POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Programmable LiM: a Modular and Reconfigurable Approach to the Logic in Memory

Relatori:
Prof. Mariagrazia Graziano
Prof. Wenjing Rao

Candidato:
Umberto Casale

Ottobre 2020

Acknowledgments

After months of work, we come to the conclusion of this amazing journey: my thesis. I learned a lot from my tutors and all the wonderful people leading me during these months. I wish to acknowledge the support of my family and friends, with which I shared happy times and suffering. They enjoyed for me and with me. Much appreciation goes out to my Italian supervisor, the Professor Mariagrazia Graziano, for the confidence she placed in me. I wish also to show my gratitude to my American advisor, the Professor Wenjing Rao, for the constant willingness to help me and the huge professionalism, despite the great distance separating us. Moreover, a special thanks to Andrea Coluccio, who has never stopped believing in me and supporting me. And finally, I'd like to express my deepest gratitude to all the professors leading me in this journey who they taught me all that I know: Mariagrazia Graziano, Maurizo Zamboni, Giovanna Turvani and Marco Vacca.

UC

0.1 Summary

Data traffic between the CPU and the memory is the bottleneck for the processor's performance: the Logic in Memory approach breaks down this wall. Memory becomes much more than a storage device: it's able to perform logic operations directly inside it. So, this approach shifts the processor architecture from a CPU center to another one where the processor is not involved in every computation. The space of logic in memory has been explored for years, with several different approaches. This thesis work explores the logic in memory with a modular approach called LEGO-Like approach. The designer builds the smart memory according to his specifications: an RTL library of components is his design space. In other words, he builds up the memory by choosing the bricks to use from the library first and then assembling them together.
First of all, the state of art of the Logic in Memory is explored for contextualizing where this thesis works. Several architectures are explored: from a technological level up to the RTL level. Then, the LEGO-Like approach is presented. It's explored at different degrees: from the high level architecture down to the RTL design of each block. The first step explains how the ecosystem embedding a smart memory works: a building block view shows which elements collaborate together for implementing SIMD operations inside the memory. Then, each component is deeply analyzed: the library of RTL components, the front-end and the back-end. The goal of the last step is to answer two questions trough the performance evaluation : what is the overhead due to the smart features? Is it worth it in comparison to a RISC-like processor? The test-bench used is a Binarized Convolution Neural Network: it was implemented on both a RISC-like processor(the DLX) and two LEGO LiM Units.

Table of contents

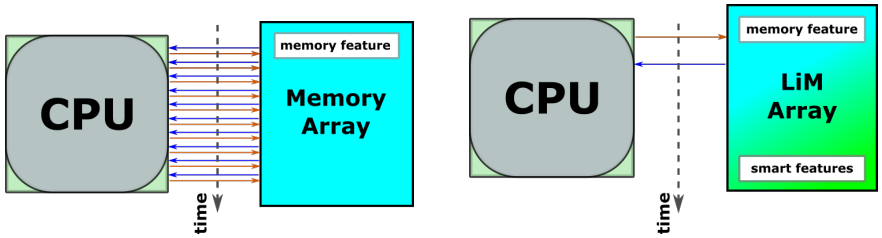
List of tables

List of figures

Chapter 1

Introduction

The Moore's law is going to reach its peak for the available technology: new architectural approaches have to be found for improving processors' performance. The **Logic in Memory** (LiM) architecture changes the standard skeleton of processors in order to solve one of its main bottleneck: the **memory wall**. Massive data exchanged between processor and memory is the issue and the LiM approach is the best way to solve that(1.1).



(a) CPU center architecture (b) LiM architecture

Figure 1.1: The LiM architecture drastically reduces the traffic of data between memory and CPU

Nowadays, several strategies for implementing a smart memory can be found in literature: from changing the electrical properties of a memory cell to modifying the analog circuits surrounding the memory array. All these approaches have one common point: the array is application dependent. In other words, the memory was designed with a view limited to specific operations, without any general purpose features. This thesis work explores a new high level approach for computation in memory: it has been called the **Lego LiM approach**. The key point is that the LiM array is fully modular and programmable : the designer has to configure it according to the complexity required. The design space for the LiM array is huge: it can look like from a standard memory to a complex configurable array, similar to an FPGA. The rest of the thesis is organized in three main blocks: the exploration

of the state of the art(chapter 2), the detailed analysis of the LEGO LiM Unit from a high level view down to the RTL level(chapter 3 to 8) and the performance estimation (chapter 9 to 10) . The last section has three main goals: comparing the standard memory array with a LiM array, finding out the impact of the Control Unit and analyzing the performance of a LiM Unit respect a standard RISC-like processor.

Chapter 2

State of the art

2.1 Von Neumann paradigm

Modern computing architectures are CPU-center according to the **Von Neumann paradigm**(VN paradigm): data are exchanged between CPU and memory for computation purpose. A Von Neumann computer(no matter the architecture, Harvard or VN) has three building blocks: the CPU, the BUS and the memory storing both data and instructions. The CPU executing the instructions of the running program changes the content of the memory in some way: the processor retrieves data from memory, performs operations with them and writes back the results in the memory.

2.1.1 Limitations

This paradigm has to main limitations : the so called Von Neumann bottleneck and the power consumption.

Von Neumann bottleneck

The terms **Von Neumann bottleneck** (coined by J.B. [2]) addresses the problems related to the rigidity of the VN paradigm: a technological problem related to the difference of performance between CPU and memory and, especially, an intellectual barrier for the programmers.

- About the latter problem, in order to minimize the enormous traffic of words between CPU and memory, the programmers spent a lot of time thinking on code optimization rater than on conceptual tasks.
- For what concern the former problem, the gap between the progress made in recent years on CMOS technology and the improvements obtained for memory

causes the difference of speed between CPU and memory: memories can't provide data to the CPU as fast as it could process that(2.1). So, no matter how fast is the processor , the overall performance are limited by the memory bandwidth. The scaling process of transistor just further increases the gap between logic and memory performance.

Power Dissipation

Furthermore, the power consumption is another critical problem intrinsically related to the VN paradigm: a huge power dissipation is generated by the continuous data movements between memory and CPU. This situation is exacerbated for data-intensive applications.

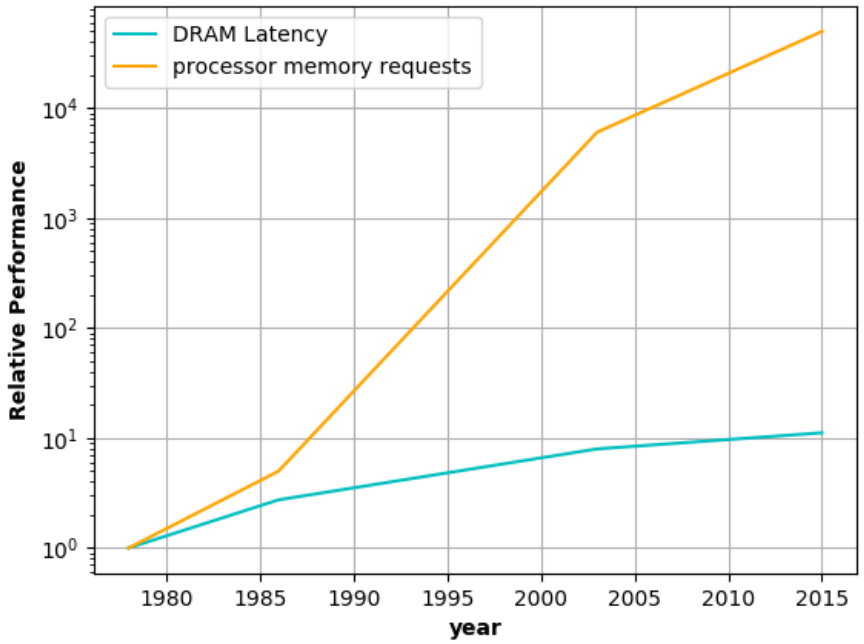


Figure 2.1: The gap between memory and CPU is measured by comparing the memory request of a single core and the latency of a DRAM access. Both of them are relative data: the baseline for the CPU performance is th VAX-11/780 ; the 64 KiB DRAM is the baseline for the memory instead.This figure is inspired to the study done in [QAP].

2.2 Breaking down the Memory Wall

To address this problem the weak components of the paradigm have to be improved : the memory, the bus or the whole paradigm itself .

2.2.1 Memory hierarchy

To attenuate the memory wall , several architectures implement memory hierarchy to hide the latency of the main memory by placing smaller but faster intermediate memories between the main one and the processor([10]).

2.2.2 Prefetching

Another approach to break the memory wall is to implement the prefetching in order to overcome the latency problem. The idea is to predict the next data to retrieve in order to begin the read operation before the fetch phase. In other words, the prefetching allows to anticipate the beginning of the fetch operation before the fetch phase: so , the latency is just hidden behind other phases([10]).

2.2.3 Beyond the Von Neumann architecture

However,the approaches analyzed so far are not enough : the problem must be tackle at the root. The idea is to go beyond the strictly separation between logic and memory unit imposed by the von Neuamnn approach: **computation in Memory** moves logic units inside the memory in order to perform computation on data without moving data outside the memory . This innovative approach brings three main benefits:

- Reduction of data traffic between memory and CPU drastically decrease the power consumption;
- Processing data directly inside the processor allows to fully exploit the internal memory bandwidth;
- Increase the degree of parallelizability admitted introducing the possibility of distributed computation(both in memory and in the processor).

Categories

According to [14] the state of art for the Logic in Memory can be categorized in 4 degree of smart memory: **Computation near Memory**(CnM), **Computation with Memory**(CwM), **Computation in Memory**(CiM) and **Logic in Memory**(LiM),

CnM Architectures still observe the Von neumann paradigm: logic and Memory are kept separate. The idea is to break down the interconnection latency by stacking memory layers and logic ones in the same 3D structure. Data are moved from one wafer to and other through a vertical electrical interconnection called TSV(trough-silicon via) : the shorter interconnections bring wider memory bandwidth and lower power consumption.
On one hand there are structures like [12] where the logic stacked in the 3D structure handle the operations required to manage the DRAM memory layer: refresh, error correction, sequencing and so on.
On the other hand, [9] maps a multicores(64 cores) architecture on a 3D-Architecture. Each core has a MIPS-like architecture with the own dedicated memory tile.

CwM Computing with Memory exploits CAM for implementing operation in a LUTlike approach: CPU generate a key and the memory directly generate the results. Different approaches can be identified.
[7] reduces the data movement of the BLAST(Basic Local Alignment Search Tool is a tool for alignment DNA sequences) by exploiting the ability to locally perform parallel comparison locally of a 3D resistive CAM(ReCAM). The architecture is name RADAR.

[8] adopts a similar approach: the electrical characteristic of a ReCAM allows to perform some basic query functions directly in memory.

More complex logic operations can be implemented by the associative processor proposed by [18]. The approach is very similar to a LUT: pre-computed results are stored inside the ReCAM and the combination of the operands is used as Key for it.

A different solution in exposed in [4], where the idea is to exploit the electrical property of a 10T NOR cell based CAM for locally implementing the XNOR-popcount of a BNN. The input images (IFMAPs) are stored in the CAM and the weights are used as key: this is the starting point that made possible the implementation of popcountt as a CAM search operation.

CiM The wish to kept intact the memory array brought to the third approach for the logic in memory architecture that [14] called Computing in Memory. Precisely, the idea is to left unchanged the memory cell structure and exploit the analog functionality of the peripheral circuitry around the array for implementing operation directly inside the memory: data are read from the array, then they are computed by the peripheral circuits and the results are written back in the array. Hence, there are not movement of data between memory and the logic unit. Technologies used for implemented this smart arrays are both nonvolatile and volatile.

PIMA architecture proposed by [1] implements complex computation in a memory based on Magnetic RAM (MRAM) technology. Precisely, the technology used is the Spin Orbit Torque MRAM(SOT-MRAM): the magnetization direction of a metal layer(the Free Layer) sets the equivalent resistance of the cell. It can assume two values: the lower indicates the logic '0' and the higher is for logic '1'. It allows to perform operation between two or three data stored into the peripheral circuits : OR, NOR, AND, NAND, maximum and minimum can be implemented.

[3] proposed a CiM architecture for accelerating Neural Network application. Cells of the array are based on the metaloxide Resistive RAM technology : the information is mapped on the resistance of the cell(low Resistance for 0 and high Resistance for 1). The array is partitioned In three region: a subarrays able to store data, a subarray with both storage and computation capability and another subarray that contains temporary results . Peripheral circuits are properly adapted for implementing the key operations of Neural network algorithm .

On the other hand, [15] and [16] respectively propose a DRAM and an SRAM array that perform in Memory operation.

LiM The key idea behind the LiM approach is to give smart features to each cell . Exactly, simple logic is integrated in the memory cell itself. In other word, a

cell has inside both a computation unit and a store unit. This approach avoids any data traffic between memory and peripheral circuits(for CiM) or processing units (for CnM).

[17] proposes an architecture called MISK where the LiM array is integrated as a section of the cache memory. The LiM array has a structure organized in four layers: memory, logic, memory and latch. The logic layer computes data coming from the surrounded memory ones and the final or partial results are stored inside the latches.

[5] introduces a LiM cell made up by three virtual layers, i.e. technological independent : Memory plane, Routing Plane and Logic Plane. Beside the traditional write/read, the proposed architecture allows to : locally perform logic operations between adjacent cells(logic-logic ops), interconnect adjacent or not adjacent cells(Toc-toc write/read and Remote write/read respectively). Moreover the paper proposes an implementation in two layers(since technological limits) based on NanoMagnet Logic technology.

A new computation in memory approach: CLiM

Which one of the computation in memory approach analyzed so far has to be implement depends the target application. The idea key idea introduced in [14] is to implement in the same structure all the different degrees of computation in memory. The approach is called Configurable Logic In Memory (CLiM)

Architecture The proposed architectures is made up by two main blocks: a CnM unit and a smart memory(LiM + CiM). The operations cannot be managed by the logic in the memory, are executed in the dedicated logic near the main memory(CnM). Each cell of the smart memory integrates both storage and computation unit(LiM). Moreover, for handling more complex operations inside the memory, there are extra-row and extra-column logic blocks(CiM).

The CLiM cell has computation unit and a storage unit. The **Computational Block** has a Full Adder and a logical block allowing to implement bit-wise operations. The inputs are the locally stored data and a signal provided by the input multiplexer: can be an external signal or data coming from other cells(the source

cells of this signal depends on the location of the cell). The **storage unit** works as a standard memory cell.

CLiM Array The interconnection among the cells of the CLiM array are managed by several multiplexers embedded in each cell. Five connection can be established :

- *intra-cell* connection for storing the result of computational block inside the storage block of the same cell;
- *inter-cell* connection with the north-east cell;
- *inter-cell* connection with the south-east cell;
- *inter-cell* connection with the east cell;
- *inter-cell* connection with the south cell;

This articulated interconnection system allows to handle five types of operations:

- *local*: data are manipulated and then stored in the same cell;
- *Intra-row/ column*: several cells of the same row/column are interconnected together for implementing an operation;
- *inter-row/column*: an operation involves cells of two rows/columns;

Interrow or column operations allows to implement logical bitwise operations between two words stored in two consecutive row/ cell. On the other hand, both inter and intrarow interconnections can be exploited for implementing more complex structure as an Array Multiplier or a Ripple Carry.

Trade off Two degrees of freedom cab be exploit by the designer: interconnection and computational unit. The configuration of the interconnection among cell is the key point for implementing operation in memory: in other words, which cells are interconnected determined what kind of operation can be implemented. However, have a fully configurable interconnection system could be expensive in terms of cost

and logical required for managing it. On the other hand, an advanced computation unit able to handle several kind of operation could be as useful as very expensive.

The approach adopted in [14] is to design special purpose CLiMA: this approach suits the complexity of the structure to the algorithm that has to be implemented.The more detailed architecture presented in [14] is a CLiMA able to accelerate operations of a Quantized Convolutional Neural Network. The cell can be configured for implementing logical bitwise operation, arithmetic operation an Shift operation.

Strong and Weak points The strong point of this approach are the configurability and the parallelism offered by the architecture. On the other hand, the weak points are the control Complexity and the Interconnection Complexity. Moreover, the whole architecture has to be redesigned each time according to the application.

Chapter 3

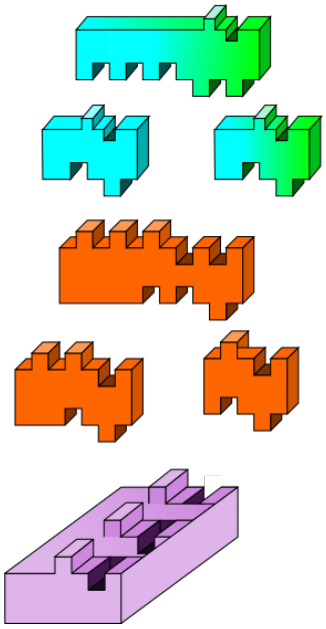
A new LiM approach: Lego-LiM architectures

3.1 Motivations

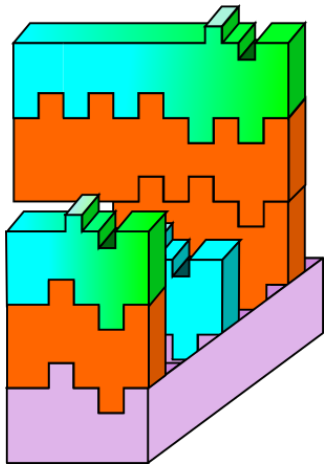
The LiM arrays implemented so far are ASIC-like unit: the design flow begins from an application and ends with an architecture optimized for that specific bunch of instructions. So, complexity and the instruction set are carved in stone. The approach explored in this thesis work follows the opposite direction: adaptability and user-friendliness are the strong points. The idea is to define a technological independent and open source library of components(RTL level), where open source means that is it can be upgraded with new units at any time. Then, the designer can combine these building blocks for generating several different smart arrays(LiM array), with different degrees of complexity. The instruction set of the memory is laid down once the designer has built up the LiM array. Then, the programmer can write programs that will run on it.

The library of building blocks is the **LEGO-LiM library**. Basically, it defines the design space for the designer of the LiM architecture : it stores all the blocks that can be used for building up the LiM array. These building blocks are called **LEGO-LiM**. The designer has to select the LEGO-LiM from the library that allow him to build a customized smart array with the proper instruction set: this approach has been defined **LEGO-Like approach**(3.1). Indeed, the design flow is the same adopted for building up a structure with LEGO bricks: a package of bricks can generate thousand of different architectures depending on both which of that are chosen and how they are placed together. Obviously, bricks must be placed in a proper order for making the connections work. *chapter 8* guides the designer to build his customized LiM array and to adapt the Conrol Unit to it.

This approach allows to fit the architecture to the required complexity: power and area loss can be avoided. The main strong points of this approach are flexibility and configurability in terms of modularity.



(a) Step1: Select the LEGO LiM



(b) Step2: Put them together

Figure 3.1: LEGO LiM approach

Chapter 4

Lego-LiM architecture: an HL overview

4.1 The high level architecture

The ecosystem embedding a smart memory(LiM ecosystem) has three main units(4.1): the **LiM Unit**, the **LiM scheduler** and the CPU, or anyone uses a memory as a storage device. The **LiM scheduler** manages the execution of LiM operations: it decides both which instructions have to be executed in the memory and when executing them. The whole process is *completely transparent* for the CPU: its interface with the LiM unit is as complex as the one with a standard memory. Word Line(WL), Read Enable(Ren), Write Enable(Wen), memory Address(Add), word and memory out are the signals exchanged between them. The **LiM Unit** is the core of this thesis project: it's the smart memory array. Due to its nature, it's able to manage **SIMD**(Single Instruction Multiple Data) operations. Indeed, a smart array is made up by stacked memory rows with smart features: SIMD operations allow to fully exploit all of them in parallel.

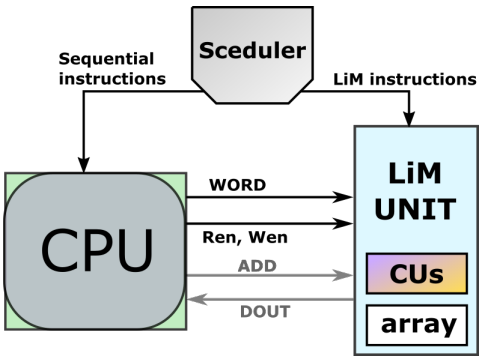


Figure 4.1: High Level View of a LiM ecosystem.

4.2 explores with more details the whole system. The LiM unit is made up by four blocks: the **LiM array**, that is the smart array itself; the**LiM control unit** managing both the LiM array and the interface with the scheduler; the **near memory unit** embedding dedicated computational units and the standard memory decoder. Although neither the memory decoder nor a scheduler were implemented for this thesis project, figure 4.2 offers a global point of view of the a complete LiM ecosystem. Notice that the LiM unit, that is the focus of this project, is much more than a storage unit: it takes the form of a co-processor.

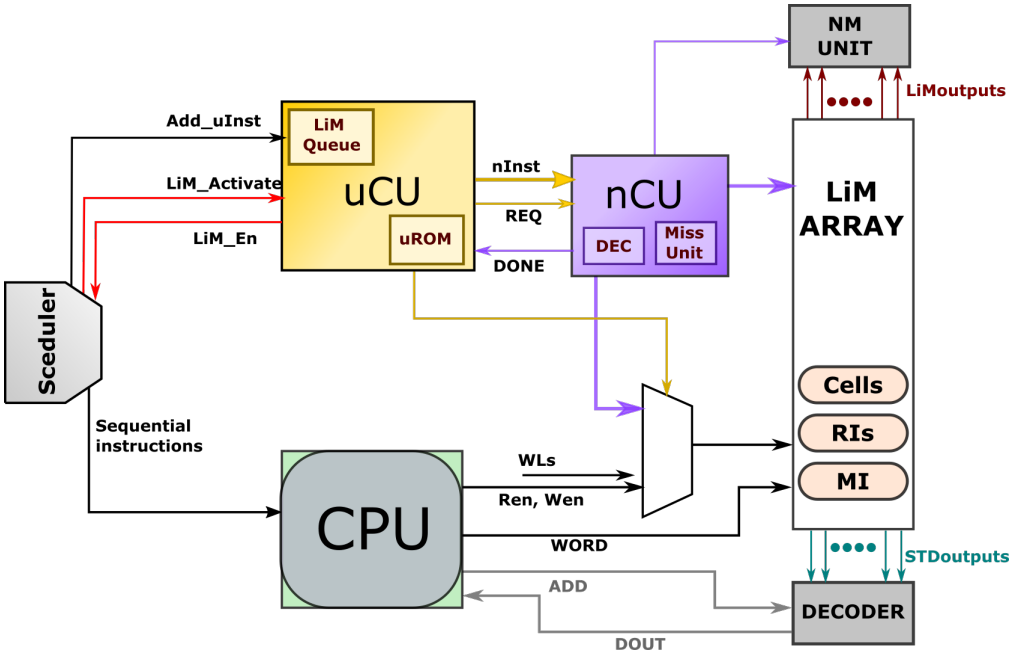


Figure 4.2: Detailed view of a LiM ecosystem. Units grey colored have not been developed for this project.

4.1.1 The control units

The control unit is implemented with an hierarchical approach(4.3): a **nCU** acts as a front-end and a **uCU** acts as a back-end. If the first one is independent from the array, the back-end has to be **partially** adapted to the implemented LiM array. The big advantages of this approach are

- easier debug for the designer;

- easier programming for the user;
- flexibility: also if the LiM array changes, the main components of the CU don't change.

The nCU suits the LiM array implemented: it's able to manage its smart features. On the other hand, the uCU doesn't see the LiM array. It manages the interface with the scheduler and acts as arbiter for the LiM array: it decides if the array has to be used as a standard memory by the CPU or if LiM operations have to be implemented. In the latter case, the **nCU** controls it.

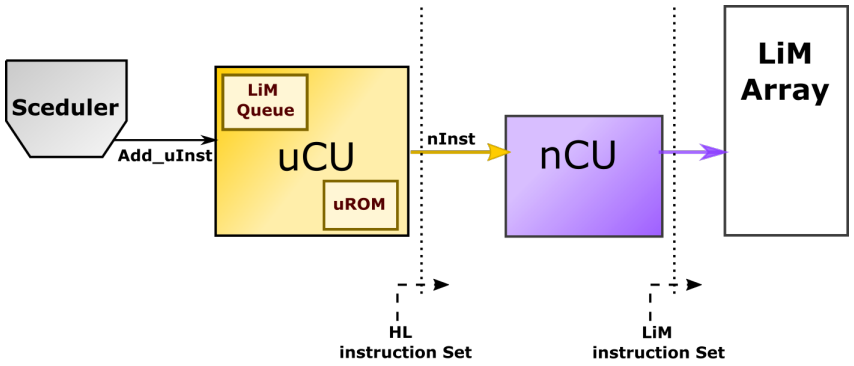


Figure 4.3: Hierarchy of the control units

The scheduler fills the **LiM queue** with the addresses of the programs to execute in the LiM unit. Once activated(**LiM_activate=1**), the uCU translates the addresses stored in the queue in flows of **nano instructions**(nInstructions) that control the nCU. Each nInstruction implement a LiM operation: the nCU generates the proper configuration signals for the LiM array.

4.1.2 Handshakes

Two different handshakes have to be observed: the one between the two CUs and the one between the nCU and the scheduler.

nCU and uCU

The interface between the CUs consists in two signals: **DONE** sent by the nCU and **Request**(REQ) sent by the uCU. REQ validates the nInstruction generated

by the uCU: the nCU fetches a new instruction as soon as the request is asserted at the rising edge of the clock. Indeed, the nCU can read each clock cycle a new nInstruction to manage. The DONE signal indicates the state of the nCU: idle if it's high and busy if it's low. The DONE signal goes down after the fetch of the first instruction, that is as the execution of instruction in the array begins. It comes back high in the last cycle of the last instruction executed. Notice that each nInstuction takes two cycle. 4.4 shows the timing for a generic stream of nInstructions.

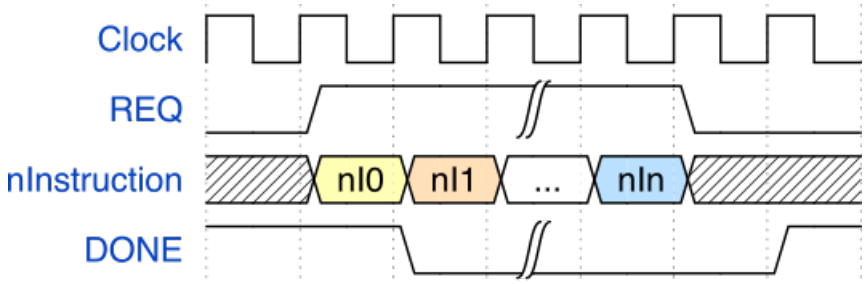


Figure 4.4: Handshake between the front-end and the back-end

uCU and scheduler

The interface between the front-end and the scheduler consists in two signals: **LiM Enable**(LiMen) and **LiM Mode**. The former is sent by the scheduler for activating the execution of LiM instructions. The latter is generated by the uCU for giving a feedback to the scheduler about the execution of LiM instructions: it's high when the LiM unit is executing a LiM operation and it's low when the LiM unit is in idle. The handshake takes four steps:

1. **Load**: the scheluder resets the CUs and loads the LiM queue. Then, the LiM unit can be activated.

2. **Start**: the scheduler asserts the LiMen for starting the execution of LiM operations. The LiM queue is fetched during the first rising edge and the addressed instruction is sent to the uCU at the following rising edge.

3. **Execution**: after the second rising edge, the LiM execution starts. So, the uCU gives the control of the array to the nCU(LiM mode asserted).

4. **End**: Once all the programs stored in the queue have been processed, the uCU gives back the control to the CPU and alerts the scheduler by denying the LiM mode.

Notice that the scheduler has to keep high the LiM enable until the LiM unit completes the execution of instructions. 4.5 shows the timing for a generic life cycle.

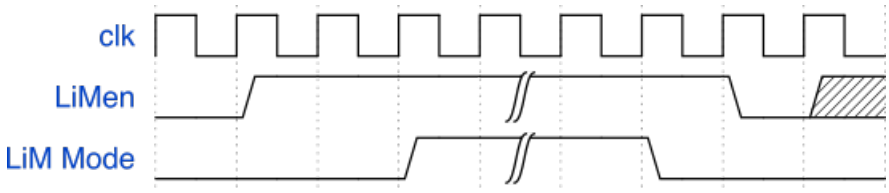


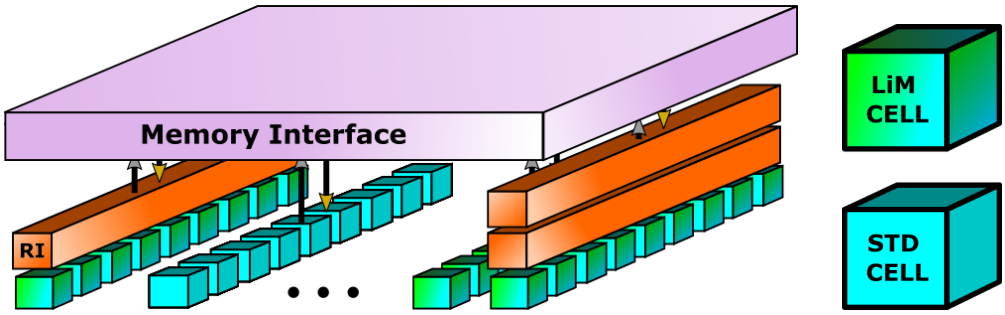
Figure 4.5: Handshake between the front-end and the scheduler

4.2 Lego LiM library

The **Lego-LiM library** defines the design space for the LiM architecture designer. It is a collection of the **Lego-LiM** that can be used by the designer for customize his architecture. This section explores the high level view of the whole library. Then, the skeleton of a LiM array is presented too.

4.2.1 Lego LiM

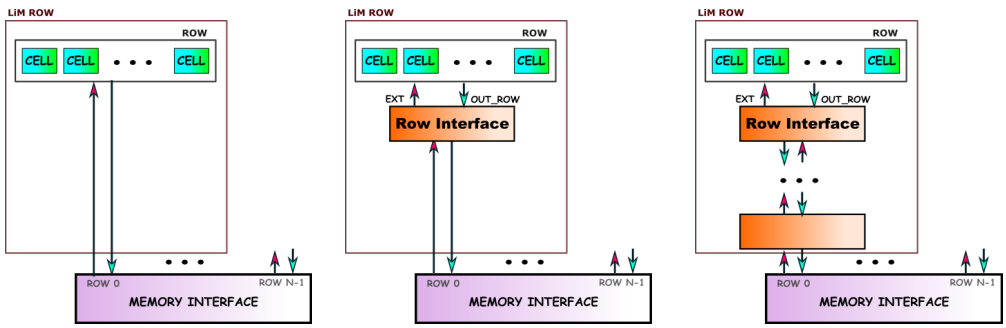
The **Lego-LiM** are the building blocks of a LiM architecture. They need to be heterogeneous enough to allow the designer to customize the architecture to the given specifications: having a library with equal size blocks it would have made it impossible. So, the winning strategy was been *divide et impera*: the blocks of the library are classified in three families according to them scope(or granularity), and each of these family is further split in classes depending on the functions implemented. The former classification places the components on three layers(4.6): LiM Cells, Row Interfaces and Memory Interface. The **LiM Cells** define how a bit, that is the smallest information stored, can be manipulated. The **Row Interfaces** (RI) define the operations that can be applied on the word stored in a row. The **Memory Interface**(MI) manages the communication between different rows(inter-row).



(a) LiM Array (b) LiM cells

Figure 4.6: HL view of a LiM array

Notice that the **row** is where a word is stored, that is a group of LiM cells sharing the same Bit Line. On the other hand, a **LiM row** is the portion of the LiM array made up by a row and its smart plug-ins, that are the RIs. Between each row and the MI there could be one or more stacked RIs. If there are one or more RIs, they filter the data exchanged between cells and the MI. However, the row is always at the lowest level of each row(bits). The scope of the CPU is limited to them. 5.1 shows the high level view of a LiM row in three cases, with three different degrees of complexity.



(a) no RI (b) 1 RI (c) n RIs

Figure 4.7: Hierarchy of a LiM row in three different cases: without RIs(a), whit one RI(b) or whit staked RI

Notice that, there is only one MI in each LiM array. It is the block with the wider scope: it can communicate with all the LiM rows and with the near memory unit, if there is one.

4.2.2 LiM Cells

The LiM cells are the components of the library with the finest granularity. Since they define if and how the bit stored can be modify, they open the instruction set of a LiM row: depending on the type of cell implemented, different **logical bit-wise operations** can be implemented. In contrast to the standard memory cells, each LiM cell has also smart capabilities. Depending on the logic feature, three families can be identified:

- **Memory Cells**(M Cells): the logic unit is as easy that no logic operation can be implemented;
- **Logic Cells**(L Cells): standard bitwise logical operations can be implemented;
- **Arithmetic Cells**(A Cells): since they embed at least an Full Adder or a Half Adder, they allows to implements also arithmetic operations.

4.2.3 Row Interfaces

The row interfaces complete the instruction set of a LiM row. Indeed, each RI can be seen as a plug-in that adds a smart feature to the row. Moreover, the RIs manage the traffic of data inside the LiM row. There could be both rows with several stacked row interfaces and rows without row interfaces.
Depending on the smart feature implemented, the row interfaces can be classified in four groups:

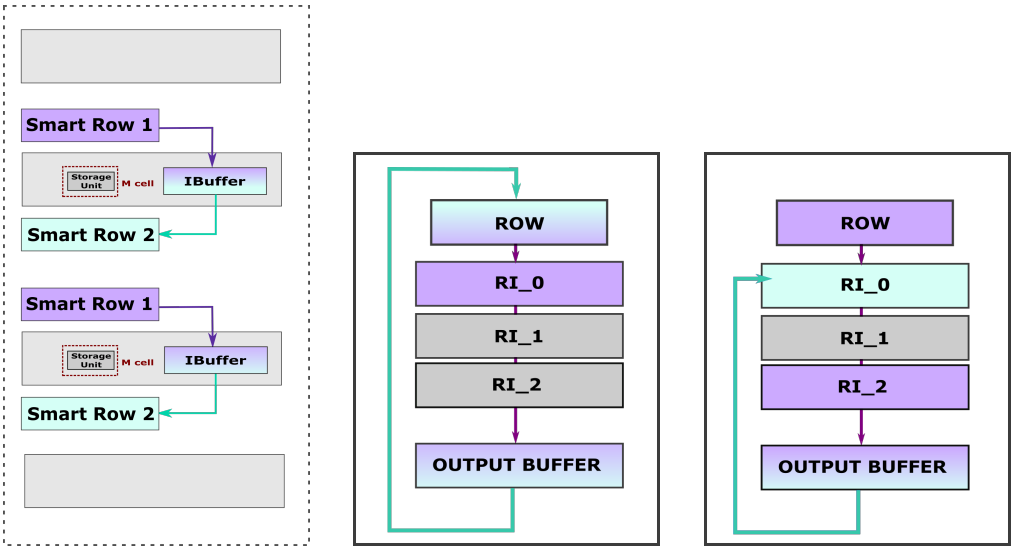
- **Shifters**: they can alter the order of the bits stored in a row;
- **Adders**: they can perform both logical and arithmetic operations;
- **Buffers**: they allow to store either the output or the input of a row;
- **Special Purpose**: they are a bunch of row interfaces for implementing peculiar operations.

The role of the Buffers

A buffer is a RI stores a temporary data for using it during the next clock cycles: the **Input Buffer** stores a data coming from another LiM row; the **Output Buffer** stores the result of the LiM row which it belongs to. Moreover, there is also an **InOut Buffer** that is able to work as either an Input or an Output buffer. Using a buffer as plug-in in a LiM row allows to increment the instruction set of the array. Indeed, some operations can't be performed without a buffer because of the architecture of the LiM array(4.8): they're feedback, complex algorithm and multi-rows operations.

The *Store Operation*(4.8.b) also allows to store the result of an operation performed in a LiM row in its row: this is what **feedback** stands for. It is implemented in two steps: storing the result in the output buffer and then writing back the result in the row. So, the output buffer is used for storing the partial result. Some **complex algorithms**(4.8.c) can't be performed in just one step: the partial results has to be stored in the output buffer for further computing it in the next cycle. A **multi-rows operations**(4.8.a) is an operation requiring two consecutive LiM rows to be completed: it begins in a LiM row exploiting its smart features, and it has to be further computed in a different one. For reasoning better explained in chapter 8, the input buffer between these two LiM rows has to be used as a checkpoint: it's written with the partial result by the first LiM row and then it's read by the second LiM row.

Moreover, the input buffer allow to use two external operands for an operations(see chapter 8).



(a) multi-rows operations (b) feedback (c) complex operation

Figure 4.8: The figure shows the complex operations that can be implemented with buffers. Grey colored components are not involved in the operation. Violet indicates component used in the first cycle and cyan the ones used in the second cycle. (a) shows the smart section of a template V2, (b) and(c) a LiM row

Notice from 4.8 that a buffer has to be placed as the last RI of a row. This makes a lot easier the configuration of RIs for managing the intra-row traffic. Although it's deeply explained in chapter 8, the key point is that this role keeps clear and sort the stream of intra-row data: data are processed going down through the RIs of the LiM rows toward the MI(Only the RIs activated processing the data, the other ones are just transparent). Then, the result in output to the RIs can be either stored in the buffer or it can be send to the MI. The only reason why a result goes upstream is to be stored in the cells.

4.2.4 Lego LiM array : templates

The LiM unit executes SIMD(Single Instruction Multiple Data). So, it must have a configuration, a shape that makes it feasible. A LiM array with all the LiM rows different from each other would be unimplementable and useless.
Unimplementable because of a huge interconnection overhead: the CUs would generate an insane number of configuration signals, one for each LiM row(4.9). Moreover,

taking into account that, first, the number of these signal increases whit the depth of the row and each configuration signal is up to 30 bits wide. So, It would became the bottleneck of the architecture.
Useless because of all the rows has to be able to handle the same smart features since the goal is to manage SIMD operations.

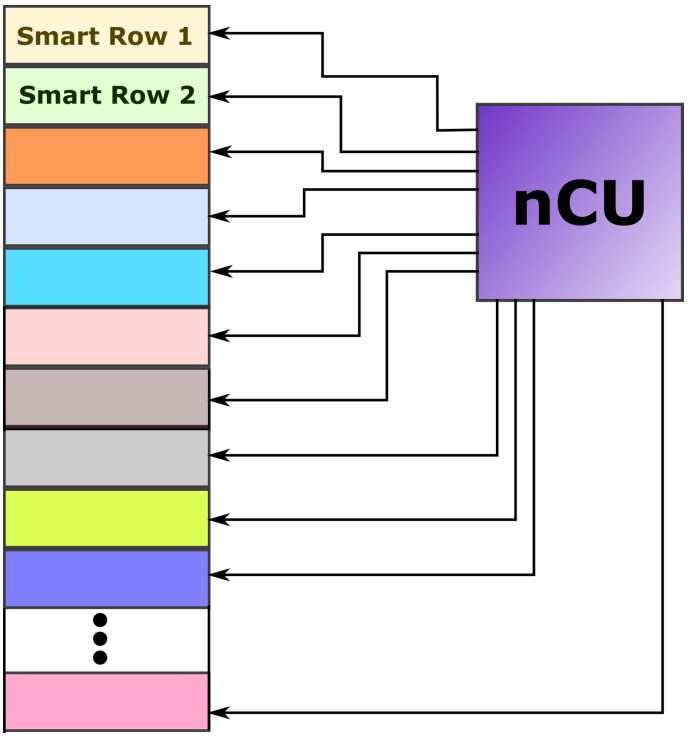
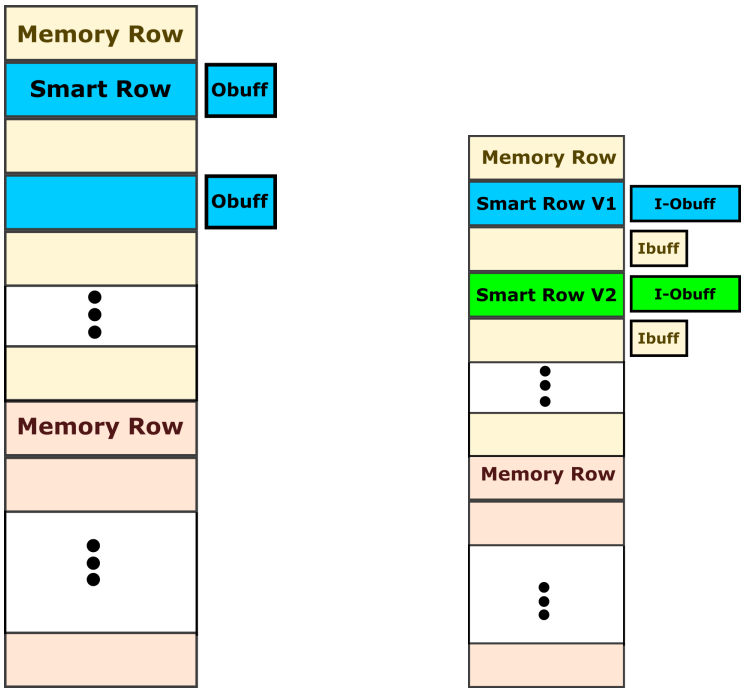


Figure 4.9: What will happen with an heterogeneous LiM array

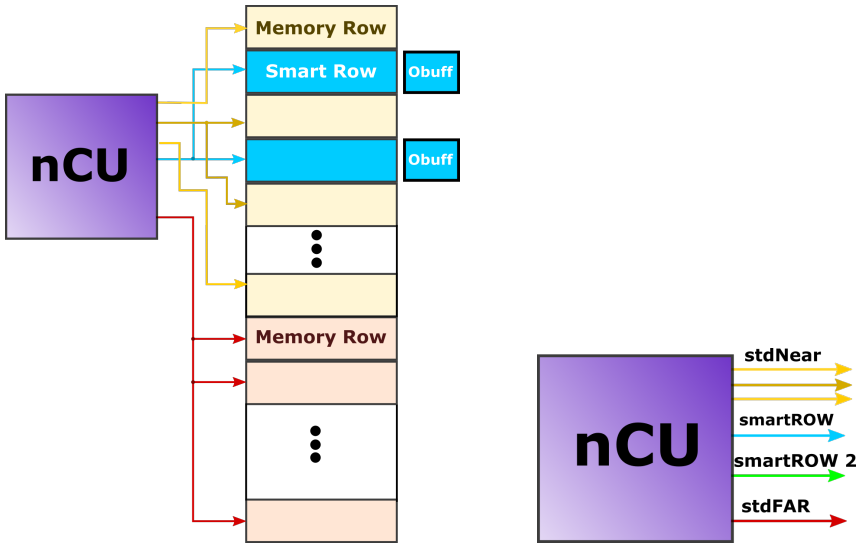
So, the idea is that skeleton of ever LiM array can assume two different homogeneous templates, that is the number of different LiM row is very limited: **Version 1**(V1) and **Version 2**(V2). 4.10 shows two possible implementations. The designer has to configure the smart row with the LEGO LiM of the library. Also buffers may or may not be placed depending on the application. Obviously, the instruction set reduces without them. If the V2 would allow to have a wider Instruction set, the V1 allows to implement an higher number of the same instructions in parallel with a given memory size.



(a) V1 (b) V2

Figure 4.10: The two available templates. Buffes belongs to the LiM rows.

These approach has two big advantages: more flexibility and the reduction of the complexity of the CUs. Flexibility means that the core of the the back-end is the same for all the implementation of the same template, independent from the Lego-LiM implemented. Moreover, since the number of different LiM rows is very small(four or five), the CU has to generates just few signals independently from the memory size. Indeed, almost all the LiM rows of the same type share the configuration signals, as shown in size(4.15). The only two exceptions are the first and the last LiM rows of the smart section(see chapter 8 for the reason).



(a) V1 (b) V2

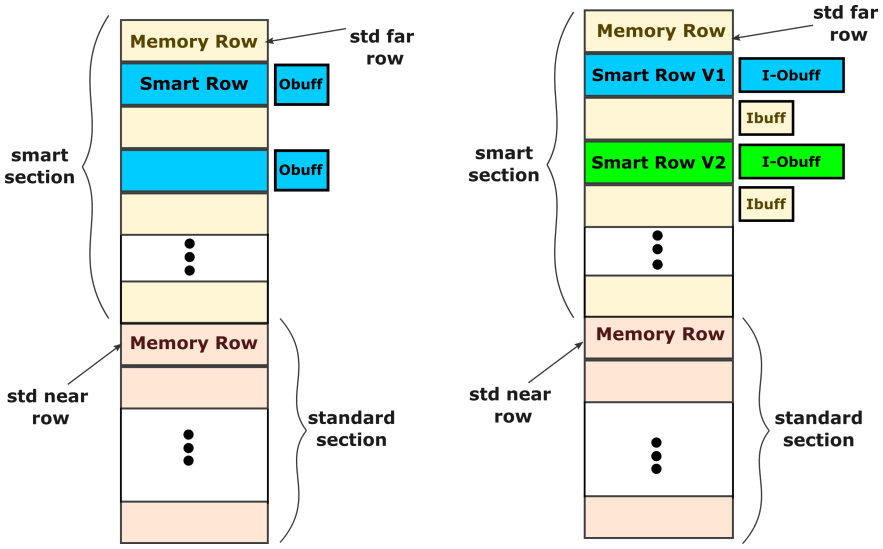
Figure 4.11: signals generated by the nCU for both the versions

Taxonomy

There are two types of LiM rows:the **smart rows** and the **memory rows**. If the latter ones is made up of LiM cells with only memory capability(M-Cells), the former ones have also smart features. Indeed, the smart rows embed either A-Cells or L-Cells and they could have one or more row interfaces: how many and which ones are choices of the designer. For both the templates, the LiM array has two sections(4.12): the Smart Section and the Standard Section. The **smart section** is where the LiM operations take place: it contains all the smart rows. How many smart rows it needs to use depends on the degree of SIMD required, that is how many operations have to be implemented in parallel. Moreover, there are also rows with just memory capability in the smart sections: **standard near rows**. The two standard rows surrounding a smart rows are usually called neighbourns of a smart row.

About the **standard section**, it has a key role: it allows to avoid redundancy, that is the storing of the same data multiple times in the memory. The rows of this section are called **standard far Rows**.

All the smart rows work simultaneously for performing an SIMD instruction: this



(a) V1 (b) V2

Figure 4.12: Taxonomy of a LiM array

is why the SIMD operation is the best choice to fully exploit all the smart features of the array.

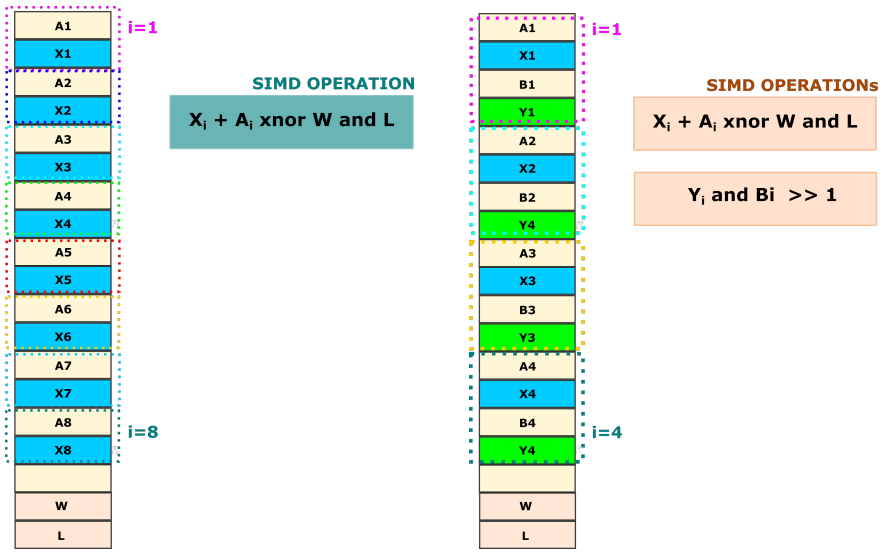
Where to store the operand

A single LiM operation of a SIMD instruction takes place in a smart row. The smart row can retrieve the operands to use:

- from one of the neighbors, that are the two rows surrounding a smart row;
- from the row of the smart row itself;
- from a row of the standard section trough the cache of the MI(see section below).

In order to avoid redundancy, the operands shared by all the operations of an SIMD instruction must be placed in the standard section and the ones change for each operation either in the smart row or in a neighbors. So, a smart row needs to read the cache just for retrieving shared operand stored in the standard section's rows. 4.13 shows an example of how to store the operands of an SIMD operation.
Notice that it has to be coherent for all the operations executed in parallel. In other

words, if a smart row stores its operands(the ones not shared) in its row and in the upper neighbor it has to be applied to all the other smart rows too(4.13)



(a) V1 (b) V2

Figure 4.13: The figure shows and example of how to store the operands in the array for an SIMD operation. In this case, the operand are stored in the smart row and in the upper neighbours. The shared operands have to be stored in the standard near instead.

4.2.5 Memory Interfaces

The memory interface is the traffic light of the LiM array: it manages data movements between different LiM rows. Moreover, it can handles the communication with the near memory unit if it's required. Since the interconnections have a key role for the operation in a LiM array, a stand alone section(5.1.6) is dedicated to the exploration of the design space for this components: the goal is to find the most flexible and the least expensive implementation.

The MI is made up by two main blocks(4.14): a **cache layer** and the **multiplexers layer**. The former allows to cache two rows of the array. The latter allows to interconnect LiM row: each smart rows can exchange data both with neighbors and with the cache registers.

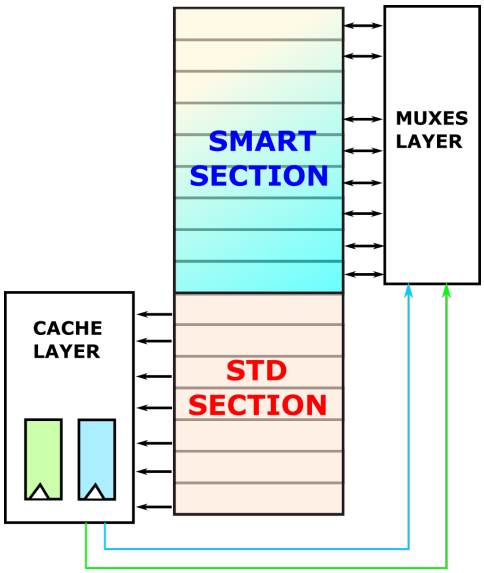


Figure 4.14: High Level view of Memory Interface with a LiM array of 16 rows

4.2.6 Standard Interfaces

The strong point of the LEGO bricks is the interface: the fixed interconnection allows to match a brick with a lot of other ones that are compatible with it and, moreover, to replace a brick with another one with the same interface , no matter its color. Having in mind that, the interfaces of the Lego LiM blocks were founded on three key points:

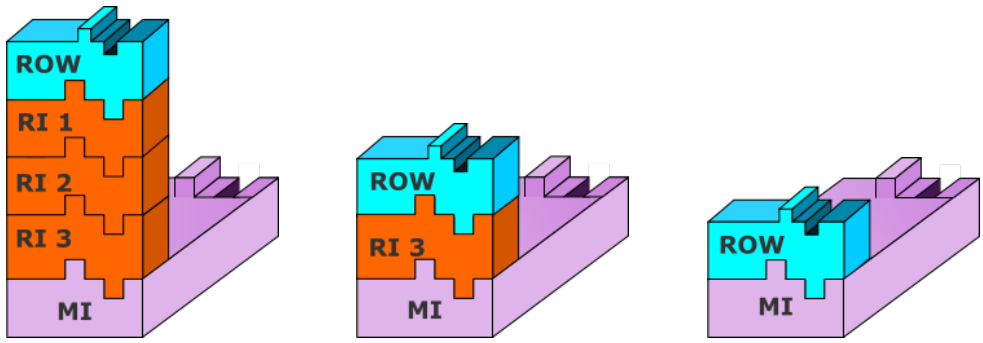
1. **Fixed Interfaces within a family**: blocks of the same family have the same interface, that is the same input ports and output ones as well;

2. **Fixed Interfaces between families**: the interface between Cells and RIs, the one between RIs or Cells and the MI and the one between two RIs are fixed no matter which kind of blocks are placed;

3. **Transparent for CPU**: the standard interface between CPU and a standard memory array must be kept unaltered .

(i) and (ii) allow to define a library of components that can be combined in a lot of different ways. (ii) allows to concatenate components in the same way, no matter which and how many are the blocks placed in each row. On the other hand, (i)

makes the components of the same families interchangeable to each other. In other words, cells and RIs can be interfaced in the same way independently from the types of components implemented; the same happens between RIs and MI, cells and MI in absence of RI and between two RIs.
Having a standard interface allow to see the LiM library as a box of LEGO bricks(4.16).



(a) (b) (c)

Figure 4.15: The figures represent a MI with just one LiM row. Figure (a) is the starting point. (b) and (c) demonstrates (ii). Moreover, (b) underlines how blocks of the same family are interchangeable(i)

The following sections focus on the data signals exchanged between LEGO LiM. About configuration signals, they are managed by the CU.

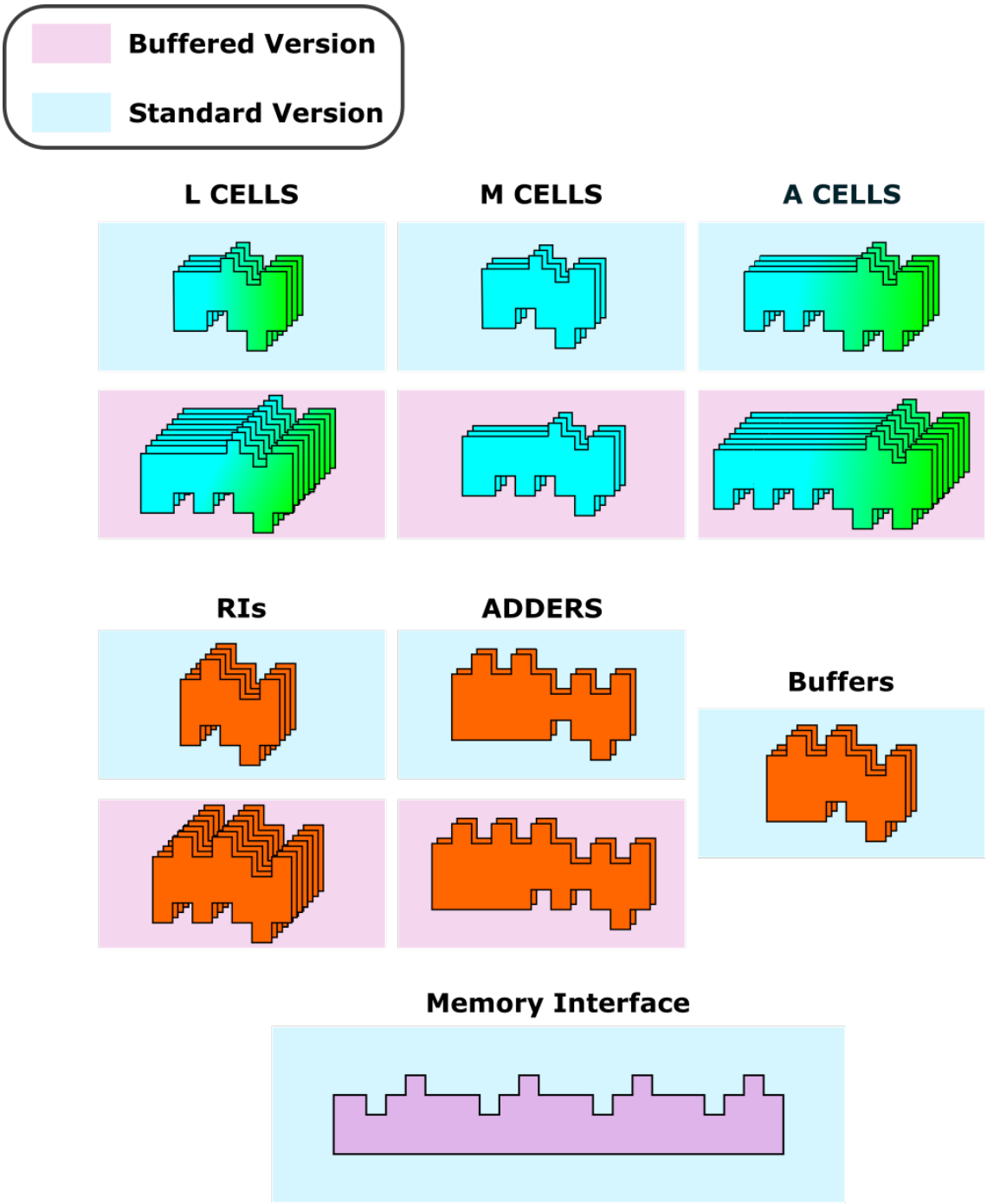
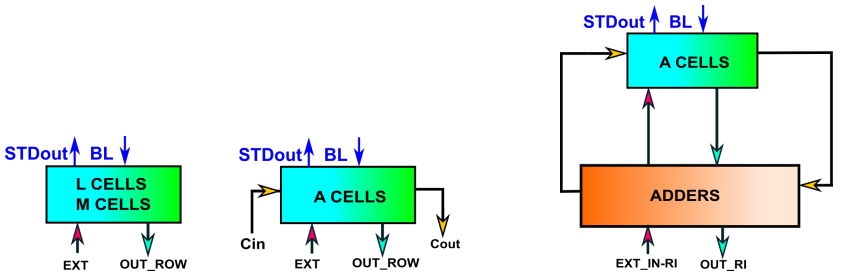


Figure 4.16: The figure shows the symbolic view of the LEGO LiM

Interface of Cells

The standard interface of cells have four ports, two of that are toward the CPU and the other two are inner to the LiM array:

- **Bit Line**(BL) and **Standard Output**(STDout) are the standard memory interface toward the CPU;
- **External**(EXT) and **ROW OUT** are the signals exchanged with the other components of the LiM array. They are totally invisible for the outside world.



(a) Standard Cells Interface (b) ACell standardized

Figure 4.17: Interfaces of standard LiM cells

ROW OUT is the only signal generated by the cell and EXT is the second operand to use for a logic operation that is performed inside the cell. However, there is one main obstacle for the standardization of the cells' interface: if standard logic gates(that are the one embedded in LCells) have three ports, both half adder(HA) and full adder(FA)(that are the ones embedded in ACells) require one and two additional ports respectively. The Idea is to use a RI after each row made up by ACells: any row of arithmetic cells must be followed by an any version of Adders. This row interface manages the additional ports required by arithmetic cells and it converts the interface in the standard one(4.17). This standardizes also the interface of arithmetic cells.

Interface of RI

Apart from the adders, that essentially are a plug in of the A cells, and the buffers, that are deeper analyzed in the next section, all the row interfaces have four ports:

two of them are toward the row(upside) and the other two are toward the memory interface(downside)(4.18). Since the two sides are the same, several of them can be concatenated. The signal coming from the memory interface(**EXT**) is propagated trough all of them until the row: it can be used in each block of the LiM row.

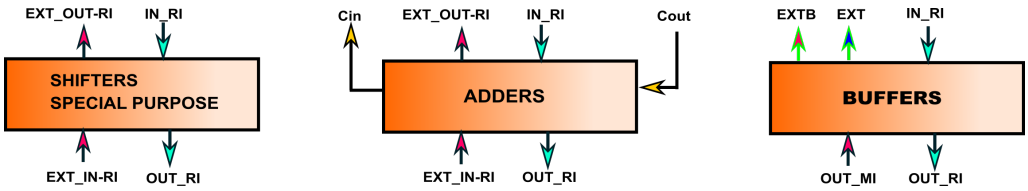


Figure 4.18: Interfaces of standard RIs

Memory Interface

The interface of a LiM row with the MI is always the same(4.19): the MI receives one signal from each LiM row and sends one signal toward each of them.

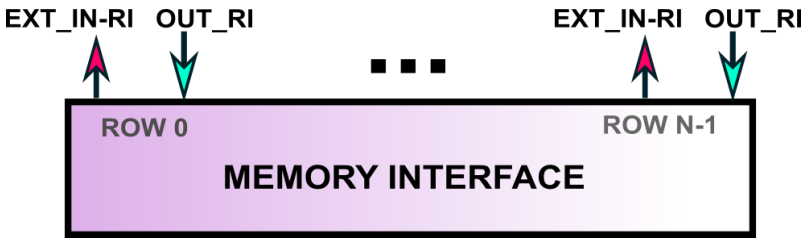


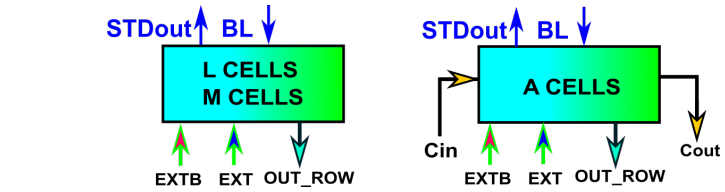
Figure 4.19: Interfaces of MI

4.2.7 How to deal with a buffer

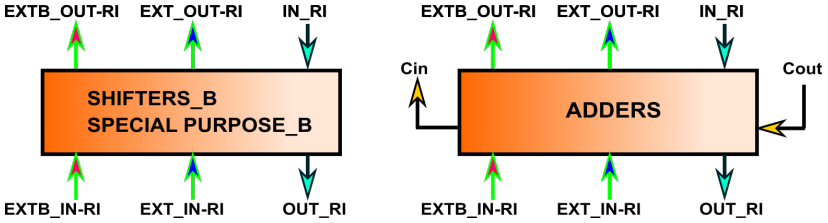
The buffers have to be handled separately. Indeed, they require one more signal to manage than the standard row interfaces: this is the output of the buffer. The output of buffer can be used as operand for the operation perform in any other blocks of the row. The idea is to create two main versions of each component, one required for the smart rows without a buffer(**Standard Version**) and the other for the ones with buffer(**Buffer Version**), in4.20.

Since the output buffer can be used in any component of the LiM row, it's propagated trough the architecture with the **extB_inRI** and **extB_outRI** signals. Then, it arrives at the input of row too(**extB**). The huge advantage of this approach

is that the additional signal is locally managed where required. In other words, the interconnection is made more intricate only in the rows with a buffer rather than made more complicated the interconnection of the whole array.



(a) LiM Cells



(b) RIs

Figure 4.20: Buffered Interfaces of Lego LiM

Chapter 5

Lego-LiM library

The previous chapter draws the shape of the Lego LiM. This chapter does a further step forward for analyzing the core of each block. Having a standard interface for each component belonging to the same family requires an equal standard inner architecture. In other words, components of the same family have also the same organization of the low level architecture. Moreover, each component is totally generic and technological independent: the user can decide both the parallelism and the technology for any Lego LiM.

5.1 LiM Cells

Each LiM cell has three main sections(5.1), no matter the family : a **storage unit** that is the traditional memory cell, a multiplexer selecting the input of the cell and a **logic unit** filtering the output of the cell.

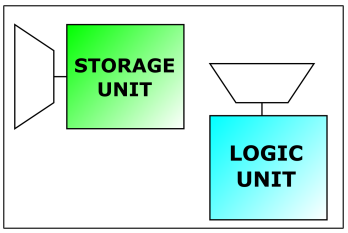


Figure 5.1: High Level view of a LiM Cell. One multiplexer selects the input of the SU and another one selects the input of the LU

Depending on both the family and the version of the component, the multiplexer and the logic unit can change. The following sections show the standard versions of the cells first, and the version for supporting buffers then.

5.1.1 Standard versions

All the Cells have the same basic behavior(5.2) :

- **storage feature**: they can store either the BL coming from the CPU or Ext , that is the signal coming from the rest of the LiM array;

- **smart feature** : the output of the cell is filtered by a logical unit before leaving the LiM cell. The second operand of the bitwise operation is the Ext signal.

Moreover, the Acells have two additional signals to manage: Cin and Cout.

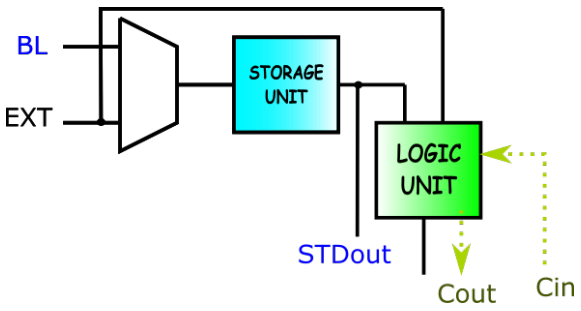
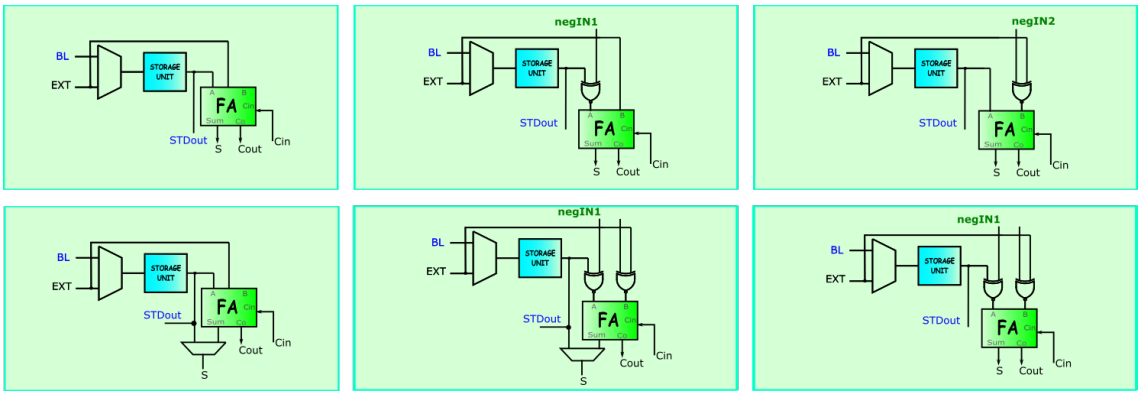
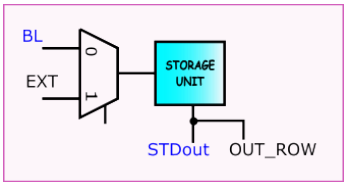


Figure 5.2: Generic view of a standard versions of the LiM cells

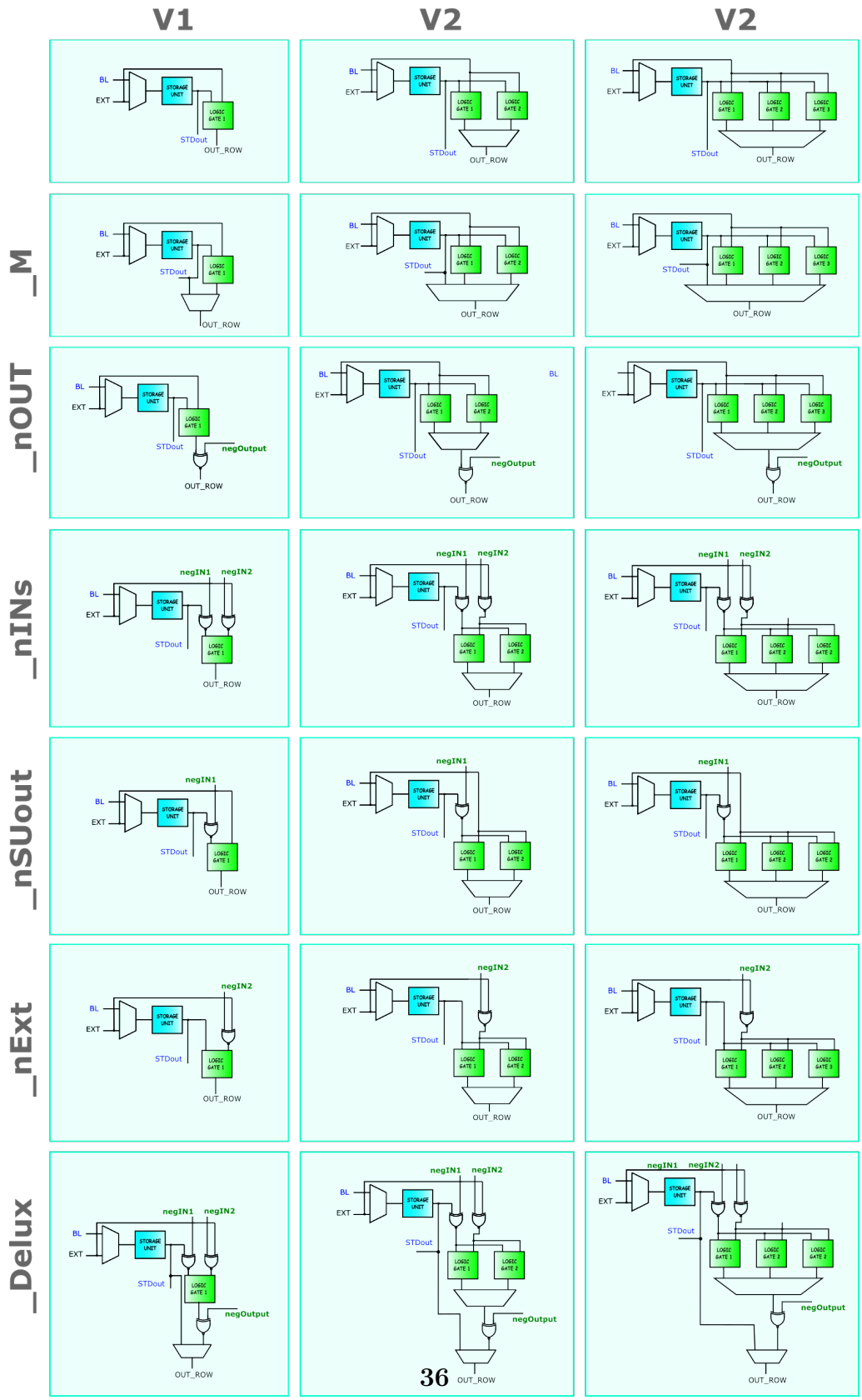
Although the interface is the same, the smart feature changes from cell to cell. 5.3 shows all the LiM cells that populate the LiM library for the moment.



(a) ACells



(b) Mcells



(c) LCells

Figure 5.3: Collection of all the LiM cells(standard version) stored in the LiMlibrary

There could be thousand of different LiM cells. However, the **design space**(5.4) was explored with a criteria that comes out clearly by looking at 5.3: starting from the easiest cells, the complexity was been progressively increased.

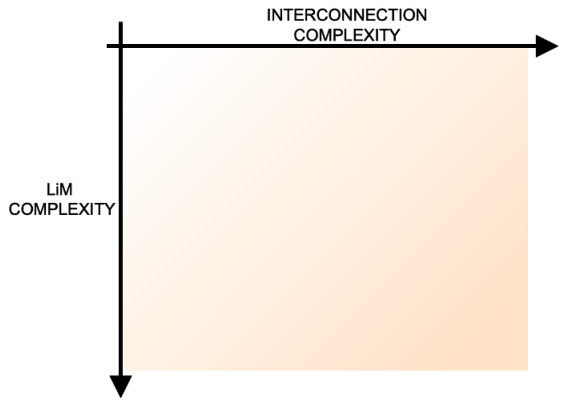
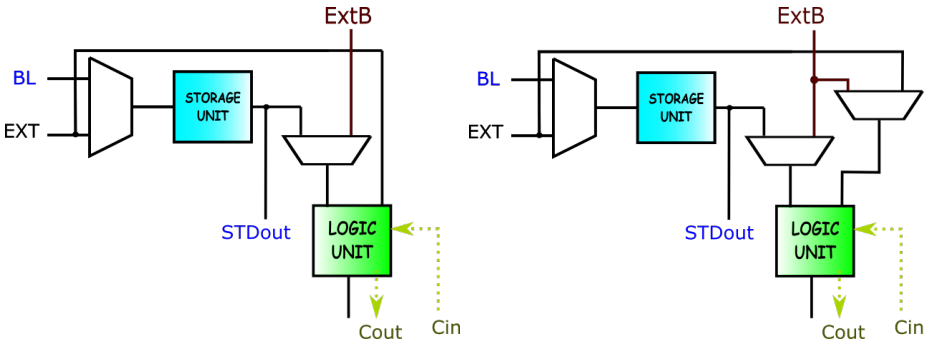


Figure 5.4: The design space develops itself in two directions: complexity of the interconnection and complexity of the logic(LiM complexity)

5.1.2 Buffer versions

The versions of LiM cells supporting buffers have the same basic behavior of standard ones but with one more degree of freedom: the output of buffer is an extra possible operand for both storage and smart operations. There are two versions for each cells shown in 5.3 with a different complexity(5.5 : **B Version** and **BB Version** . Both of them allows to storage either the BL, the Ext signal or the data coming from the buffer (extB). The only difference is how many configurations for the input signals of the logic units exists.



(a) Version B (b) Version BB

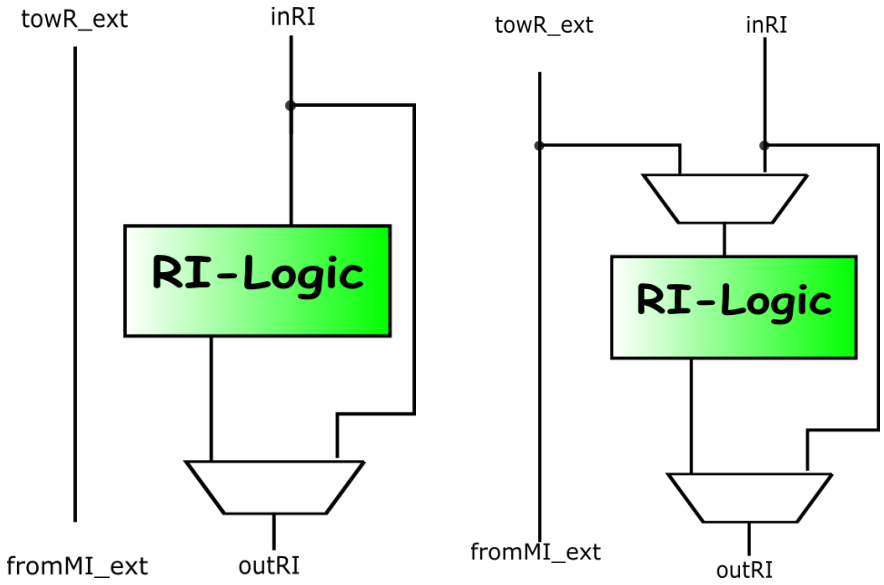
Figure 5.5: Generic view of a buffer versions of the LiM cells

5.1.3 Row interfaces

There are three main families of RIs: the buffers, the adders and the others. Since the former two must be places in a proper level of the smart row, after an ACell the first and at the end of the row the second, they require a peculiar inner architecture, that is different from the standard one of the RI. So, they are treated in separate sections.

Standard Version

The standard low level structure of the RIs has two main components(5.6): the logic implementing the function characterizing the RI and three multiplexers, one for the input of logic, another one for selecting the output toward the upper component and the last one for the output toward the lower component.

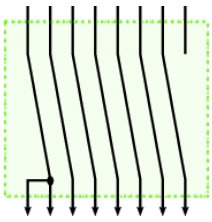


(a) Version 1D (b) Version 2D

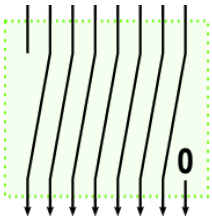
Figure 5.6: Generic view of a standard versions of the RIs

There are two versions for each RI: **Up Version** (1D) and **Up/Dwn Version** (2D). They change for how many signals can be used as operand for the operations implemented by the RI: (a) allows to use just data from an upper block and (b) also a data from a lower block of the same LiM row. The data coming from the MI(ext) is propagate trough all the RIs of a smart row until the row: it can be used as operand in every block of a row. Notice that the flow of the computation can be just upside-down. It's deeply explained in chapter 8.

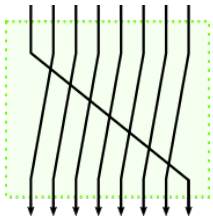
On the other hand, toward MI(downside) data are sent for computational task or for sending data to a neighbour: either the logic or a data coming from upper blocks of the smart row can be sent. 5.7 represents the LL view of standard version of all the RIs in the library. About right and left shifters, more versions could be implementable just varying how many bits can be shifted.



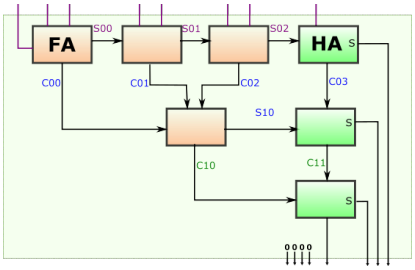
(a) Shifter: Right 1 shifter



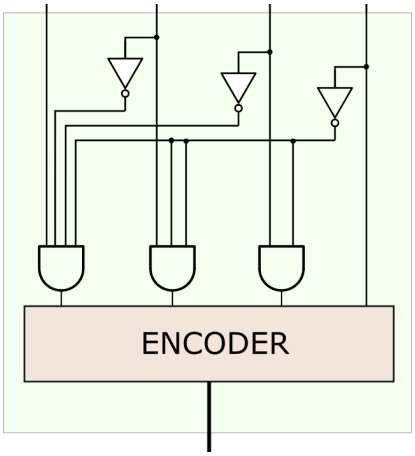
(b) Shifter: Left 1 shifter



(c) Shifter: rounder



(d) Special Purpose: 1Counter

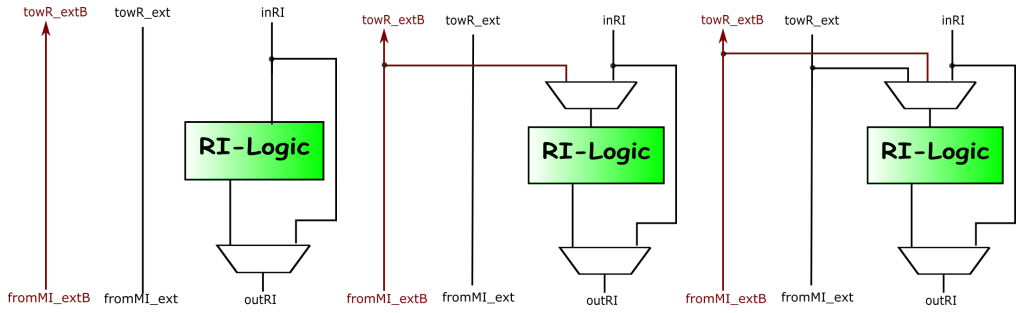


(e) Special Purpose: BSF

Figure 5.7: Logic Unit of the RIs with a word of 8 bits

Buffer Version

The versions of RIs supporting buffers have the same basic behavior of standard ones but with one more signal to manage: the output of the buffer. Each RI delivers it to the upper component, that can be either another RI or the row. This allows to use the output of the buffer in each component of the smart rows. There are three versions of buffered RIs which change for the number of operands allowed(5.8): just the data coming upward(**Up Version**)(1DB), also the data coming downward(**Up/Dwn Version**)(2DB) or also the output of the buffer (**Up/DownB Version**)(2DBB).



(a) Version 1D B (b) Version 2D B (c) Version 2D BB

Figure 5.8: Generic view of a buffered versions of the RIs

5.1.4 Adders

Adders must be placed after a row of A-cells for implementing either logic or arithmetic operations. There are two adders: the **RCA** and the **RCA&Logic**. 5.1 shows the operations that can be implemented by each adder.

Table 5.1: OPERATIONS IMPLEMENTABLE WITH THE ADDERS (A,B ARE THE TWO SELECTED INPUTS OF THE ADDER)

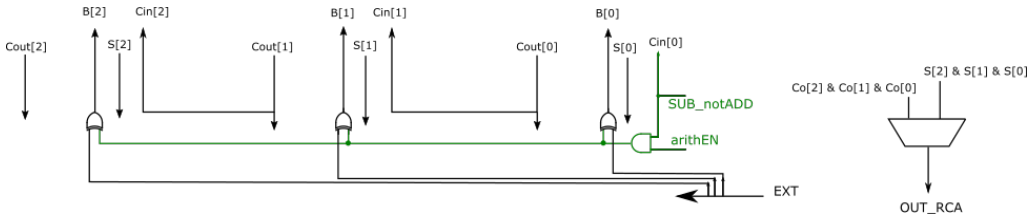
RCA	
Standard	Buffer
$A+B$ $A-B$	
	$-A+B$ $-A-B$

RCA&Logic	
Standard	Buffer
$AorB$ $AandB$ $A \oplus B$ $\overline{A \oplus B}$ $AorB$ $A \oplus \overline{B}$	
$A+B$ $A-B$	
	$-A+B$ $-A-B$
	$\overline{A}orB$ $\overline{\overline{A} \oplus B}$ $\overline{A}or\overline{B}$ $\overline{\overline{A} \oplus \overline{B}}$

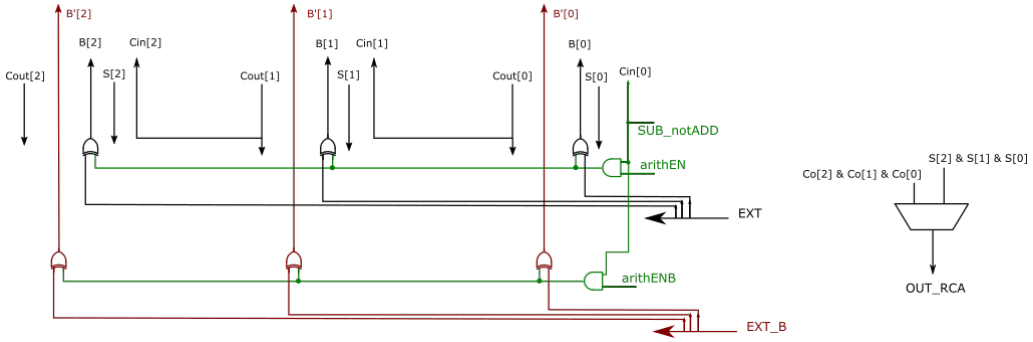
About the **standard versions**, the two data involved in the operation are the content of row and the external signal. Both RCA and RCA&Logic can implement either addition or subtraction between them like a ripple carry adder. The data send toward the RI can be either the carries or the sum. Moreover, the RCA&Logic

can perform complex logical operations. It's implemented by interrupting the propagation of the carries: the control signal SUB_notADD replaces them.

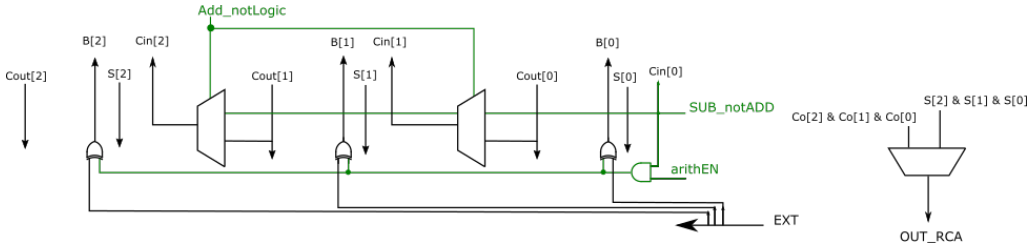
About the **buffered versions**, the second operand can be either the buffer's content or the external signal. Moreover, with buffered cells, the content of the row can be left out from the operation and the operations colored in cyan can be performed between the external signal and the content of the buffer. Chapter 8 shows the configuration required for each operation too.
5.9 shows the low level architecture for both standard and buffer versions of the adders.



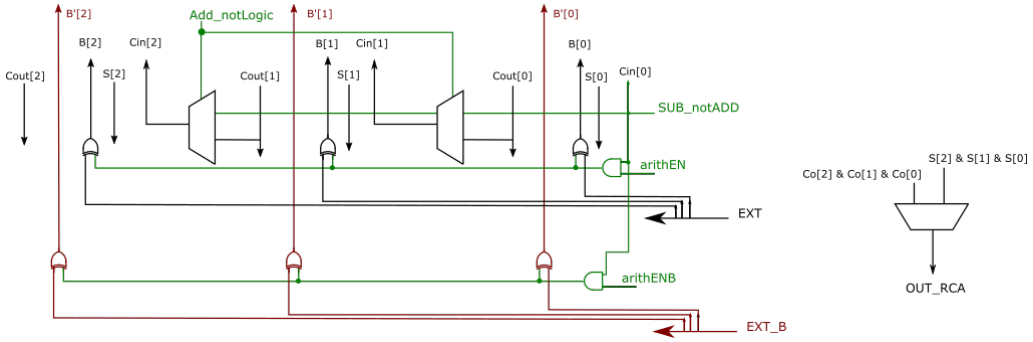
(a) RCA standard



(b) RCA buffered



(c) RCA&Logic standard

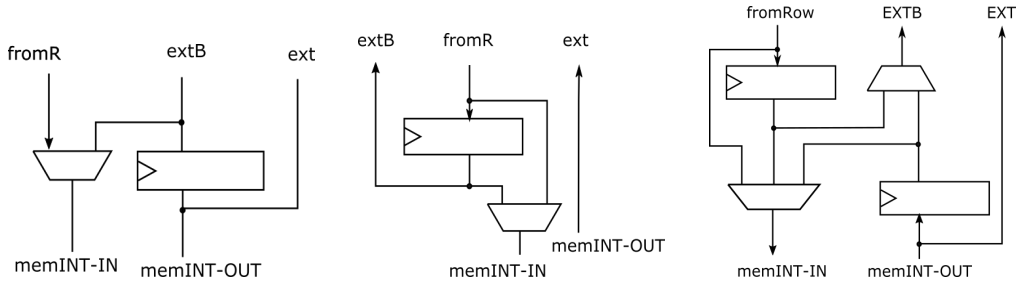


(d) RCA&Logic buffered

Figure 5.9: architecture of adders for a word 3 bits wide

5.1.5 Buffers

Buffers are the only RI with storing capabilities: they can stores either input or output of a LiM row to which they belong. So, the main component of a buffer is a register: it's fed by the input of its LiM row for the input buffer and with the output of its LiM row for the output buffer(5.10).



(a) Input Buffer (b) Output Buffer (c) I-O Buffer

Figure 5.10: BUFFERS' architectures

The buffer sends toward the row(upside) both the output of the register and the data coming from the MI toward the row. For the Input-Output buffer, either the output of the input register or the one of the output register can be sent. On the other side, since the interface provides just one output toward MI, only one between the output of the register and the output of the RI can be read by the MI.

5.1.6 Memory interfaces

The memory interface has a quite hard task: it interconnects all the rows together. Since it must be placed in every LiM architecture, the best implementation must be designed. The goal of the study done is to find the lightest MI being able to **fully connect** the LiM array, where lightest means the one with the lowest power and area impact and fully connect means that each row can communicate with any row. Three solutions were explored: **Standard MI**, **Pipelined MI** and **Cache-like MI**. Spoiler: the latter one is the best.

5.1.7 Cache-like MI

The main blocks of the **Cache-like MI** are the **cache registers** and the **row multiplexers**(5.11). The **row multiplexers** are able to manage the interfaces of all the rows belonging to the smart section in parallel:

- all the smart rows can be fed simultaneously either by the output of their neighbors or by the output of the caches;

- all the standard near rows can be fed by one of the two surrounding smart rows.

The **cache registers** feeds all the smart rows and they are fed by the standard section of the array. The cache can read just the standard section because a smart row needs it just for retrieving shared operands stored in the standard section(4.2.4). Just like a standard Cache, if the shared operand required is in one of the cache registers(**hit**), the communication happens without delay. On the other hand, if the row it's not in the cache, a **miss** occurs: the data has to be retrieved from the LiM array and stored in the associated cache line, that is one of the two registers. Using the cache terminology, the mapping of the LiM array on the cache is a *direct associative mapping*, where each register is dedicated to an half of the standard section of a LiM array, that is the **cachable memory**. No write strategy is required since there aren't problems of cache coherence(section 7.6).

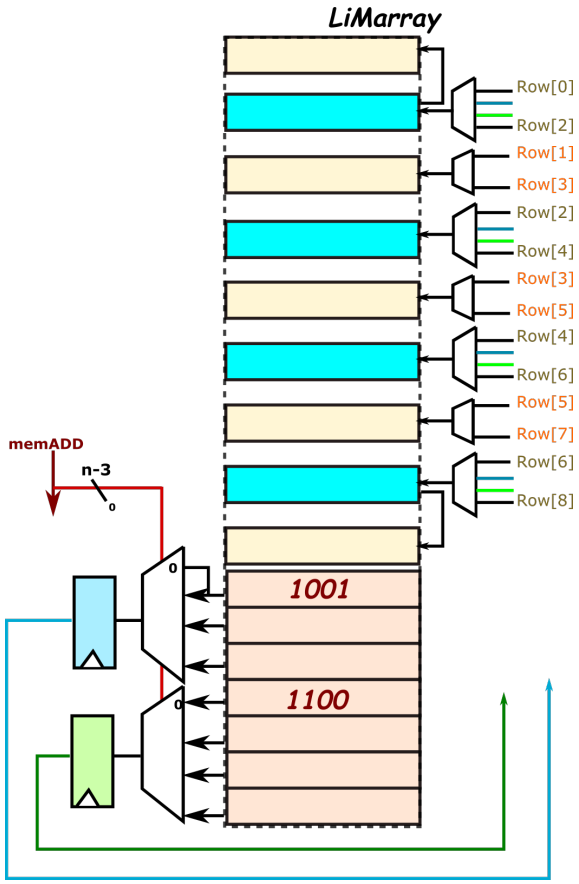


Figure 5.11: MI for a LiM array(V1) with 7 rows in the standard section

Design choice

Each implementation fully connects all the rows.
The **Standard MI**(5.12) implements the interconnections in the most intuitive way: a multiplexer wide as the memory size is placed at the input of each row. Each of them is fed by the whole memory array. The price for a fully connect array is the huge number of very large multiplexers required.

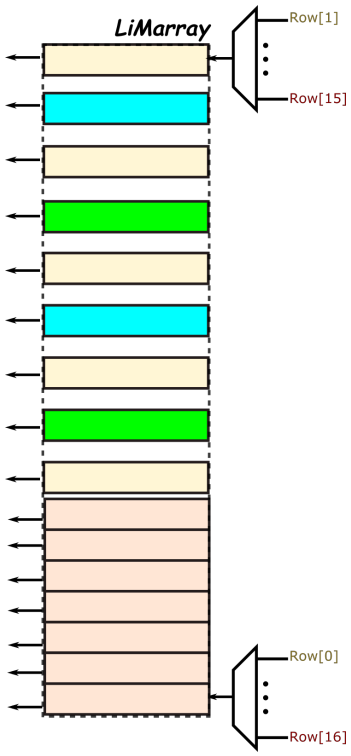


Figure 5.12: The standard MI for an array(V2) with 16 rows

The **Pipelined MI**(5.13) solves the problem of the number of multiplexers with just one pipelined multiplexer handling all the interconnections: there is just one multiplexer that is as wide as one of the N multiplexers of the previous version. It's an n-ways multiplexer implemented with layers of 2-ways multiplexers where each two consecutive layers are interrupted by a registers' layer. So, there is a reduction of the number of multiplexer but an increasing number of registers. Moreover, although each row can communicate with any other row, just one interconnection for clock cycle can happen.

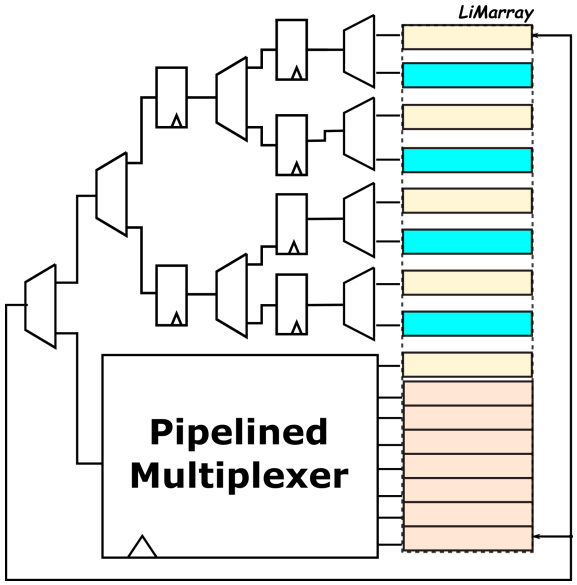


Figure 5.13: The pipelined MI for an array(V1) with 16 rows

The last one implementation solves both the previous problems: the **Cache-like MI** it has smaller multiplexer and just two registers(5.14).

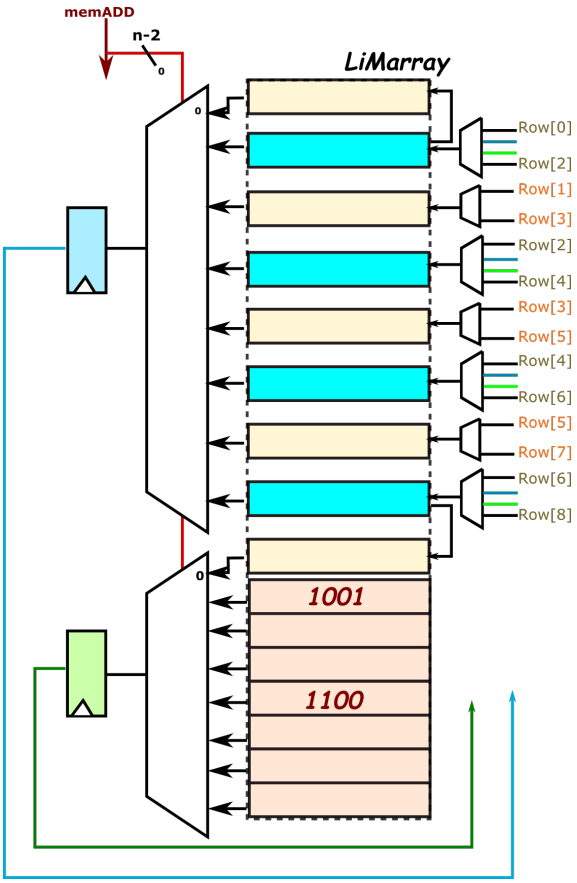


Figure 5.14: The cache-like MI for an array(V1) with 16 rows

5.2 summarizes the **behavioral comparison** between the three MIs, where the focus is on four parameters: the complexity of multiplexers(MuxC), the overhead of registers(OReg), the latency of 1 movement(L) and the parallelizability(Par), that means how many movements can be simultaneously performed. The complexity of multiplexer is defied as $\sum_i MUX[i] \cdot N_input[i]$

Table 5.2: COMPARISON BETWEEN MIS

MI	MuxC	OReg	L	Par
standard	$N \cdot (N-1)$	0	1	N
pipelined	$1 \cdot (N-1)$	$log_2(N)$	$log_2(N)$	1
cachelike	$N \cdot 5$	2	1(2)	N

The table underlines how the **Cache-like MI** combines the strong points of the

other two versions:

1. N movements can be done in parallel without the overhead due to the huge multiplexers of the STD architecture, where N is the number of rows;
2. the number of registers and the size of row multiplexers are independent from the number of the rows;
3. the latency of one movements is just one cycle, or two cycles in case of miss.

The **performance comparison** was been tested by stimulating the three implementations with the same algorithm. The focus of the analysis is on the critical path, the power consumption and the area. These parameters have been evaluated to vary of the memory depth: it sweeps from 8 rows to 1024 rows. On the other hand, the word width is kept fixed at 16 bits. The evaluations were done with the Back-annotation approach(Appendix A) before the place and route step.

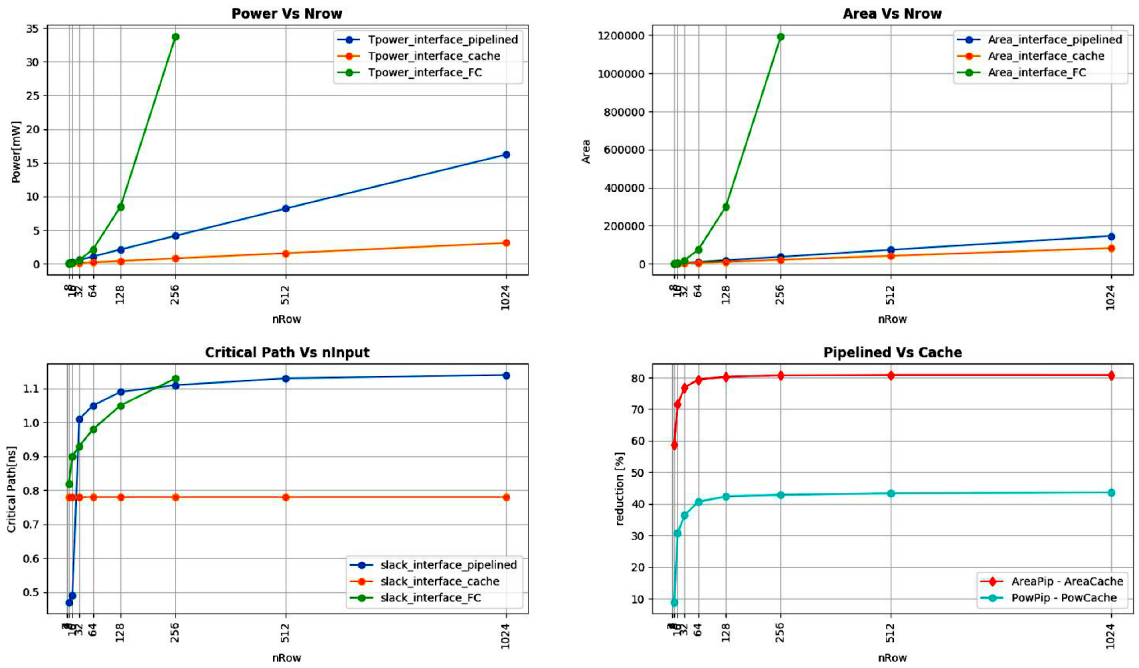


Figure 5.15: Comparison between the different implementations of a MI

5.15 summarizes the results obtained. One consideration is clear: the standard solution is unusable due to an almost exponentially growing of both power consumption and area with the number of rows. On the other hand, the Cache-like

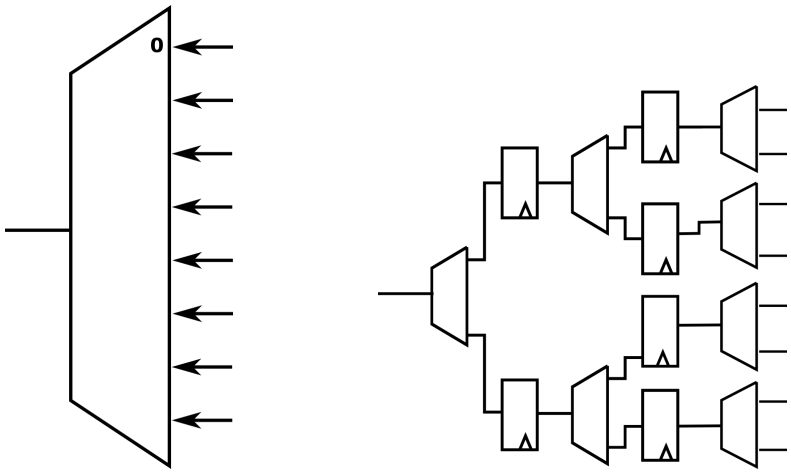
MI is just slightly affected by the number of rows: in other words, the impact of a bigger memory on its performance is very light. That's a huge advantage of this approach. The fourth plots in 5.15 compares it with the second best implementation: the Cache-like approach allows to have a reduction saturated at almost 40% on the power consumption and at 80% on the area overhead.

5.1.8 Definitive Version

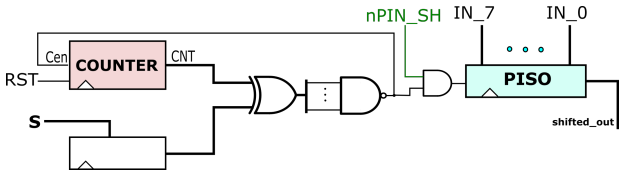
Given that the best solution is the Cache-Like one, 5.14 is not the final version. About the row multiplexers, two simplifications have to be done. First, since the standard rows can be just used as destination of LiM operations, they don't have to receive also the output of the cache registers. Second, multiplexers are useless at the input of standard far rows: they aren't involved in LiM operations. They can just be read for retrieving shared operands trough the cache, but they can't be written. About the cache, the correct use of the array is hardwired forced due to hazard and cache coherence problems. These issues are analyzed in the chapter 7. Basically, as explained in 4.2.4, a smart rows needs to reads for retrieving operands either one of the neighbors or the standard section. Since the connection with neighbors is hardwired, the cache is just need for retrieving the shared operands in the standard section. So, the cache needs to read just the standard section: this is the correct behavior of the cache that is not hardwired in 5.14. Moreover, since just one cache register is written at a time, notice that the same configuration signal control both of the cache multiplexer. The width of the control signal for the cache multiplexer depends on the size of standard far section, that is the reading space of the cache.

Multiplexer: finding different implementations

The key element of all the MIs is the multiplexer. So, the first step for the analysis of the MIs was the exploration of the design space of the multiplexer. Three solutions were explored: standard multiplexer(5.16.a), pipelined multiplexer(5.16.b) and sequential multiplexer(5.16.c).



(a) standard multiplexer (b) pipelined multiplexer



(c) sequential multiplexer

Figure 5.16: Three implementations of a multiplexer

The **pipeliend multipexer** is an N-ways multiplexer where the stages of a 2-way multiplexers are cut with registers' stages. The **sequential multiplexer** has three main elements: a counter, a register and a Parallel-IN-Parallel-OUT(PISO) register. The fist step is to store the selection signal into the register and all the inputs of the multiplexer into the PISO register. Then, the counter starts to count until it reaches the value of the selected signal stored in the register. For each counting the PISO shifts out one input.

The **performance** are evaluated with the Back-annotation approach(Appendix

A) before the place and route step: the number of ways sweeps from 2 to 1024 and the data width is kept fixed at 16 bits. The same algorithm stimulates all the entities: it leads the farthest Input to Output. 5.17 summarizes the results.

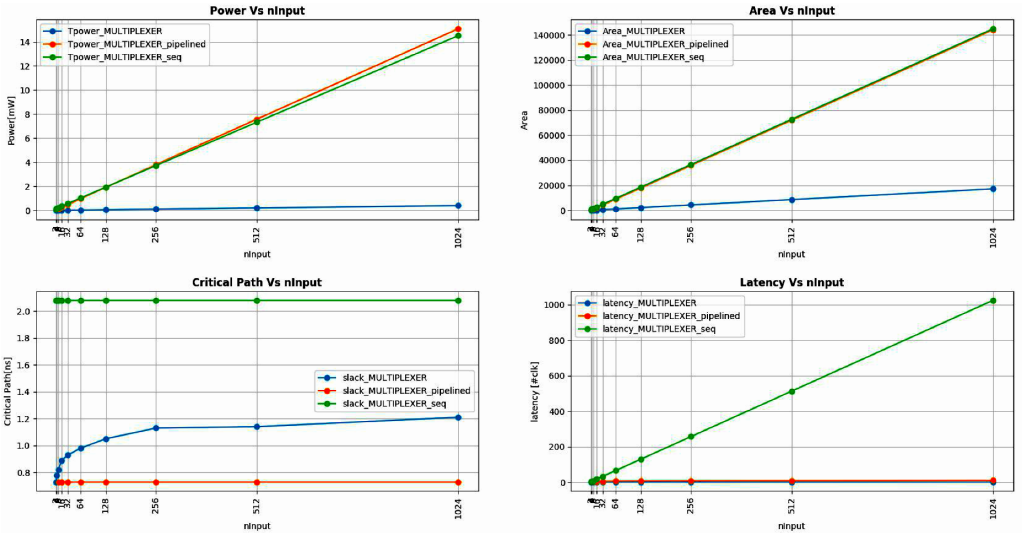


Figure 5.17: Results of the comparison between the three multiplexers

The sequential multiplexer is unusable: it's the slowest and the most expensive both for area and power consumption. Moreover, the latency for getting out the selected input linearly increases with the number of ways: indeed, the number of shifts required linearly increases. Here the bottleneck is of course the counter. About the pipelined version, it has both strong and weak points. The bottleneck is the register's overhead, that means an heavier power consumption and a bigger latency(in terms of clock cycle numbers) than the standard version. On the other hand, first, the latency is one clock if the pipe if fully fed, second, the critical path is fixed and it's the one of a two way multiplexer. Third, just one pipe multiplexer can manage the interconnection between several input - output ports.

Chapter 6

The front-end

6.1 Overview

The front-end(uCU) is implemented with a **micro-programmed** Control Unit. The huge advantages of this approach are the flexibility and accessibility. Flexibility means that new instructions can be easily implemented just increasing the size of the micro instruction memory, the **micro ROM**(uROM). Moreover, the micro-programming is very user-friendly: it happens just writing 0 and 1 in the uROM. The instruction written in this uROM are called **micro-Instructions**(uInstructions). A **micro-program**(uprogram) is a bunch of uInstructions.

The implemented control unit manage **explicit addressing** for the evolution from a uInstruction to the next one. It avoids the additional hardware required for an implicit addressing. Explicit addressing means that the address of the next sequential uInstruction is written in the uInstruction itself. About **branches**, just 2 ways branches are allowed: the next uInstruction can be the sequential one or another one, that is the branch one. It's managed with a kind of explicit addressing: the address of the branch and the one of the sequential uInstruction change just for the lowest significant bit. It reduces a lot the size of uROM required for managing the explicit addressing. Moreover, the uprogram can also have calls to function : they are called **micro-calls**(ucalls). They are managed almost as the branch instructions: the call address and the return addresses can change just for the lowest significant bit. The call address ends with logic 0.

6.2 Architecture

The building blocks of the uCU are(6.1):

- **LiM Queue**: it stores the queue of uPrograms to execute, that are the addresses of the first uInstructions of each uProgram;
- **micro address register**(uAR): it's the next uProgram to execute;
- **micro program counter**(uPC): it stores the address of the uInstruction in execution;
- **uROM** : it's the core of the uCU. It acts both as memory of the configuration signals for the back-end and as the next uInstruction generator;
- **sequencer** (uSeq) : it selects the next uInstruction to execute leading by th output of uROM ;
- **micro return address register**(uRAR): it store the address where to return after the execution of the function called.

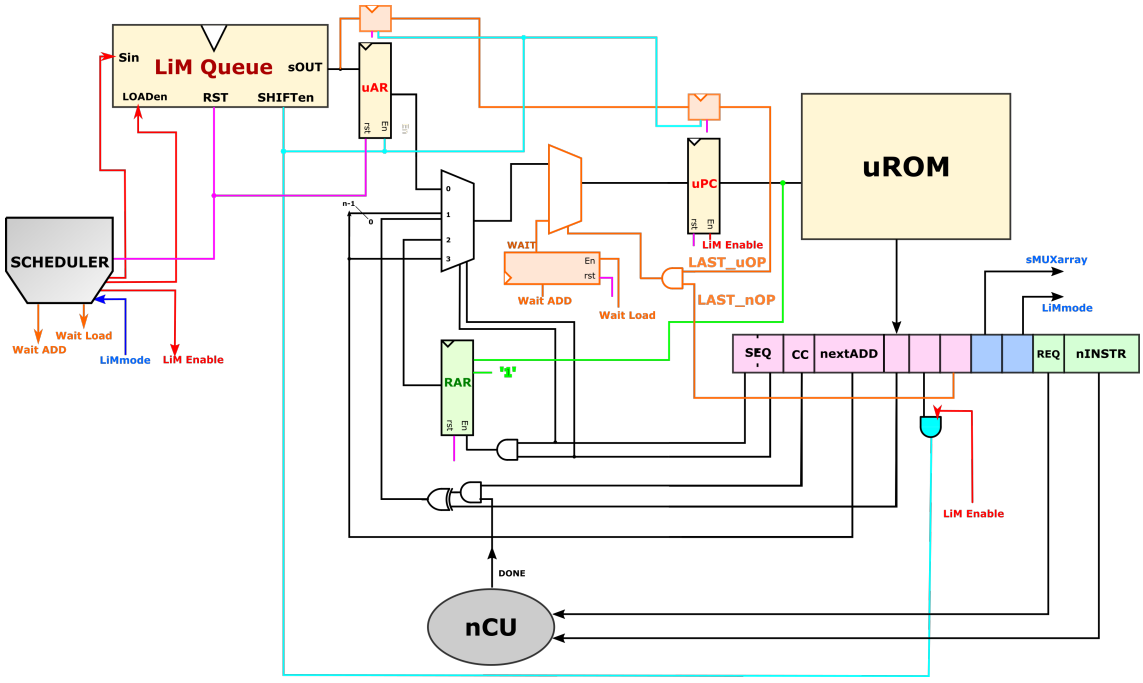


Figure 6.1: Architecture of the front-end

The **uAR** stores the previous output of the Queue, that is the next uProgram that has to be executed. When a new program has to be executed, the **uPC** reads

the uAR : the output of the uPC addresses the first instruction of the uProgram just started. So, the output of the **uROM** becomes the first uInstruction of this uProgram. The **uInstruction** has three section: the signals for configuring the uSeq(the pink ones in 6.1), the state signals(the blue ones in 6.1) and the configuration signals for the back-end(the green ones in 6.1). The instructions sent to the back-end(nCU) are called **nInstruction**. Each of them takes three rising edge for being executed.

The signals for the **uSeq** leads it in the generation of the next input of the uPC, that is the next uInstruction. It can be the next instruction of the uProgram in execution, the return address or the first instruction of the next uProgram to run, that is the output of the uAR. Actually, the next instruction might or might not be the branch target. When all the uInstruction of a uProgram has been executed, the uPC is updated with the uAR: the programmer of the uROM has to proper write the last uInstruction of each uProgram in order to alert the uSeq that the output of the uROM is the last instruction of a uProgram. The **state signals** are the two signals for the handshake with the scheduler and for managing the arbiter of the array.

The orange portion of the architecture in 6.1 is need to end the LiM execution. When all the uProgram stored in the queue has been executed, the uCU asserts the LiM mode signal and the sMUXarray for giving back the control of the array to the CPU. However, how could the uCU know which is the last uProgram? In other words, how could the uCU know when the queue is empty? It does care because the instruction following the last uInstruction of the last uPorgram must be the **Wait Instruction**. It allows to complete the last nInstruction before getting back the control of the array to the CPU. So, the programmer of the LiM queue(scheduler programmer), that is the sequencer, and the programmer of the uROW(uProgrammer) must talk. The interface between them is done such that they don't have to talk a priori. It means that the uProgrammer writes the uROM without being aware about how they will be executed, that is in which order. They communicate run-time with two signals: the scheduler programmer asserts a flag with the last uProgram to execute and the uProgrammer asserts a flag with the last uInstruction of each uProgramm. When both of them are true, the uInstruction in execution is the last one: the next one has to be the Wait instruction. The address of this instruction is stored in a register in the uSequencer. The scheduler must write it

before starting.
Notice from 6.1 that the flag of a uProgram moves forward with it in the uCU and the **reset** signals(violet one) is shared by all the registers.

6.3 Timing

7.2 shows the timing describing a complete life cycle of a uCU. After the reset and the writing of the WAIT address' register, the uPC addresses the **idle instruction**: in this state both the CUs and the LiM array doesn't work and the CPU controls the memory. Once the scheduler enables the LiM unit, the output of the queue is written in the uIR and then it's shifted. It happens for two rising edges, that is until the uPC is updated with the first output of the queue. Two rising edge after the enabling of the LiM execution, the uPC is updated with the first uInstruction of the first uProgram, the uAR with the second uProgram to exectued and the output of the queue becomes the third uProgram to execute. So, the two starting cycles make the uAR one step further than the uPC and the queue two steps further.
The flag of a uProgram moves forward with it in the uCU.

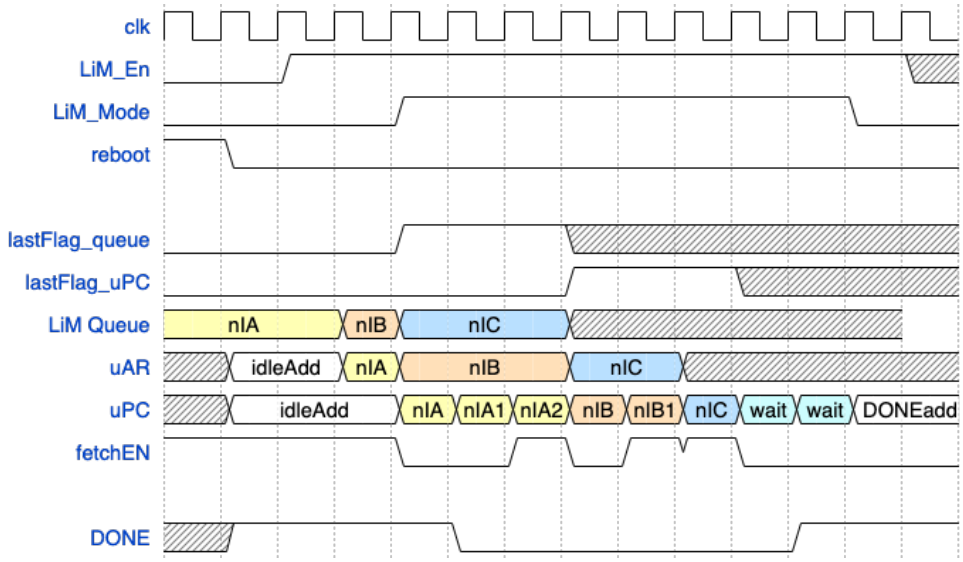


Figure 6.2: Example of a complete LiM execution from the uCU point of view

Notice that the uAR and Queue are updated at the same time when the uPC

reads the uAR, that is at the last nI of a uProgram. Instead, the uPC is always updated when the LiM unit is activated. Indeed, the uCU fetches one uInstruction for clock cycle. When the last uInstruction is written into the uPC, the uSeq selects the WAIT address as the next uInstruction to run. This uInstruction waits for the DONE signal before jumping to the **DONE** instruction. It means that the branch is not taken until the nCU completes the execution of the last nInstruction. The difference between the DONE and the IDLE instruction is the fetch enable signal: in the former one the uAR is not fetched.

Chapter 7

The back-end

7.1 Overview

The back-end, that is the nCU, manages the execution of LiM operations, that are SIMD. It's a pipelined CU with two pipe stages. It translates the instructions received by the front-end, **nano Instructions**(nInstructions), in configuration signals for the LiM array. It's made up by two building blocks(7.1): the **Decoder Unit**(Dec unit) and the **Memory Interface Unit**(MIU). The first unit parses the nInstructions received for generating configuration signals for the LiM cells and the Row Interfaces of the array. On the other hand, the MIU reads the source field of the nInstruction for managing the MI. Since each block controls a different portion of the LiM array, they can work in parallel. The big advantage is that also if a miss happens in the MI, the execution will not be delayed.

The nCU generates five or six configuration signals depending on the version of the array: V2 required an additional signal due to the two types of smart row. As shown in figure 7.1 the rows of the same type share the configuration signal. However, notice that two different signals are required for the first and the last row of the smart section. The chapter 8 explains why it happens. Notice that additional logic outside the units is required for the read operation of standard far rows: they can be read just for writing cache on miss. So, the read enable signals (Ren) of that is the miss signal generated by the MIU. Since the word line(WL) has to be enabled for accessing to a row, it becomes high as the Ren is asserted.
The **Miss Enable** signal activates the MIU on a operation required the cache. Moreover the decoders send to the cache also the configuration for the row multiplexers of the MI. Indeed,if the cache is not involved, the input of the of a smart row depends on the operand field that the MIU doesn't see.
The reset signal (**Reboot**) resets both all the registers of the back-end and the

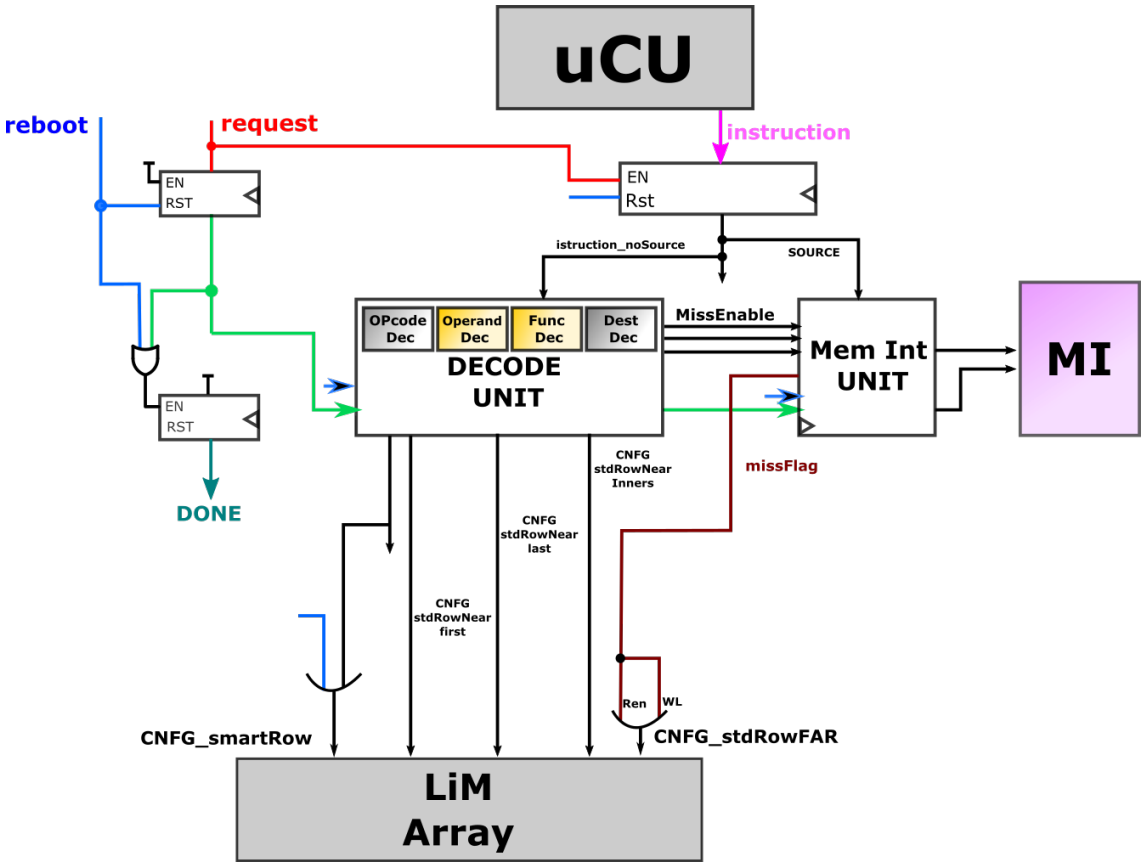


Figure 7.1: High Level Overview of the back-end for V1

registers of the LiM array, that are the cache registers and the buffers.
The **Request** is propagated in the CU for enabling the registers storing the configuration signals entering in the execution stage.
Notice that the generation of the **Done** signal is hardwired: the request signal lets it down. After one clock, the delayed request pulls it up again. Actually, also the reboot signal can make it high.

7.2 Execution of a nano Instruction

As explained in the chapter 8, each nInstruction has 5 fields: OpCode, Operand, Results, source address and function. If a row that is not a neighbour has to be read, the source address indicates its address. It will accessed trough a cache register.

The execution of each operation takes two cycles(7.2): **Decode & Miss** and **Execution**.

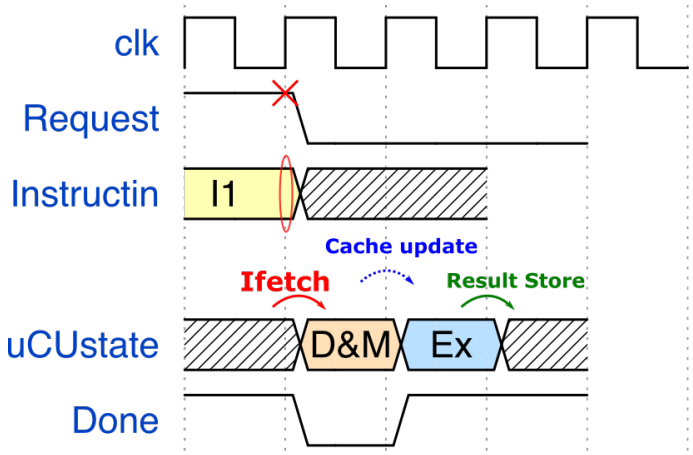


Figure 7.2: Execution of one nInstruction

The **nInstruction register** fetches a new nInstruction as soon as the request signal is high at the rising edge of the clock. This nInstruction is splitted in two subsections: the MIU reads the source address field and the decoder unit receives the rest of the nInstruction. During the first cycle after the fetch, the MIU checks for a miss and the decoder unit decodes the nInstruction. If the instruction involves the cache, the decoder enables the MIU(MissEnable=1): if there is a miss, the MIU proper updates the content of the cache registers during the next rising edge. Otherwise, the signals generated by the MIU can't affect the cache registers. Moreover, the MIU generates the signal for the row multiplexers of the MI. During, the next clock cycle the configuration signals generated are sent to the array. If it's required by the nInstruction, either the buffers or the smart rows are updated with the result of the operation performed.
The pipelined structure allows to manage two instruction in parallel(7.3). The **DONE** signal goes down at the decode stage of the first instruction and it returns high at the execution cycle of the last nInstruction.

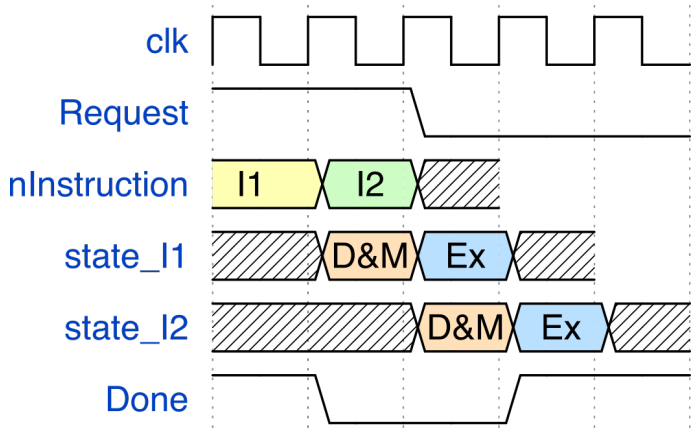


Figure 7.3: Execution of two nInstructions in parallel

7.3 Decoder Unit

The decoder unit receives four of the fields of a nInstruction. There is a decoder dedicated to each field(7.4). The huge advantage of this approach is the flexibility. It means that two decoders are independent from the array: the result decoder and the operand decoder. Actually, they are the same for all the array designed starting from the same template. On the other hand, the other two decoders strongly depends on the LiM array, since they describe its smart features.

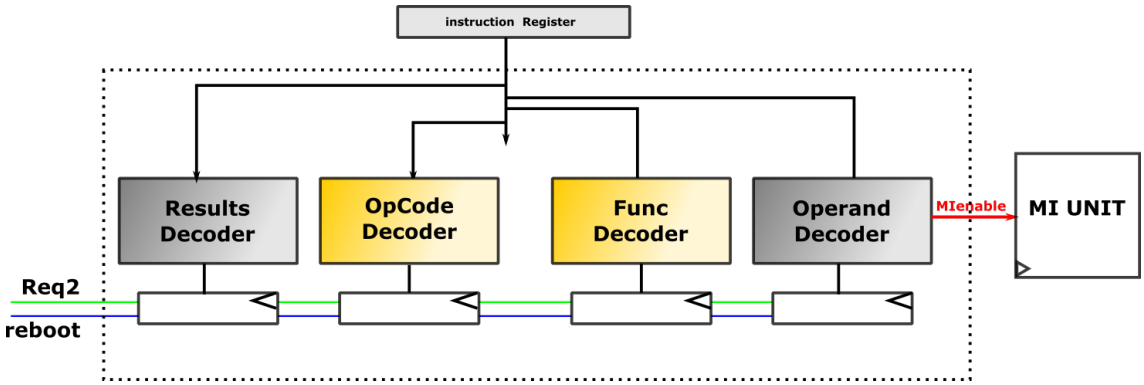


Figure 7.4: High level view of the Decoder Unit

7.3.1 Generation of Configuration signals

Overview

The **Operand Decoder** receives information about where the smart rows have to retrieve them operand/s. So, it controls the configuration of the MI for intra-row movements and the multiplexers of both cell's logic unit and row interfaces' logic for selecting the operands. About the latter signal, it's the same for all the row interfaces; then, the op Code Decoder decides which ones have to be activated. Moreover, it activates the MIU (miss flag ='1') if the cache needs to be read.

The **Op Code Decoder** is aware about which units and in which order have to be activated for performing the operation. It configures the RIs and selects the input of cells' storage unit. It generates a flag for each RI that is asserted: it's high if the RI is the highest logic unit of the LiM row involved in the operation($flagACTIVE\text{-}MODE\_RI_i$). Moreover, for V2, it generates a flag for each one of the two smart row's type($flag\ SR_i$): it is asserted if the operation involves the smart row.

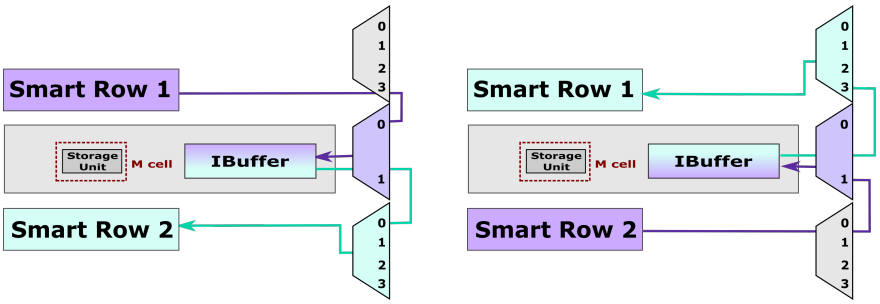
The **Result Decoder** receives information about where to store the result: it can enable the writing of either output buffers or smart rows.

The **Function Decoder** just configures the more complex RIs, as a RCA, and LiM cells. Indeed, all the RIs that don't have to be configured have the same configuration signal's width. The same for the cells.

More complex cases

Actually, there are two situations where the decoders have to collaborate for generating the signals: they're shown in 7.6 and in 7.7 . The fist situation is about the configuration of the row multiplexers of the MI for the V2. The input buffers of standard near rows are used for multi-row SIMD operations(7.5) as a checkpoint: the smart row 1 has to read or write the lower input buffer and the smart row 2 the upper one.

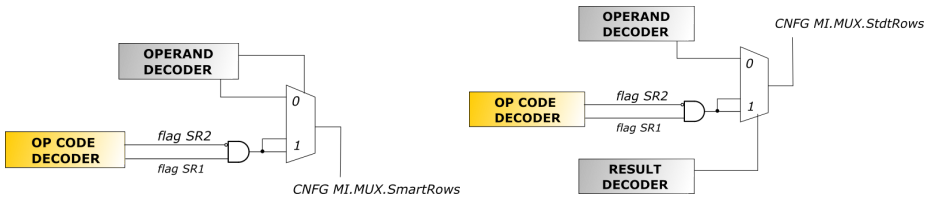
So, when the checkpoint is involved in the operation, the configuration of row multiplexers depends on which smart row is activated. If the operand decoder alerts that the input buffer of standard near rows has to be used as operand(7.6.a.), the row multiplexers of the smart rows need to read the lower input buffer if the



(a) SR1−− >SR2 (b) SR2−− >SR1

Figure 7.5: Extract of the smart section of a LiM array V2 showing the two types of multi-row operation. Violet indicates the components used in the first cycle and cyan the ones used in the second cycle.

smart row 1 performs the operation(7.5.b) and the upper one if the smart row 2 is activated(7.5.a). On the other hand, if the result decoder alerts that the partial result has to be stored in the checkpoint(7.6.b), the multiplexers of standard row near need to read the upper smart row if the smart row 1 performs the operation(7.5.a) and the lower one otherwise(7.5.b).



(a) (b)

Figure 7.6: Configuration of MI's row mutliplexers

The second situation involves the configuration of the RIs(7.7). The idea is that the opCode decoder configures the RI such that they can perform the operation required. Then, the operand decoder configures the input of the only one RI receiving that. It means that if the operation require the concatenation of more row interfaces, the operand decoder controls the input of the highest RI involved(RI-1 in 7.9.), that is the one receiving the operands.

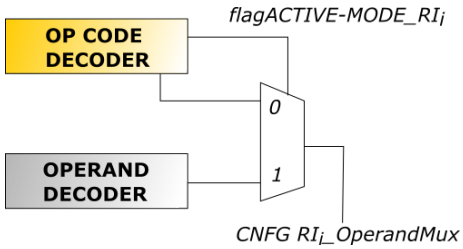
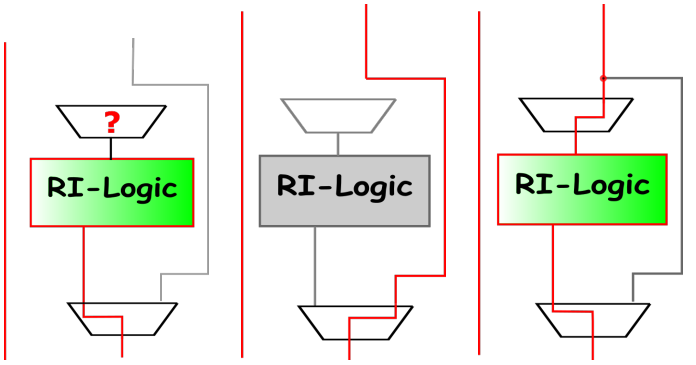


Figure 7.7: Configuration signal of RIs

Configurations of a RI

Depending on the operation, the RI can be configured in three ways: active mode or concatenate mode if it's involved in the operation, transparent mode if it's not(7.8). RIs not involved in the operation are configured in **transparent mode** : it's taken apart, it just connects the upper with the lower unit. On the other hand, the RI involved in the operation has to be configured in the **active mode** if it's the highest unit among the ones activated. In this mode, it sends the logic result toward the MI(for storing it in the buffer or for sending it out). The input of the RI depends on the operand required. On the other hand, if the RI is involved in the operation but it's not the highest unit activated, it has to be configured in the **concatenate mode**: the operation is performed on the result coming from the upper RI and the result is send toward the MI. The ext signal coming from lower RI is propagated upward for having the possibility to use it in the activated RI as an operand.



(a) active mode (b) transparent mode (c) concatenate mode

Figure 7.8: Configuration of RIs

An example is shown in 7.9.

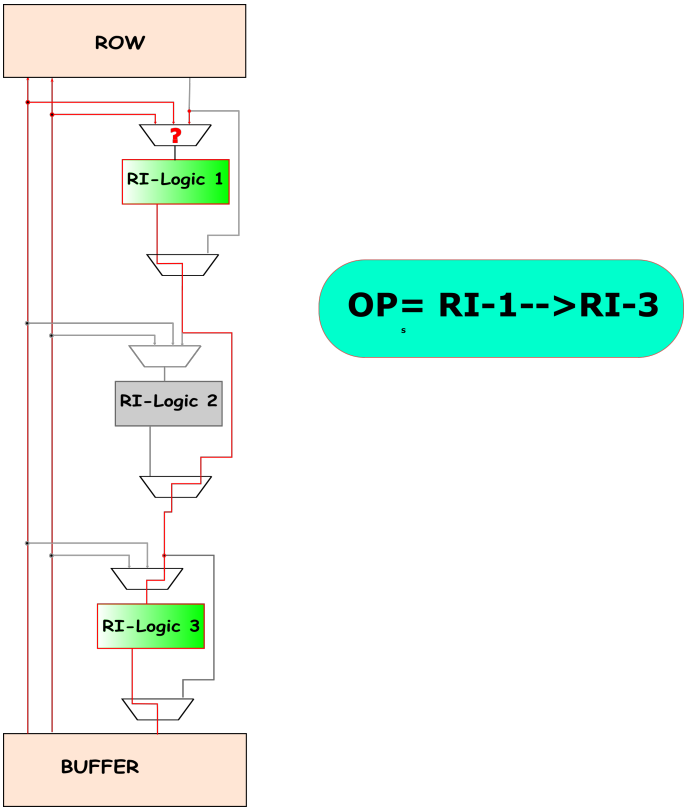


Figure 7.9: Example of the configuration of a smart row

7.3.2 Reading and Writing rows

The decoders have also to generate signals for reading and writing the rows. A distinction has to be done: smart rows, standard row far and standard row near.

About the generation of **read enable signals**, it's completely managed by the operands decoder: it knows where the operand has to be retrieved. Actually, it's a little bit more complicated for V2. Indeed, only the activated smart row type is read. It means that a row of a smart row is read only if the decoder alerts the smart rows is involved in the operation($flag\ SR_i = 1$).

About the generation of **write signals**, it depends on the LiM row. The smart rows can be written just on a store operation. It happens if the opCode decoder alerts that there is a store operation(storeAc_xx $=' 1'$), and the operand decoder

double checks only the write enable of the row that has to be written. So, there are two security steps to go further in order to enable the writing of a row. About V2, the store operation can be done either for one or for both the smart row's types. Actually, a further distinction has to be done among the standard near rows: there is a signal for the inner rows, one for the first one and another one for the last one. Indeed, when the upper rows have to be written, the last row does not have to be written; on the other hand, when the lower rows have to be written, the first row does not have to be written(7.10).

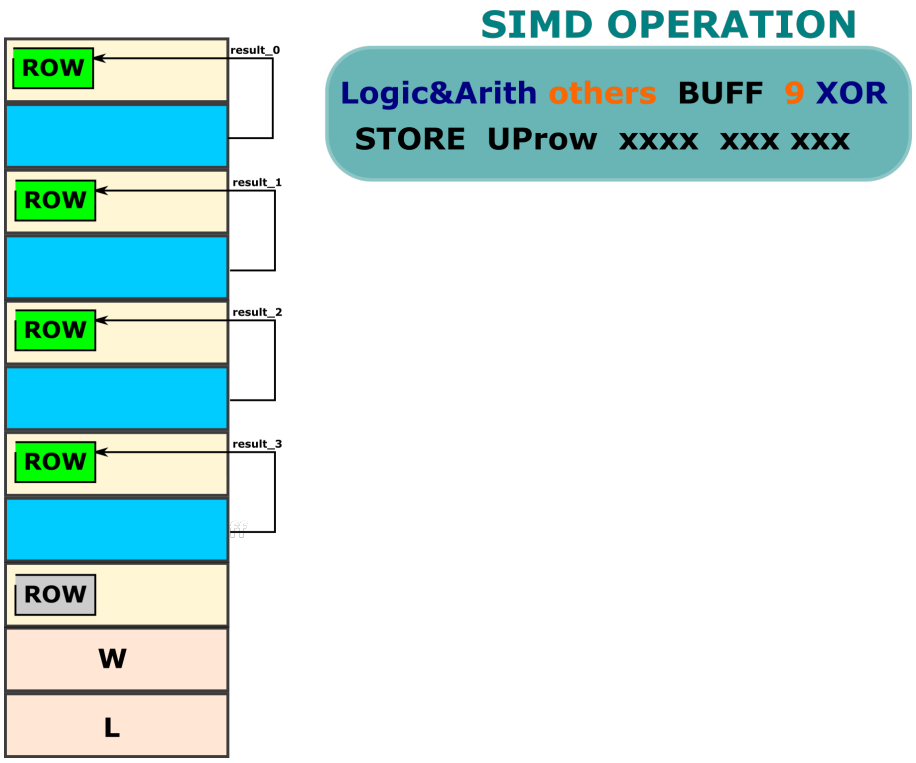


Figure 7.10: Example of an SIMD operation required to perform a XOR between the word stored in the smart row and a shared operand(W). The result is then written into the upper neighbour. The last row doesn't have to be written

7.4 Memory Interface Unit

The MIU has two main sections: the Miss Management Unit and the registers(7.11). There are two registers, one for each cache register: the **Address Register Up**

(Add RegU) and the **Address Register Down** (Add RegD). They stores the address of the word stored in the caches. The **Miss Management Unit**(MMU) generates a flag if a miss is find out: this signal is the **Miss Flag**.

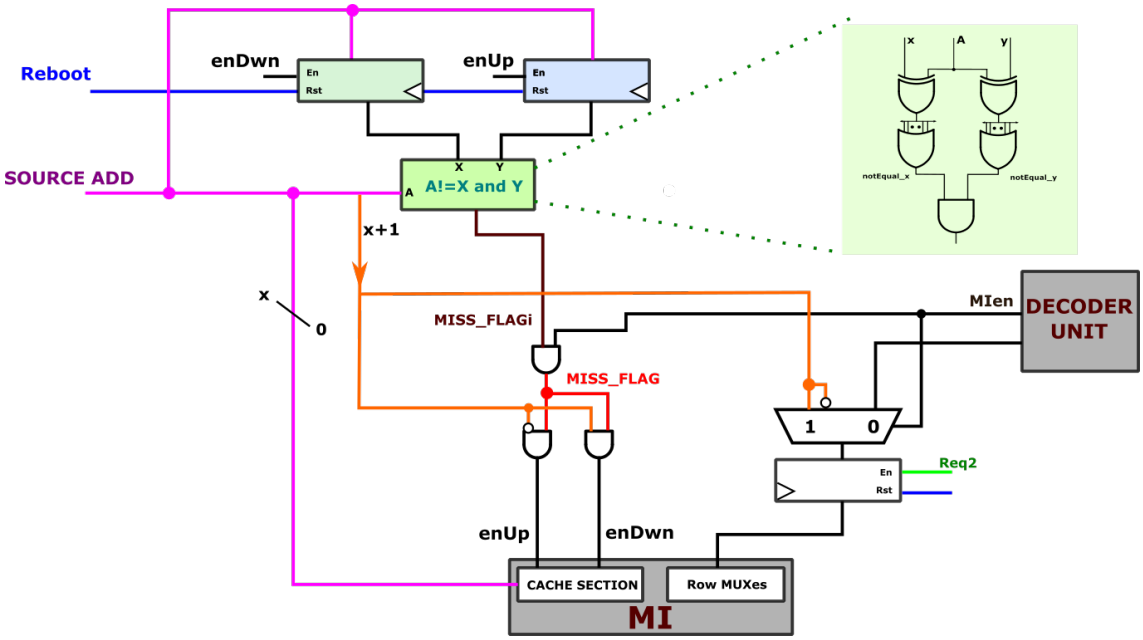


Figure 7.11: High Level View of MMU

The MIU operates in two cycles. After the fetch of a new nInstruction, it receives the source address. If a miss happens, the MIU generates signals for managing that, both for the inner component and for the cache. However, if it's not enabled by the decoder unit, these signals have no effect. During the next rising edge of the clock, the cache is updated if a miss happens.

Moreover, the MIU also generates the signals for the row multiplexers during the first cycle: these configuration signals are sent toward the row multiplexers in the next clock cycle. The MIU selects one of the cache register as input of the smart rows: which one depends on the source address. Actually, if the instruction doesn't require cache(Miss enable = 0) these signals come from the decoder unit.

It has a key role to find out which half of the memory is the addressed word in. The first bit of the source address isn't used for the control of cache multiplexer accomplish that. It indicates if the addressed word is in the lower or in the upper half of the cacheable memory (as explained in chapter 4, the size of the control

signal for cache multiplexer depends on the depth of the standard section of the LiM array) : if it's zero, the word is in the upper half; if it's one the word is in the lower half. This information is used both for generating the control of the row multiplexers and for enabling just one cache register and address register during a miss.

7.4.1 Cache Miss

If a word of the standard memory has to be used as operand, its address is indicated with the source address field. This word can be accessed just trough the associated cache register: the upper cache register if its in the upper half of the standard memory and the lower cache register otherwise.

A cache miss happens if the word addressed by the source address is not stored in the cache. It has to move into the associated cache register before starting the execution. The MIU asserts the flag miss if the source address is different from both the addresses of the words stored in the caches, that are the contents of the address registers. The **Miss routine** requires two actions(7.12): the source address is stored in the proper address register and the cache enabled is updated with the addressed word. Which register is updated depends on the **half bit**: the Add RegU register and the upper cache register if it's zero, the other two if it's one.

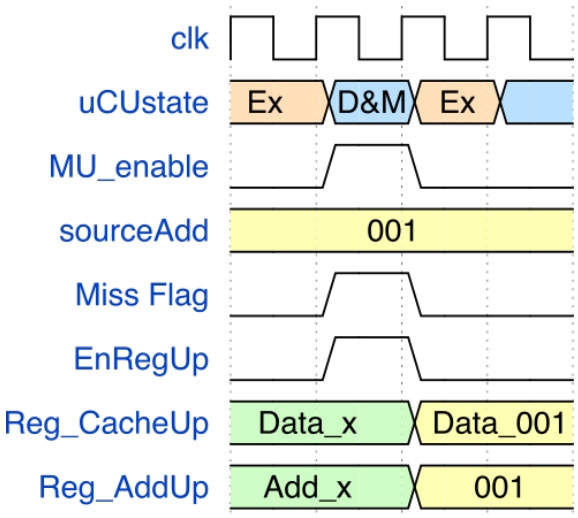


Figure 7.12: Miss routine for a miss in the upper cache register

7.5 Hazards

The **Hazards** are conflicts raising when the executions of two instructions overlaps with each other: in other words, it happens in a pipelined execution. They can be classified in three categories: data hazards, structural hazards and control hazards. The **control hazards** arises when an instruction changes the sequential flow of the instruction in the pipeline. Since there aren't instructions allowing to do that, are not taken into account by the analysis below. The **structural hazards** happens when two instructions would want to use the same hardware unit in the same cycle. The **data hazards** arise when the same data is shared among consecutive instructions: conflicts can happen. There are three kinds of data hazard depending on how the consecutive instructions access to the data: Read after write (RAW), Write after read (WAR) and write after write (WAW).

Summarizing, no hazard can happen in a LiM unit. About structural hazard, 7.13 shows how each cycle stimulates a different unit: so two overlapped cycles work on different units.

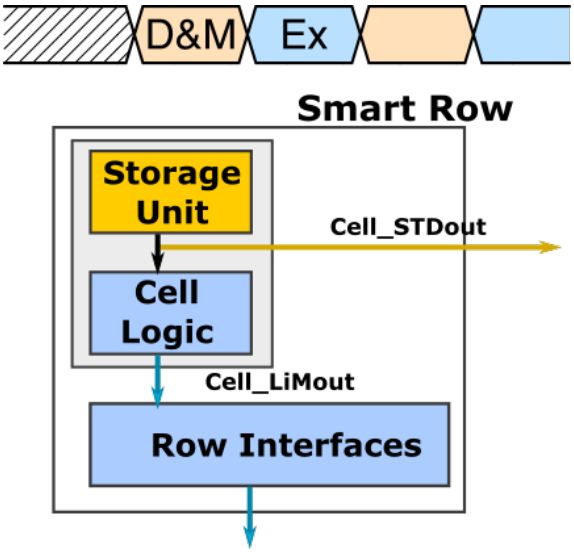


Figure 7.13: When units of a smart row are used

About data hazard, it's fundamental to understand when data are accessed. A row of the smart section can be written only by a LiM operation at the end of the execution stage. A cache register can be written after that a miss arises at the end

of the decode&Miss stage. About reading, a LiM operation can retrieve an operand either from a cache register and from a row of the smart section. On the other hand, the standard section can be read for a miss. Moreover, a cache register can be read after a miss too. So, the reading space of the cache is the standard section and the writing space of LiM operations is the smart section 7.14.

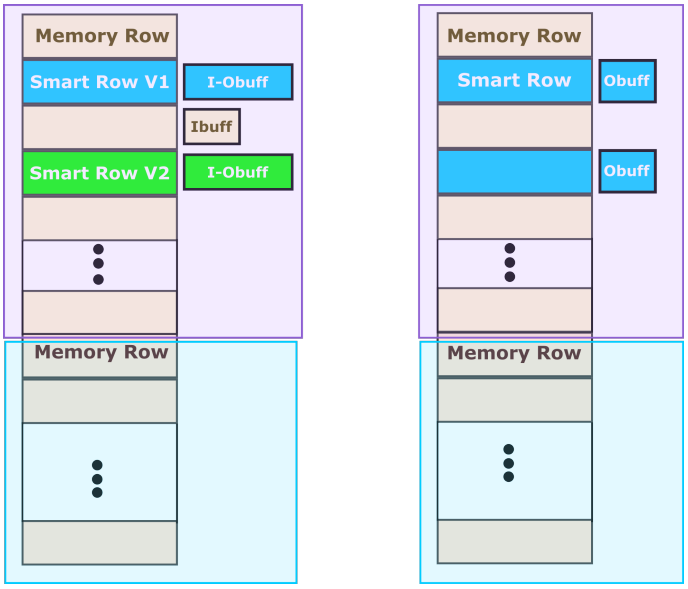
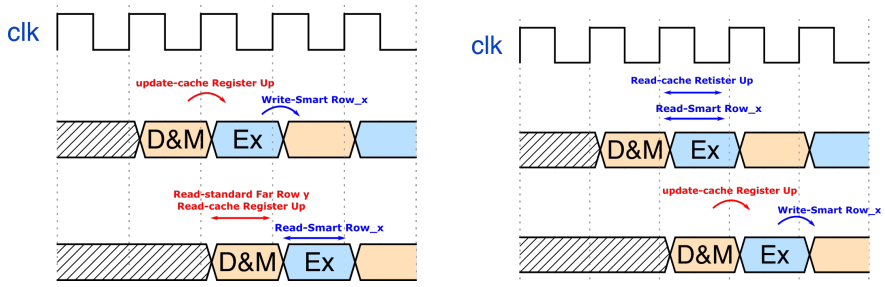


Figure 7.14: Writing space in violet and cache reading space in cyan

7.15.a shows how it's impossible to have a WAR or a WAW: there is only one point where a certain register (either a row or a cache) cane be written and it happens after that the previous instruction reads it. About RAW the situation is more tricky. Since two types of writing can occur, there are tow possible scenarios:

- the first operation writes a smart row and the next one reads it for another operation
- the first operation writes a cache for a miss and the next one reads it, either for a miss or for an operation

7.15.b underlines that there aren't hazard since the data is updated before that the next one reads it.



(a) RAW (b) WAR and WAW

Figure 7.15: Hazards' analysis. The blue operations are due to LiM instructions, the red ones due to cache miss

Notice that wherever the cache would read also the smart section, an hazard arose when the first operation writes a smart row and the following one reads it for a miss: the data read is not yet updated by the previous instruction. This is the reason why in chapter 5 the proper behavior has been hardwired forced: it avoids the overlapping between the reading space of the cache and the writing space of LiM operations.

7.6 Cache Coherence

A cache coherence issue happens when the content of the cache registers is outdated with respect to the memory content. In other word, if a row stored in the cache is written . A row can be written either by the CPU when the LiM array acts as a std memory or by a LiM operation. About the first situation, when the LiM CUs left the control of the memory to the CPU, they can't track what happens to the array. So, the scheduler has to reset the CUs, and so the cache registers too, before of a LiM execution. On the other hand, the second situation can't never happen: the cache reading space and the writing space of LiM operations are hardwired separated(7.14).

Chapter 8

Operating Instructions

This chapter is an handbook for the user. There is explained how to adapt the LiM unit to the specifications of the user. In other words, how the designer is supposed to create his own LiM unit, that is which sections have to be designed. The designer has to intervene on the uROM for writing the uPrograms, on the decode unit for managing his LiM array and he has to build the LiM array itself.

8.1 How to write the uROM

The uROM stores several uInstructions. Each of them has several fields. Focusing on the signals required for the operation of the uCU, the fields are(6.1):

- **Sequencer Configuration**(Seq): It controls the input of the uPC. 8.1 describes the different configurations allowed.
- **Conditional Code**(CC): It enables or not the branch. The branch is taken if the CC is high and the Done signal arrives from the nCU. As explained below, it has to be asserted just for the wait instruction.
- **next Address**(nextADD): It's the address of the next sequential uInstruction. So, its size depends on the depth of the uROM.
- **End uPorgram Flag**(endPflag): it alerts the uCU that the output of the uROM is the last uInstraction of a uProgram.
- **Fetch Enable**(FetchEN): if the uCU is activated, it enables the update of both Queue and uAR. It happens in the last uInstruction of each uProgram, that is before reading the uAR in the uPC.

Table 8.1: SEQ SIGNALS

SEQ	Effect
00	uPC <=uAR
01	uPC <=next sequential Address or jump Address
10	uPC <=return Address
11	uPC <=call Address & store return Address

Then, there is the **sMUXarray** signal. It arbiters on who has to control the LiM array. It has to be enable for all the uInstructions of each uProgram. Moreover, there are the signals for the handshakes. The **LiM mode** is the signal for the handshake with the sequencer. It is low only when the LiM array doesn't execute instructions. At the end, the **Request**(Req) validates the nInstruction sent to the nCU. It has to be asserted for each uInstruction of a uProgram.

How many uInstruction are stored in the uROM depends on the application. However, it need to write three standard instructions in each uROM for making the uCU properly works.: the idle instruction, the wait instruction and the end instruction.

The **Idle** instruction is the uInstruction addressed by the uPC when it's reset, that happens before starting a LiM processing. So, this instruction must be written in the first location on the uROM. This instruction need to enable the update of both the uIR and the LiM queue: the fetch enable signal must be asserted and the input of the uPC must be the uAR(seq= "00"). About the interfaces with the external world, the control of the array has to be left to the CU(sMUXarray = '0') and the scheduler has to be alerted that the LiM Unit is in idle(LiM mode ='0'). Moreover, this instruction has to not make working the nCU: the request signal must be zero. The other signals are don't care.

The **End** instruction is very similar to the idle one but it doesn't enable the update of the queue and the uIR. Indeed, it's the last nInstruction executed: it doesn't need to read new programs.

The **Wait** instruction must be the one after the last uInstruction of the last uProgram executed. This instruction must enable the jump to the End uInstruction as soon as the Done signal arrives from the nCU(CC='1'). So, the input of the uPC has to be the next Add field with the last bit controlled by the Jump Unit(uSeq =

"01"). Moreover, for making the branch possible, the address of the wait instruction and the one of the End instruction has to change just for the LSB. Then, since no new nInstructions are sent to the nCU, the request signal has to be disabled.

8.2 Nano ISA

The nano Instruction set architectur (nISA) describes which SIMD operations can be implemented with the LiM array built.

8.2.1 Types of LiM instructions

All the LiM instructions(nInstructions) can be classified in two main categories: executive and movements. The **executive instruction** are operations performed in the smart rows. There are the bitwise operations, either logical or arithmetic, and the operations performed on just one operand like shifting and rounding. On the other hand, the **movement instructions** allow to exchange data between different rows, that are inter-row movements, or to perform intra-row movements. There are two movement instructions. The **LOAD** moves a data coming from the a neighbourn into the input buffer of a smart row. The **STORE** moves the content of a smart row's output buffer into a row, either the one of a neighbourn or the one of the smart row itself.

8.2.2 Instruction Format

The nInstruction has a **Fixed field format** for a given memory size. Indeed, since a memory address is included in the instruction, the size of the instruction depends on the size of the memory. However, a LiM instruction has five fields(8.1):

- **OPCODE**: it identifies the LiM operations to do
- **OPERAND**: it identifies the addressing mode, that is where to take the operand/s
- **RESULT**: it encodes where to store the result,

- **SOURCE ADDRESS** : it's a plug-In of the operand. It's used in case of far operand
- **FUNC**: it's a plug-In of opcode. It can be used for managing complex configurable RI or cell, like adders

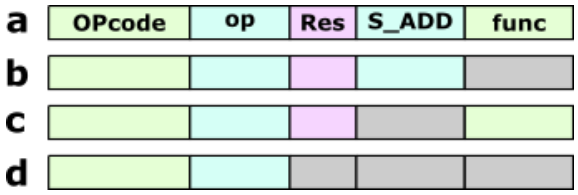


Figure 8.1: Instruction format(a). (b), (c), (d) show special cases: instructions don't involve programmable RIs, instructions don't require cache, and movements instructions respectively. The grey sections are don't care.

If the operand is not a neighbourn, its address has to be specified trough the source address. It will be read trough the cache. Operand, Result and source address are independent from the array. They just depends on the template chosen and on which buffer is used. On the other hand, both OpCode and Func strongly depend on the LegoLiM of the LiM array : they describe the smart features of the smart rows. Func field allows to manage components with an higher number of configuration signals, that are the configurable cells of RIs.

With movements operations the last three fields has to be left void(8.1.d). For these operations there is just one variable: where to send the output buffer for STORE and what the input buffer has to be loaded with for LOAD. These variable are fixed with the operand field.

8.2.3 Addressing modes

An SIMD operation takes place into smart rows. From where the smart unit of the smart rows retrieve the operands is specified by the addressing mode encoded into the operand field. In other words, the addressing modes are the possible sources of the instruction's operand. It depends on the template used, on the buffer implemented in the smart rows and on the version used for the components of the smart rows.

An instruction can call one or two operands : the bitwise operations require two operands, the other ones just one. The main role is that just one of the two operands can be external to the smart row because of the interface with memory interface has just one input signal. External means a neighbor or the cache registers. Moreover, since the operand decoder is just one, three constrains are laid down: the same version has to be used for the LiM Cells of all the smart rows(std, B or BB), all the RIs must be of the same Version(1D, 2D, 1D B, 2D B or 2D BB) and the same buffer has to be used for all the smart rows. So, the same addressing mode is forced for both the smart rows of a template V2. 8.3 summarizes all the addressing modes. Depending on the buffer used and the versions of components, only the proper ones can be implemented.

1 Operand	2 Operands
V1	
Smart Row(Row)	Row ☐ Row.Output-buffer (Row.Obuffer)
Up Neighbor(UpRow)	Row ☐ UpRow
Down Neighbor(DwnRow)	Row ☐ DwnRow
Standard Near Row(Other)	Row ☐ Other
Row.Obuffer	
	Row.Obuffer ☐ UpRow
	Row.Obuffer ☐ DwnRow
	Row.Obuffer ☐ Other
Row.Input-buffer(Row.Ibuffer)	Row ☐ Row.Ibuffer
	Row.Ibuffer ☐ UpRow
	Row.Ibuffer ☐ DwnRow
	Row.Ibuffer ☐ Other
Additional modes for V2	
Neighbor.Input-buffer(Neighbor.Ibuffer)	Neighbor.Ibuffer ☐ Row
	Neighbor.Ibuffer ☐ Row.Obuffer

Figure 8.2: Signal generated by the decode unit

Figure 8.3: Addressing Modes:The text color indicates the buffer required for implementing that addressing mode and square indicates the version of component required for implementing that addressing mode. **Color legend**: purple for array with output buffer in the smart row, cyan for array with input buffer in the smart row. **Square legend**: white for the addressing modes always allowed, yellow and orange for the version of LiM cells(yellow for at least version B and orange for version BB), green and blue for RIs(blue for buffer version 2D or 2D BB,if there is a buffer, and green for at least version 2D B)

The standard sources for an operand are: the neighbourns, that are the standard row up and the standard rows surrounded the smart rows, the content of the smart row itself or a row of the standard section(Others). In the latter case, the source address has to be used for specifying the address of the row. Moreover, for V2 the neighbourns can be either the content of the row or the content of the input buffer.

If a buffer is implemented in the smart rows, also it can be used as operand. An input buffer added to the smart rows allows to partially overcome the limits of using just one external operand(8.4). Indeed, exploiting LOAD is possible to perform an operation with two external operands. The cost is an additional cycle : move a neighbour in the input buffer of the smart row during the first cycle; perform the operation with input buffer, that is a neighbour, and an external signal in the second cycle.

About operations required two operands, the addressing modes are just the proper combinations of the addressing modes for 1 operand, where proper means that the role of using just one external operand must be observed.

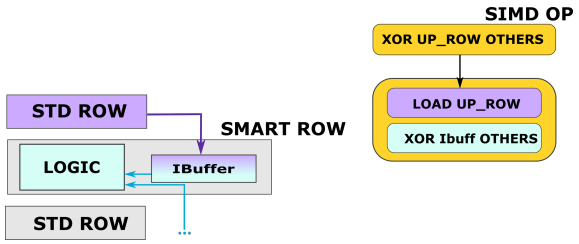


Figure 8.4: Signal generated by the decode unit

8.2.4 Destination

Where the result of a smart row operation has to be send is laid down by the destination encoded in the result field. It depends on the template and on the buffer of the smart rows. Also for the destination the approach is the same of the addressing mode: there is one destination decoder generating the same configuration for for all the smart rows. 8.2 summarizes all the possible destinations.

Table 8.2: DESTINATION MODES. THE PURPLE TEXT INDICATES THAT THE DESTINATION MODE IS JUST FOR ARRAY WITH OUTPUT BUFFER IN THE SMART ROW, THE GREEN MODE IS THE ADDITIONAL ONE WITH V2.

Destination Modes
To the Near Memory Unit(Ext)
Row.Output-buffer (Row.Obuffer)
Neighbor.Input-buffer(Neighbor.Ibuffer)

The result can be directly stored either somewhere into the smart row or it can be send to the near memory section(**Ext**). Inside the smart row itself, it can be just stored in the the output buffer, if there is it. Moreover, V2 allows to store the result also in the input buffer of neighbourns for multi-rows operations(7.5).
These basic options for result's destination can be significantly increased trough an output buffer. Indeed, the store operation allows to store the result either in the smart row itself or in one of the neighbours at the cost of one additional cycle. First, the result of the operation is stored in the output buffer, then it's written in the destination(smart row, upper neighbor or lower neighbor) with a store operation(**??**.b).

8.2.5 OPCODE

The opCodes describe the smart features of the array. So, they strongly depends on the Lego LiM of the array. However, in order to generalize, some key points can be fixed.

executive instructions

There are two types of executive instructions: the ones involving just one unit of the smart row and the ones involving from 2 to all the smart units of the row. Starting with the first, each unit can used alone. So, the first step is to write down as many opCodes as many are the units of the smart row. Each opCode can stimulated each unit just one time. However, if there is an adder, it's not enough. The opCode alerts that the adder is involved but the func field has to specify which operation has to be performed. 8.3 shows the func for adders. The same would happen with every configurable RI(at the moment, the library has just the adders).

Table 8.3: CONFIGURATION OF ADDERS. FOR THE OPERATIONS SEE **??**

RCA		
func	**Operations**	
	Standard	Buffer
001	$A+B$	
010	$A-B$	
011		$-A+B$
100		$-A-B$

RCA&Logic		
func	**Operations**	
	Standard	Buffer
0111	$A or B$ (OR1)	
0110	$A and B$(AND1)	
0101	$A \oplus B$(XOR1)	
1011	$\overline{A \oplus B}$(XNOR1)	
1000	$A or \overline{B}$(OR2)	
1100	$\overline{A \oplus \overline{B}}$(XNOR2)	
0001	$A+B$	(SUM)
0010	$A-B$	(SUB1)
0011		$-A+B$(SUB2)
0100		$-A-B$(SUB3)
1001		$\overline{A} or B$(OR3)
1101		$\overline{\overline{A} \oplus B}$(XNOR3)
1010		$\overline{A} or \overline{B}$(OR4)
1110		$\overline{\overline{A} \oplus \overline{B}}$(XNOR4)

Then, more units can be involved. They has to be concatenated in all the possible way just observing one role: the units involved can be concatenated just in order(8.5). It means that the logic in the row can be just executed for first in a stream of operations and the furthest RI from the row as the last one.

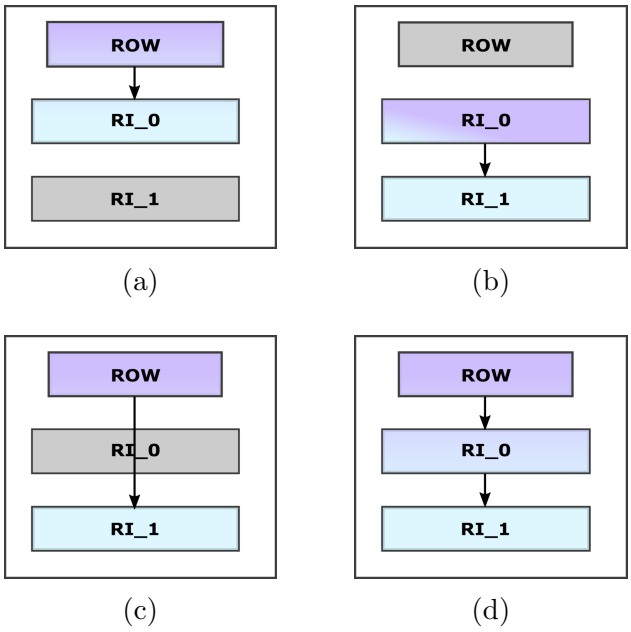


Figure 8.5: Possible concatenation of smart row's bricks. The starting point is the violet block and the end point is the cyan one. The grey colored blocks are not involved.

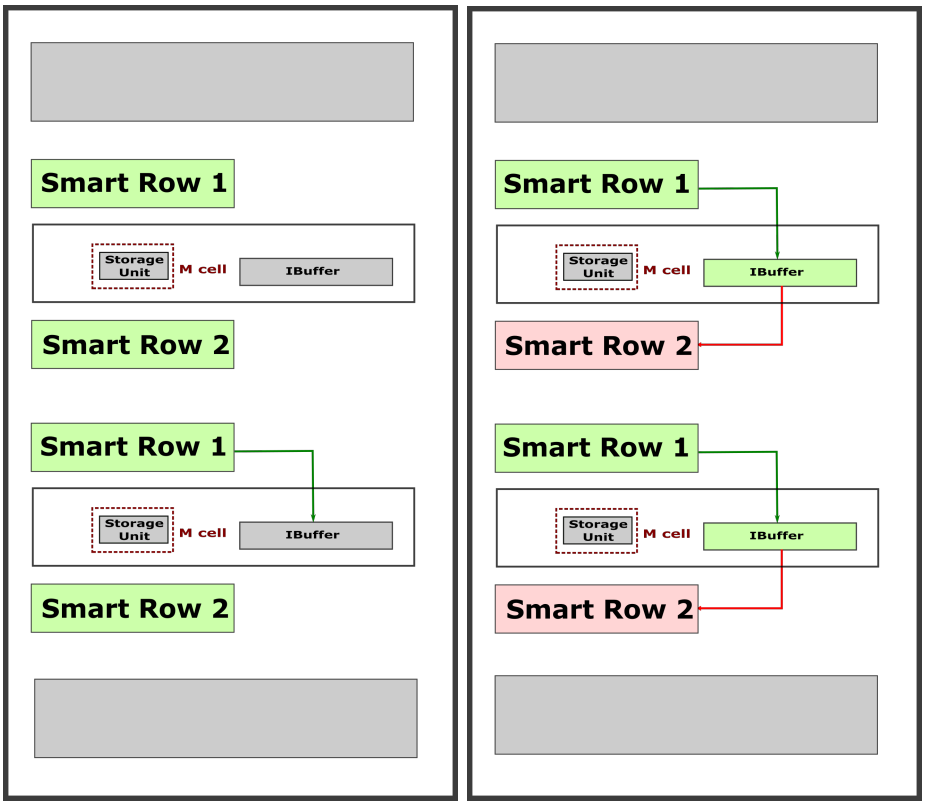
With V2 the situation is a quite more tricky: the smart features of both the smart rows have to be manages. There are two modes of operation: just one or both the rows can work(8.6).

For **one row mode**, the other row is in idle. This mode can be activated for two reasons: the operation implemented by the other row are useless for the running algorithm or a **multi-row SIMD operation** has to be performed. The second case is the most interesting. It means that the operation required both the rows : the input buffer of standard near row is used as checkpoint. This kind of operation takes two two cycle:

1. $1^s tcycle$: one row performs the operation . The result is moved into the input buffer of the standard row.

2. $2^s tcycle$: the other row reads the partial result from the standard row and performs the operation.

An additional cycle could be required because of a load operation for bringing inside the partial result. It allows to use another external signal.



(a) 1 Row (b) 2 Rows

Figure 8.6: Portion of the smart section showing the modes of operation of V2. The green bricks are used in the first cycle and the red ones in the second cycle. The grey bricks are not involved.

About the **two rows mode**, both the rows work independently from each other. So, for this mode input buffer of standard near row can be used neither as operand nor as destination.

Movement instructions

8.4 shows all the possible movement instructions. Notice that, for V2, load can be performed just for one type of smart row at a time because of all the multiplexer of the smart row share the same configuration signal. So, it could be impossible to simultaneously load the Ibuffer of a neighbourn in both of the smart rows.

Table 8.4: MOVEMENT OPERATIONS FOR THE TWO VERSIONS

V1		V2		
STORE	SmartRow.Obuffer −− > SmartRow (Row) SmartRow.Obuffer −− > Upper Neighbor (UpRow) SmartRow.Obuffer −− > Lower Neighbor (DownRow)	**STORE**	$SmartRow_i$.Obuffer −− > Row $SmartRow_i$.Obuffer −− > UpRo) $SmartRow_i$.Obuffer −− > DownRow	i=1,2 or 1&2
LOAD	SmartRow.Ibuffer < −− Row SmartRow.Ibuffer < −− UpRow SmartRow.Ibuffer < −− DownRow	**LOAD**	SmartRow.Ibuffer < −− Row SmartRow.Ibuffer < −− UpRow SmartRow.Ibuffer < −− DownRow	i=1 or 2

8.3 How to write the instruction decoders

Although there are four decoders, just two of them have to be configured by the designer: the function one and the opCode one. About the other two decoders, they are explained in 8.3.3.
The function decoder is just a plug-in of the opCode decoders for the configuration of more complex RI such as an adder. About the opCode decoder, it completely depends on the LiM array but it slightly changes between the two versions of templates. However, its skeleton is always the same. The following sections analyzes the opCode decoder for V1 first; then, the differences with V2 are underlined. For both the version, the most complex case is analyzed: it's supposed to implement I-Obuffer for both the versions.

8.3.1 OpCODE decoder for V1

There are five types of signal:

- **SmartRow.S_EXT_notBL** selects the input of the cell of the smart rows. It cares just for store operations: the output of theb(extB) has to be selected;
- **stdROWnear.S_EXT_notBL** selects the input of the standard rows near. It cares just for store operations: it has to select the data coming from the smart rows(ext);
- **WriteActive** has to be asserted just for store operation;
- for each RI a $\mathbf{RI_i}$**.flagACTIVE-MODE** and a $\mathbf{cnfgRI_i}$ are required. $\mathbf{RI_i}$**.flagACTIVE-MODE** has to be asserted when the RI is used in active mode. $\mathbf{cnfgRI_i}$ is

the configuration of the RI: it selects the mode for the RI;

- The **IOBuff.TowMI** and **IOBuff.enIbuff** are required only if the buffer of smart row is used. **IOBuff.enIbuff** need to be enabled just for load operations. **IOBuff.TowMI** selects the output toward MI of the smart rows. It has to be always the output of the row with the exception of the store operations: the output of the Obuffer is send toward MI for storing it in the neighbourns.

8.3.2 OpCODE decoder fo V2

there are three differences with V1:

- the **writeActive** is split in three signals: one for smart rows 1, one for smart rows 2 and the latter one for the standard rows. Depending on the store operation, one or both the writeActive signals for smart rows are activated. The one for standard row is asserted in all the store operations;

- two signals enabling the smart rows' types are required: **flag SR2**, **flag SR1**. The flag signal of a smart row is asserted when the operation requires that smart row. Otherwise it's zero;

- there are two loads operation and three store operations(refmovementOP)

8.3.3 Fixed Decoders

An extract of the VHDL source of a LEGO LiM unit is reported below. It shows the result and the operand decoder for the most complex case: template V2, with IObuffer, Cells of version BB and RIs of version 2D BB. In case of simpler array, the tables has to be pruned.

```
--------------------------------
--------Operands DECODER--------
--INPUT: Operands
opDEC: process( Operands )
       begin
         Wen_stdROWnearFirst_Operands<='0'; Wen_stdROWnearLast_Operands<='0';
               Wen_stdROWnearInner_Operands<='0';
         case Operands is

```

```
            when ROW    => selRowMux_std <= '0'; selRowMux_smart <="00";
                  EN_missCheck <='0'; S_inLogic <="00"; Ren_SmartROW <='1';
                     Wen_smartROW_Operands <='1';
                       Ren_stdROWnear <='0'; Ren_stdROWfar <='0';
                             cnfg1Count_MSBs_Operands <="00";
                             cnfgSH_MSBs_Operands <="00";

            when Buff   => selRowMux_std <= '0'; selRowMux_smart <="00";
                  EN_missCheck <='0'; S_inLogic <="01"; Ren_SmartROW <='0';
                     Wen_smartROW_Operands <='0';
                       Ren_stdROWnear <='0'; Ren_stdROWfar <='0';
                             cnfg1Count_MSBs_Operands <="01";
                             cnfgSH_MSBs_Operands <="01";

            when DwnR   => selRowMux_std <= '0'; selRowMux_smart <="11";
                  EN_missCheck <='0'; S_inLogic <="10"; Ren_SmartROW <='1';
                     Wen_smartROW_Operands <='0';Wen_stdROWnearLast_Operands <='1';
                           Wen_stdROWnearInner_Operands <='1';
                     Ren_stdROWnear <='1'; Ren_stdROWfar <='0';
                             cnfg1Count_MSBs_Operands <="11";
                             cnfgSH_MSBs_Operands <="11";

            when B_DwnR => selRowMux_std <= '0'; selRowMux_smart <="11";
                  EN_missCheck <='0'; S_inLogic <="11"; Ren_SmartROW <='0';
                     Wen_smartROW_Operands <='0';
                       Ren_stdROWnear <='1'; Ren_stdROWfar <='0';
                             cnfg1Count_MSBs_Operands <="00";
                             cnfgSH_MSBs_Operands <="00";

            when UpR    => selRowMux_std <= '1'; selRowMux_smart <="00";
                  EN_missCheck <='0'; S_inLogic <="10"; Ren_SmartROW <='1';
                     Wen_smartROW_Operands <='0';
                           Wen_stdROWnearFirst_Operands <='1';
                           Wen_stdROWnearInner_Operands <='1';
                     Ren_stdROWnear <='1'; Ren_stdROWfar <='0';
                             cnfg1Count_MSBs_Operands <="11";
                             cnfgSH_MSBs_Operands <="11";

            when B_UpR  => selRowMux_std <= '0'; selRowMux_smart <="00";
                  EN_missCheck <='0'; S_inLogic <="11"; Ren_SmartROW <='0';
                     Wen_smartROW_Operands <='0';
                       Ren_stdROWnear <='1'; Ren_stdROWfar <='0';
                             cnfg1Count_MSBs_Operands <="00";
                             cnfgSH_MSBs_Operands <="00";

            when OthR   => selRowMux_std <= '0'; selRowMux_smart <="00";
                  EN_missCheck <='1'; S_inLogic <="10"; Ren_SmartROW <='1';
                     Wen_smartROW_Operands <='0';
```

```
                        Ren_stdROWnear<='0'; Ren_stdROWfar<='1';
                              cnfg1Count_MSBs_Operands<="11";
                              cnfgSH_MSBs_Operands<="11";

            when B_OthR => selRowMux_std<= '0'; selRowMux_smart<="00";
                  EN_missCheck<='1'; S_inLogic<="11"; Ren_SmartROW<='0';
                     Wen_smartROW_Operands<='0';
                       Ren_stdROWnear<='0'; Ren_stdROWfar<='1';
                              cnfg1Count_MSBs_Operands<="00";
                              cnfgSH_MSBs_Operands<="00";

            when others => selRowMux_std<= '0'; selRowMux_smart<="00";
                  EN_missCheck<='0'; S_inLogic<="00"; Ren_SmartROW<='1';
                     Wen_smartROW_Operands<='0';
                       Ren_stdROWnear<='0'; Ren_stdROWfar<='0';
                              cnfg1Count_MSBs_Operands<="00";
                              cnfgSH_MSBs_Operands<="00";
          end case;
       end process;



---------------------------------
------- Resource DECODER--------
--INPUT: Res
redDEC: process( Res )
    begin

       case Res is
          when StoreBuff => EnObuff<= '1';
          when StoreExt  => EnObuff<= '0';
          when others    => EnObuff<= '0';
            end case ;
        end process;
```

8.4 How to build a Lego LiM array

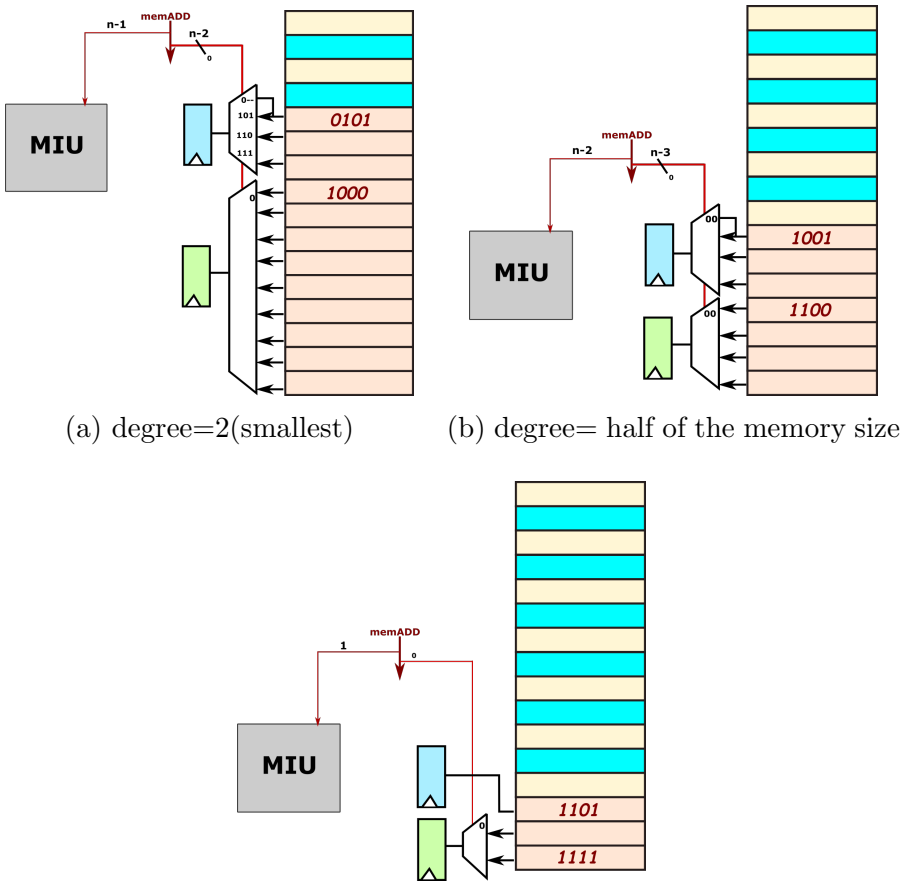
Whatever is the template chosen, each LiM array is made up by a smart section and a standard section. Since the standard section have just rows without smart features, the designer has to just build the smart section. The designer has to lay down: the template to use, the size of the sections and the Lego LiM of the smart section(how many? which ones? which versions?).

8.4.1 Template: V1 or V2 ?

The starting point is required to be the algorithms to implement: depending on the degree of SIMD required, one of the two template has to be chosen. V1 allows to have an higher parallelism but a less wide instruction set(4.13). Indeed, it can handle just one SIMD operation for clock cycle but this operation can be performed on an higher number of data rather than the V2. On the other hand, since V2 has two different types of smart row, two SIMD operations can be done in parallel. However, the same operation can be performed less times in parallel with respect to V1, giving a certain memory size.

8.4.2 Template: sizes

Once the template has been established, it has to be configured: the size of the sections are two variables to fix. It depends on the parallelism required: increasing the size of the smart section allows to have an higher number of operations done in parallel. Of course, the drawback is the overhead in terms of area and power. Depending one the size of standard section, the width of the cache multiplexers of the MI changes: the parallelism of the control signal has to be adapted according to $\left\lceil log_2(\frac{|StandardFar|}{2}) \right\rceil$. Moreover, the MIU has to read the first bit of the source address that is not used for controlling the cache multiplexers(8.7 shows it with different degrees of SIMD).
Actually the size of the standard section depends on the data too. Indeed, the operands shared by all of them have to be stored in the standard section(4.2.4). So, the standard section has to be deep enough to store all of them.



(a) degree=2(smallest) (b) degree= half of the memory size

(c) degree= highest

Figure 8.7: MI for different degrees of SIMD

8.4.3 LEGO LiM: which ones? in which order?

The next step is the design flow of the smart rows: which LEGO LiM use for building them. Seven rules have been observed during the construction of a smart row:

- the same cell has to be used for a row
- an adder must be placed after an arithmetic cell
- If there is a buffer in the smart row, the same buffer version has to be used for all the components of the row
- buffer has to be the last RI of the row

- for V2, all the smart rows must have the same buffer
- all the LiM cell of smart rows must be of the same version
- one version of RI can be used for all the array

In order to chose the bricks to use, the algorithms to implement have to be analyzed in order to extract the operations to implement. Each **RI** adds to the instruction set a new smart feature, that is an operation performed on one operand, i.e. one counting and shifts. The **adders** are the only exception. They are the plug-in of arithmetic cell allows to implement complex logic and arithmetic operations. The type of **cell** lays down the bitwise operations that the cell can perform.
Summarizing the key points from the nISA :

I a LiM operation can involves from one to all the unit of a smart rows;

II each unit of the smart row can be used just one time for clock cycle;

III the units of a smart row involved can be concatenated only observing the hierarchy;

IV a LiM operation can use just one operand that is external to the smart rows, either a neighbour or a cache register.

So, the last step is to analyzes the flow of the instructions, that is in which order they have to be executed: it lays down where to place the RIs selected inside the smart row. Then, the buffer in output of the smart row may or may not be placed. The output buffer allows to implement SIMD requiring more cycles because of either (ii) or (iii). On the other hand, the input buffer allows to overcome (iv). Remember that the reset of each buffer is the reboot signal sending by the scheduler.
Moreover, which version to use for both smart rows' cell and RIs need to be chosen. It depends on the addressing mode required by the operations: the one operands operation fixes the version of RI and the bitwise operations lays down the cells' version. Of course, if buffers are implemented the buffer versions are required.

8.4.4 Example

You want to design a LiM array implements just one algorithm, that is the one used as test-bench in chapter 10. The first step is to parse the model of the algorithm for obtaining the instructions required. 8.1 describes the algorithm.

$$2 \cdot 1count(\overline{X_i \oplus W}) - L \qquad i \quad in[1,6] \tag{8.1}$$

Template

Version Since just one algorithm has to be implemented, it's better to have V1: it allows to have an higher parallelism with the same memory size.

Sizes About the sizes of section, the degree of SIMD required and the shared data has to be analyzed. Since six operations have to be done in parallel, six smart rows are enough. Then, since W and L has to be shared by all the SIMD operations, they has to be store in the standard far; in which row doesn't care: so a standard section of at least two rows is require. About the other data(The X_i), they could be placed both either in the upper neighbourns, in the lower neighbourns or in the smart row. However, since just one operand can be external and W and L are already external, it's useful to store it as internal, that is in the smart rows(8.8).

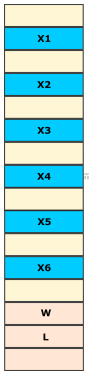


Figure 8.8: Algorithm's data-memory mapping

LEGO LiM

Which one About the LEGO LiM to select, the operations required are: two bitwise (xnor and difference) and two one operands operations(one counting and left shift). About bitwise operation, since there is an arithmetic operation, an Arithmetic cell as well as an Adder as plugin are required. Since also a logic operation has to be performed, the *RCA&Logic* adder is used. Then, it needs a shifter and a one counter for implementing the one operand operations.

Hierarchy Then looking to the steps required(8.9), it can be identified the hierarchy of the cell, if a buffer is required and which version of the bricks have to be used.

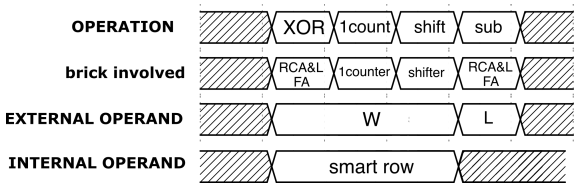


Figure 8.9: Algorithm's flow

About the hierarchy, the 1 counter has to be placed before the shifter because of it has to be executed first. About buffer, an output buffer is required because of the RCA&Logic has to be used two times. On the other hand, since the bitwise operation require to use just one external operand, an input buffer would be useless. So, two steps are required(8.10): the first step performs the operations until the shift and the partial result is written in the buffer; the second step reads the buffer and performs the subtraction.

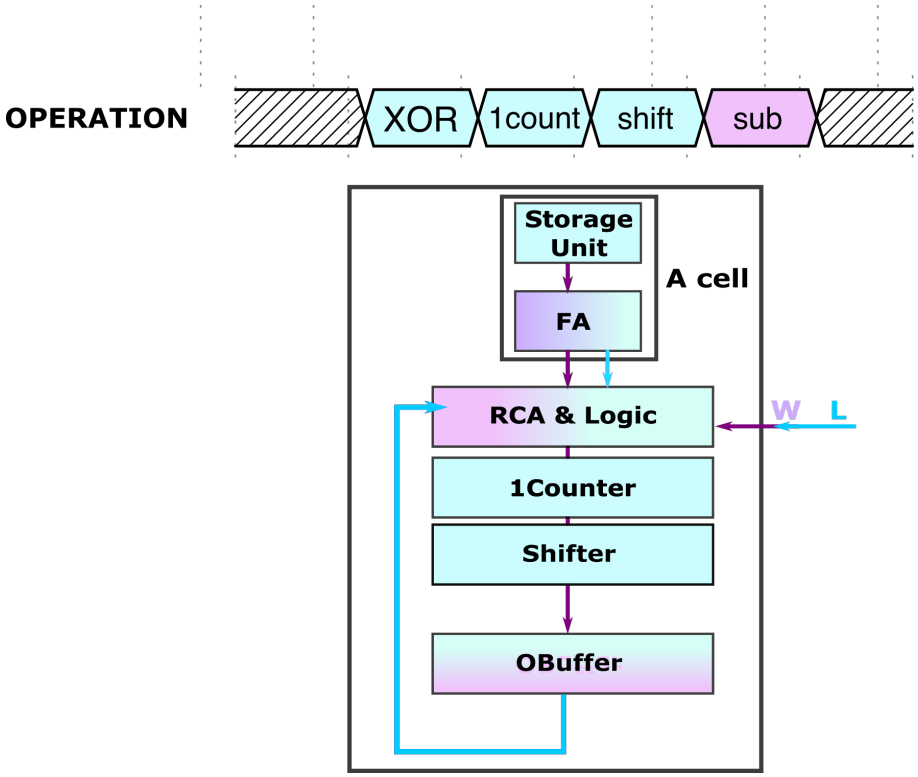


Figure 8.10: Algorithm's flow

Version About the version of the bricks, the operands required by the operations have to be analyzed: one operand operations for the RIs and bitwise ones for the cells of smart rows. For one operand operations, just the external signal has to be used as operand: version 2D BB is required. About the bitwise operation, since the external signal don't have to be left out, version B is enough for the cells. However, in order to analyze the worst case, the version BB is implemented for the chapter 9.

Chapter 9

Impact of Smart Features

9.1 Introduction

This chapter analyzes two key aspects of the architecture : the impact of the smart features and the impact of the Control Units. The two architectures that has to be compared are stimulated trough the same test-bench and they are synthesized with the same time constrains : clock with a period of 7 ns, skew of 0.07 ns and both input and output delays of 0.5 ns. If the parallelism of the array is kept fixed, the number of rows sweeps from 8 to 512. The performance after the place & route are recorded following the approach described in appendix A. The test-bench stresses the arrays: it writes one time all the rows. This approach was adopted for both the comparisons.
The two LiM arrays analyzed are the ones described in the next chapter. Wherever is the number of rows, the smart section is an half of the whole memory.

9.2 LiM array vs standard array

The first analysis evaluate the impact of the smart features: a standard array is compared with the two LiM arrays described in the next chapter. Moreover, the consumption of the standard array has a key role: they fixes the lower bound of the array. In other words, it's impossible to have less consumption than this array.

9.1 shows the performance of the three architecture in terms of area, total power an interconnection at different memory depths. Moreover, the last subplot compares the LiM arrays with the standard array: the smart features duplicate in average the power consumption. Notice that increasing the number of row the gap enlarges because of more smart rows. Indeed, the size of the smart section is an half of the memory size: so, the number of smart rows increases with the memory depth.

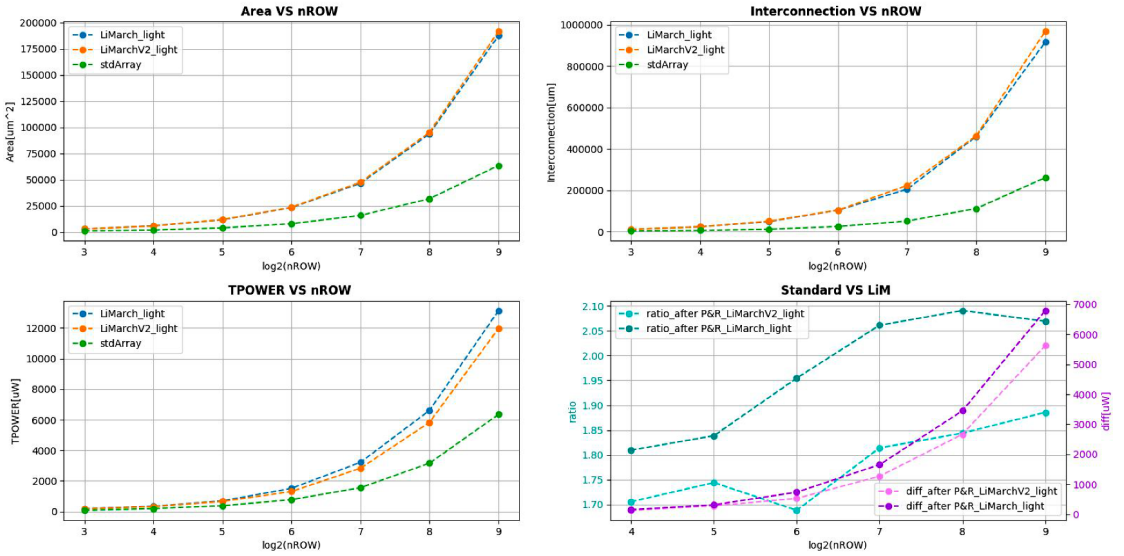


Figure 9.1: Impact of smart features.

However, it has sense considering the huge additional hardware of the LiM array.

9.3 LiM array vs standard array

Once the impact of smart features has been evaluated, the further overhead due to the CUs must be evaluated: the LiM arrays with CUs are compared with a LiM arrays without them(9.2). Again, the impact has been evaluated in terms of area, total power an interconnection to vary the number of array rows, with the smart section kept as an half of the whole memory. Two observations can be done. First, the impact of control unit decreasing by increasing the memory size: the ratio between the power consumption of architecture with and without CUs goes toward 1. 9.3 demonstrates why it happens: the impact of the CU on the whole LiM architecture decreases with higher number of rows. In other words, increasing the number of rows the complexity of the array increases but the overhead of the CU remains almost the same. So, the impact in percentage of the CUs decreases. Second, notice that the impact of the CU is bigger for the V2 because of the CU has to generate more signals. It underlines the power of templates: they drastically reduces the number of signals generated by the CU.

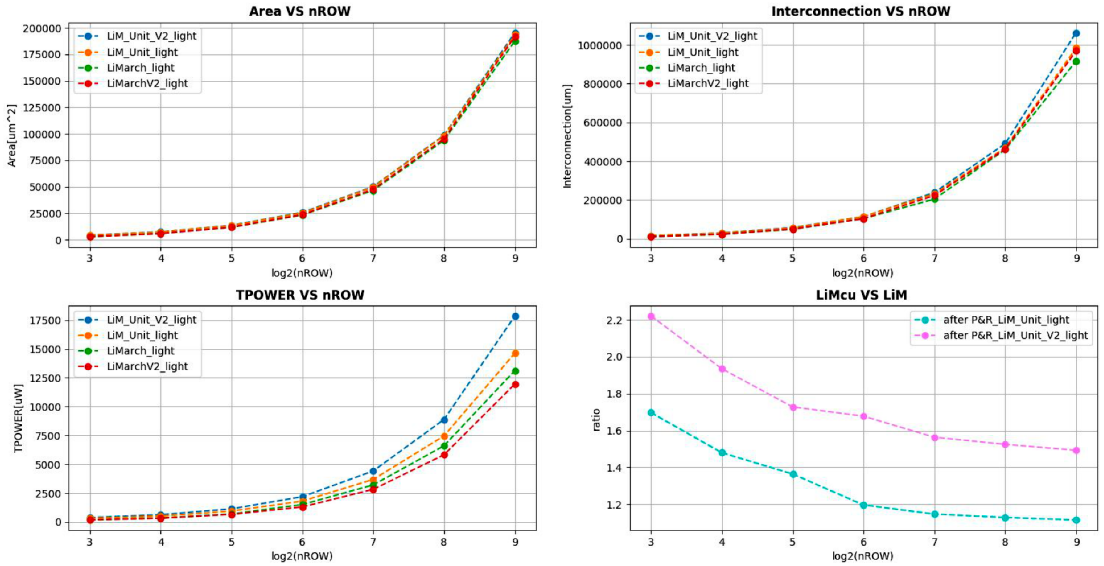


Figure 9.2: Impact of smart features.

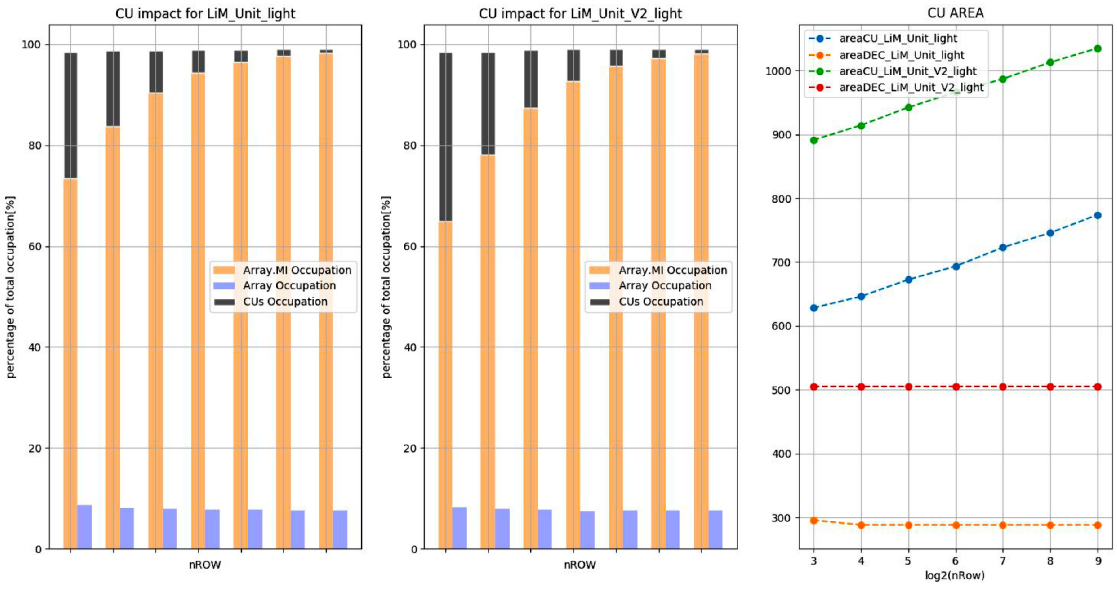


Figure 9.3: Impact of smart features.

Chapter 10

Implementations

A LEGO LiM Unit can handle several operations in parallel(SIMD operations). So, the applications better fit the nature of a LiM array are the ones with an parallel-based computation core. Neural Networks(NNs) fully belong to this category: hundred, thousand of input neurons are processing in parallel trough a NN. As done in [6], among the several model of NN in literature, the Computation neural networks (CNNs) have been chosen. Specifically, one step of the computation is implemented: the convolution operation. Three architectures have been compared on this test-bench: a LiM array V1, a LiM array V2 and the **DLX**.

10.1 DLX: an overview

The DLX is a conceptual RISC microprocessor with a LOAD-STORE architecture pipelined in 5 stages. There are four main units: the data memory, the instruction memory, the pipelined datapath and the Control Unit (CU). The CU is an hardwired control unit.

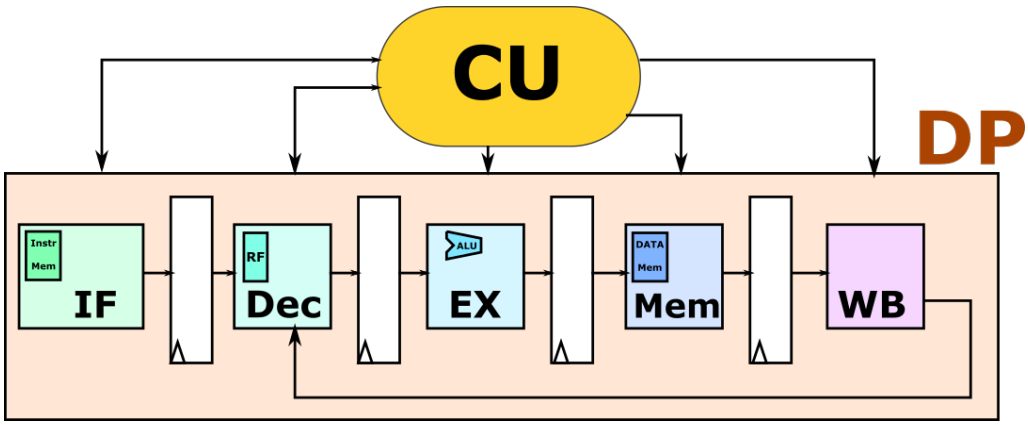


Figure 10.1: High level view of a DLX.

10.1.1 Stages

In the **Instruction Fetch** (IF) stage, the instruction memory is addressed for fetching the new instruction. Then, the new instruction is decoded and the register file is addressed: this happens in the **Instruction Decode** (ID) stage. The **Execution**(EX) stage is the core: it's where the ALU(Arithmetic Logic Unit) performs operation on the input data. The ALU implements logic and arithmetic operations, comparisons and shifts. The **Memory Access** (MEM) stage is focused on the access to the data memory. The last stage is the **Write Back**(WB): the access in writing to the Register File is performed.

10.1.2 ISA

10.2 INSTRUCTIONS

- **FORMAT** : fixed

- **PARALLELISM** : 32 bits
- **FAMILIES** : I-TYPE, J-TYPE, R-TYPE (10.2)

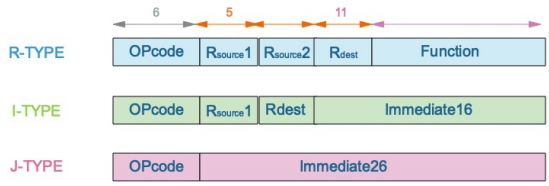


Figure 10.2: Instruction format of DLX.

10.3 REGISTERS

- **TYPE** :INTEGER GPRs (General Pourpose Register)
- **PARALLELISM** : 32 bits
- **NUMBER** :32
- **DEDICATED REGISTER** :
  - **R[0]** must be always 0
  - **R[31]** stores the return address of a *Jump And Link* instruction.

10.4 DATA MEMORY

- **TYPE** : alligned memory
- **PARALLELISM** : 32 bits (virtually, figure 1.3)
- **Allignment** :32

10.5 Neural Network: the model chosen

10.5.1 Introduction of NN

A neural network is a mathematical model inspired by the human brain. It's described by an algorithm that's able to recognize paths on the input data for achieving task like classification, clustering, prediction and so on. The core computational unit is the **artificial neuron**(10.3): it receives a weighted signal from each dendrite and the sum of all these signals is filtered by an activation function. The weights are the implementation of the synapsis and the activation function models the axon. A NN is made up by several layers of neurons interconnected in some way.

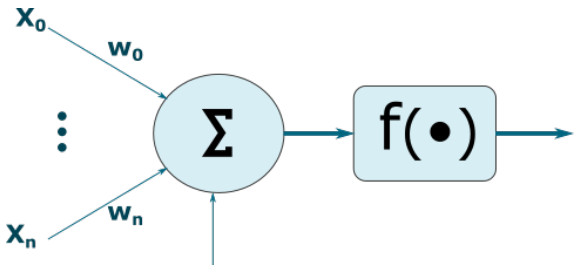


Figure 10.3: Neural Network's neuron

Before using a NN, a training phase is required: the weights are updated by feeding the NN with a set of labeled samples. Then, the accuracy is evaluated by showing to the NN data never seen.
The **Convolutional Neural Network**(CNNs) are one of the most common used structures of NN for immage recognition: they extract information from the input immage in several steps, each one with a higher degree of details. The matrix of pixels representing the input image is called **Input Feature Map** (IFMap). There are three kind of layers:

- the **convolutional layer** performs the convolution operation between the IFMAP and a matrix of weights called **kernel**. After that a convolution is performed, the matrix is shifted of a value equal to the stride(10.4).
- the **pooling layer** records one single value for each portion of the input matrix analyzed. There are several types of these layer, from the maximum layer to the average one(10.4).

- the last layer is the NN itself. It receives as input the image processed trough the convolutional and pooling layers

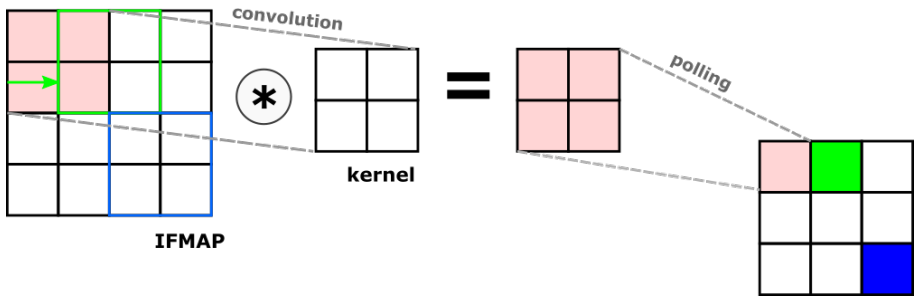


Figure 10.4: Neural Network's neuron

Implementing a CNN in hardware would be very tough because of its complexity. The idea is to introduce some approximations: the **binarized NNs** (BNNs) limits the possible values of both weights and input pixels to logic '0' and logic '1'. There are several models in literature. [6] analyzes several of them: the best one is the **XOR-Net**. There are two reasons: it has an high accuracy and the convolution operation is very simple to be implemented in hardware. The model of the net is 10.1, where $\circledast$ is binary convolution.

$$Xnor - net \approx (IFMAP \circledast Kernel) \cdot Kxa \tag{10.1}$$

[13] proves that the binary convolution can be implemented as shown in 10.2, where X_i is the portion of the IFMAP that is used for each convolution, L the length of the word an W the kernel matrix. This is the operation implemented by the architectures below.

$$2 \cdot 1count(\overline{X_i \oplus W}) - L \qquad i \quad in[1,n] \tag{10.2}$$

10.5.2 Implementation in LiM

The algorithm was implemented with both of the two templates in order to explore the whole design space.

V1

8.4.4 explains the design flow for the construction of a LiM array V1 that is able to manage six convolutions. However, a more generic approach it needs to be adopted. Two variables have to be fixed: the size of the memory and how to map the input matrices on the array. Indeed, 8.4.4 just explains in which rows storing the operands but there is no mention about matrices. 10.5 shows how the matrices have to be stored in the array: they are converted in vectors and then stored in the array. So, the complexity is spread in depth.

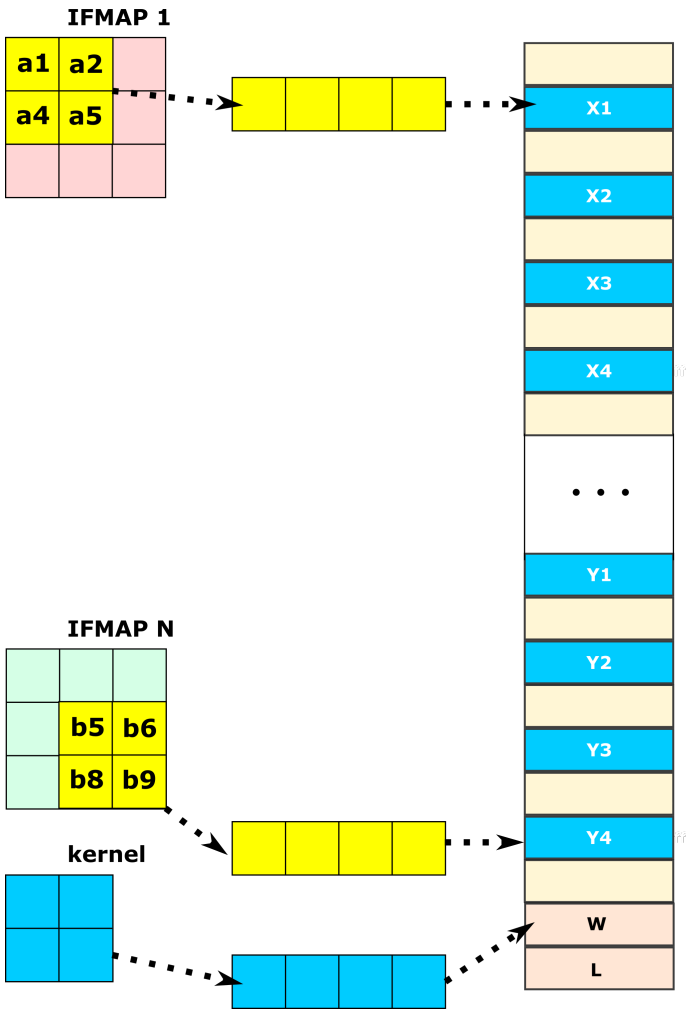
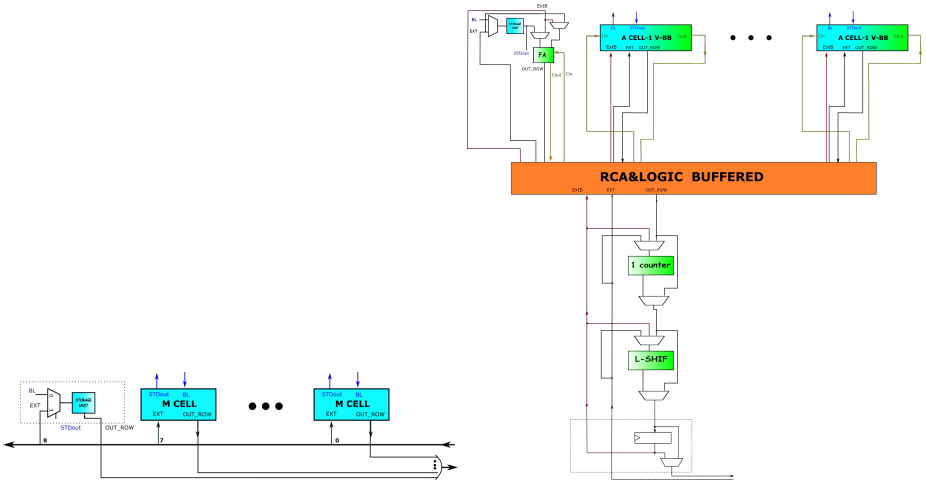


Figure 10.5: algorithm to array mapping

About memory size,10.5 shows that : first, the parallelism depends on the area

of kernel, second, the number of smart rows depends on how many convolutions have to be performed in parallel. That depends on both the number of IFMAPs to analyzes in parallel and the size of each of them. 10.6 shows how the rows of the LiM array looks like.



(a) (b)

Figure 10.6: Rows of the LiM array V1. (a) the standard row and (b) the smart row.

V2

8.4.4 explains how template V2 is not optimal for this algorithm since just one kind of operation has to be performed in parallel. However, just for figuring out the difference between the two versions, the algorithm is mapped also in a template V2. The design flow is the same:

1. Operations required are: xnor, difference, one counting and left shift.
2. W,L are share operands, so they have to be placed in the standard far rows
3. The X_i could be placed both either in the upper neighbourns, in the lower neighbourns or in the smart row
4. The flow is : xnor $--> $ one counting $--> $ left shift $--> $difference

In order to exploit the multi-row SIMD feature of V2, the idea is to split the operation in two steps as shown in figure 10.7:

- The smart row 1 performs the operation until the left shift. The partial result is stored in the input buffer of standard row. The only external operand is the W
- The smart row 2 performs the difference between the partial result stored in the input buffer and the L.

Actually, before the second step a LOAD operation is required: it cannot be used to external operands. So, the partial result is brought inside the input buffer of smart row 2 before performing the difference. So, since the two rows have to be the same buffer, an IO buffer must be used.

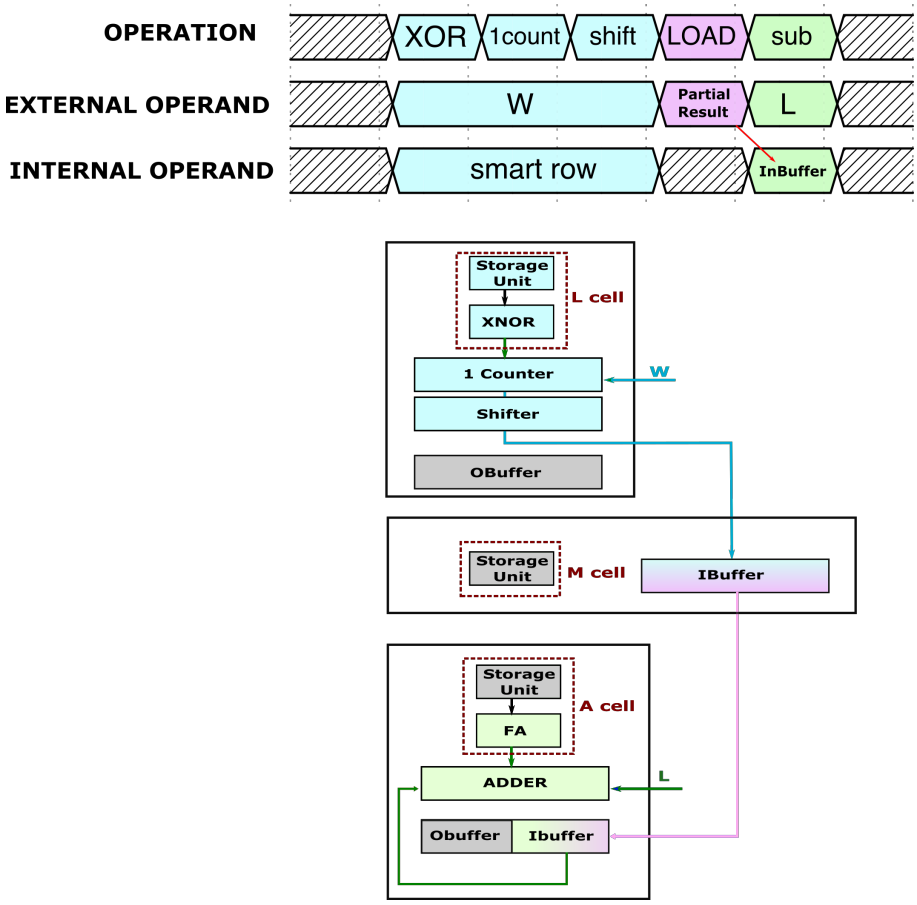
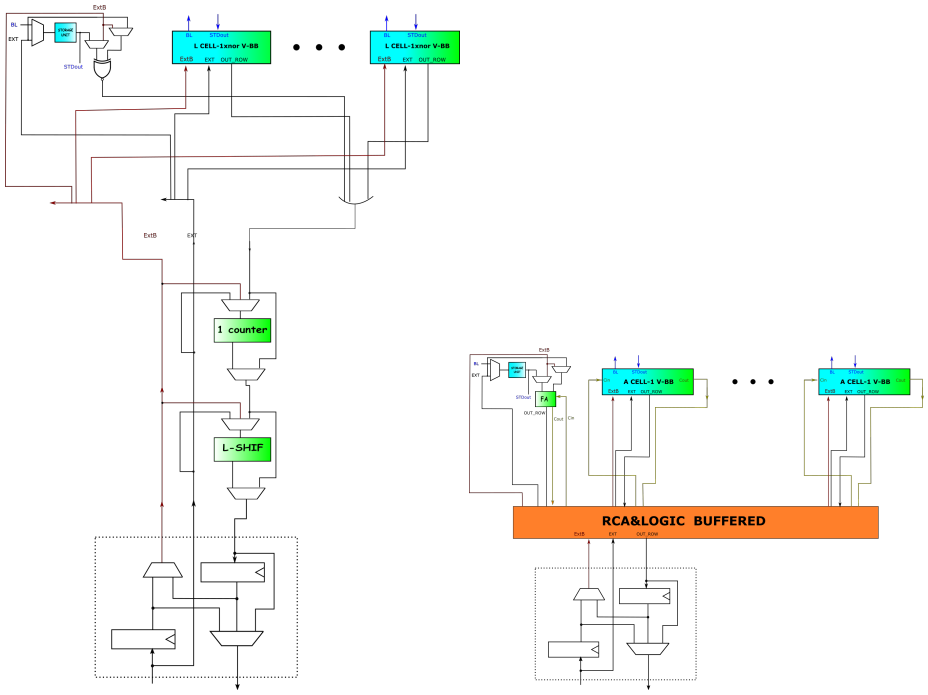


Figure 10.7: Schedule of the operations

?? shows how the row of the LiM array looks like. Notice that the adder is just a RCA.



(a) (b)

Figure 10.8: Rows of the LiM array V2. (a) the smart row 1 and (b) the smart row 2.

10.5.3 Implementation in DLX

The DLX is a sequential processor: this algorithm requires a lot of steps to be implemented. The most complicated section to implement is the one counting. The pseudo code implementing it is shown below

Algorithm 1: one Count algorithm

```
begin
    int oneN=0;
    #check all the bits of the word trough the LSB;
    for i ∈ [0,31] do
        lastb=x and 1 #extracting the LSB;
        oneN+=lastb #if 1 increment the number of 1s;
        x=x>>1 #shift out the LSB just checked;
    return oneN;
```

Only one convolution was implemented on the DLX since 126 instructions are required just for it.

10.5.4 Results

The three architectures were synthesized with the same time constrains : clock with a period of 7 ns, skew of 0.07 ns and both input and output delays of 0.5 ns. The idea for having a fair comparison was to sweeps together the data memory of DLX and the LiM array: if the the parallelism of the array is kept fixed at 32 bits, the number of rows sweeps from 8 to 512. The performance before the place & route are recorded following the approach described in appendix A. The comparison was evaluated in terms of area, total power, execution time and bandwith, that is how many instructions are performed(10.9 and 10.1).

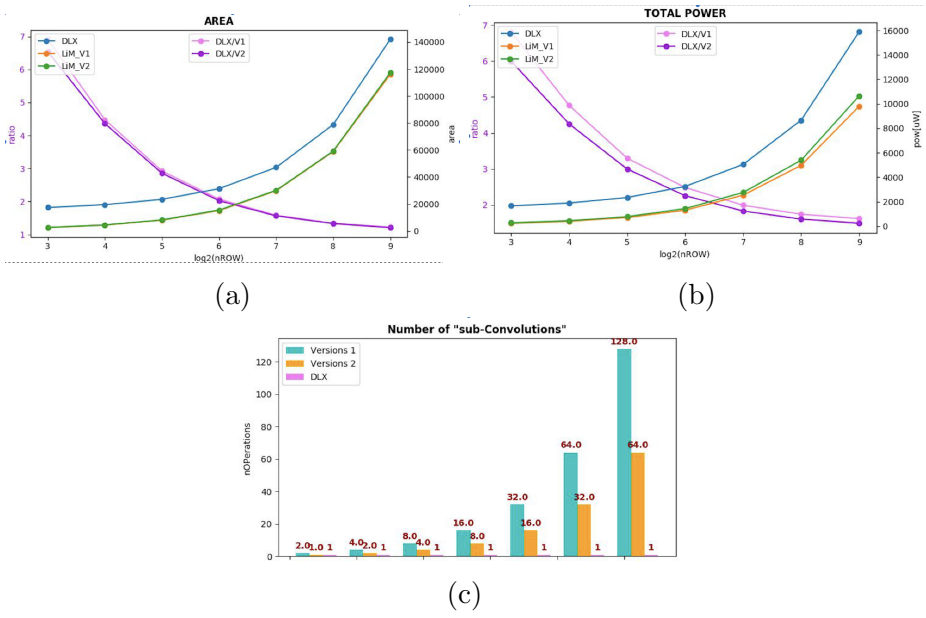


Figure 10.9: Comparison between DLX and LiM array pre P&R: area(a), power(b) and bandwidth(c)

Table 10.1: EXECUTION TIMES DLX VS LIM

Execution Time		
LiM V1	**LiM V2**	**DLX**
14 ns	21 ns	3.5 us

The result is clear. Since the algorithm is highly parallelizable, the advantage of a SIMD ISA is huge: less power, less Area and much less time for managing a lot more operations. More accurately, choosing the LiM units leads to an 85% reduction of both the power consumption and area occupation for an 8 rows memory and a reduction of 40% for the power consumption with a memory of 512 rows. Moreover, on one hand, the LiM Units are able to implement a number of convolutions in parallel that is proportional to the number of smart rows; on the other hand, the DLX can implement just one convolution for each memory size. In other words, it would be unfeasible to manage 128 convolutions with a DLX: almost sixteen thousand operations were required. Moreover, the execution time of the LiM array is independent from the number of convolutions to perform: how many operations are executed in parallel just depends on the smart section size. On the other hand, the execution time of the DLX strongly depends on both the number of operation and on the word size because of its sequential nature.
About the instruction set, they are similar. Appendix D shows the three instruction set. 10.10 shows comparison repeated after the P&R: it leads to the same conclusion.

A further step has been repeating the comparison with a more recent technology: the 28nm. 10.11 underlines how the situation doesn't change: there are no differences with the results obtained with the 45nm based implementations. It happens because what 10.11 analyzes is the ratio between the performance of two architectures, either the DLX and the V1 or the DLX and the V2. In other words, the advantages lead by the technology shrinking(which are underlined in 10.13) are canceled by the ratio itself. However, 10.13 shows the benefits of a smaller technology: faster gates, smaller power consumption and lighter area impact.

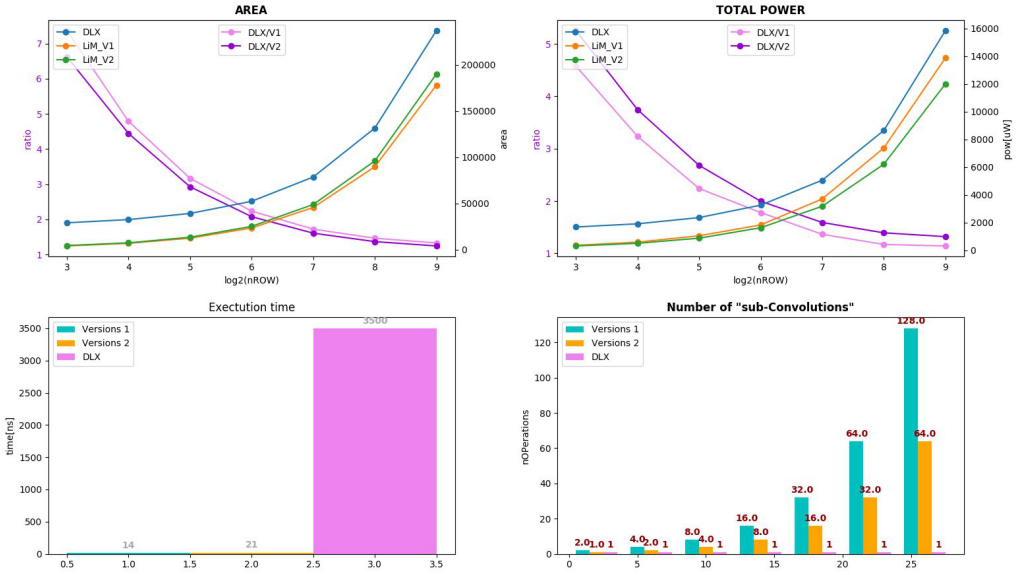
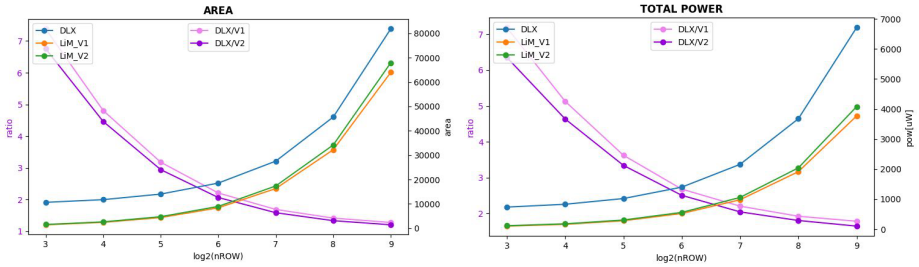


Figure 10.10: Comparison between DLX and LiM array after the P&R



(a) (b)

Figure 10.11: Comparison between DLX and LiM array pre P&R with 28nm: area(a) and power(b)

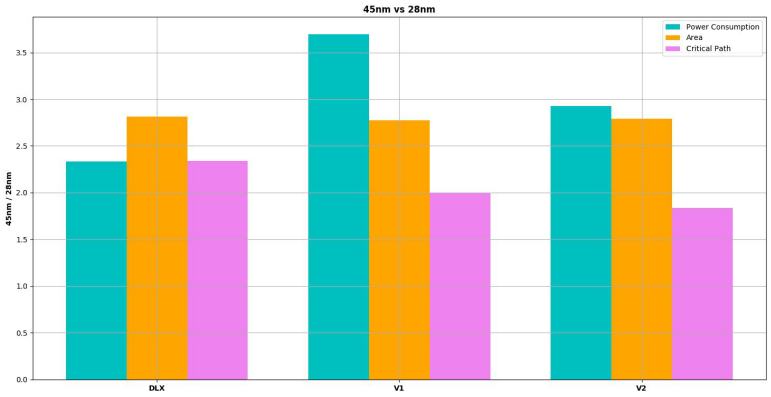


Figure 10.12

Figure 10.13: Comparison between 45nm based implementations and ones based on 28nm technology

Chapter 11

Conclusion

The logic in memory is the approach explored to overcome the Von Neumann's bottleneck. Basically the idea is to spread the computation between the memory and the CPU. In other words, the concept of the memory array has totally been changed: from a storage unit, it has became a smart-unit able to perform operations directly inside it. This thesis work thinks outside the box of the standard approaches that can be find in literature. The LEGO LiM unit is a modular, programmable device able to collaborate with the processor for executing data intensive algorithms. It's a unit able to perform SIMD operations. Once the designer has built its own LEGO LiM Unit exploiting the LEGO LiM library, the smart array can be used either as a standard memory array or for performing SIMD operations. The smart features are totally transparent for the CPU. There are three building blocks in common to all the smart array: the front-end, the back-end and the LiM array. The former manages the interface with the world outside the LEGO LiM unit: both the processor and the scheduler. The back-end translates the high-level instructions in configuration signals for the LiM array. The LiM array is the smart array itself: it has both smart and storage features. It's made up by three families of LEGO LiM acting at different levels of granularity: from the word down to the bit. The handbook(chapter 8) guides the designer to build up his customized LEGO LiM ecosystem.

The second portion of this thesis work evaluates the performance of the LEGO LiM Units. This evolves in two directions: evaluating the overhead of the smart features and comparing the LEGO LiM ecosystem with a standard sequential processor(DLX).
The first step underlines the significant impact of the smart features on the performance of the array: giving a huge instruction set to the memory array causes an overhead on both the area occupation and the power consumption. An array able

to handle complex SIMD operations can't consume as an array that is just able to store words. If the smart features almost duplicate the power consumption , the impact of the CUs weakens itself increasing the memory size: it's almost negligible on array with more that 512 rows.
Now the question is if this overhead is worth it. The result obtained from the comparison with the DLX is clear: there is a huge advantage due to the parallel nature of the test-bench used(a B-CNN). Choosing the LEGO LiM Unit allows to consume much less power for performing simultaneously more than one hundred times the operations performed by the RISC-like process. Moreover, the execution time required by the DLX for performing one convolution is an order of magnitude bigger that the times spent by the smart array for performing N convolution in parallel, where N is the number of smart rows. Notice that if the execution time of the sequential processor increases with the number of convolutions to perform, for the LiM array it remains constant: this is the advantage of a parallel computation over a sequential execution. The **Landau's Law**([11]) allows to formalize this performance gap. It helps to find out the speedup due to a an improvement of a certain unit of an architecture: it's the memory array in this case. With a memory size of $32x512$ bits, the speedup factor is over 99.

The future work could evolve in two directions: technological and architectural. The library of components should be updated with new LEGO LiM. The library characterization should be performed again with a more recent technology, since it has been evaluated with a 45 nm CMOS technology, which is quite dated .

Appendix A

Run-time SIMD

A new interesting LiM arrays feature has been implemented: the run-time decision of the SIMD's degree. The smart section is divided in blocks(A.1), each one containing the same number of smart rows. Then, the user can decide which blocks enabling for each instructions. The blocks not activated don't perform any instructions. This approach avoids unwanted operations, that means useless power consumption and unwanted memory writings.
Configuring the blocks is very easy. An additional field is included in the uInstruction : **BlockEN**. This signal is n-bits wide, where n is the number of blocks. Each bit of this signal controls a block: if the n-th bit is asserted, the n-th will be activated for that operation.

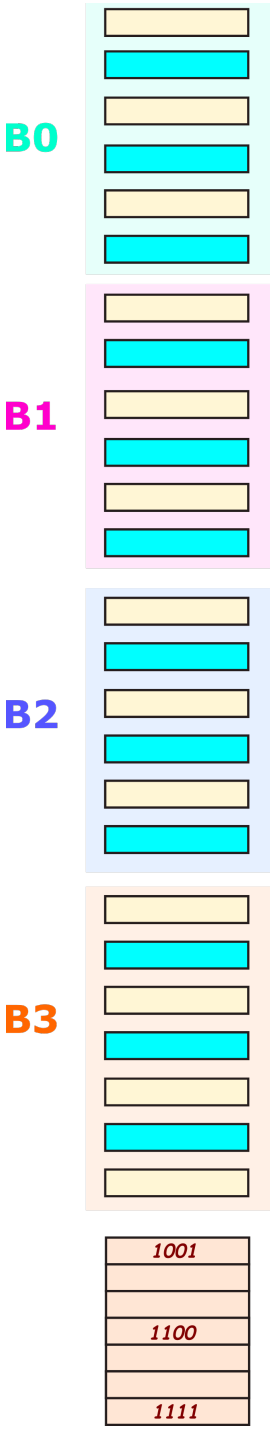


Figure A.1: LiM array with 4 blocks, each one with 3 smart rows.

Appendix B

User Manuals

Building a LiM array from scratch could be very challenging and timing consuming for someone different from me, who invented this approach. So, in order to make the design flow of a LiM array much more user-friendly, the idea is to provide two ready-to-use arrays with two manuals for each one of them:

- a ISA leading the user to write a uPrograms for the chosen array
- a guide allowing the user to customize the template according to the specifications. It guides step by step the user in the whole process, specifying which files and which signals have to me modified. The degrees of freedom are: the type of LiM Cell, the number of RIs and the type of RIs.

So, the user can either directly use one of the arrays or customize it's own array starting from one of the two templates. The key difference between the two provided arrays is that the **arrayLogic** array has to be selected if the specifications require LCells. On the other hand, the array to use is the **arrayArithmetic** , which contain ACells.
The structure of both the smart feature is the same:

- **RIs**: 1s' Counter and IN-OUT buffer;
- **Blocks**: 5 blocks;
- **Memory size**: n bits, n smart Rows, 7 standard far rows.

Appendix C

Performance evaluation

C.1 Simulation

The performance estimations are performed by combining three tools: Modelsim (Altera), Design Vision (Synopsys) and innovus(Cadence).
A first step allows to record area occupation and power estimation after the synthesis: both Modelsim and Design Vision perform that. The second step records the area occupation, the interconnections length and the power estimation after the place and route: both Modelsim and Innovus are required. Both the power estimations are performed with the **back-annotation** approach.
All the steps are managed by bash scripts that are completely generic. The wrapper is the main.sh(figure xx). It receives three information : the entity to analyze, the clock to use and the parameter to swap. Indeed, also parametric simulation can be performed: in each iteration, the wrapper

I adapts vhdl files to the value of the variable

II runs the first flow of simulations (modelsim & synopsys)

III runs the second flow of simulations (modelsim & innovus)

IV collects the results

The technology library used is the 45 nm.

C.2 Back Annotation

The key idea of the back annotation (BA) is to record the power consumption including fair switching activities of netlist's nodes. It means that the netlist is

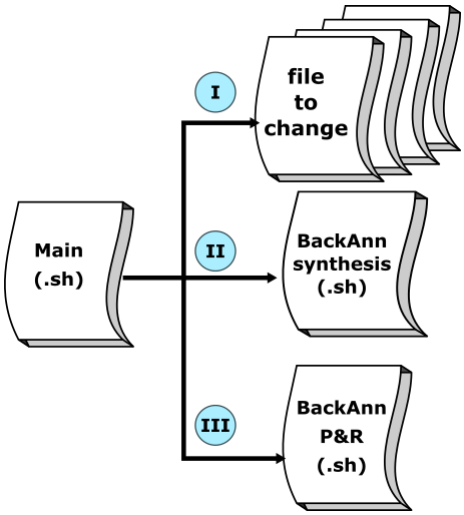


Figure C.1: A figure with the maximum width you can use

stimulated with a testbench for recording the activity of the nodes first; then, the power performance is estimated.
Two BA analysis are performed: one with the netlist of the synthesized architecture and another one with the netlist obtained after the place and route. Notice that they are very similar.

C.2.1 BA after synthesis

Three steps are required: static synthesis, switching activity annotation and power estimation. The first step is managed by Design Vision. Two files are generated: the netlist of the synthesized architecture and the file with the delay of the netlist (.sdf).
During the second step, Modelsim stimulates the synthesized netlist with a proper testbench. Then, the activities of the nodes are recorded in a file (.vcd). For this step two files are required: the .sdf and the compiled model of the cells in the technology library.
Then, Design Vision reads the performs again the power estimation but the real switching activity are taken into account. Actually, since Design Vision can't read .vcd file, it has to convert in a .saif fail before.

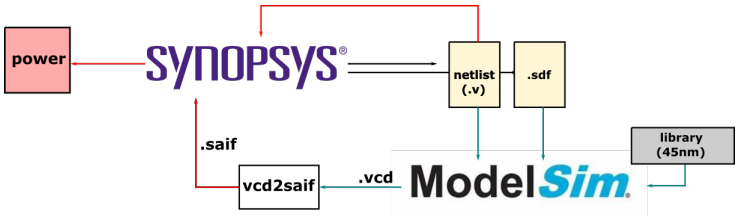


Figure C.2: low of the operations required for the BA after synthesis . Arrows of the same color belongs to the same step

C.2.2 BA after Place & Route

There are only two differences with the BA done after synthesis: the starting point is the synthesized architecture, Innovus is used instead of Design Vision and Innovus can read .vcd file.

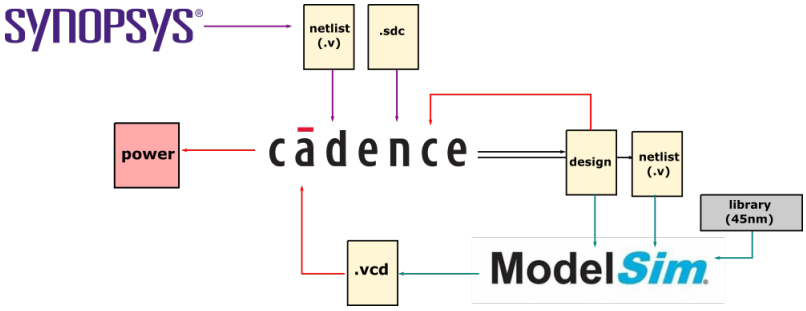


Figure C.3: low of the operations required for the BA after P&R. Arrows of the same color belongs to the same step

Appendix D

Characterization results

D.1 characterization

The Lego LiM have to be characterized in order to have useful data for the power estimators. The results of the characterization were the Static power consumption, the average dynamic power consumption, the Area occupation and the interconnection length for each block. The performances were estimated through the back-annotation after place&Route approach explained in the appendix A. All the blocks of the same family are stimulated with the same clock frequency.

D.1.1 Average Dynamic Power

The dynamic power consumption is obtained as the average of the results obtained with 4 different simulations: in each one of these the component is stimulated with a different algorithm. The **Idle0** (**Idle1**) algorithm forces the output to logic 0 keeping in idle the storage unit for cells and the logic for RIs. The **write01** (**write01**) stimulates the blocks in order to switch the output from logic 0(1) to logic 1(0) in the worst case. So, one hand the storing unit is written and its output is propagating for cells; on the other hand, the logic unit of RI is fed in order to properly switch the output. Moreover, each RI is analyzed four times, each time with a different parallelism: 8, 16, 32 and 64 bits.

D.1.2 Storage Unit

Since a memory cell can't be synthesised with Synopsys, it's described as shown in figure D.1. It's a flip flop with read and write enable and it works like a standard memory cell: no operation can be done with the word line not asserted.

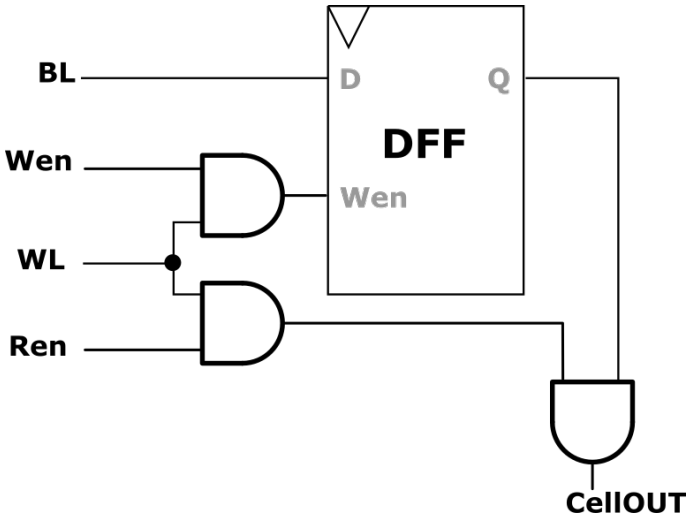


Figure D.1: Storage Unit of LiM cells

D.1.3 Characterization File

An example of the characterization files obtained for some bricks of the LegoLiM is shown below.

#----------------------------------#
#--   LiM Cells   --#
#----------------------------------#

#CellName	avStaticPow[uW]	avDynamicPow[uW]	avTotalPow[uW]	Area[um^2]	AreaNoPH[um^2]	INTlenght[um]	Critical Path[ns]
Acell_V1	0.295772	1.837352	2.133125	27.930000	15.428000	120.600000	0.24000
Acell_V1_F	0.365515	2.809953	3.175467	36.176000	19.684000	161.425000	0.26000
Acell_V1_M	0.334820	2.070215	2.405035	31.920000	17.822000	154.155000	0.30000
Lcell_V1and	0.229243	0.811222	1.040465	21.546000	11.438000	86.770000	0.11000
Lcell_V1and_B	0.305390	1.162578	1.467968	29.526000	16.226000	130.525000	0.17000
Lcell_V1and_F	0.267363	0.989843	1.257205	25.536000	13.832000	100.570000	0.12000
Lcell_V1and_F_B	0.337168	1.342868	1.680035	32.984000	18.088000	149.595000	0.18000
Lcell_V1and_M	0.254368	0.057572	0.311940	25.536000	13.832000	100.075000	0.19000
Lcell_V1and_M_B	0.346128	1.468922	1.815050	32.984000	18.620000	144.060000	0.25000
Lcell_V1and_nExt	0.252022	0.906197	1.158220	23.940000	13.034000	103.915000	0.13000
Lcell_V1and_nINs	0.305972	0.892530	1.198503	26.334000	14.364000	113.800000	0.17000
Lcell_V1and_nOUT	0.256215	0.989735	1.245950	23.142000	12.768000	101.415000	0.16000
Lcell_V1and_nSUout	0.252920	1.072512	1.325433	23.940000	13.034000	105.115000	0.18000
Lcell_V1or	0.221635	0.880370	1.102005	21.546000	11.438000	87.330000	0.12000
Lcell_V1or_B	0.298030	1.156595	1.454625	29.526000	16.226000	126.545000	0.19000
Lcell_V1or_F	0.264575	1.068540	1.333115	25.536000	13.832000	108.205000	0.13000
Lcell_V1or_F_B	0.313642	1.281278	1.594920	32.984000	17.024000	147.265000	0.19000
Lcell_V1or_M	0.266468	1.168832	1.435300	25.536000	13.832000	109.195000	0.21000
Lcell_V1or_M_B	0.342740	1.490660	1.833400	32.984000	18.620000	145.860000	0.27000
Lcell_V1or_nExt	0.248298	0.759340	1.007637	23.940000	13.034000	103.635000	0.15000
Lcell_V1or_nINs	0.291845	0.870450	1.162295	26.334000	14.364000	112.850000	0.16000
Lcell_V1or_nOUT	0.252978	1.013422	1.266400	23.142000	12.768000	100.655000	0.17000
Lcell_V1or_nSUout	0.247657	1.080893	1.328550	23.940000	13.034000	104.305000	0.19000
Mcell_V0	0.177255	0.723002	0.900257	15.428000	9.044000	64.960000	0.08000
Mcell_V1	0.199497	0.820213	1.019710	19.152000	10.374000	85.730000	0.08000

#----------------------------------#
#--   Row Interfaces   --#
#----------------------------------#

#CellName	avStaticPow[uW]	avDynamicPow[uW]	avTotalPow[uW]	Area[um^2]	AreaNoPH[um^2]	INTlenght[um]	CPath[ns]
InOut_Buffer_8	2.657220	24.228920	26.886140	254.562000	151.620000	1382.820000	0.23000
InOut_Buffer_16	5.289045	48.824382	54.113428	502.740000	300.580000	2804.150000	0.29000
InOut_Buffer_32	10.540553	98.920995	109.461548	1058.414000	597.170000	7451.930000	0.35000
InOut_Buffer_64	20.826780	201.265217	222.091997	2110.976000	1187.690000	18174.510000	0.44000
In_Buffer_8	1.204980	10.874345	12.079325	113.582000	68.096000	673.530000	0.17000
In_Buffer_16	2.397848	22.194063	24.591910	226.100000	135.128000	1781.800000	0.23000
In_Buffer_32	4.722145	45.289753	50.011898	470.820000	267.862000	4393.650000	0.29000
In_Buffer_64	9.380290	91.905637	101.285927	938.448000	533.330000	10982.200000	0.38000
L_shifter_8	0.399757	6.674415	7.074172	51.870000	29.792000	435.810000	0.17000
L_shifter_16	0.802720	13.635145	14.437865	102.410000	60.648000	1034.970000	0.23000
L_shifter_32	1.597985	28.713675	30.311660	234.080000	120.232000	2700.810000	0.29000
L_shifter_64	3.202865	60.353450	63.556315	470.820000	239.932000	6655.520000	0.38000
L_shifter_B_8	0.609090	9.415815	10.024905	78.204000	45.220000	742.630000	0.24000
L_shifter_B_16	1.208022	19.860748	21.068770	153.216000	90.972000	1749.840000	0.30000
L_shifter_B_32	2.403353	41.390210	43.793563	352.716000	180.082000	4528.600000	0.36000
L_shifter_B_64	4.826995	86.696580	91.523575	703.836000	360.962000	11128.990000	0.45000
Out_Buffer_8	1.193623	11.129252	12.322875	113.582000	68.096000	684.340000	0.17000
Out_Buffer_16	2.378625	22.541200	24.919825	226.100000	135.128000	1748.630000	0.23000
Out_Buffer_32	4.712710	45.822905	50.535615	470.820000	268.128000	4174.220000	0.29000
Out_Buffer_64	9.322398	92.933380	102.255777	938.448000	533.330000	10184.130000	0.38000
RCA1_8	0.430925	7.532617	7.963542	48.944000	29.260000	561.605000	0.21000
RCA1_16	0.829252	15.389275	16.218528	95.760000	56.924000	1418.505000	0.27000
RCA1_32	1.661202	32.903170	34.564373	210.140000	112.784000	3650.795000	0.29000
RCA1_64	3.326283	68.292348	71.618630	417.088000	225.036000	9451.895000	0.38000
RCA1_B_8	0.430925	7.717767	8.148692	48.944000	29.260000	702.855000	0.21000
RCA1_B_16	0.829252	16.075582	16.904835	95.760000	56.924000	1724.915000	0.27000
RCA1_B_32	1.661202	32.535297	34.196500	210.140000	112.784000	4300.495000	0.29000
RCA1_B_64	3.326283	73.407250	76.733533	417.088000	225.036000	11577.455000	0.38000
RCAandLogic_8	0.626132	9.845115	10.471248	78.204000	42.826000	709.190000	0.21000
RCAandLogic_16	1.238895	21.136340	22.375235	155.344000	85.386000	1603.090000	0.28000
RCAandLogic_32	2.507147	45.369980	47.877128	342.342000	171.304000	4705.320000	0.37000
RCAandLogic_64	5.028742	92.105257	97.134000	684.684000	343.672000	9989.140000	0.38000
RCAandLogic_B_8	0.626132	10.178513	10.804645	78.204000	42.826000	923.790000	0.21000
RCAandLogic_B_16	1.238895	20.977430	22.216325	155.344000	85.386000	2097.240000	0.28000
RCAandLogic_B_32	2.507147	44.461280	46.968428	342.342000	171.304000	5524.415000	0.37000
RCAandLogic_B_64	5.028742	93.432685	98.461428	684.684000	343.672000	13739.480000	0.38000
R_A_shifter_8	0.411693	6.872212	7.283905	51.870000	30.590000	449.860000	0.17000
R_A_shifter_16	0.802918	13.933972	14.736890	102.410000	60.648000	1047.830000	0.23000
R_A_shifter_32	1.604052	28.998312	30.602365	234.080000	120.498000	2691.980000	0.29000
R_A_shifter_64	3.215145	60.967955	64.183100	470.820000	240.996000	6680.300000	0.38000
R_A_shifter_B_8	0.620730	9.809023	10.429753	78.204000	46.284000	751.130000	0.25000
R_A_shifter_B_16	1.208220	20.227002	21.435222	153.216000	90.972000	1751.020000	0.31000
R_A_shifter_B_32	2.415633	41.993125	44.408758	352.716000	181.146000	4480.480000	0.37000
R_A_shifter_B_64	4.830468	87.197438	92.027905	703.836000	361.494000	11101.300000	0.45000
oneCounter_8	0.920508	14.582517	15.503025	98.686000	56.392000	642.700000	0.71000
oneCounter_16	1.982077	27.644578	29.626655	212.800000	122.094000	1561.830000	1.21000
oneCounter_32	4.127763	52.983272	57.111035	494.760000	263.340000	3399.850000	1.93000
oneCounter_64	9.310655	143.664775	152.975430	1064.532000	576.422000	11749.560000	2.74000
oneCounter_B_8	1.001475	13.160255	14.161730	124.754000	70.224000	902.560000	0.86000
oneCounter_B_16	2.150650	27.796688	29.947337	263.340000	151.886000	2301.570000	1.42000
oneCounter_B_32	4.760580	58.486400	63.246980	633.080000	326.382000	5977.640000	1.92000
oneCounter_B_64	11.100487	161.615178	172.715665	1283.450000	701.974000	14536.050000	2.77000

Appendix E

Instruction Set

Portion of vhdl sources are reported below for comparing the instruction sets of the three architectures analyzed: DLX, V1 and V2.

```
-------------------------------------------------------------------------
-------------             INSTRUCTIONS DLX                      ------
-------------------------------------------------------------------------
   ----------------
   --------OP codes

    constant RTYPE       : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "000000" ;    -- RTYPE
    constant JTYPE_J     : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "000010" ;    -- X02 J
    constant JTYPE_JAL   : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "000011";    -- X03 Jal
    constant ITYPE_BEQ   : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "000100" ;    -- X04 beqz
    constant ITYPE_BNEQ  : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "000101" ;    -- X05 benqz
    constant ITYPE_ADD   : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "001000" ;    -- X08 addi
    constant ITYPE_SUB   : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "001010" ;    -- X0a subi
    constant ITYPE_AND   : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "001100" ;    -- X0c andi
    constant ITYPE_OR    : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "001101" ;    -- X0d ori
    constant ITYPE_XOR   : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "001110" ;    -- X0e xori
    constant ITYPE_SL    : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "010100" ;    -- X14 slli
    constant NOP    : std_logic_vector(OP_CODE_SIZE - 1 downto 0)       :=
          "010101" ;    -- X15 nop
    constant ITYPE_SR    : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "010110" ;    -- X16 srli
    constant ITYPE_NE    : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "011001";    -- X19 snei
    constant ITYPE_LE    : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
          "011100" ;    -- X1c slei
```

```
22     constant ITYPE_GE    : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
           "011101" ;    -- X1d sgei
       constant ITYPE_LW    : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
           "100011" ;    -- X23 lw
24     constant ITYPE_SW    : std_logic_vector(OP_CODE_SIZE - 1 downto 0)  :=
           "101011" ;    -- X2b sw

26    ---------------------------------
      --------FUNC(for RTYPE)
28     constant funcSL  : std_logic_vector(FUNC_SIZE - 1 downto 0) := "00000000100" ;
           -- x04 sll
       constant funcSR  : std_logic_vector(FUNC_SIZE - 1 downto 0) := "00000000110" ;
           -- x06 srl
30     constant funcADD : std_logic_vector(FUNC_SIZE - 1 downto 0) := "00000100000" ;
           -- x20 add
       constant funcSUB : std_logic_vector(FUNC_SIZE - 1 downto 0) := "00000100010" ;
           -- x22 sub
32     constant funcAND : std_logic_vector(FUNC_SIZE - 1 downto 0) := "00000100100" ;
           -- x24 and
       constant funcOR  : std_logic_vector(FUNC_SIZE - 1 downto 0) := "00000100101" ;
           -- x25 or
34     constant funcXOR : std_logic_vector(FUNC_SIZE - 1 downto 0) := "00000100110" ;
           -- x26 xor
       constant funcNE : std_logic_vector(FUNC_SIZE - 1 downto 0) :=  "00000101001" ;
           -- x29 sne
36     constant funcLE : std_logic_vector(FUNC_SIZE - 1 downto 0) :=  "00000101100" ;
           -- x2c sle
       constant funcGE : std_logic_vector(FUNC_SIZE - 1 downto 0) :=  "00000101101" ;
           -- x2d sge
38
------------------------------------------------------------------------
40 -------------            INSTRUCTIONS V1                      ------
------------------------------------------------------------------------
42    ----------------
      --------OP codes
44    constant logicArith              : std_logic_vector(2 downto 0):= "000";
      constant shift                   : std_logic_vector(2 downto 0):= "001";
46    constant count1                  : std_logic_vector(2 downto 0):= "010";
      constant count1_shift            : std_logic_vector(2 downto 0):= "011";
48    constant logicArith_shift        : std_logic_vector(2 downto 0):= "100";
      constant logicArith_count1       : std_logic_vector(2 downto 0):= "101";
50    constant logicArith_count1_shift : std_logic_vector(2 downto 0):= "110";
      constant store                                : std_logic_vector(2 downto 0):=
          "111";
52


54    ---------------------------------
      --------FUNC(for logicArith)
56    constant SUM      : std_logic_vector( 3 downto 0):=  "0001";
      constant SUB1     : std_logic_vector( 3 downto 0):=  "0010";
```

```
58    constant SUB2          : std_logic_vector( 3 downto 0):=  "0011";
      constant SUB3          : std_logic_vector( 3 downto 0):=  "0100";
60    constant XOR1      : std_logic_vector( 3 downto 0):=  "0101";
      constant AND1      : std_logic_vector( 3 downto 0):=  "0110";
62    constant OR1           : std_logic_vector( 3 downto 0):=  "0111";
      constant OR2       : std_logic_vector( 3 downto 0):=  "1000";
64    constant OR3       : std_logic_vector( 3 downto 0):=  "1001";
      constant OR4       : std_logic_vector( 3 downto 0):=  "1010";
66    constant XNOR1     : std_logic_vector( 3 downto 0):=  "1011";
      constant XNOR2     : std_logic_vector( 3 downto 0):=  "1100";
68    constant XNOR3     : std_logic_vector( 3 downto 0):=  "1101";
      constant XNOR4     : std_logic_vector( 3 downto 0):=  "1110";
70    constant nullOP        : std_logic_vector( 3 downto 0):=  "0000";

72


74  ------------------------------------------------------------------------
    -------------            INSTRUCTIONS V2                        ------
76  ------------------------------------------------------------------------
      ----------------
78    --------OP codes
      --mode 1row
80    constant Arith                                : std_logic_vector(4 downto 0):=
            "00001";
      constant XNOR1                                : std_logic_vector(4 downto 0):=
            "00010";
82    constant shift                                : std_logic_vector(4 downto 0):=
            "00011";
      constant count1                               : std_logic_vector(4 downto 0):=
            "00100";
84    constant count1_shift                         : std_logic_vector(4 downto 0):=
            "00101";
      constant XNOR1_shift                          : std_logic_vector(4 downto 0):=
            "00110";
86    constant XNOR1_count1                         : std_logic_vector(4 downto 0):=
            "00111";
      constant XNOR1_count1_shift                   : std_logic_vector(4 downto 0):=
            "01000";
88    --mode 2row
      constant Arith_&_XNOR1                        : std_logic_vector(4 downto 0):=
            "01101";
90    constant Arith_&_shift                        : std_logic_vector(4 downto 0):=
            "01110";
      constant Arith_&_count1                       : std_logic_vector(4 downto 0):=
            "01111";
92    constant Arith_&_count1_shift         : std_logic_vector(4 downto 0):= "10000";
      constant Arith_&_XNOR1_1count         : std_logic_vector(4 downto 0):= "10011";
94    constant Arith_&_XNOR1_shift          : std_logic_vector(4 downto 0):= "10011";
      constant Arith_&_XNOR1_1count_shift   : std_logic_vector(4 downto 0):= "10100";
```

```
96    constant Store1                                : std_logic_vector(4 downto 0):=
          "01010";
      constant Store2                                       : std_logic_vector(4
          downto 0):= "01011";
98    constant Store1_2                              : std_logic_vector(4 downto 0):=
          "01100";
      constant Load1                                        : std_logic_vector(4
          downto 0):= "10101";
100   constant Load2                                        : std_logic_vector(4
          downto 0):= "10110";
      constant nop                                   : std_logic_vector(4 downto 0):=
          "00000";
102
      -------------------------
104   --------FUNCN(for Arith)
      constant SUM                                          : std_logic_vector( 2
          downto 0):=  "001";
106   constant SUB1                                         : std_logic_vector( 2
          downto 0):=  "010";
      constant SUB2                                                 :
          std_logic_vector( 2 downto 0):=  "011";
108   constant SUB3                                                 :
          std_logic_vector( 2 downto 0):=  "100";
      constant nullOP                                               :
          std_logic_vector( 2 downto 0):=  "000";
```

Bibliography

[1] Shaahin Angizi. "PIMA-Logic: A Novel Processing-in-Memory Architecture for Highly Flexible and Energy-Efficient Logic Computation". In: ().

[2] JOHN BACKUS. "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs". In: (1977).

[3] Ping Chi. "PRIME: A Novel Processing-in-memory Architecture for Neural Network Computation in ReRAM-based Main Memory". In: ().

[4] Woong Choi. "Content Addressable Memory Based Binarized Neural Network Accelerator Using Time-Domain Signal Processing". In: ().

[5] M. Cofano. "Logic-In-Memory: A NanoMagnet Logic Implementation". In: (2015).

[6] Andrea Coluccio. "Logic-in-Memory: is it worth it? A Binary Neural Network case of study". 2019.

[7] Wenqin Huangfu. "RADAR: A 3D-ReRAM based DNA Alignment Accelerator Architecture". In: ().

[8] Mohsen Imani. "Efficient Query Processing in Crossbar Memory". In: ().

[9] Joe Jeddeloh. "Hybrid memory cube new DRAM architecture increases density and performance". In: (2012).

[10] D.A. Patterson J.L. Hennessy. *Computer architecture: a quantitative approach, 6th edition*. Morgan Kaufmann, 2019. Chap. 2.

[11] D.A. Patterson J.L. Hennessy. *Computer architecture: a quantitative approach, 6th edition*. Morgan Kaufmann, 2019, pp. 49–51.

[12] Dae Hyun Kim. "Design and Analysis of 3D-MAPS (3D Massively Parallel Processor with Stacked Memory". In: (2015).

[13] Rastegari M. *Xnor-net: Imagenet classification using binary convolutional neural networks*.

[14] Giulia Santoro. “Exploring New Computing Paradigms for Data-Intensive Applications”. 2019.

[15] Vivek Seshadri. “Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology”. In: ().

[16] William Simon. “A Fast, Reliable and Wide-Voltage-Range In-Memory Computing Architecture”. In: ().

[17] Kai Yang. “Interleaved Logic-in-Memory Architecture for Energy-Efficient Fine-Grained Data Processing”. In: ().

[18] Leonid Yavits. “Resistive Associative Processor”. In: ().