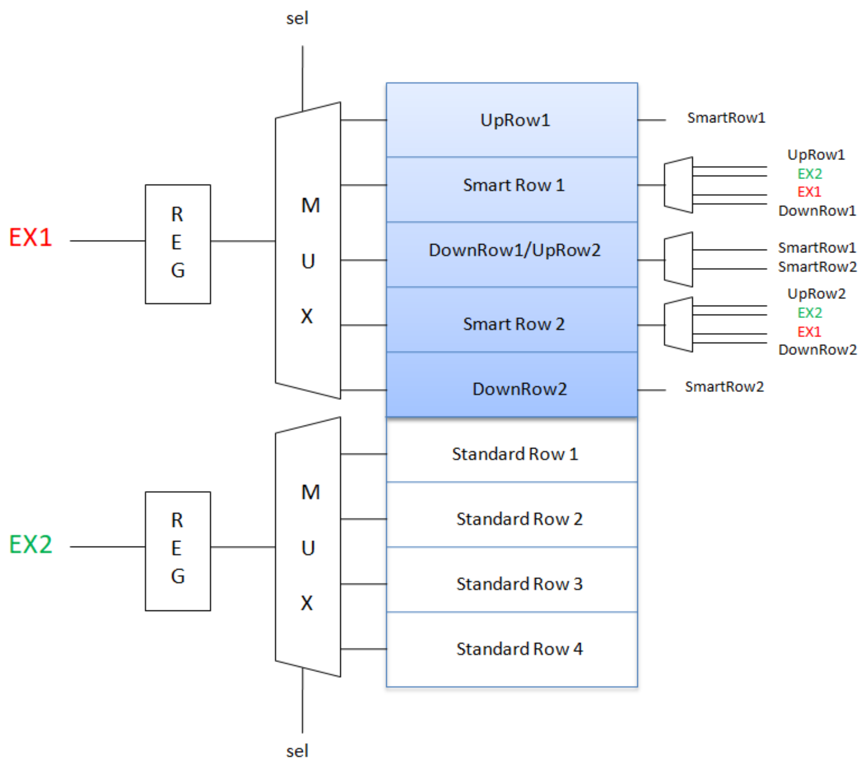


Figure 3.6. Memory Interface as the unit that manages data traffic through PLiM

The memory interface administrates the traffic of data through the PLiM structure. As it is noticeable from Figure 3.6, it is basically composed of multiplexers and registers that work like a cache. Figure 3.6 just provides an idea of the possible communications between memory rows, the important point to highlight is that a direct communication, that is within one clock cycle, is permitted just between rows belonging to the same smart row environment (UpRow n, Smart row n, DownRow n)

3.1.4 Row Interfaces

Row Interfaces (RIs) represent the engine of PLiM, they are extra hardware added to smart rows that can perform operations directly inside the memory.

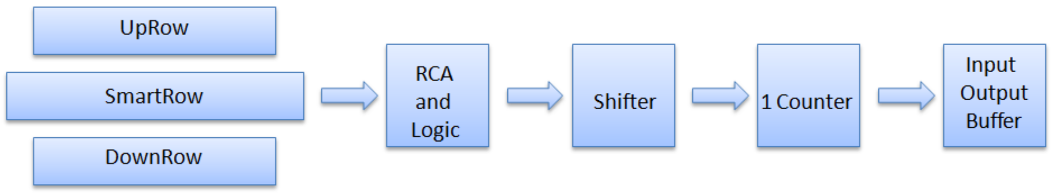


Figure 3.7. Row Interfaces organized in a chain

Row Interfaces set the type of instructions that can be handled in LiM mode depending on the application purpose. Visual feedback of RIs organization and management is reported in Figure 3.7. For design choices, the first RI must be the RCAandLogic block, which is able to perform a lot of logic functions, additions, and subtractions while the last block of the chain must be the Input/Output buffer, which is used to store the output of previous blocks and to manage load and store operations. The designer builds the Row Interfaces chain depending on the application, in this example a block able to perform shift operations and, a block able to count the number of ones of data are added to the structure. The chain sets the order of the operations that can be handled in one single instruction of the uROM section 3.2. Once again referring to Figure 3.7, this configuration allows PLiM to perform in one instruction, so within a clock cycle, a sum with the RCAandLogic block, a shift of the result, the count of ones of the shifted data. The final result of a PLiM instruction is always saved in the Output buffer with the exception of Load and store operations. The user is not forced to use all the blocks that compose the chain; if one or more of them are not needed in a precise algorithm step, they can become transparent during the execution of a LiM instruction. The execution of a very simple algorithm is reported in Figure 3.8 to better understand this point. The algorithm requires first a shift operation of the content of the smart row and then to sum the shifted data with the original content of the smart row; the algorithm cannot be executed in one instruction since the two operations do not respect the order of blocks but

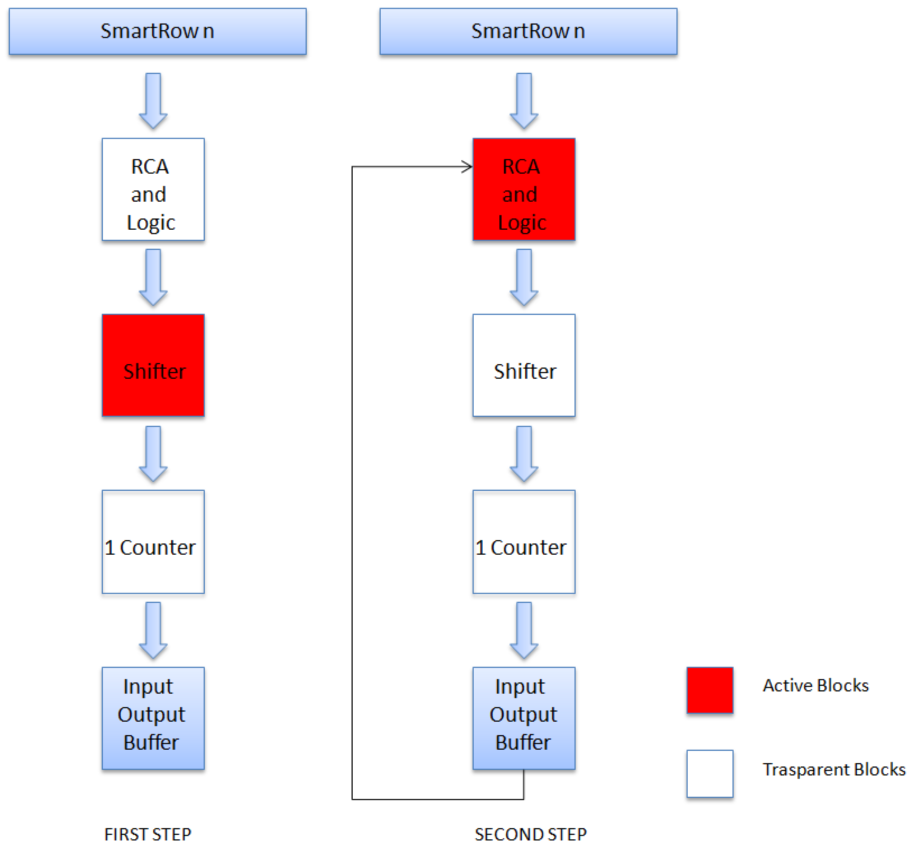


Figure 3.8. Row Interfaces configuration to accomplish a very simple algorithm made of a shift and a sum

in two steps:

- **Shift**: RCAandLogic and 1counter are not needed, they become transparent, only the shifter is active and it takes as input the content of the smart row, the result of the shift operation is stored in the output buffer.
- **Sum** : The RCAandLogic block is the only one to be active and it takes as inputs the content of the smart row and the data previously stored in the output buffer. The result of the sum is stored in the output buffer.

In the example of Figure 3.8 the same smart row is taken into account and it is shown how PLiM handles concatenated operations with the storage of partial results. RIs also shares the great feature to all have the same interface as shown in Figure 3.9 In this way for a user, for a designer adding a RI becomes very intuitive and simple, the steps to follow are:

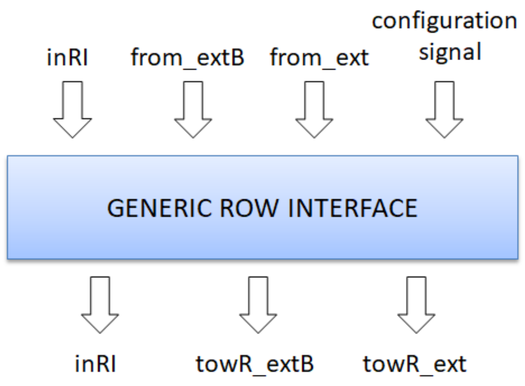


Figure 3.9. The common interface of each RI

- **Define the operational functionality of the row interface** : The block with the interface shown in Figure 3.9 is built describing the operation it is able to perform.
- **Add the new block to the chain of RIs** As discussed before, RIs are organized in a chain; to insert a new block its inputs have to be connected to the outputs of the previous block and its outputs have to be connected to the inputs of the following block.
- **Add the Opcodes** Since a new operation has been introduced, it is necessary to add the Opcode for the single operation itself and all the opcodes linked to all the concatenated operations in which the new RI can be involved.

3.2 Control Section

In this section, the unit that manages the flow of instructions through the PLiM structure is analyzed. The PLiM is a micro-programmed machine and an illustrative scheme that depicts its way of operating is reported in Figure 3.10, again it is not a perfect portrait of the real implementation but it clarifies its working mode.
A micro-programmed machine is able to execute programs thanks to the micro-instructions stored in the μROM, a program is a set of instructions

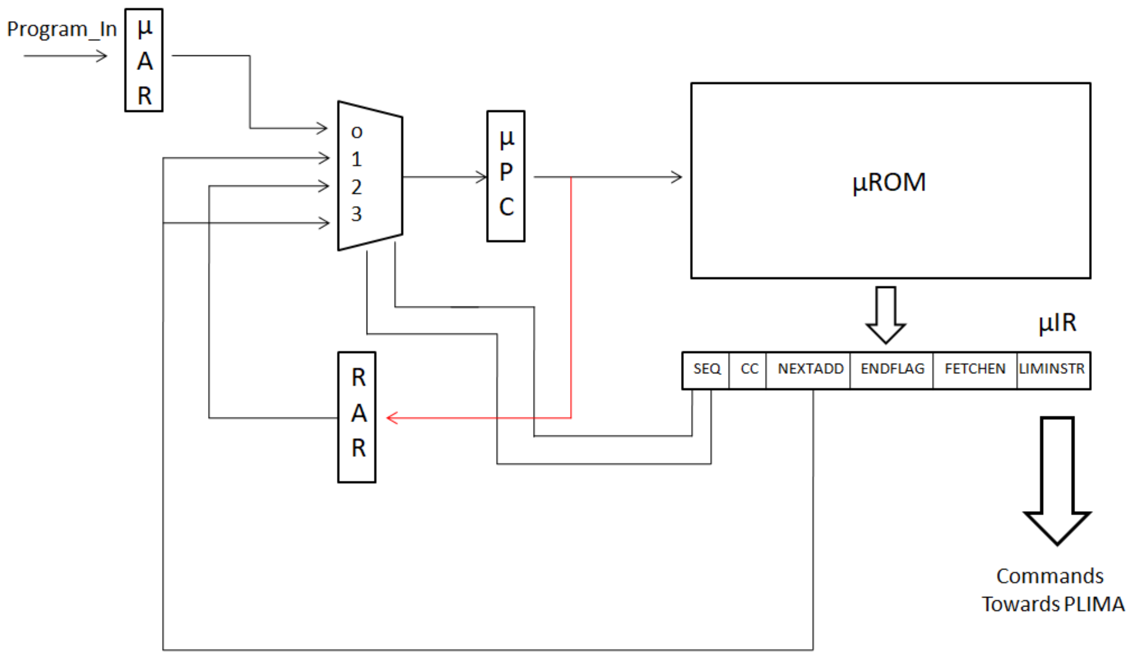


Figure 3.10. The micro-programmed machine that handles the flow of instructions in a PLiM

and a lot of them can be stored in the μROM . When a specific program has to be executed, the μAR (Micro Address Register) is filled with the address that points at the location where the first micro-instruction of that program is stored, then this address is copied into the μPC (Program Counter) that accesses the proper μROM location. After the execution of the first instruction, the machine will evolve according to what has been previously stored in the memory. Evolving through instructions means updating at each algorithmic step the content of the μ PC with the address of the instruction that has to be executed at each specific point. Every time the μROM is accessed by the address stored in the μPC, the addressed content is taken from the memory and copied in the μIR(Micro-Instruction-Register). The rules that manage the succession of instructions are set in some of the μIR fields; the remaining fields are filled with the commands that have to be sent to the PLiM to make it perform the proper operation at each step. In general, programming a machine by compiling a μROM is extremely flexible and user-friendly since this operation consists just of writing zeros and ones in the right fields.

3.2.1 μROM fields that manage the succession of instructions

Once more referring to Figure 3.10, the first five fields of the μROM deal with the evolution of instructions and programs and they are:

- **SEQ**: It gives information about the next instruction to execute, it is composed of two bits and therefore it can assume four possible configurations:
  - When SEQ = "00" The content of the μAR is copied into the μPC to let a program start with its first instruction.
  - When SEQ = "01" The machine fills the μPC with the content of the NextAdd field.
  - When SEQ = "11" it means that a subroutine call has to be performed, the μPC is filled with the address at which the subroutine begins, this is always stored in the NextAdd field, and the return address is stored into the RAR (Return-Address-Register). Once the machine has satisfied the subroutine execution, the μPC will be filled with the content of the RAR to continue the normal execution of the program.
  - When SEQ = "10" it indicates that the subroutine has terminated its execution and that the μPC has to be filled with the content of the RAR whatever the content of the NextAdd field is.
- **CC**: It is the conditional code and it indicates if a jump has to be taken or not. In this architecture, it has to be asserted just for the wait instruction.
- **NextAdd**: The micro-programmed machine is based on explicit addressing, therefore each instruction contains in this field the address of the next instruction to execute.
- **EndFlag**:It is needed to handle the flow of different programs and it alerts the control unit when the last instruction of a program is being executed.

- **FetchEn**: Also this field manages the protocol to start the execution of a new program, its function is to update the content of the μAR to start the execution of a new program.

Summarizing the crucial points of this section, it is important to understand that PLiM is able to execute instructions in sequence, to perform jumps to a subroutine, to return to the normal application execution that was running before the subroutine call.

3.2.2 μROM fields containing commands for the operating unit

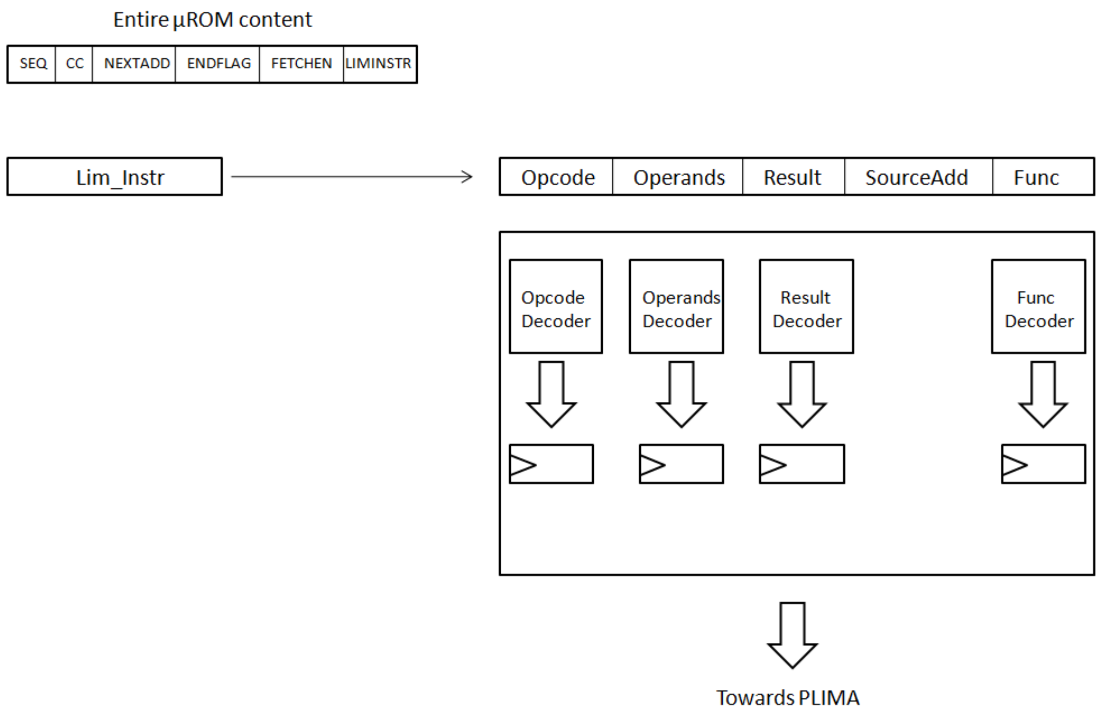


Figure 3.11. μROM operative commands and their related decoders

The μROM fields left to be explained are those that manage the actions to be carried out by PLiM; they are coded commands to be sent to the PLiM structure to let it work in the desired way, after a decoding phase. This section of the μROM is composed of 5 fields but just four of them need to be decoded, a representation is provided in Figure 3.11. More in detail the fields with their function are:

- **Opcode** It represents the coded string to refer to a specific operation that PLiM has to perform through the row interfaces or the input/output buffer. This field of the μROM manages the activation and deactivation of row interfaces as discussed in subsection 3.1.4. For instance, if the Opcode of a μinstruction is "RCAandLogic", the adder is the only active row interface required by that instruction, all the other blocks become transparent and the Opcode decoder translates the string "RCAandLogic" into its associated operative code. Every single operation and every possible chained operation must be associated with an operative , which is a set of bits whose length depends on the number of row interfaces inserted by the designer.

- **Operands** This field refers to the operand or to the two operands sent as input to the block that is meant to be used for the corresponding μinstruction. All the possible patterns regarding this section will be discussed more in detail in chapter 4. Each possible input pattern for a row interface is coded with a literal string and associated to a string of bits, the literal string has to pass through the operand decoder before being sent to PLiM structure in order to command it.

- **Result** This is the field with the fewest possible options, in fact it establishes where the final result of an operation has to be stored, the two possible alternatives are the input buffer or the output buffer. The input buffer is linked to the "StoreExt" literal string while the output buffer is linked to the "StoreBuff" literal string; the result decoder translates these two strings into their associated operative code in order to let PLiM understand where the result has to be saved.

- **Address** This field does not require to be decoded, it provides PLiM the functionality to take operands stored in the standard section as possible inputs for the row interfaces. The field, in particular, indicates to the structure the address where the desired data is stored in order to retrieve it.

- **Function** This field is linked just to the RCAandLogic block; when the adder has to be used, in the last field of the μROM the function that it has to accomplish has to be specified. The RCAandLogic block is capable of performing sums, three types of subtractions (A-B, B-A, -A-B), and the most common logic functions such as AND, XOR, OR functions. Once again, all the possible operations the adder is able to perform are coded with meaningful names such as "SUM", "XORop", "SUB1", "SUB2", "SUB3", they are sent to the function decoder that translates them into the operative commands to let PLiM understand the kind of operation the RCAandLogic block has to accomplish. When the row interface involved in a μinstruction is different from the RCAandLogic one, the function field has to be filled with the "nullFunc" literal string.

Chapter 4

How to implement an algorithm on PLiM

When an algorithm has to be implemented, the μROM has to be filled in the proper way and the Programmable-Logic-in-Memory (PLiM) structure has to be suitable for the requirements of the algorithm. The μROM is the heart that moves the structure and the algorithm must be precisely described within it, μinstruction after μinstruction. It is significant to specify what are the possibilities that the μROM offers, but at the same time also what it is not intended to achieve, it is a crucial point to have a broad vision on all the ways in which an algorithm could be implemented to choose the best one. A generic μROM instruction with the attention focused on the field of major interest can be the following one:

Multiplier & Bo_OthR & StoreBuff & "111111" & nullFunc

As it is noticeable these are the five fields discussed in subsection 3.2.2 and represent the commands that has to be sent to PLiM to make it perform the desired operation. The paths to follow in the algorithm implementation phase, and as a consequence PLiM functionalities will be listed and explained through these μROM fields. The first field is related to the specific operation that has to be performed at a specific algorithmic step, it can be filled with each available operation depending on the instruction set determined by the row interfaces added to the structure. The fourth field (address field) and the fifth field (function field), have been discussed in subsection 3.2.2 The second filed related to the source operands and the third one related to the final destination of data have to be analyzed more in detail since they represent part of the novelties proposed by PLiM.

4.1 Source Operands

This is the field that establishes what is the operand or what are the two operands to be sent as input to the row interface chosen in the opcode field. There are a lot of possible patterns and therefore many different ways of implementing an algorithm.
First of all, when in the operand field just one input data is declared, it is understood that the other input is always the content of the smart row. The following operand patterns belong to this category:

- **ROW** The input of the Row Interface is only the content of the smart row.
- **UpR** The first input of the row interface is the content of the smart row while the second one is the content of the UpRow.
- **DwnR** The first input of the row interface is the content of the smart row while the second one is the content of the DownRow.
- **BuffO** The first input of the row interface is the content of the smart row while the second one is the content stored, from a previous operation, in the output buffer.
- **BuffI** The first input of the row interface is the content of the smart row while the second one is the content stored, from a previous operation, in the input buffer.
- **Othr** The first input of the row interface is the content of the smart row while the second one is the content of the shared section location whose address is retrieved from the address field of the μROM.

Then, considering the case in which two inputs are declared in the operand field of the μROM, the possible combinations are:

- **Bo_UpR** The first input of the row interface is the content stored in the output buffer, the second one is the content of the UpRow.
- **Bi_UpR** The first input of the row interface is the content stored in the input buffer, the second one is the content of the UpRow.

- **Bo__DwnR** The first input of the row interface is the content stored in the output buffer, the second one is the content of the DownRow.
- **Bi__DwnR** The first input of the row interface is the content stored in the input buffer, the second one is the content of the DownRow.
- **Bo__Othr** The first input of the row interface is the content stored in the output buffer, the second one is the content of the shared section location whose address is retrieved from the address field of the μROM.
- **Bi__Othr** The first input of the row interface is the content stored in the output buffer, the second one is the content of the shared section location whose address is retrieved from the address field of the μROM.

When mentioning UpRow, DownRow, BuffO, BuffI, they are referred to the same smart row, it is not possible to make operations between elements belonging to different smart rows directly. Each smart row has its UpRow, DownRow, Input/Outbuffer whose content can be sent as input to the row interfaces linked to the smart row they are associated with. As it is noticeable PLiM does not allow to perform directly, that is within a single μinstruction, operations between:

- The UpRow and the DownRow
- The UpRow and a data coming from the shared section
- The DownRow and a data coming from the shared section

These mentioned interactions are anyway achievable in two instructions thanks to Load and Store functions, explained in detail in section 4.2.

4.2 Result

This is the μROM field that specifies where the result of a Lim instruction has to be stored. It could be filled with the literal string "StoreExt", which is referred to the input buffer, or with the literal string "StoreBuff", which is referred to the output buffer. Every time an operation involving row interfaces is performed, the result has always to be stored in the output buffer. In order to clarify this concept, when a computational block is used during the execution of a μinstruction such as the "RCAandLogic" block or the Multiplier block, the field of the μROM has to be filled with the

"StoreBuff" text. When a move operation has to be performed, that is a store or a load function, the result filed has to be thought in a different way. Load and store functions are very useful in this kind of architecture for managing data organization and movement inside PLiM. One algorithm likely needs to make an operation between two data to be retrieved from precise locations of the smart section; without the move function this is not achievable, for this reason, it is meaningful to understand how PLiM manages these two functions.

4.2.1 Load Function

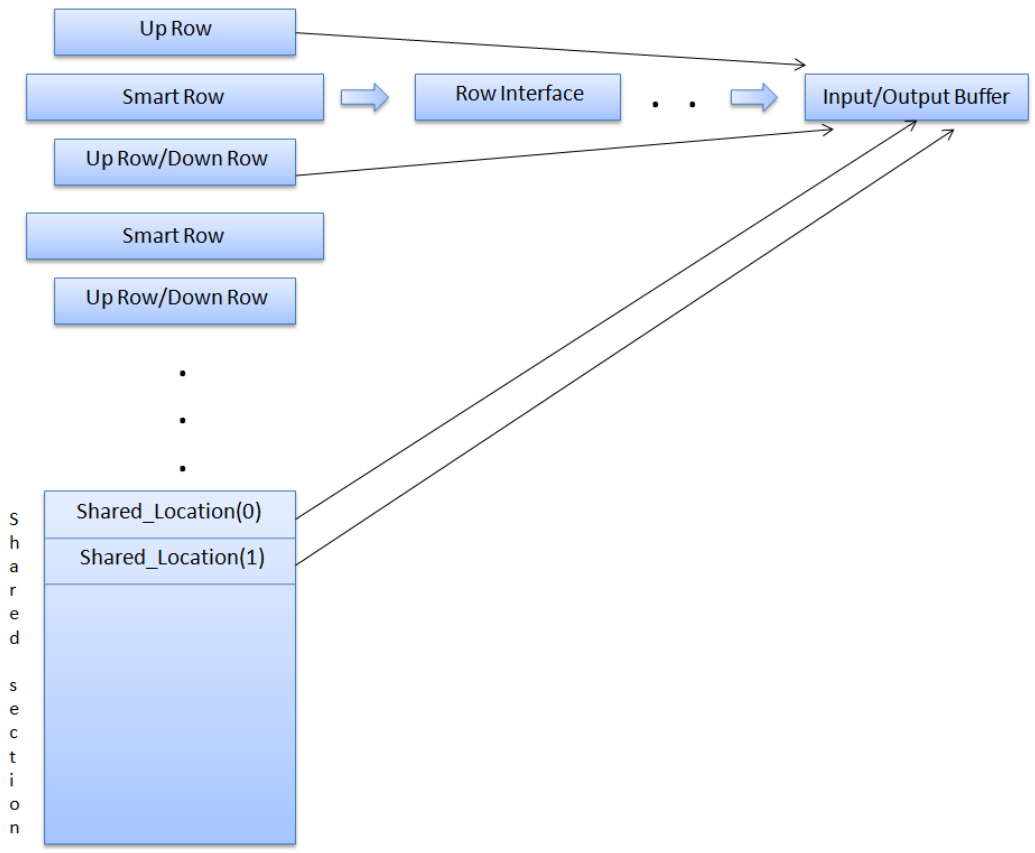


Figure 4.1. Data movement allowed by load instruction

Load function allows to perform operations cited in section 4.1 that are not achievable directly within one μinstruction. In particular, this function enables all the data movements marked in Figure 4.1, it is able to manage the

transfer of data from a location of the shared section, from the UpRow, or from the DownRow to the input buffer. As a consequence, the result of a load instruction has always to be stored in the input buffer, therefore the related result field of the μROM must be filled with the "StoreExt" text. Summing up, load instruction takes charge of data movement from the memory array to the buffer.
For instance the two following instructions shows how it is possible to make an operation involving the content of the UpRow and a shared data in two instructions.

Load & Othr & StoreExt & "11111" & nullFunc
RCAandLogic & Bi_UpR & StoreBuff & nullAdd & nullFunc

The firts instruction retrieves the data stored at address 31 and moves it towards the input buffer, then the second instruction perform a sum between the content of the input buffer, which corresponds to the desired shared data, and the content of the UpRow. The final result as explained is stored in the output buffer.

4.2.2 Store Function

The Store function manages data movement in the opposite direction with respect to the load function, that is from the buffer to the memory array as shown in figure Figure 4.2. As it is noticeable, it allows to copy data from the buffer to the UpRow, to the DownRow, or to the smart row itself. The syntax to be used in the μROM for this kind of instruction is particular and it represents the only exception to the rules listed in this work. For instance two instructions accomplishing to store in the DownRow the result of a sum are reported.

RCAandLogic & Othr & StoreBuff & "11111" & SUM
Store & DwnR & StoreExt & nullAdd & nullFunc

The first instruction performs the sum between the content of the smart row and the data located at address 31 of the memory and stores the final result in the output buffer, the second instruction copies this result from the buffer to the Down row. As it is noticeable, the fields function seems to be inverted in this specific case, in fact, the operand field, the second one in the order, has to be filled with the location in which the data coming from the

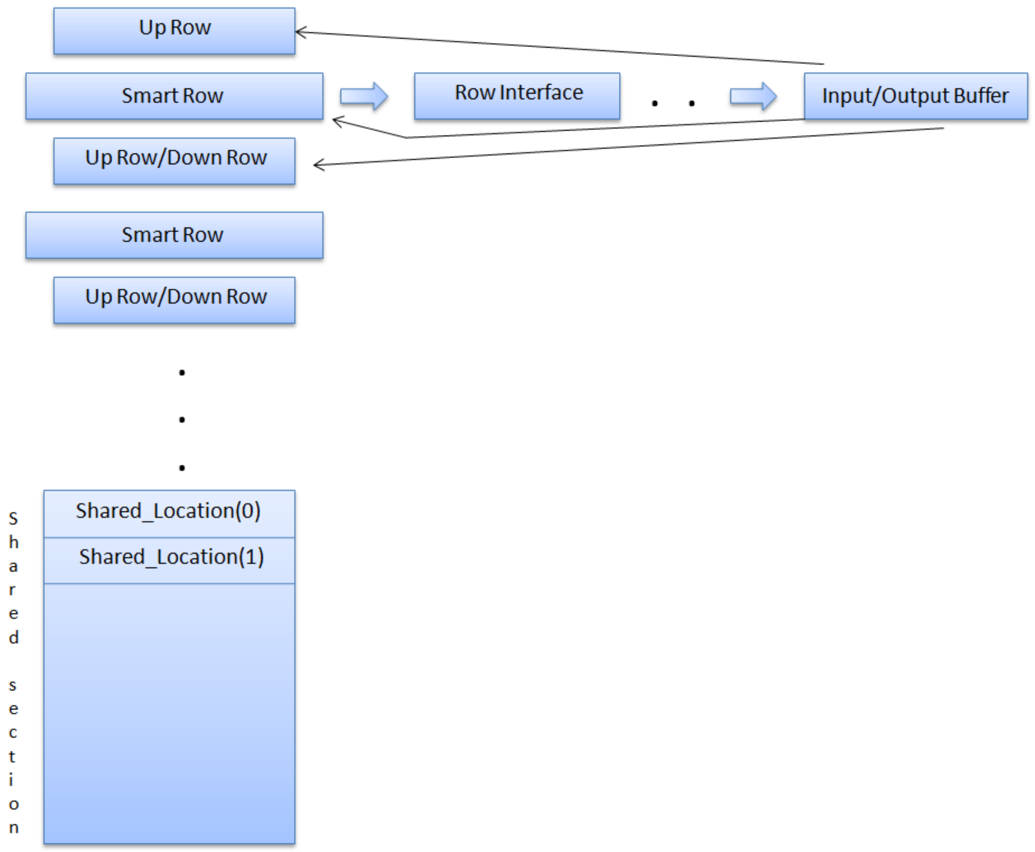


Figure 4.2. Data movement allowed by store instruction

buffer has to be stored, the result field instead becomes now the source from which data has to be retrieved. In order to perform the store instruction, the result field has to be filled with the "StoreExt" literal string.

4.3 Step to follow for the implementation of an algorithm

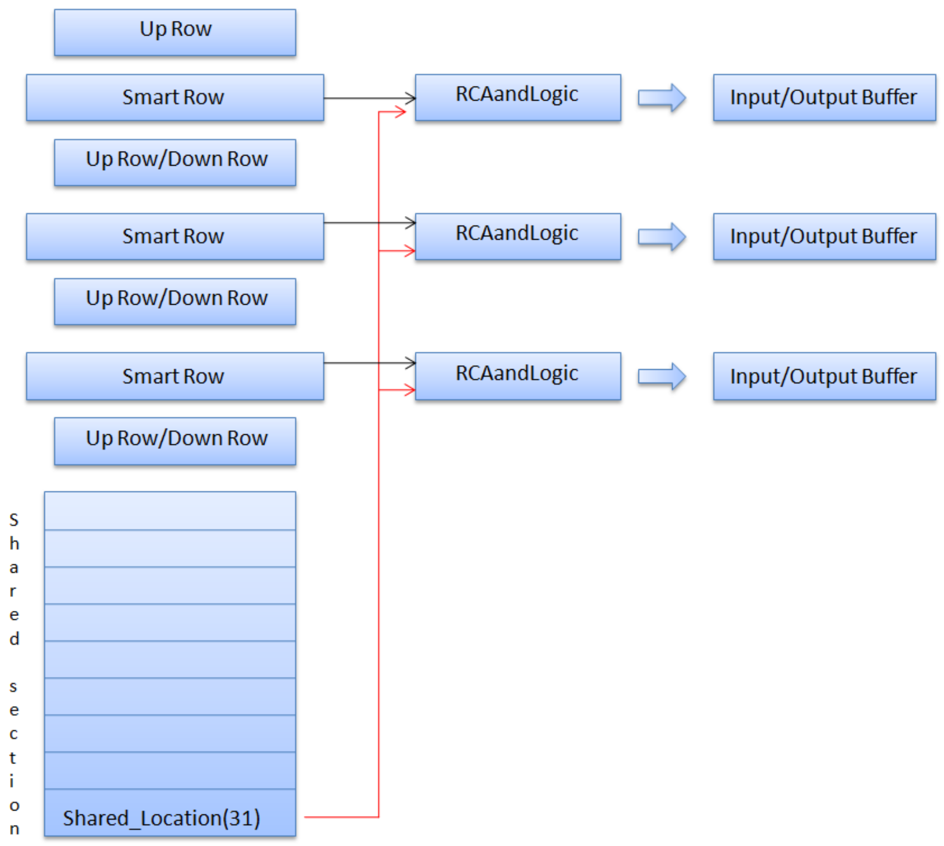


Figure 4.3. Single instruction multiple data protocol to perform operations inside PLiM

Once all the PLiM functionalities and capabilities have been explained, all the steps required for the implementation of an algorithm on it are listed. One important concept has to be underlined; PLiM handles SIMD operations (Single-Instruction-Multiple-Data), that is that all the commands sent by the μROM to the PLiM has to be applied to the whole memory unless some blocks are deactivated subsection 3.1.1. Smart section division into blocks is limited and tricky to manage, for this reason, blocks deactivation is not suggested unless the number of smart rows is restricted or it is strictly necessary. For instance the following μinstruction is considered:

RCAandLogic & OthR & StoreBuff & "11111" & nullFunc

It makes PLiM perform, at the same time, the sum between each smart row content and the data stored at address 31 of the standard section as conceptually depicted in Figure 4.3, the final result is, as usual, stored in the output buffer related to each smart row. Summing up PLiM is able to apply the same operative commands described by one μinstruction to every smart row composing the smart section array. If n smart rows have been instantiated in the architecture, PLiM can achieve to carry out n identical operations; identical means same involved row interface, same source operands, same result location, in case of RCAandLogic is involved, the same operation among the possible ones this block is able to achieve. As a consequence, for instance, it is not possible to make a sum within one smart row and one subtraction in another one at the same time, it is not possible to use two different shared data within one single μinstruction, it is not possible to realize a shift operation within one smart row and simultaneously a multiplication in another one. This sometimes represents an advantage since it allows to parallelize some steps of one algorithm, sometimes it may force PLiM to realize unwanted computations or data movement, for this reason, data organization is fundamental in this architecture.
There could be a lot of different ways to implement one specific algorithm on PLiM, some common steps of evaluation in order to choose the optimum are presented:

1. Analyze the algorithm that has to be implemented in PLiM and all the operations required to execute it.

2. Considering an architecture with only the "RCAandLogic" row interface and the input/output buffer instantiated by default for design choices, give PLiM the capability of performing such operations by inserting the row interfaces to accomplish the desired functions. Row interfaces have to be described in their functionality and with the same default interface as discussed in subsection 3.1.4.

3. Examine the algorithm in order to understand which steps can be parallelized and which cannot.

4. Schedule operations on the basis of the considerations made in the previous point relying on all the features of PLiM. All the possibilities described in section 4.1, subsection 4.2.1, subsection 4.2.2, have to be taken

into account to exploit the entire solution space provided by PLiM.

5. Modify the PLiM structure in order to make it suitable for executing the algorithm. The PLiM architecture is described with a full parametric code, as a consequence, its conformation can be easily modified to adapt it to every demand. Different algorithms may require a different number of smart rows to be implemented, the smart section size and the shared section depth can be easily changed depending on the requirements.

6. Fill properly through the data maker both the smart section and the standard section.

7. Write the μROM instructions in order to give PLiM commands to execute each required algorithmic step.

8. If necessary modify the μROM size if a specific algorithm needs more memory space to write the μcode to execute it.

9. Simulate the algorithm to check if it has been implemented in the correct way checking the final result.

Chapter 5

Algorithms implementation and testing

In this chapter, the implementation of different algorithms on Programmable-Logic-in-Memory (PLiM) is presented in order to evaluate how they fit the architecture, how the structure has to be shaped in order to execute them. In the end, the system is evaluated in terms of performance through some meaningful parameters to check if some benefits may come from this new approach.

5.1 Advanced Encryption Standard 128 algorithm introduction

The first algorithm to consider is the Advanced-Encryption-Standard 128 (AES128), it is the standard set by the U.S. National Institute of Standards and Technology in 2001 for the encryption of electronic data. Before analyzing in detail the algorithm, the steps of an encrypted communication are presented in order to clarify the context and the reason why the AES128 has been taken into account in this thesis work. As depicted in Figure 5.1, an encrypted communication starts with the exigency of sending a message called plain text. In order to be sure that the message can only be understood by the intended recipient, the sender modifies it using a key, the cipher text is sent to the other user who is able to decode it since he knows the key and

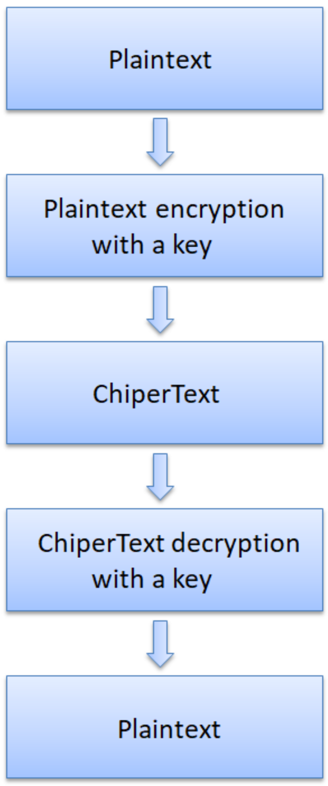

Plaintext
Plaintext encryption with a key
ChiperText
ChiperText decryption with a key
Plaintext

Figure 5.1. Encryption-Decryption procedure

the rules of encryption applied by the sender. In particular, when the key used in the encryption phase is the same involved in the decryption phase, the communication is fulfilled with a so called symmetric key. Symmetric encryption is faster and more efficient than asymmetric one, therefore it is typically used for encrypting large amounts of data, e.g. for databases. The AES is a symmetric encryption algorithm where the word to encrypt, that is the plain text is 128 bit long, while the key can vary depending on the degree of security required, it may be 128, 192, or 256 bit long. In this work the AES128, where 128 is referred to the key length, is considered. The AES is based on a substitution and permutation network that accomplishes the Shannon's cryptographic principle of "confusion and diffusion" to provide the algorithm the desired security. In most of the cases and also for the AES, the substitution boxes (S-Box) and the permutation boxes (P-Box) are made of simple operations. As a consequence, the AES algorithm is fast whether if it is developed in software or in hardware. It is relatively simple to implement, requires little memory and offers a good level of protection, reasons why its implementation has been preferred to other encryption algorithms.

5.2 AES128 operations involved

The AES algorithm is composed of a number of iterations called rounds involving the same operations; since AES128 is being analyzed, the number of rounds is fixed to ten. The algorithm can be summed up by the following pseudocode:

```
$Addroundkey0$

$for(j = 1; j = 9; j++)\{$
$SubBytes()$
$ShiftRows()$
$MixColumns()$
$AddRoundkey(j)\}$

$SubBytes()$
$ShiftRows()$
$AddRoundkey10$
```

As it is noticeable the round zero is composed of only the "Addroundkey" operation while the last round does not require the "MixColumns" operation. Before explaining all the operations involved the role of the key in this algorithm has to be clarified. The 128-bit long key is used to generate the eleven sub-keys required in the eleven Addroundkey operations; the sub-keys are only functions of the original key and they are processed through the "Rijndael key schedule" whose composition will not be analyzed in this work since it is not meaningful for the final implementation and results. However the whole AES128 algorithm with the sub-key generation has been implemented in Matlab environment to check if the encryption has been carried out in the correct way. In order to make everything clear, the 128-bit plain text can be seen as a sequence of 16 ASCII characters, each represented on 8 bits, which compose a short sentence or a word, translated in their corresponding hexadecimal format as shown in Figure 5.2. The 128-bit word will be always considered as a sequence of 16 hexadecimal characters. ASCII elements will

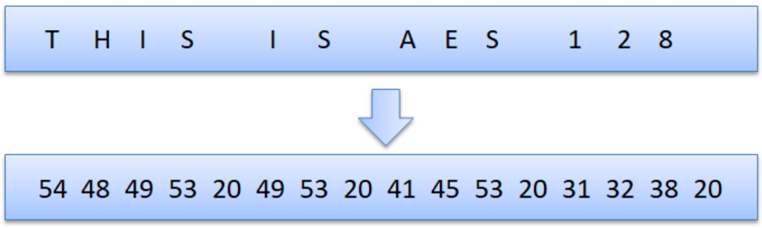


Figure 5.2. The 128-bit word to encrypt as the composition of 16 ASCII or hexadecimal characters

be handled in the hexadecimal format during the whole algorithm analysis for the features of this representation format which is convenient for the algorithm development.
Another important concept to be underlined is that the AES128 is made of simple operations unique to this algorithm, for this reason, special purpose row interfaces have to be instantiated in the architecture in order to implement it.

5.2.1 AddRoundKey operation

After having separately generated the eleven sub-keys starting from the 128-bit long key chosen, the AddroundKey operation simply consists of performing a sum between the subkey of the corresponding round and the cipher text at that specific algorithmic step. Since the AES128 operation are thought in the GF(2^8), which is a finite element field, the result of a sum corresponds to the modulo 2 operation of its result. The sum in this field behaves in the following way:

- $0 + 0 = 0 mod 2 = 0$
- $1 + 0 = 1 mod 2 = 1$
- $0 + 1 = 1 mod 2 = 1$
- $1 + 1 = 2 mod 2 = 0$

As is noticeable considering all the possible input patterns, working on the single bit, the sum operation is completely equal to a bitwise XOR operation

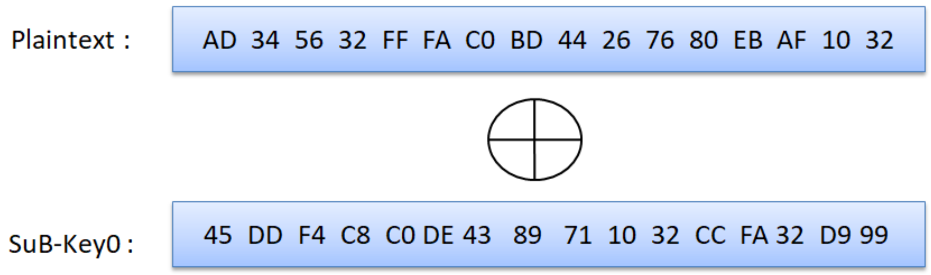

Plaintext :
AD 34 56 32 FF FA C0 BD 44 26 76 80 EB AF 10 32
SuB-Key0 :
45 DD F4 C8 C0 DE 43 89 71 10 32 CC FA 32 D9 99

Figure 5.3. AddRoundKey step as a simple xor operation in GF(2^8)

and it can be performed directly between the ciphertext and the corresponding sub-key as clarified in Figure 5.3. If round zero is considered, the plain text, which is the original word to encrypt, has to be summed with the sub-key zero and this is achieved by a simple bitwise XOR operation between each bit of the plaintext and the corresponding bit of the sub-key. If a round different from round zero is considered, the text that has to be put in XOR with the corresponding sub-key is obviously the output of previous operations.

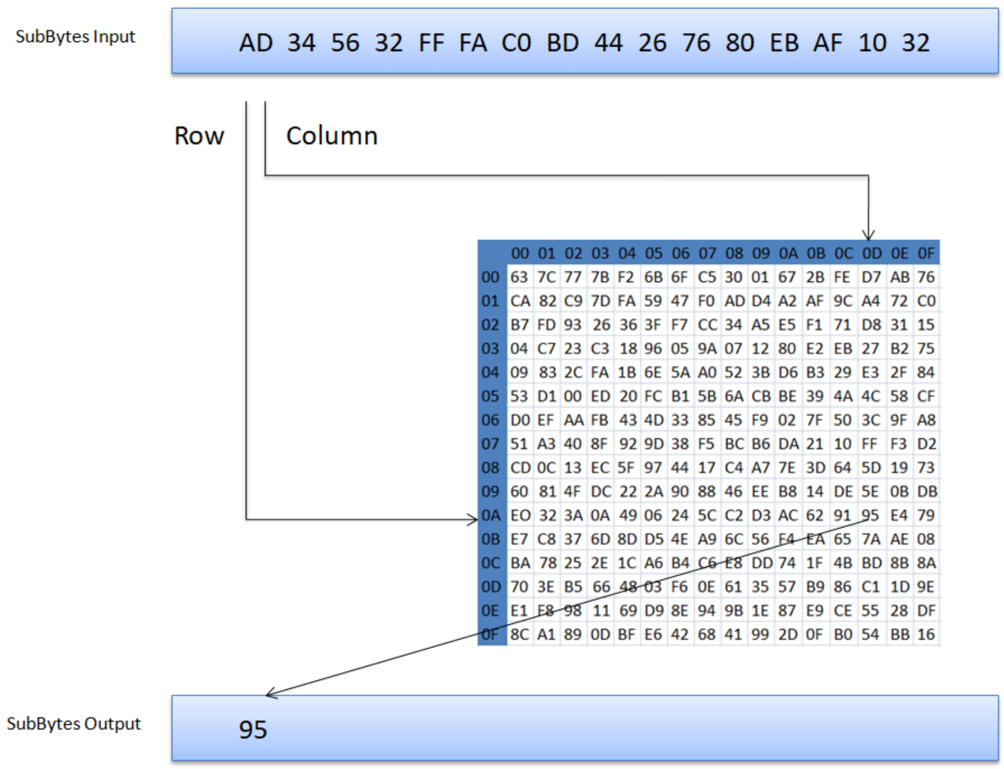
5.2.2 Row Interface required for the Addroundkey operation

As discussed in subsection 3.1.4, the architecture is provided by default the RCAandLogic row interface which is capable of performing sums, subtractions, and several bitwise logic functions, including the XOR one. For this reason, no extra row interface is required to achieve the Addroundkey operation and it is also performed within a single clock cycle with the following μinstruction:

RCAandLogic & OthR & StoreBuff & "11111" & XORop

For convenience the eleven sub-keys have been stored in eleven different locations of the shared section, therefore considering round zero, the XOR operation is performed between the plain text to encrypt, stored in the first smart row, and the sub-key zero stored at address 31.

5.2.3 SubBytes operation



	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
00	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
01	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
02	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
03	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
04	09	83	2C	FA	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
05	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
06	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
07	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
08	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
09	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
0A	EO	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
0B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
0C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
0D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
0E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
0F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Figure 5.4. SubBytes step consisting of a LUT access with row and column address

The SubBytes operation consists of substituting each hexadecimal character, composing the 128-bit long word, with another hexadecimal value retrieved by a special Look-Up-Table proper of the AES algorithm. The Look-Up-Table under exam and the way each data is substituted is illustrated in Figure 5.4. As previously discussed, each hexadecimal character is composed of eight bits, the first four bits select the row address of the LUT while the last four bits select the column address, in this way the LUT is accessed in a univocal way for all the possible 256 input patterns. In Figure 5.4, the substitution of just one element is depicted but this operation has to be performed sixteen times for all the elements that compose the 128-bit long input word of the block. This is an example of a Substitution Box previously discussed in section 5.1. In particular, considering the scheduling of the algorithm, the input of the SubBytes operation of round n is always the output of the AddroundKey operation of round n-1.

5.2.4 Row Interface required for the SubBytes operation

The AES is an algorithm that requires computations unique to this algorithm, therefore special purpose blocks are mandatory to be instantiated. In the SubBytes case, a block capable of performing simultaneously all the sixteen substitutions is needed; this means that for each 128-bit word or sentence to encrypt, sixteen Look Up Tables are required. It may be possible to schedule the whole SubBytes process in sixteen consecutive steps by sharing a single LUT, but this has not been achieved because of synthesis issues, and because of PLiM intrinsic features. The management of single vector elements turn out to be difficult considering that every row interface must have the same parallelism of the memory; the hexadecimal characters that compose the complete word can be considered elements of a vector.

5.2.5 ShiftRows operation

The ShiftRows operation is the simplest to implement among the ones required by the AES algorithm, it just consists of switching the position of some of the elements composing the 128-bit word according to some fixed rules. The process is shown in Figure 5.5. The input to the ShiftRows row interface fills by column a 4x4 matrix, then the following rules are applied to the resulting matrix:

1. The first is not shifted
2. The Second row is circularly shifted left by one position
3. The Third row is circularly shifted left by two positions
4. The Fourth row is circularly shifted left by three positions

After these operations are performed, the output is reconstructed in its format following the same rules with which it has been organized in a matrix.

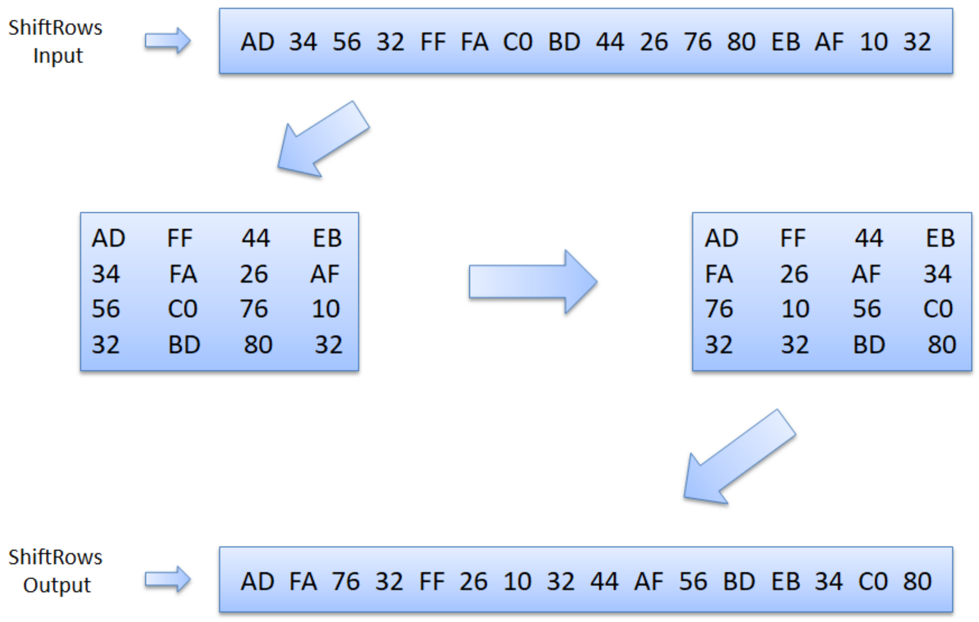


Figure 5.5. Elements switching demanded by the ShiftRows operation

5.2.6 Row Interface required for the ShiftRows operation

The row interface required to accomplish this operation is very light in terms of extra hardware and it implements a very simple function. As it is noticeable from Figure 5.5, each hexadecimal character is always substituted with an element in a specific position, the substitution criterion is fixed, therefore the implementation of the functionality of this block just consists in swapping wires in the proper way.

5.2.7 MixColumns operation

MixColumns operation is the primary source of diffusion within the AES algorithm and it is also the trickiest one to handle in PLiM environment. It consists of performing a multiplication between a fixed S_Matrix, proper of the algorithm, and the 4x4 matrix made up of the sixteen hexadecimal elements coming in input to this block organized by column. The order of this matrix multiplication and the S_Matrix layout are shown in Figure 5.6. The

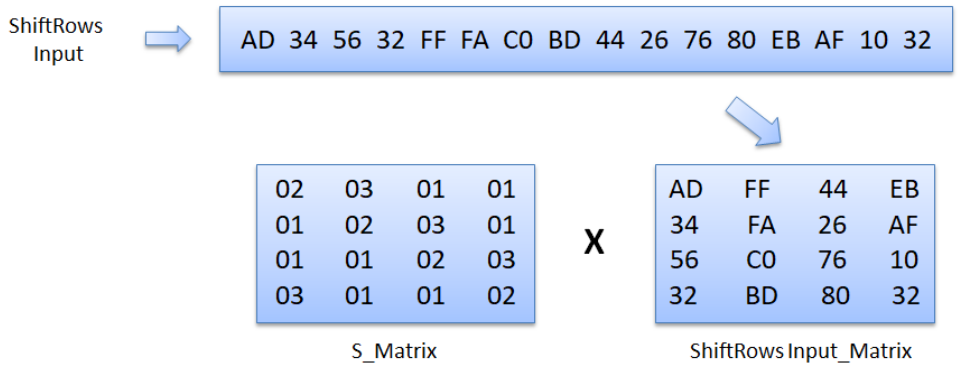


Figure 5.6. MixColumns operation performed between the corresponding ciphertext matrix and a fixed S_Matrix

MixColums is an AxB operation where the A matrix is always the S_Matrix of Figure 5.6, its elements are fixed and do not change along with the algorithmic rounds. The S_Matrix demands to perform multiplication by one, two, or three, this represents a huge advantage for how multiplications are meant in the GF(2^8) finite element field. In particular, considering a general hexadecimal element "FF":

- FF * 01 = FF
- FF * 02 = LeftShiftByOne(FF)
- FF * 03 = (FF * 02) XOR FF = LeftShiftByOne(FF) XOR FF

It is also necessary to take into account when multiplying by two or by three that overflow may occur. When an overflow case is detected the final result of these multiplications has to be corrected with an extra XOR operation with the decimal number "27" or the corresponding hexadecimal value "1B".

5.2.8 Row Interface required for the MixColumns operation

As discussed in subsection 5.2.7, the mixcolumns matrix multiplication can be translated into left shift and xor operations due to the features of the finite element field in which the AES algorithm works. Despite this, PLiM architecture does not allow to handle the whole mixcolumns computation by splitting it into different algorithmic steps, relying on the already existing RCAandLogic row interface for XOR operation and on a Left Shift row interface, which may be easily implemented, to manage the required shift operations. This happens for two main reasons, the first one is linked to the conditional overflow detection that can only be managed by a special purpose block, the second one is linked to the fact that the AES cannot be parallelizable at all in PLiM architecture since each hexadecimal character is subjected to different operations. The final choice is to build a special purpose mixcolumns block that is able to compute, within one single μinstruction, four elements of the resulting 4x4 matrix. In order to compute all the sixteen elements, the mixcolumns process is subdivided into four steps all of them involving the mixcolumns row interface. This special purpose block does not integrate a multiplier within it but just XOR functions and shift operations.

5.3 AES128 scheduling and implementation choices

As discussed in section 4.3, after having instantiated the required row interfaces, the structure must be adapted to the algorithm demands. The PLiM layout that fits these exigences is portrayed in Figure 5.7. Since the AES cannot be parallelized according to PLiM features, the choice is to perform the entire encryption of a word in a single smart row. If one word has to be encrypted, one smart row is required, if three words have to be encrypted, three smart rows are needed to cipher all the 128-bit input words at the same time. Hence the n-smart rows have to be filled with the n-words to encrypt while all the UpRows and DownRows are left uninitialized and their functionality is not exploited in the implementation of this algorithm. The standard section has to be filled with the eleven sub-keys required for the AddRoundKey operation and with the other four data necessary to schedule into four steps the MixColumns operation. The parallelism of the memory and of each row interface must be 128.

In order to make everything clear, in Figure 5.8 the scheduling of round zero and round one is provided where the operation involved are all referred to the same word, to the same smart row. A brief description of these five steps is reported:

- **AddRoundKeyRound0** Only the RCAandLogic block is active to perform the XOR operation between the sub-key0 retrieved from the standard section and the original word to encrypt stored in one smart row.

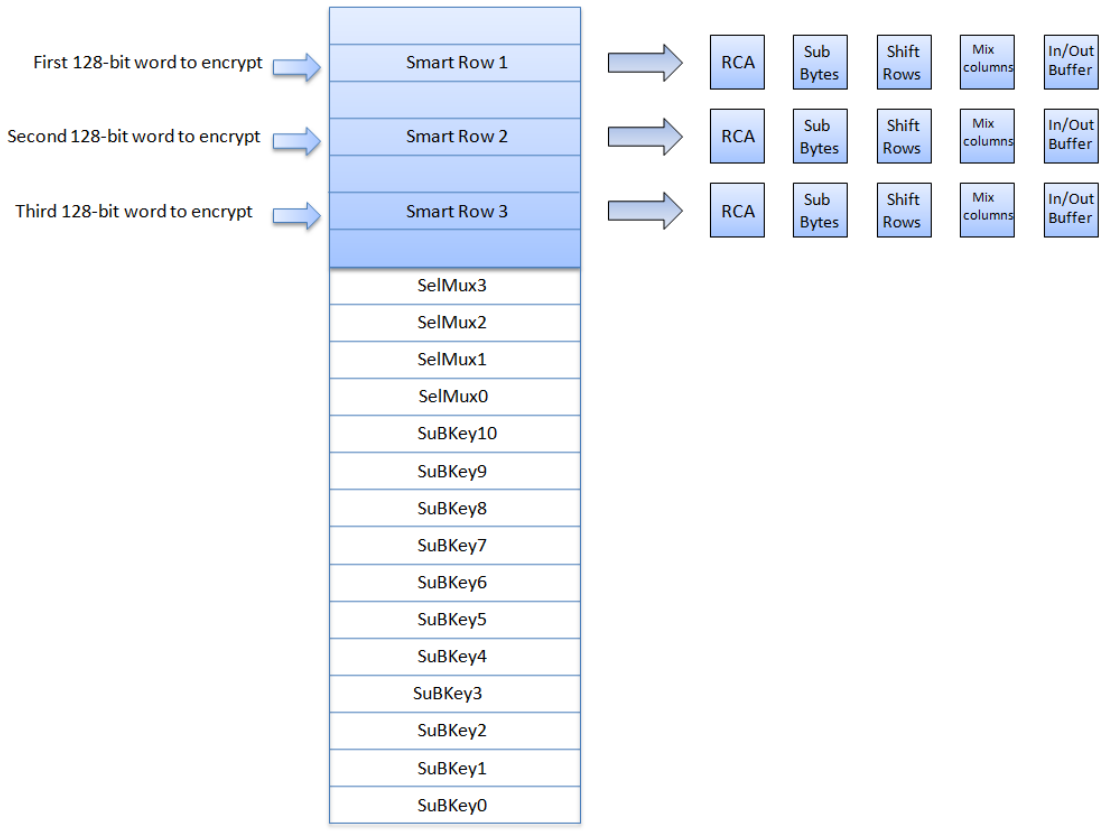


Figure 5.7. PLiM structure to fulfil the AES128 execution on it

As usual, the final result is stored in the output buffer.

- **SubBytesRound1** The result of the previous operation is retrieved from the output buffer and it is sent as input of the SubBytes row interface, the only one active within this step, and the substitution of all the sixteen hexadecimal characters is performed through the LUTs at the same time. The final result is stored in the output buffer.

- **ShiftRowsRound1** The result of the ShiftRows operation, stored in the output buffer is sent as input to the ShiftRows row interface, where the position of all the sixteen hexadecimal elements is switched at the same time. The resulting string is saved into the output buffer.

- **MixColumnsRound1** The operation is performed in four μinstructions time, the first input of the MixColumns row interface is taken from the

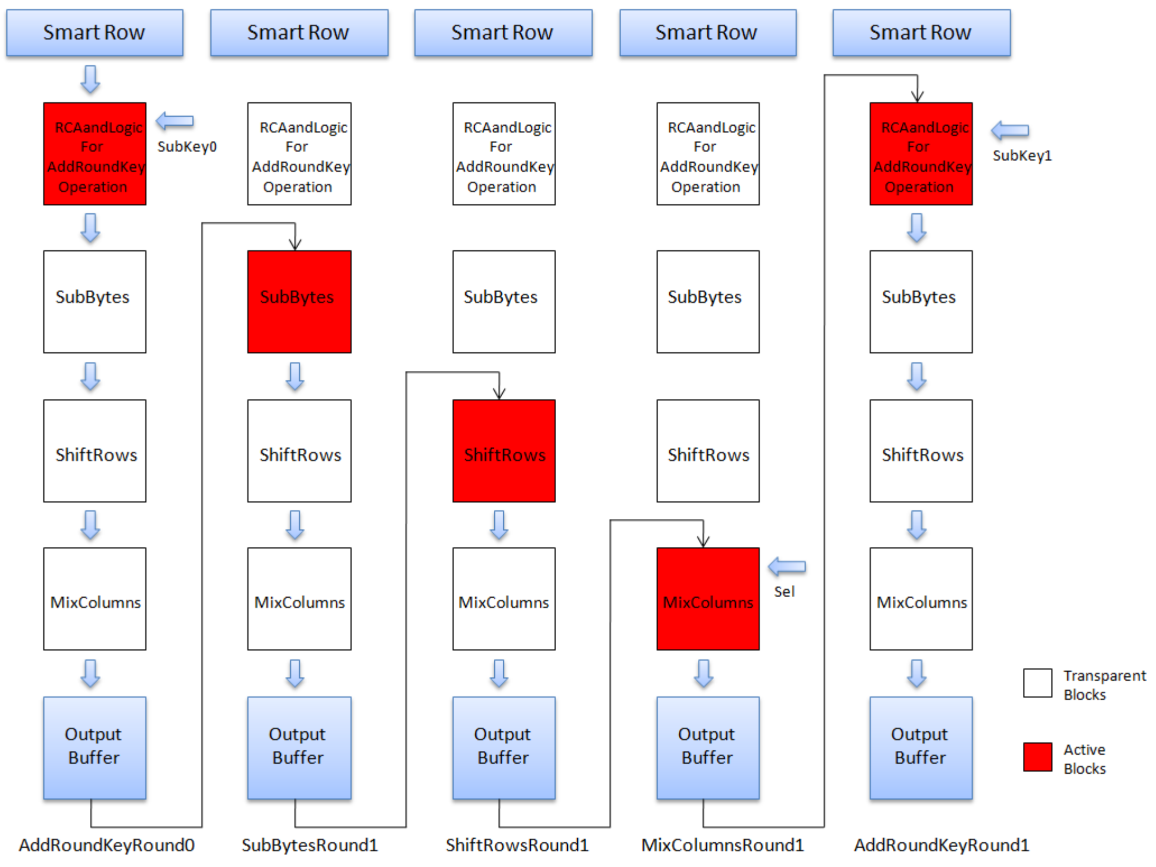


Figure 5.8. AES128 scheduling, in particular with the μinstructions involving the row interfaces to execute round zero and round one.

output buffer while the second one is a selection signal retrieved from the standard section whose function is to schedule the matrix elements computation.

- **AddRoundKeyRound1** The first input of the RCAandLogic block is the sub-key1, retrieved from the standard section, while, unlike round zero, the second input is no longer coming from the smart row but from the output buffer, since the result of MixColumns operation is needed. The final result is stored in the output buffer and a new round is ready to start

All the remaining rounds are managed as round one with the exception of the last one where the MixColumns operation is not required.

5.4 Approximate Message Mapping algorithm presentation

The Approximate Message Mapping (AMP) algorithm belongs to the compressed sensing world, a sort of evolution of the classic compression mechanism. Image compression is useful since it allows to save data reducing the storage space required by a certain file. Considering an image, it is composed of millions of pixels, if the information of each pixel should be saved, the image file would occupy so much memory space, therefore it has to be compressed in order to be stored. In particular, in Figure 5.9, the compres-

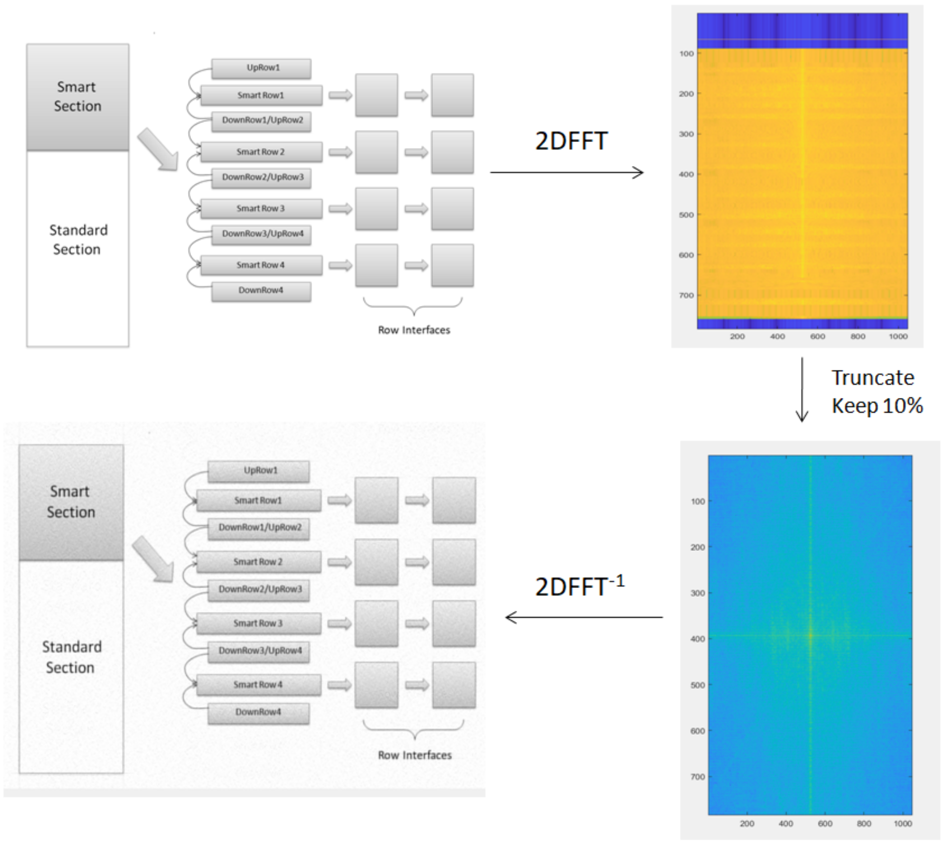


Figure 5.9. Example of standard image compression

sion and the reconstruction of an image using a common method is provided. The 2D Fast Fourier Transform (FFT) is applied to the original image in order to obtain millions of Fourier coefficients; most of the coefficients composing an image are very very small, therefore they can be truncated. Despite

this truncation, it is possible to reconstruct the original image, it is just a little bit blurred.
The key idea of compressed sensing, instead, is to start with a massively downsampled version of the original image; a brief and simplified explanation of this is reported since it has been found very interesting, and more details about this are available in [29]. If one signal, data is expressed in an appropriate chosen basis, only a few parameters are necessary to characterize the active modes. Compressed sensing is based on the sparsity principle for which a generic compressible signal x can be expressed as a sparse vector s within a transformation basis Ψ, that is:
$x = \Psi s$
A vector is considered sparse when most of its elements are zero.
The difference with respect to the classic compression method is that, instead of directly measuring the signal x, of collecting all the Fourier coefficients, it is possible to reconstruct a signal by taking fewer random measurements and then solve for the sparse vector s. In other words, the compressed sensing consists of collecting some few measurements y of the original signal x where $y = Cx$ and as a consequence
$y = C\Psi s$ where C is called measurement matrix
and then from these measurements infer how the sparse vector should look to be consistent with these measurements; in the end from the inferred sparse vector s, the full signal x can be reconstructed. The reconstruction is usually performed through convex algorithms and in this subject the AMP is introduced. These convex algorithms may be very expensive in large-scale data applications and for this reason, fast iterative thresholding algorithms have been analyzed as alternatives to convex procedures. The AMP algorithm belongs to this category and it may lead to important improvements since it is fast and the operations involved are not so complex. A simplified version of the AMP algorithm is described by the following formulas:

$$AMP = \begin{cases} x^{t+1} = \eta(A^* z^t + x^t) \\ z^t = y - Ax^t \end{cases} \tag{5.1}$$

Where:

- y is the Nx1 matrix containing the acquired measurements of the original signal X according to $y = Ax$
- x^t is the Mx1 estimation matrix of signal X at iteration t
- A is the measurement matrix of dimensions NxM, where N<M

- A* is the transposed A matrix of dimensions MxN
- η is a scalar threshold function

In [30], besides the algorithm explanation, a possible implementation of the AMP using memristive crossbar arrays to perform matrix-vector multiplications is presented.

5.5 Operations and row interface required by the AMP algorithm

As shown in section 5.4, the operation involved in the AMP algorithm are sums, subtractions, and multiplications, in particular, all of these computations are performed between matrices. Even if such operations are difficult to handle within PLiM environment, because of data dependencies, it has been achieved to implement the whole algorithm by using the already instantiated RCAandLogic row interface and a multiplier, whose functionality has been easily described. The multiplier row interface simply performs the multiplication between the two incoming inputs.
In order to iterate the algorithm multiple times, some data have to be saved, since more storage space than the one provided by the input/output buffer is required, two temporary registers have also been introduced in the row interface chain. The first register is used to store the A matrix elements required at each iteration, the second one is used to store the x^t elements

5.6 AMP algorithm implementation choices

In order to implement this algorithm on PLiM architecture, some assumptions have to be done; the two matrices multiplications Ax^t and A*z^t set the PLiM structure that satisfies the algorithm demands. A matrix multiplication is composed of two phases, row-column multiplication and sum of the partial results. Since all the multiplications can be parallelized, all the partial results have to be computed at the same time therefore the structure must have enough smart rows to achieve this kind of parallelization. In a real application of the AMP algorithm, all the matrices involved have conspicuous dimensions, in the provided PLiM application instead, some limitations have to be applied to the amount of data. In particular the following starting conditions have been chosen:

- y is a 2x1 matrix
- x^t is a 4x1 matrix
- A is a 2x4 measurement matrix
- A* is 4x2 matrix
- η is just a constant

The subtractions and the sums are simply performed relying on the RCAand-Logic functionality while the procedure to handle the matrix multiplications is explained in detail since it makes the most of PLiM functionalities exploiting UpRow, Smart row, DownRow exchange of information. For the chosen measurement matrix, eight smart rows are required to obtain all the partial results to be summed at the same time, the following four images clarifies the entire process. Considering the first iteration where x^0 is required, in Figure 5.10, the matrix multiplication to be performed is shown, in particular, both the first and the second row of the A matrix have to be multiplied by the only column of x, as a consequence eight partial results are produced and six sums, three for each element of the resulting matrix, have to be carried out. In Figure 5.11, the way the smart section is filled to achieve the desired operation is reported. The x matrix data are stored in the eight smart rows in a redundant way since they are multiplied twice by the two rows of the A matrix and thanks to the following μinstruction:

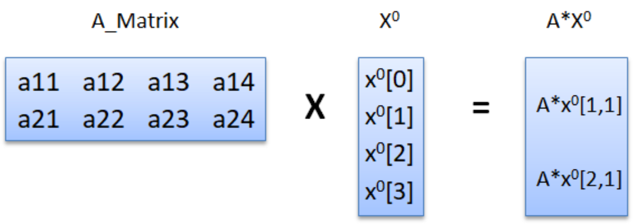


Figure 5.10. Example of a matrix multiplication demanded by the AMP algorithm

Multiplier & UpR & StoreBuff & Nulladd & nullFunc

all the smart rows contents, highlighted in red, are multiplied by their cor-

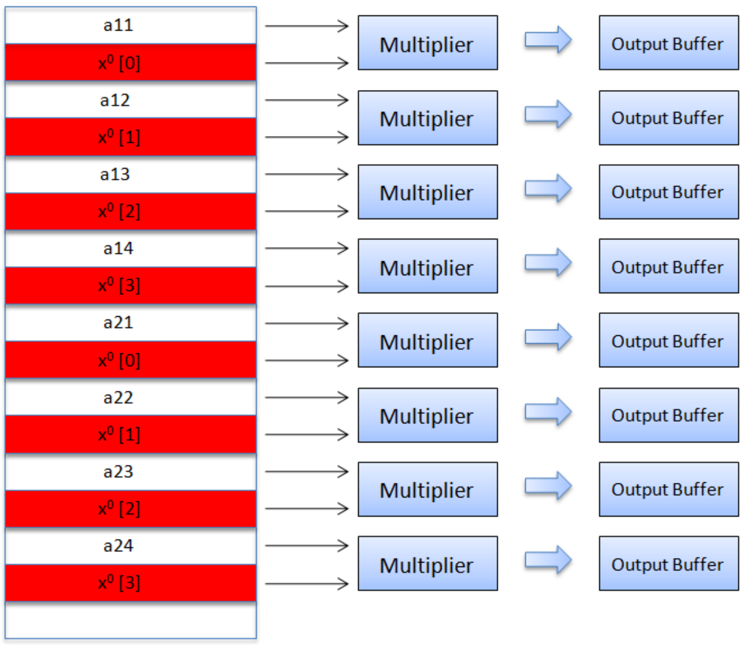


Figure 5.11. Parallel multiplication between the corresponding elements of the considered matrices

responding UpRow content and the eight partial results are stored in the output buffers. Then after having stored all the partial results in the output buffers, thanks to the store function subsection 4.2.2, these values are copied in each smart row and DownRow as shown in Figure 5.12. It is important to underline that after this operation, smart section data are lost since an overwrite occurs, for this reason, it is mandatory to store the A matrix coefficients and the x^t elements that will be used for further computations within the algorithm.
In Figure 5.12, the first sum between partial results is performed; the two inputs to the row interface are once again the content of each smart row and the content of their corresponding UpRow. The only useful partial sum is obtained in the second and sixth smart row groups.

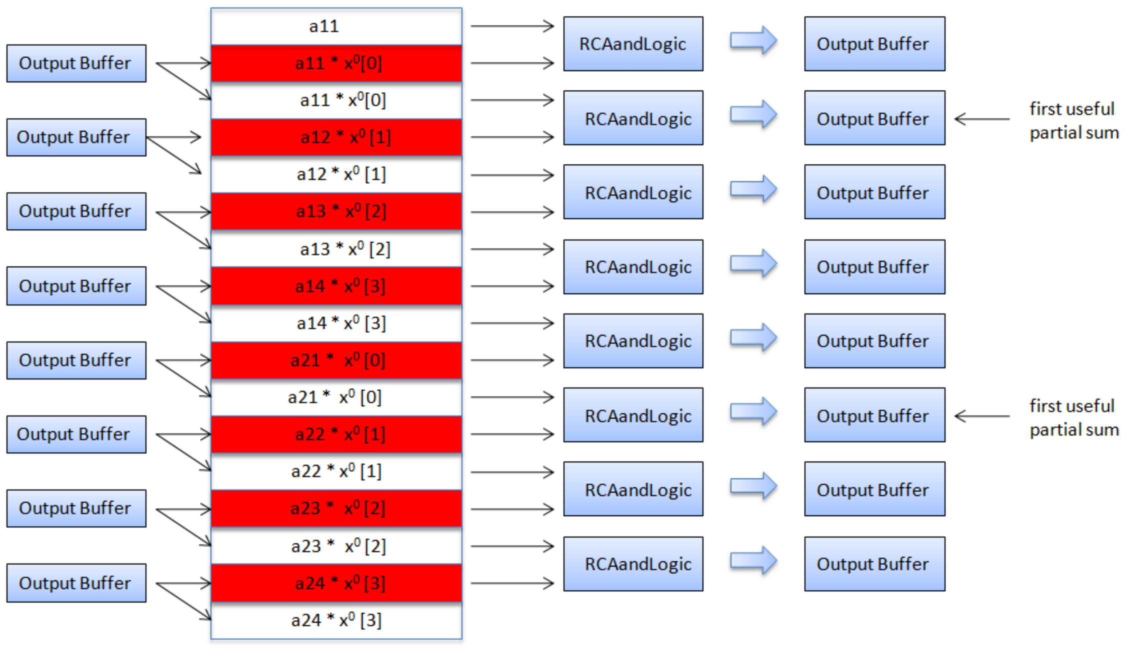


Figure 5.12. Execution of the first sum between the two first partial products

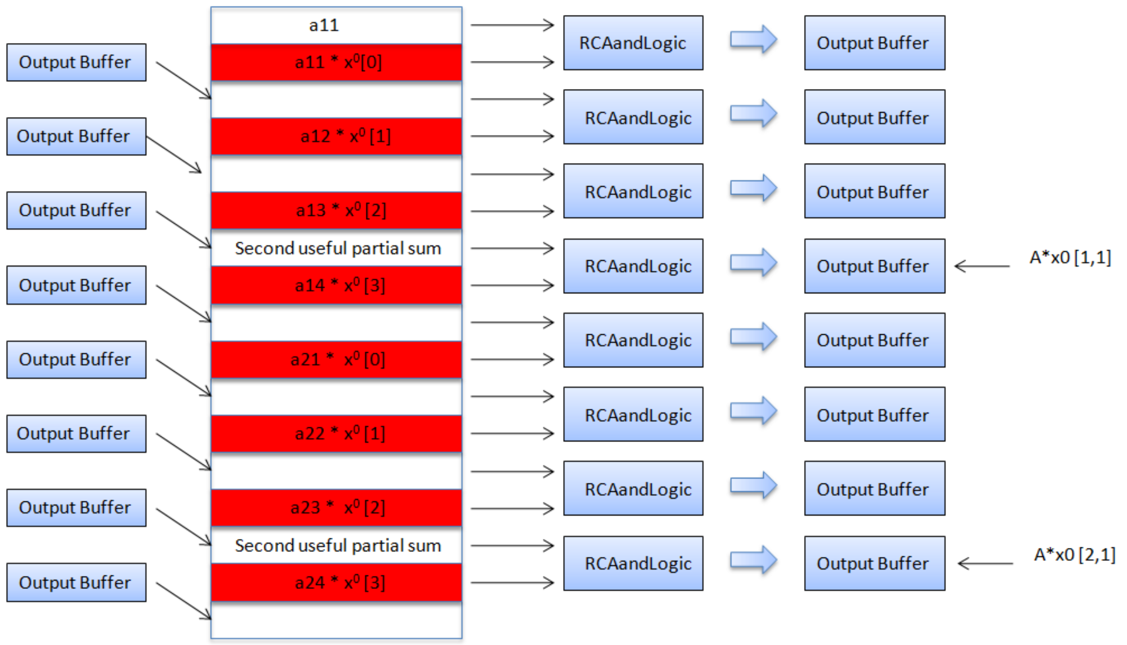


Figure 5.14. Execution of the third sum between the partial sum obtained in the previous step and the fourth partial product

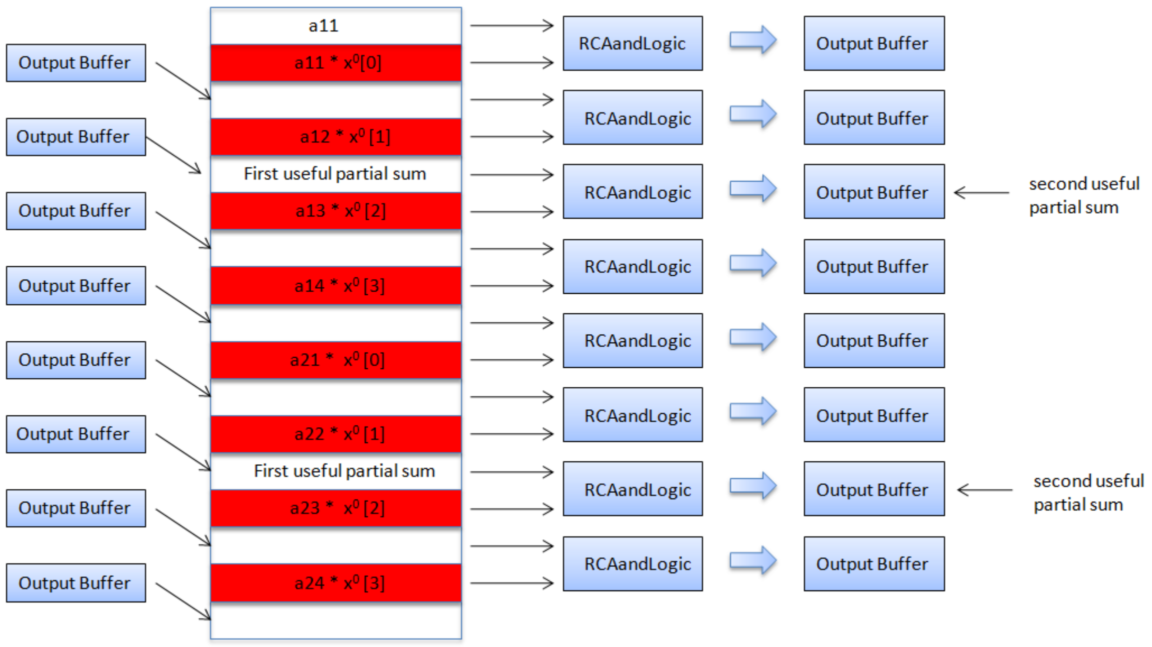


Figure 5.13. Execution of the second sum between the partial sum obtained in the previous step and the third partial product

The second and the third sums of the partial results are handled as reported in Figure 5.13 and Figure 5.14; now the store function copies the result of the previous μinstruction only in each DownRow, in this way with two equal procedures the other two partial results are added to the sum of the first two partial results. In the end, the elements of the resulting matrix are located in the output buffer of smart row number four and smart row number eight. As it is noticeable from this scheduling, when making sums, just one of them is really required, the others are completely useless but in any case, they are performed because of the Single-Instruction-Multiple-Data (SIMD) feature of the architecture. On the one hand, in the case of the initial multiplication, the SIMD feature is fully exploited and it turns out to be very useful, from the other hand in some cases it might even be destructive since it leads to unwanted computations or worse to the overwrite of useful stored data. For this reason, the trickiest aspect in the implementation of the AMP algorithm is data movement and management in order to handle all the data dependencies and to make the architecture suitable for iterating the algorithm as many times as required. In order to achieve this, most of the μinstructions are store and load functions. The last parameter to consider is the parallelism of the memory and as a consequence of all the row interfaces, a 32-bit parallelism has been chosen since it is compliant with a real application.

5.7 Results

In this section, the different PLiM structures built to implement the described algorithms are evaluated in terms of performance and consumption. For each algorithm the followed procedure steps are:

1. Modelsim simulation to check if with given inputs the structure provides the correct outputs.
2. Synopsys Design Compiler synthesis to extract timing parameters and a first consumption evaluation. Then the Back Annotation of simulation signals is performed to perform a more accurate power analysis that takes into account a more realistic activity of nodes through the whole algorithm execution time.
3. Thanks to the netlist obtained with the Back Annotation, the structure is synthesized with Innovus where the Place and Route procedure is applied, this step consists of placing all the circuitry, logic in a limited amount of space, and of establishing all the interconnections. With the new netlist, obtained after Place and Route a more accurate power estimation is performed.

5.7.1 AES results

Two different AES128 implementations are analyzed in this section; the first one encrypts just one 128-bit word, the second one encrypts four 128-bit words. From an architectural point of view, the first implementation requires one single smart row while the second one needs four smart rows; this last approach is clearly heavier in terms of extra hardware since all the row interfaces presented in section 5.2 have to be instantiated for every smart row composing the PLiM structure. As a first analysis, some information about performance is provided in table Table 5.1

As it is noticeable, in terms of timing, the extra hardware, implemented directly inside the memory array to encrypt four words at the same time, forces

Algorithm	nInstructions	Max Frequency	Execution Time
AES128_1_WORD	70	56 MHz	1245 ns
AES128_4_WORDs	70	49 MHz	1430 ns

Table 5.1. AES128 algorithm encrypting one and four words timing information

the structure to work at a slightly lower maximum operating frequency compared to the case with a single smart row. However, the μcode written in the μROM is exactly the same for both the implementations that require the same number of μinstructions to complete the algorithm run.
In terms of consumption, the results after Innovus Place and Route are reported in Table 5.2. These values are taken considering for each implementation its corresponding maximum operating frequency of Table 5.1. The power consumption value encloses both the total dynamic power and leakage power and as expected a growing number of smart rows leads to a corresponding increase of the dissipated power.

Algorithm	Power Consumption after P&R	nSmart rows
AES128_1_WORD	6.9 mW	1
AES128_4_WORDs	16.64 mW	4

Table 5.2. AES128 algorithm encrypting one and four words power dissipation information

5.7.2 AMP results

Also for the AMP analysis, two different implementations are considered; in particular, the first one provides just one iteration of the algorithm, that is until x^1 is obtained, the second implementation is made of three iterations, that is until x^3 is processed. From an architectural point of view, the two different implementations rely on the same structure, the same number of smart rows, same kind of row interfaces, the only different aspect lies in the μROM filling. However the μROM size is almost equal in the two cases since each full iteration of the AMP algorithm is handled as a subroutine, so really little extra μcode has to be written. In table Table 5.3 timing information about the synthesized structure that fulfills the AMP implementation is reported As expected, since the involved structure is exactly the same for the

Algorithm	nInstructions	Max Frequency	Execution Time
AMP_1_ITERATION	59	158 MHz	373 ns
AMP_3_ITERATIONS	174	158 MHz	1100 ns

Table 5.3. AMP algorithm iterating one and three times timing information

two algorithms, the maximum achievable frequency is also the same, the only difference is obviously the number of instructions required and as a consequence the execution time.
Considering the power consumption after the place and route procedure the results are shown in Table 5.4. Also in this case the estimates are carried out

Algorithm	Power Consumption after P&R	nSmart rows
AMP_1_ITERATION	35.66 mW	8
AMP_3_ITERATIONS	55.44 mW	8

Table 5.4. AMP algorithm, iterating one and three times, power dissipation information

considering the maximum operating frequency Table 5.3 achievable by the structure. It is evident that the growing number of iterations leads to more pins commutation and therefore to an increase of the total dynamic dissipated power. Even if it is not so meaningful to make comparisons between different algorithms implemented on PLiM architecture since the parameters that most affect performance are completely different, some considerations can be done. At first, the maximum achievable frequency, so the clock period, is strongly affected by the row interfaces chain, in fact, the critical path for the AMP implementation, where just registers, a multiplier, and an adder are instantiated, is minor with respect to the AES128 case, where heavy, high parallelism blocks have to be integrated.
Considering instead power values of Table 5.2 and Table 5.4, the AMP consumption is major compared to the AES128 one, since a higher clock frequency has been taken into account. However, the smart section impact is appreciable, in fact, the minor complexity of AMP row interfaces instantiated, anyway brings to a power consumption comparable to AES128 approach even if the AES128 works with higher parallelism and with more

complex blocks. This is linked to the AMP demand for a deeper smart section that brings with it a major number of interconnections to be performed, more row interfaces to be integrated, more pins switching at the same time because of the SIMD feature of PLiM.

5.7.3 Results Comparison with the RISC-V architecture

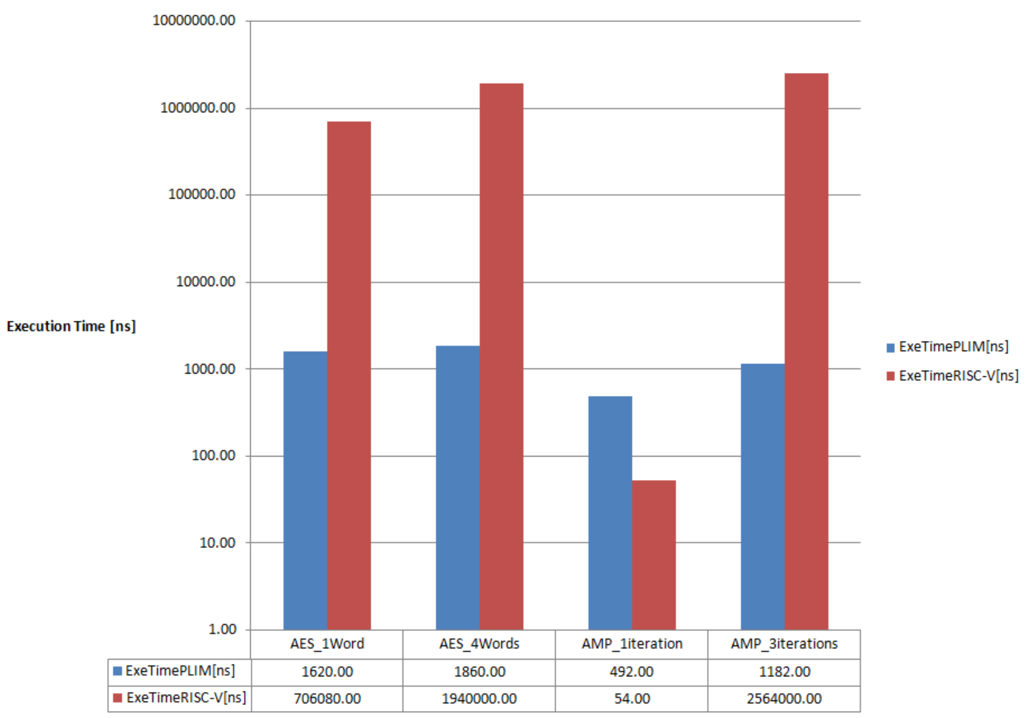
The main purpose of this work of thesis is to understand if PLiM architecture may bring some advantages compared to a standard memory-processor working system. In particular, the attention will be focused on the comparison of the same algorithm running on PLiM and on the RISC-V architecture. RISC-V performance values have been extracted and a comparison, in terms of execution time, is shown in Table 5.5 As it is noticeable, in almost ev-

Algorithm	Exe Time PLiM	Exe Time RISC-V
AES128_1_WORD	1.245 μs	706.08 μs
AES128_4_WORDs	1.430 μs	1940 μs
AMP_1_ITERATION	0.373 μs	0.054 μs
AMP_3_ITERATIONS	1.100 μs	2564 μs

Table 5.5. Comparison between execution time values regarding algorithm implementations on PLiM and on the RISC-V processor

ery case, the algorithm runs much faster on the PLiM with respect to the RISC-V architecture. Considering the AMP implementation iterating just one time, the separate memory processor approach still wins, but if the algorithm is iterated three or more times all the PLiM advantages come out and the profit in terms of execution time becomes evident. Graphical feedback of these results is provided in Figure 5.15.
Then the consumption parameters are taken into account in order to check if the execution benefits achieved are balanced or lead to disadvantages in terms of consumption. Rather than making a comparison in terms of dissipated power, it is more meaningful if it is made in terms of energy; power is the amount of energy transferred per unit time, that is:

$$Power[W] = \frac{Energy[J]}{Time[s]}$$



	AES_1Word	AES_4Words	AMP_1iteration	AMP_3iterations
🟦 ExeTimePLIM[ns]	1620.00	1860.00	492.00	1182.00
🟥 ExeTimeRISC-V[ns]	706080.00	1940000.00	54.00	2564000.00

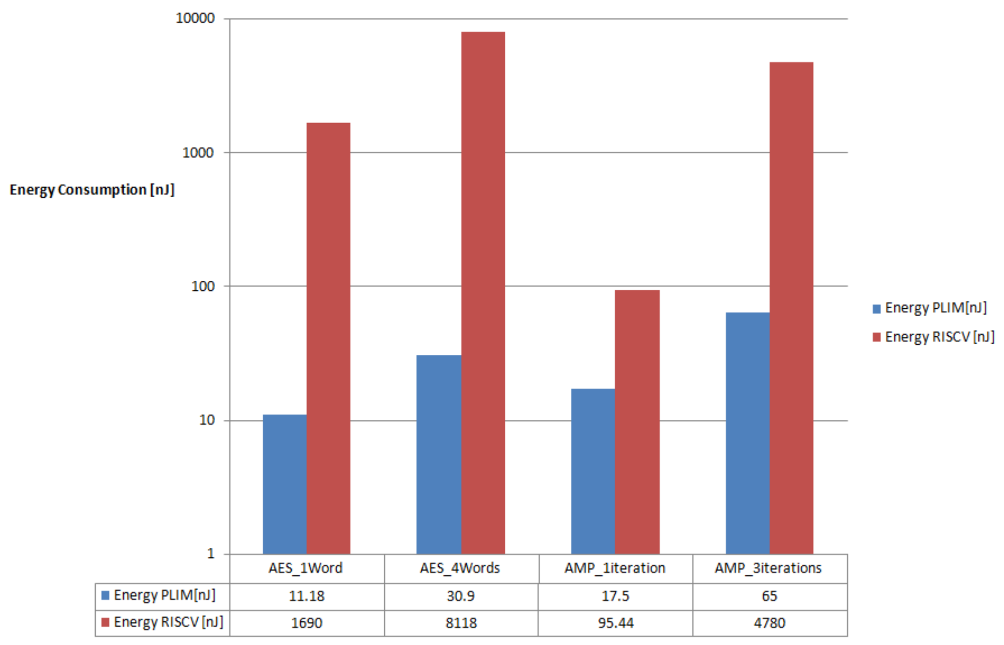
Figure 5.15. Graphical comparison between the execution time of the involved algorithm running on PLiM and on the RISC-V architecture

By multiplying the obtained execution time and power consumption values for PLiM, the architecture energy amount required for the execution of all the involved applications is calculated and displayed in Table 5.6 where the RISC-V energy estimates are reported too.

Algorithm	Energy PLiM	Energy RISC-V
AES128_1_WORD	8.59 nJ	1690 nJ
AES128_4_WORDs	23.79 nJ	8118 nJ
AMP_1_ITERATION	13.3 nJ	95.44 nJ
AMP_3_ITERATIONS	60.9 nJ	4780 nJ

Table 5.6. Comparison between the energy required to implement the considered algorithms on PLiM and on the RISC-V architecture

Also considering the architecture consumption, great benefits are accomplished, in most cases, the energy consumption of the PLiM is several orders of magnitude lower than the RISC-V processor. This can be appreciated in



	AES_1Word	AES_4Words	AMP_1iteration	AMP_3iterations
🟦 Energy PLIM[nJ]	11.18	30.9	17.5	65
🟥 Energy RISCV [nJ]	1690	8118	95.44	4780

Figure 5.16. Graphical overview of the energy consumption of the considered algorithms on PLiM architecture and on the RISC-V processor

the graphic in Figure 5.16

5.7.4 RISC-V Vs PLiM Vs AES accelerators

In order to check how a PLiM application may behave in the real world, an extra comparison is made between the PLiM structure encrypting four 128-bit words at the same time and some encryption accelerators found among the related works, specifically proposed for the AES algorithm. Since the AES128 algorithm developed in Electronic-Codebook (ECB) mode has been tested, the comparison is done considering this specific context. Some meaningful parameters have been extracted to make a comparison with the RISC-V processor, in this section some extra meaningful values are needed and in particular:

- **Throughput** It is the data amount that is processed or transmitted in the unit of time, it can be calculated considering the amount of the output data of the system and the interval of time with which they are provided. Considering the AES128 implementation on PLiM encrypting four words the throughput expressed in Megabit per second(Mbs) is 358 Mbs

- **Cycles per Byte** It is commonly used as a partial indicator of performance in the cryptographic world and it indicates how many clock cycles are needed by the processor to computed a byte within the involved algorithm. For the AES128 PLiM application that encrypts four words, the cycles per byte value is 1.07.
- **Energy Efficiency** It correlates the dissipated power with the achieved throughput, it gives information about the rate at which the processor can provide data in output per Watt. For the AES128 PLiM application encrypting four words, the energy efficiency is 21.5 Gbs/W

In table Table 5.7 the PLiM AES128 application is compared to some architectures found among the related works. Considering the AES128 implementation on PLiM encrypting four words, the energy efficiency is 21.5 Gbs/W.

Design	Tech	Frequency	Voltage	Throughput	Power
PLiM	45 nm	48 MHz	1.1 V	358 Mb/s	16.64 mW
Erbaagci [31]	65 nm	2.2 GHz	1.0 V	275 Gb/s	524 mW
Aesthetic arch [32]	0.25 μm	66 MHz	1.0 V	844 Mb/s	260 mW
Verbauwhede [33]	0.18 μm	125 MHz	1.8 V	1.6 Gb/s	56 mW
Shan [34]	28 nm	875 MHz	0.9 V	991 Mb/s	3.8 mW
Mathew [35]	28 nm	2.1 GHz	1.1 V	53 Gb/s	125 mW
Mangard [36]	0.6 μm	64 MHz	NA	241 Mb/s	NA
Chih-Pin Su [37]	0.35 μm	200 MHz	NA	2.28 Gb/s	NA
Intel Westmere [38]	32 nm	2.67 GHz	NA	16 Gb/s	NA

Table 5.7. Evaluation of some meaningful parameters for AES128 algorithm running on PLiM and on different AES accelerators

It is noticeable that the results for the examined application on PLiM are comparable in terms of throughput with the oldest works, the ones which exploit a wider CMOS technology. If modern applications are considered, in particular, specific chips with the goal of speeding up the AES execution, PLiM implementation is not able to reach their throughput level. In Figure 5.17 the difference in terms of throughput is appreciable, PLiM performance are two or three orders of magnitude minor compared to the most modern applications. If a comparison considering the throughput as a figure of merit between the RISC-V processor and PLiM architecture is performed, the obtained result for the RISC-V approach is three orders of magnitude

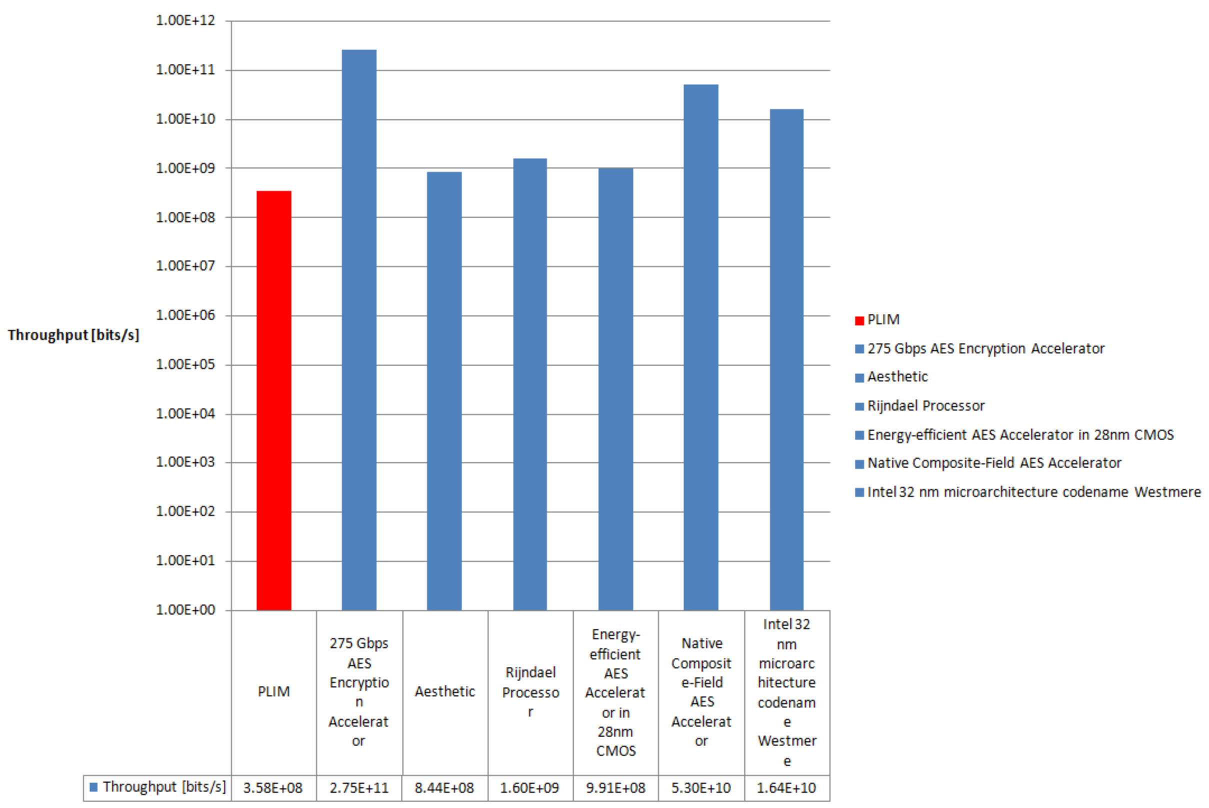


Figure 5.17. Graphical comparison between the PLiM achieved throughput and the one achieved by different AES accelerators for the AES128 application

lower with respect to the PLiM approach. This is visually shown in the graphic of Figure 5.18. Considering performance, on the one hand, PLiM architecture implementing the AES algorithm leads to great benefits with respect to a RISC-V processor, on the other hand, it cannot achieve the rate of most modern, deeply pipelined accelerators.
If besides the throughput, also the dissipated power of the structures is taken into account, a meaningful comparison can be made in terms of energy efficiency. The results, considering the information reported in Table 5.7, are visually shown in Figure 5.19. It is noticeable how the energy efficiency, which is the ratio between the throughput and the dissipated power, for the AES specific application, does not follow the throughput trend. The result obtained for the PLiM implementation is at most about one order of magnitude minor compared to modern AES accelerators. Considering instead the comparison between the PLiM energy efficiency and the RISC-V energy

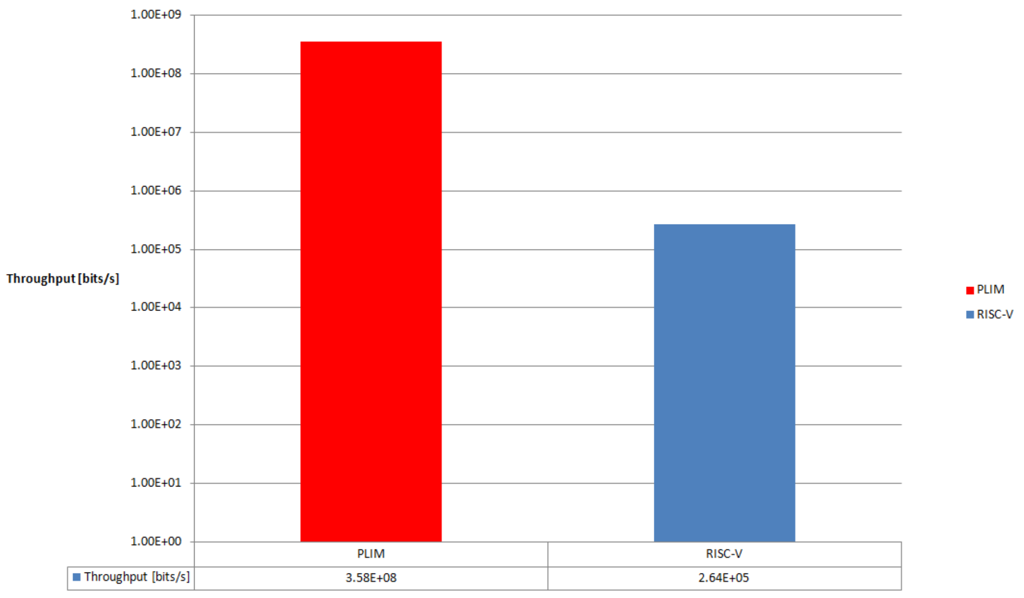
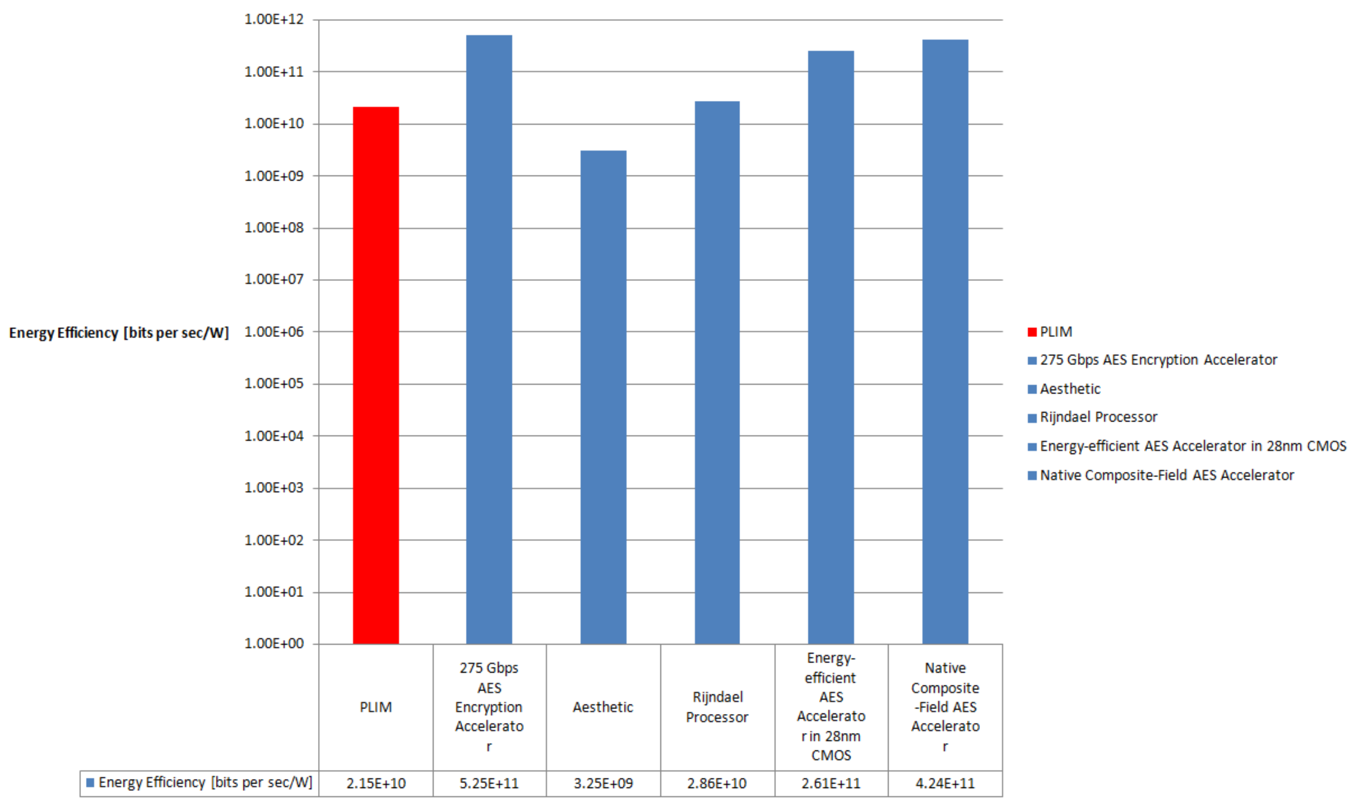


Figure 5.18. Graphical comparison between PLiM achieved throughput and the one achieved by the RISC-V considering the AES128 algorithm encrypting four 128-bit words

efficiency, the trend is the same found for the throughput, that is that also in this case PLiM performs three orders of magnitude better, as shown in Figure 5.20.

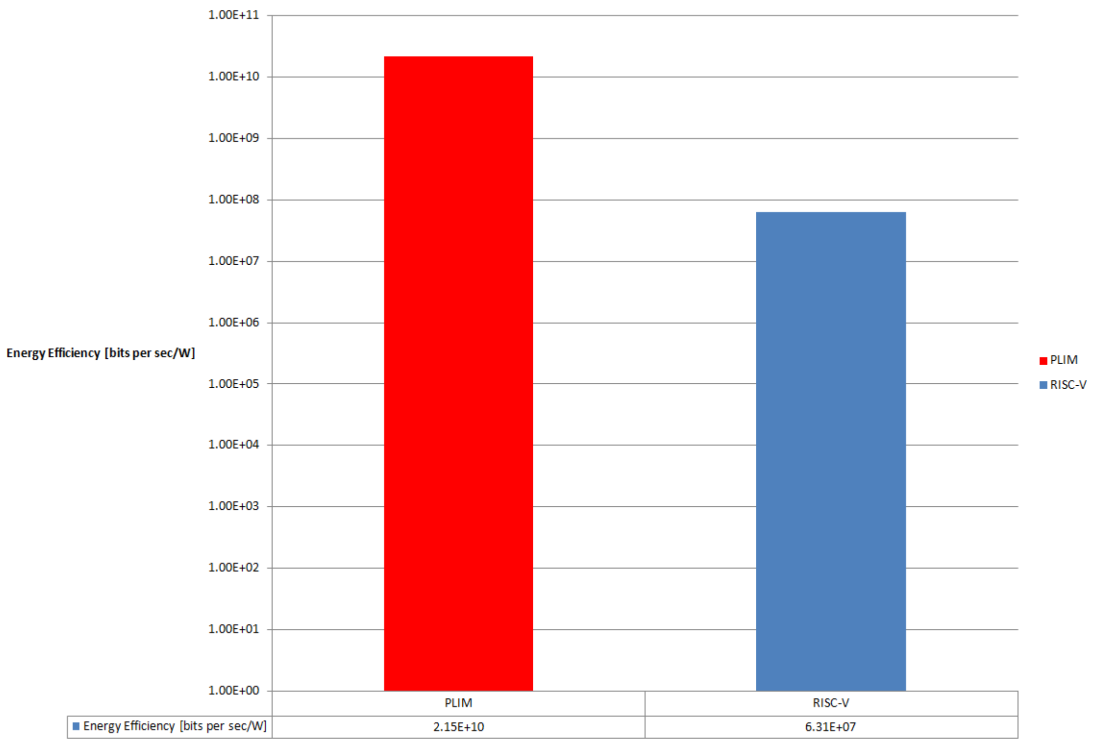
Analyzing the last figure of merit, the cycles per byte parameter, the result obtained for the PLiM implementation is compared to the results obtained for the considered AES accelerators. In particular, the smaller the value of this parameter, the more efficient the encryption algorithm is. As it is shown in Figure 5.21, even if PLiM is not a pipelined structure, it can accomplish a good outcome in general and in particular not so far even from the modern applications exploiting a similar technology.
These estimates are interesting because they give an idea of the potential of the PLiM structure. It aims at overcoming the Von Neumann bottleneck of the typical architecture with separate memory and processor and for the considered applications, it accomplishes to do that. Besides this, it may be seriously taken into account as a general purpose system with average performance and good efficiency. Considering the AES128 approach, it can be inferred that PLiM may adapt to different demands; in particular, if a higher throughput is required, the structure can be enriched with more smart



	PLIM	275 Gbps AES Encryption Accelerator	Aesthetic	Rijndael Processor	Energy-efficient AES Accelerator in 28nm CMOS	Native Composite-Field AES Accelerator
Energy Efficiency [bits per sec/W]	2.15E+10	5.25E+11	3.25E+09	2.86E+10	2.61E+11	4.24E+11

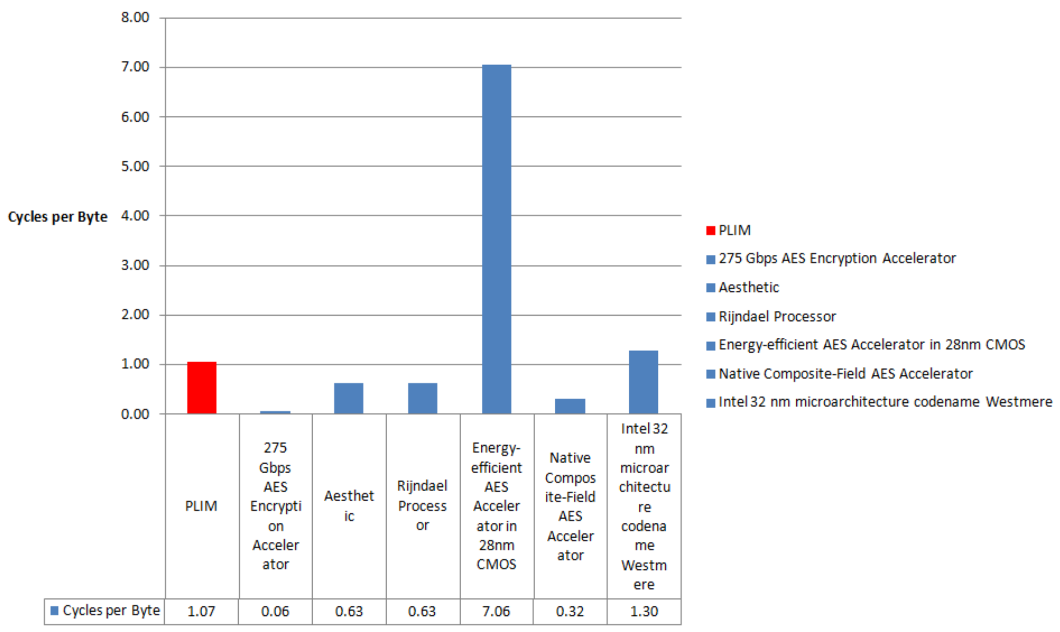
Figure 5.19. Graphical comparison between the PLiM achieved energy efficiency and the one achieved by different AES accelerators for the AES128 application

rows. The consequence of this is the reduction of the maximum operating frequency, a growing need for power but in terms of efficiency, these elements may be balanced by the higher throughput.



	PLIM	RISC-V
🟦 Energy Efficiency [bits per sec/W]	2.15E+10	6.31E+07

Figure 5.20. Graphical comparison between PLiM achieved energy efficiency and the one achieved by the RISC-V considering the AES128 algorithm encrypting four 128-bit words



	PLIM	275 Gbps AES Encryption Accelerator	Aesthetic	Rijndael Processor	Energy-efficient AES Accelerator in 28nm CMOS	Native Composite-Field AES Accelerator	Intel 32 nm microarchitecture codename Westmere
Cycles per Byte	1.07	0.06	0.63	0.63	7.06	0.32	1.30

Figure 5.21. Graphical comparison between PLiM achieved cycles per byte value and the one achieved by different AES accelerators, considering the AES128 application

Chapter 6

Conclusions and Future Works

Programmable-Logic-in-Memory (PLiM) structure represents a novelty to exploit the Logic-in-Memory principle, it allows the execution of an algorithm by processing all the required operations directly inside the memory space. While other possible solutions belonging to this environment focus their attention on configuring and interconnecting cells together in order to accomplish more complex functions, the real peculiarity of PLiM approach is that its operating functionalities are provided by hardware blocks, the row interfaces, instantiated in the smart section. In this way, such a structure ideally is suitable for each application since every function can be implemented through the row interfaces. Being a general-purpose solution is the primary goal of PLiM and it accomplishes to be that. PLiM structure can also rely on its feature of handling Single-Instruction-Multiple-Data (SIMD) computing mode, that is that every μinstruction targets all the smart section elements. This may be useful to parallelize some operations within the execution of one single algorithm, as it happens for the Approximate-Message-Passing (AMP) case, or to parallelize the execution of the same algorithm multiple times, as it happens for the Advanced-Encryption-Standard (AES) case. In particular, referring to the AMP implementation, and in general, to all the applications whose operations may be broken in parallel steps, PLiM architecture presents some limitations. The problem is that the smart section size directly depends on the input data amount, considering the following examples:

- if a multiplication between a 3x3 matrix and a 3x2 matrix has to be performed, 18 smart rows are needed to compute all the 18 partial products at the same time.

- if a multiplication between a 4x4 matrix and a 4x2 matrix has to be performed, 32 smart rows are needed to compute all the 32 partial products at the same time.

- if a multiplication between a 5x5 matrix and a 5x2 matrix has to be performed, 50 smart rows are needed to compute all the 50 partial products at the same time.

It is clear that:

- Similar operations require very different structures and μcode.

- Even operations involving small vectors or matrices require a very heavy smart section configuration.

Therefore in the case of parallelizing one or more operations within one single algorithm, it is evident that PLiM architecture is not suitable for each possible application since the smart rows number grows with the parallelization demand. If one PLiM architecture is enriched with n smart rows, n operations can be performed in parallel at maximum, algorithms that require more operations to be carried out in parallel cannot run. As a consequence, considering this aspect, the PLiM configuration can be extremely application dependent, considering the AMP synthesized structure, it allows to run the algorithm just for the input data chosen for that particular implementation. One possible solution is to perform all the operations that must be parallelized not all at the same time but in different algorithmic steps. The following example is considered:

$$Y = A[0]*X[0]+A[1]*X[1]+A[2]*X[2]+A[3]*X[3]+A[4]*X[4]+A[5]*X[5]$$

Imagining that the structure does not have six smart rows but just three, it is possible to break this operation into two main steps:

1. The three available smart rows are filled respectively with X[0], X[1], X[2] while the three UpRows are filled with A[0], A[1], A[2], the three multiplications are performed in parallel and the two required sums are calculated. The sum of the first three partial results will be stored in a temporary register.

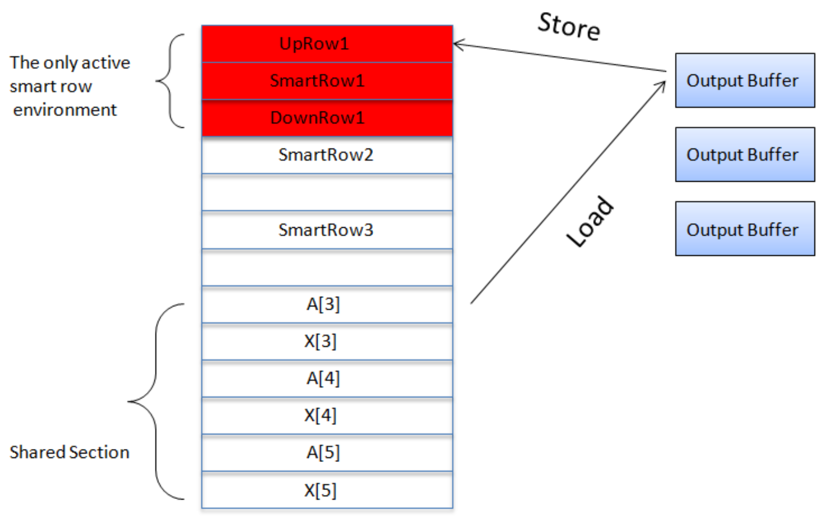


Figure 6.1. Example of collecting one data from the standard section to be saved in a single precise location of the smart section

2. By relying on move functions (Load and Store) and by deactivating blocks subsection 3.1.1, it is possible, as shown in Figure 6.1, to retrieve the unused A and X coefficients, previously stored in the shared section, and to store them, one data per clock cycle, in the appropriate way in the available UpRows and smart rows as done in step number 1, then the multiplications are performed and the sum A[3]*X[3]+A[4]*X[4]+A[5]*X[5] is computed. In the end, this sum is added to the other partial sum retrieved from the temporary register.

It must be clear that without the block deactivation this is not achievable because of the SIMD feature of PLiM. The SIMD feature always represents an advantage when an operation can be parallelized but it is harmful with data movements and management within the smart section. In order to remain inside the memory space, all the required operands that will be used in the latter algorithmic steps have to be saved in the shared section and from there retrieved when they are needed. The problem is that without the possibility of activating just one smart row environment within one clock cycle, it is not possible to store one precise data in one precise location leading to the impossibility of managing the demanded parallel computations with the desired data. In particular, the examined PLiM structure permits to subdivide the smart section into four blocks at maximum. As a consequence, if one PLiM structure has eight smart rows, there could be 4 groups with 2 smart rows each, 2 groups with 4 smart rows each, or just one group containing all the smart rows. Anyway in the best case, each operation, every data movement must be applied at least to two smart rows environments making the algorithm implementation impossible or not effective. When an operation has to be split into parallel computations within PLiM environment, data organization is fundamental, all the data that must be computed together with the same row interface must be stored accurately in the corresponding locations (UpRows, Smart rows, Down rows). Data organization in the middle of the algorithm execution time is not achievable without the maximum granularity in terms of Smart rows activation or deactivation. The tested architecture can handle in parallel not more than four smart row environments in an effective way, that is four parallel computations at each clock cycle; if a higher parallelization is required to speed up the algorithm or to handle more input data, the subdivision of the smart section in n blocks where n is the number of smart rows is mandatory to give the architecture the maximum flexibility it wants to achieve. This feature has an impact on the μinstruction length where an activation/deactivation bit has to be associated with each smart row. In order to clarify the concept, the current PLiM implementation is able to perform more than four operations within one single μinstruction, but not in an effective way since mandatory data movements are applied to more than the single smart row group and this in almost all the cases is destructive. The approach used for the AES implementation instead is completely different, here there is no communication among different smart row environments, the whole algorithm is computed in one single smart row and all the involved operations can be applied to all the other smart rows composing the smart section in order to encrypt as many words as many smart rows are available. Since parallelization cannot be achieved, the AES does not have particular structural demands besides the required row interfaces; simply the PLiM is able to encrypt in parallel n-words where n is the number of available smart rows. Anyway, considering both the two different tested approaches, it has been shown that PLiM performs better than the RISC-V processor in terms of performance and consumption, therefore the examined logic-in-memory approach may lead, for some applications, to some advantages with respect to a common approach based on separate processor and memory structures. Considering the modern architectures found in the related works, exploiting a technology similar to the one with which PLiM has been synthesized, it is clear that the tested structure cannot challenge them, this happens for different reasons. First of all, PLiM is not built with the specific intent of speeding up one particular application but it is thought to be a general purpose structure. Actually, the configuration chosen to implement the AES algorithm is application-specific but it has been implemented following the PLiM rules that are thought for a general purpose architecture. Then the considered accelerators are deeply pipelined, as a consequence, the rate at which they can provide data in output is greater than the one PLiM can achieve since it is not pipelined, therefore a new word encryption within a smart row can begin only once the previous one has finished. It must be also taken into account that the maximum operating frequency is set by the chain of row interfaces; despite just one row interface is involved in each μinstruction, the critical path is always the one that runs through the entire chain of RIs. This considerably limits the maximum operating frequency of the implemented structure, the problem is that the whole chain of RIs is not broken by registers and it sets the critical path of the system even if it is never taken. However, in general, it has been found that the Logic-in-Memory (LiM) principle exploited following PLiM rules can lead to notable advantages compared to a RISC-V architecture and that it may be considered as a valid general-purpose solution that can rely on good performance and efficiency statistics.

Bibliography

[1] E. Azarkhish, C. Pfister, D. Rossi, I. Loi and L. Benini, "Logic-Base Interconnect Design for Near Memory Computing in the Smart Memory Cube," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 1, pp. 210-223, Jan. 2017, doi: 10.1109/TVLSI.2016.2570283.

[2] J. -S. Kim et al., "A 1.2 V 12.8 GB/s 2 Gb Mobile Wide-I/O DRAM With 4 × 128 I/Os Using TSV Based Stacking," in IEEE Journal of Solid-State Circuits, vol. 47, no. 1, pp. 107-116, Jan. 2012, doi: 10.1109/JSSC.2011.2164731.

[3] D. U. Lee et al., "25.2 A 1.2V 8Gb 8-channel 128GB/s high-bandwidth memory (HBM) stacked DRAM with effective microbump I/O test methods using 29nm process and TSV," 2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014, pp. 432-433, doi: 10.1109/ISSCC.2014.6757501.

[4] Bong Hwa Jeong et al., "A 1.35V 4.3GB/s 1Gb LPDDR2 DRAM with controllable repeater and on-the-fly power-cut scheme for low-power and high-speed mobile application," 2009 IEEE International Solid-State Circuits Conference - Digest of Technical Papers, 2009, pp. 132-133, doi: 10.1109/ISSCC.2009.4977343.

[5] R. Nair, "Evolution of Memory Architecture," in Proceedings of the IEEE, vol. 103, no. 8, pp. 1331-1345, Aug. 2015, doi: 10.1109/JPROC.2015.2435018.

[6] G. Singh et al., "A Review of Near-Memory Computing Architectures: Opportunities and Challenges," 2018 21st Euromicro Conference on Digital System Design (DSD), 2018, pp. 608-617, doi: 10.1109/DSD.2018.00106.

[7] Santoro, G., Exploring new computing paradigms for data-intensive applications, 2019.

[8] Casale U.,Programmable LiM: a modular and recon
gurable approach to the Logic in Memory, 2020

[9] F. Alibart, T. Sherwood and D. B. Strukov, "Hybrid CMOS/nanodevice circuits for high throughput pattern matching applications," 2011 NASA/ESA Conference on Adaptive Hardware and Systems (AHS), 2011, pp. 279-286, doi: 10.1109/AHS.2011.5963948.

[10] R. Kaplan, L. Yavits, R. Ginosar and U. Weiser, "A Resistive CAM Processing-in-Storage Architecture for DNA Sequence Alignment," in IEEE Micro, vol. 37, no. 4, pp. 20-28, 2017, doi: 10.1109/MM.2017.3211121.

[11] M. Imani and T. Rosing, "CAP: Configurable resistive associative processor for near-data computing," 2017 18th International Symposium on Quality Electronic Design (ISQED), 2017, pp. 346-352, doi: 10.1109/ISQED.2017.7918340.

[12] A. Rahimi, A. Ghofrani, M. A. Lastras-Montano, K. Cheng, L. Benini and R. K. Gupta, "Energy-efficient GPGPU architectures via collaborative compilation and memristive memory-based computing," 2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC), 2014, pp. 1-6, doi: 10.1109/DAC.2014.6881522.

[13] Y. Chen, L. Lu, B. Kim and T. T. -H. Kim, "Reconfigurable 2T2R ReRAM Architecture for Versatile Data Storage and Computing In-Memory," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 28, no. 12, pp. 2636-2649, Dec. 2020, doi: 10.1109/TVLSI.2020.3028848.

[14] M. Imani, Y. Kim and T. Rosing, "MPIM: Multi-purpose in-memory processing using configurable resistive memory," 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), 2017, pp. 757-763, doi: 10.1109/ASPDAC.2017.7858415.

[15] C. Eckert et al., "Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks," 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA), 2018, pp. 383-396, doi: 10.1109/ISCA.2018.00040.

[16] V. Seshadri et al., "Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology," 2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2017, pp. 273-287.

[17] W. Simon, J. Galicia, A. Levisse, M. Zapater and D. Atienza, "A Fast, Reliable and Wide-Voltage-Range In-Memory Computing Architecture," 2019 56th ACM/IEEE Design Automation Conference (DAC), 2019, pp. 1-6.

[18] M. Imani, S. Gupta and T. Rosing, "Ultra-efficient processing in-memory for data intensive applications," 2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC), 2017, pp. 1-6, doi: 10.1145/3061639.3062337.

[19] D. Bhattacharjee, R. Devadoss and A. Chattopadhyay, "ReVAMP: ReRAM based VLIW architecture for in-memory computing," Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017, 2017, pp. 782-787, doi: 10.23919/DATE.2017.7927095.

[20] P. Chi et al., "PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), 2016, pp. 27-39, doi: 10.1109/ISCA.2016.13.

[21] M. Soeken, P. Gaillardon, S. Shirinzadeh, R. Drechsler and G. De Micheli, "A PLiM Computer for the Internet of Things," in Computer, vol. 50, no. 6, pp. 35-40, 2017, doi: 10.1109/MC.2017.173.

[22] W. Qian, P. Chen, R. Karam, L. Gao, S. Bhunia and S. Yu, "Energy-Efficient Adaptive Computing With Multifunctional Memory," in IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 64, no. 2, pp. 191-195, Feb. 2017, doi: 10.1109/TCSII.2016.2554958.

[23] G. Papandroulidakis, I. Vourkas, A. Abusleme, G. C. Sirakoulis and A. Rubio, "Crossbar-Based Memristive Logic-in-Memory Architecture," in IEEE Transactions on Nanotechnology, vol. 16, no. 3, pp. 491-501, May 2017, doi: 10.1109/TNANO.2017.2691713.

[24] S. Angizi, Z. He and D. Fan, "PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-Efficient Logic Computation," 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), 2018, pp. 1-6, doi: 10.1109/DAC.2018.8465706.

[25] K. Yang, R. Karam and S. Bhunia, "Interleaved logic-in-memory architecture for energy-efficient fine-grained data processing," 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS), 2017, pp. 409-412, doi: 10.1109/MWSCAS.2017.8052947.

[26] Pala, Didem and Causapruno, Giovanni and Vacca, Marco and Riente, Fabrizio and Turvani, Giovanna and Graziano, Mariagrazia and Zamboni, Maurizio, "Logic-in-Memory architecture made real," Circuits and Systems (ISCAS), 2015 IEEE International Symposium on security and privacy, 2015, pp. 1542-1545, doi: 10.1109/ISCAS.2015.7168940.

[27] V. K. Joshi, P. Barla, S. Bhat and B. K. Kaushik, "From MTJ Device to Hybrid CMOS/MTJ Circuits: A Review," in IEEE Access, vol. 8, pp. 194105-194146, 2020, doi: 10.1109/ACCESS.2020.3033023.

[28] J. S. Friedman and A. V. Sahakian, "Complementary Magnetic Tunnel Junction Logic," in IEEE Transactions on Electron Devices, vol. 61, no. 4, pp. 1207-1210, April 2014, doi: 10.1109/TED.2014.2306395.

[29] Steven L. Brunton and J. Nathan Kutz, "Data Driven Science & Engineering, Machine Learning, Dynamical Systems, and Control", Seattle, WA, March 2018.

[30] Manuel Le Gallo , Abu Sebastian, Giovanni Cherubini, Heiner Giefers, Evangelos Eleftheriou, "Compressed Sensing With Approximate Message Passing Using In-Memory Computing", IEEE TRANSACTIONS ON ELECTRON DEVICES, VOL. 65, NO. 10, OCTOBER 2018.

[31] B. Erbagci, N. E. C. Akkaya, C. Teegarden and K. Mai, "A 275 Gbps AES encryption accelerator using ROM-based S-boxes in 65nm," 2015 IEEE Custom Integrated Circuits Conference (CICC), 2015, pp. 1-4, doi: 10.1109/CICC.2015.7338448.

[32] M. Wang, C. Su, C. Horng, C. Wu and C. Huang, "Single- and Multi-core Configurable AES Architectures for Flexible Security," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 4, pp. 541-552, April 2010, doi: 10.1109/TVLSI.2009.2013231.

[33] I. Verbauwhede, P. Schaumont and H. Kuo, "Design and performance testing of a 2.29-GB/s Rijndael processor," in IEEE Journal of Solid-State Circuits, vol. 38, no. 3, pp. 569-572, March 2003, doi: 10.1109/JSSC.2002.808300.

[34] W. Shan, A. Fan, J. Xu, J. Yang and M. Seok, "A 923 Gbps/W, 113-Cycle, 2-Sbox Energy-efficient AES Accelerator in 28nm CMOS," 2019 Symposium on VLSI Circuits, 2019, pp. C236-C237, doi: 10.23919/VLSIC.2019.8778189.

[35] S. K. Mathew et al., "53 Gbps Native $GF(2^4)^2$ Composite-Field AES-Encrypt/Decrypt Accelerator for Content-Protection in 45 nm High-Performance Microprocessors," in IEEE Journal of Solid-State Circuits, vol. 46, no. 4, pp. 767-776, April 2011, doi: 10.1109/JSSC.2011.2108131.

[36] S. Mangard, M. Aigner and S. Dominikus, "A highly regular and scalable AES hardware architecture," in IEEE Transactions on Computers, vol. 52, no. 4, pp. 483-491, April 2003, doi: 10.1109/TC.2003.1190589.

[37] Chih-Pin Su, Tsung-Fu Lin, Chih-Tsiun Huang and Cheng-Wen Wu, "A high-throughput low-cost AES processor," in IEEE Communications Magazine, vol. 41, no. 12, pp. 86-91, Dec. 2003, doi: 10.1109/MCOM.2003.1252803.

[38] S. Gueron, "White Paper: Advanced Encryption Standard (AES) Instruction Set," July 2008, Intel Mobility Group, Israel Development Center.