POLITECNICO DI TORINO

Master's Degree
in Computer Engineering

Master's degree thesis

Assessing the Logic-in-Memory paradigm within a RISC-V framework

Thesis supervisor
prof. Mariagrazia Graziano
prof. Marco Vacca
prof. Maurizio Zamboni
firma dei relatori

. . . . . .
. . . . . .

Candidate
Francesca Silvano

firma del candidato

. . . . . .

Anno Accademico 2023-2024

Sommario

The growing gap between the performance of memories and processors over time has resulted in the well-known problem often called the "Memory-wall". This term refers to the performance degradation issues stemming from the classical von-Neumann computational paradigm, where the cpu and memory are separated. Especially for data-intensive applications, the continuous transfer of data between these two causes high power consumption and increased execution times. A possible solution to this problem is represented by the Logic in Memory (LiM) paradigm. The LiM architecture integrates computational elements inside the memory array, enabling the parallel processing of data and reducing data movement between the processor and memory.

This thesis considers a new type of LiM architecture called Mempa(General Purpose Logicin-Memory Architecture) pre-developed at the VLSI Laboratory of Politecnico di Torino. GPLiMA is a general-purpose LiM Architecture that integrates processing functionalities within each memory row, the ensemble of memory row and related processing unit is referred to as smart block. This also facilitates internal data transfer, eliminating the need for data to be fed to the processor. As a result, it aims at reducing the time the processor spends on data transfer and instruction execution, along with the associated power consumption.

This thesis work aims at integrating Mempa within a RISC-V based system using it as an accelerator connected to the processor, thus exploring the associated benefits and challenges. To enable the connection of Mempa to Ri5cy, two modules have been designed. A controller is needed to receive request signals from the Ri5cy and translate them into configuration and GPLiMA request signals, depending on whether it should be used as standard memory or as LiM. A switch used to multiplex signals from the Ri5cy to the memory or the Mempa based on the sent address. Thanks to the added modules, the processor can access Mempa as a standard memory with read and write operations, but it can also start the execution of an algorithm inside Mempa. Hence, a special Mempa address is reserved to serve as trigger for starting the execution.

Several tests have been conducted to evaluate the speed-up benefits provided by the integration of Mempa. Suitable algorithms have been run on both Ri5cy-only and Ri5cy+Mempa systems, thus allowing the assessment of such comparison.also by checking the execution of more complex algorithms like CNN LeNet-5 (Convolutional Neural Network). LeNet-5 was one of the early models of convolutional neural network algorithm that comprises 16x16 pixel grayscale images, the algorithms execute by the Mempa consists of the first layer of convolution.

Suitable algorithms have been run on both Ri5cy-only and Ri5cy+Mempa systems, thus

allowing the assessment of such comparison. This work paves the way for future advancements in LiM design and its practical application in real-world computing systems.

Ringraziamenti

At the end of this Master's thesis journey, I would like to express my heartfelt gratitude to my family, who has been there for me and made me feel loved throughout this entire process, even though we were separated by kilometers. A special thanks goes to my mom, who has been by my side through thick and thin in the past years.

I would also like to extend an important thank you to Edoardo, who has shown me that love and affection are not dependent on distance, but on the care and attention you put into building a relationship. A huge thank you also goes to Matteo and Riccardo, who have taught me that true friendship does not hinder you but accompanies you in your wildest ideas, sharing both tears and laughter. I would like to express my gratitude to Andrea as well, a loyal friend who has made these past years lighter and more enjoyable. A big, big thank you to Sara, who has always managed to find and communicate the positive side even amidst a thousand problems. I am grateful to all the members of the choir with whom I have shared many enjoyable moments and subsequent sore throats. Each one of them has helped me understand that sometimes taking moments for oneself and disconnecting is the best medicine.

Furthermore, I would like to thank the little friends that have been by my side. First and foremost, Milo, with whom I shared an understanding with just a single glance, and who was a valuable companion during the quarantine. Likewise, Cherie and Bernie, who have helped me study in the sweetest possible way. Last but certainly not least, a special thank you to Professors Marco Vacca and Mariagrazia Graziano, who introduced me to and guided me through this thesis journey. A tremendous thanks to Andrea Coluccio and Alessio Naclerio, who have supported me throughout the long term and during all the choices I have made.

Indice

Elenco delle tabelle

Elenco delle figure

Parte I

Background

Capitolo 1

Von Neumann Architecture and the Memory wall problem

The Von Neumann architecture was developed in the 1990s and represents the most widely used among hardware architecture types. Its characteristic feature is the sharing of memory space between data and executable instructions.
It comprises the following components:

- **Control Unit (CU):** A part of the system, known as the control unit, is responsible for managing and interpreting instructions stored in memory. This unit controls the flow of instructions, the sequence of execution, and read and write operations to memory.
- **Arithmetic Logic Unit (ALU):** The arithmetic logic unit performs arithmetic and logical operations required by instructions. The results are often stored in the central memory, whether they are data or instruction indices.
- **Input/Output Unit (I/O):** The input/output unit is a subsystem that provides the interface link to computer peripherals, enabling the exchange of information between the computer and the external world.
- **System Bus:** A system bus connects the various components of the computer, allowing the transfer of data and instructions between the control unit, arithmetic logic unit, and memory.
- **Central Memory:** The model includes central memory containing both data and instructions. This memory is organized sequentially, with each address associated with a unique memory cell. Central memory is not necessarily the only memory; multiple memories can exist, typically larger, and are viewed and used as I/O units.

This architecture [1.1] with its clear organization of instructions and data, has become a foundational concept for modern computers.

This architecture is straightforward to implement, but the use of a single bus for both data and instructions creates a bottleneck for transfer. This limitation has led to the

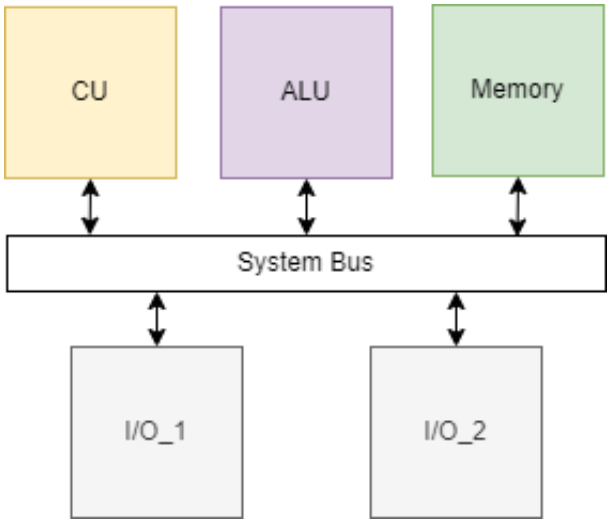


Figura 1.1. Von Neumann General scheme architecture

opposition of the Harvard architecture, which can enhance performance in scenarios where simultaneous access to instructions and data is crucial. Unlike Von Neumann, the Harvard architecture [1.2] incorporates two physically separate memories—one for data and one for instructions—along with two distinct buses. However, it may necessitate a more intricate and costly design. Consequently, it is primarily employed by specialized architectures, such as those utilized in certain microcontrollers and Digital Signal Processors (DSPs), aiming to optimize performance.

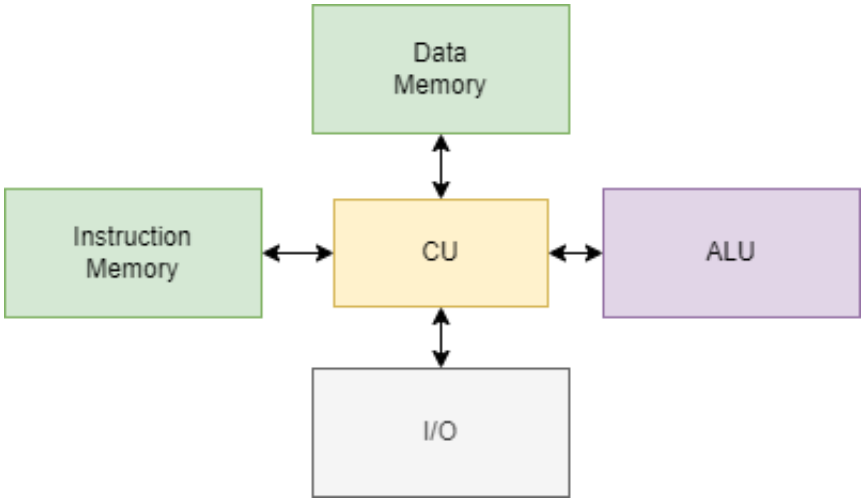


Figura 1.2. Harvard General scheme architecture

1.1 Memory Wall

The term "memory wall" refers to a performance challenge or limitation in a computer system arising from the disparity between the processing speed of the CPU and the access speed to the main memory (RAM). This speed gap can lead to a slowdown in system performance because the CPU must wait for data access from the main memory before proceeding with further operations.
In other words, while processor processing speeds have increased significantly over time, the access speed to main memory has not kept pace with the same rate of improvement. This discrepancy creates a performance "wall" as the CPU might be idle, waiting for data from memory, even though it can process instructions at a higher speed.
Several solutions and strategies have been proposed to address the memory wall challenge and improve overall system performance. Some key approaches include:

- **Cache Optimization:** The use of cache memory, which is a small, fast, and volatile type of memory located between the main memory and the CPU, helps reduce the time the CPU spends waiting for data. Frequently used data is stored in the cache, allowing quicker access.
- **Memory Hierarchy:** Implementing a memory hierarchy with multiple levels of memory (L1, L2, L3 caches, and main memory) helps bridge the speed gap. Each level of the hierarchy is larger but slower than the previous, allowing for a balance between speed and capacity.
- **Improved Memory Architectures:** Developing more advanced Dynamic Random-Access Memory (DRAM) technologies, such as DDR4 and DDR5, with higher data transfer rates and increased bandwidth, helps enhance memory performance.
- **Parallelism:** Leveraging parallelism in both hardware and software design allows multiple tasks or instructions to be processed simultaneously. This can help overlap memory access time with computational tasks.
- **Advanced Algorithms:**
  - Prefetching: Smart prefetching algorithms predict and fetch data from main memory before it is actually needed, reducing the time the CPU spends waiting for data.
  - Out-of-Order Execution: Allowing the CPU to execute instructions that are not dependent on the memory access concurrently with memory-bound instructions.
- **Non-Volatile Memory (NVM):** Exploring the use of non-volatile memory technologies, such as Flash or emerging memory types like Resistive RAM (RRAM) or Phase Change Memory (PCM), which may offer faster access times compared to traditional hard drives.
- **Hardware Accelerators:** Using hardware accelerators or specialized co-processors for certain tasks, reducing the reliance on the main memory for specific types of computations.

- **Memory Bandwidth Improvement:** Increasing the width of memory buses and channels to allow more data to be transferred simultaneously.

These solutions are often employed in combination, and ongoing research and advancements in hardware and software continue to address the challenges posed by the memory wall.

1.2 Logic in Memory (LiM)

The Logic in Memory (LiM) architecture is an approach that integrates logic operations directly within memory cells, eliminating the need to transfer data between memory and the processing unit for certain operations. This contrasts with traditional architecture, where data is moved between memory and the processing unit to perform operations.
In LiM architecture,simple operations such logical operations can be executed directly within the memory, leveraging the intrinsic parallelism of the memory cell matrix. This approach can significantly reduce the time and energy required to perform logical operations, as it avoids frequent data transfers through the system bus.
LiM architecture can lead to performance improvements in specific applications, especially those where logical operations are predominant. However, its effectiveness depends on the nature of the application and the characteristics of the workload. Designing a LiM system requires attention to managing logical operations within memory and addressing challenges associated with coexistence with other data processing operations.
The primary goal of LiM architecture is to enhance computational efficiency by reducing latency and energy associated with logical operations, thereby addressing challenges posed by the memory wall and improving overall system performance.

1.2.1 CGRA: Coarse Grain Re-configurable Architectures

The term CGRA, as discussed in [Santoro [2019]], refers to reconfigurable architectures based on a multi-bit data path, capable of processing instructions on data longer than 1 bit. This architecture comprises a limited set of processing units that can concurrently operate on multiple bits and can be programmed dynamically, allowing instructions to change during program execution, or statically, where the device behavior remains fixed.
However, one drawback of this architecture lies in its interconnections, which are constructed based on multiple bits, resulting in a larger area requirement. To maintain the same area in the device, the number of processing elements within the device must be reduced. This introduces a trade-off: there will be fewer blocks inside the CGRA, but they will be more powerful. Consequently, fewer blocks are utilized in executing algorithms, enhancing silicon utilization. Furthermore, the reduction in the number of blocks leads to a decrease in configuration time, as there is a smaller amount of data to configure.

1.2.2 PLim: Programmable Logic in memory

PLiM, as part of the LiM family architecture, incorporates processing elements within the cell, as discussed in [Casale [2020]]. What distinguishes PLiM is the utilization of a uCu,

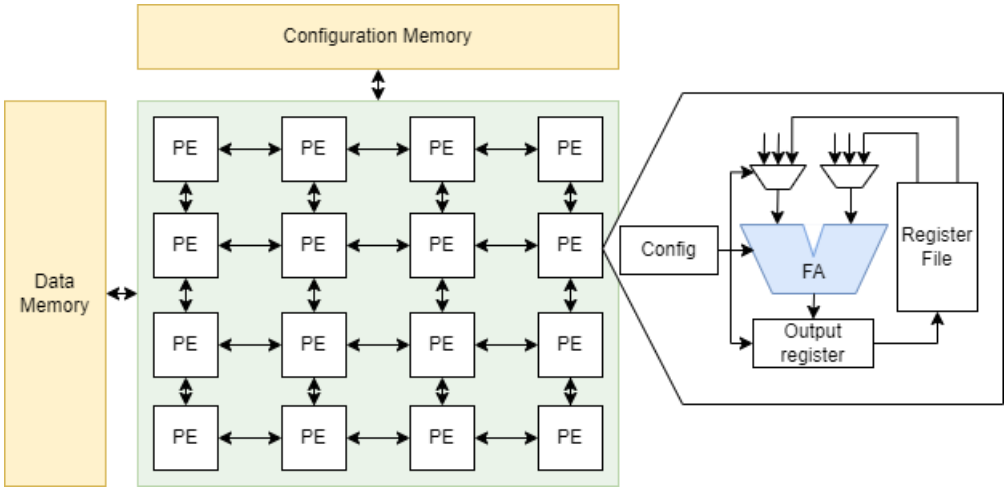


Figura 1.3. CGRA General scheme

responsible for fetching instructions from LiM memory and managing their execution. Importantly, the LiM memory can be altered, even at runtime, with different instructions. This implies that the architecture remains unchanged for different programs.
The Cu in PLiM can interact with it either as standard memory or through a scheduler. The scheduler provides the starting address and initiates the processing mode, facilitating versatile and dynamic operation.

Capitolo 2

RISCV Overview

The computer architecture chosen for this thesis is a RISC-V core. RISC-V is an open-source Instruction Set Architecture (ISA) Goti [2022] used in most cases for academic purposes because it is free and modular. Its modularity is the key to a simple and isolated architecture that leaves space for the addition of other blocks with the possibility of extending the ISA. It is proposed as an alternative to the CISC ISA, differing in the higher number of registers and the lower number of instructions, especially the store and load instructions.

2.1 Ri5cy microprocessor

Around the RISC-V families, the Ri5cy microprocessor is the chosen one for this thesis. It is a simple 32-bit core with 4 stages, developed and maintained by PULP (`https://github.com/openhwgroup/cv32e40p`). As we can see in Figure 2.1 [Andreas Traber and Multitherman Lab], Ri5cy has connections with the memory to fetch instructions and load/store data. These signals are the ones that are of interest for the connection of our LIM. Also useful is the interrupt interface, which is used in this case to activate the LIMA.

The RI5CY core communicates with the memory using the same protocol for data as for instructions Goti [2022]. However, the data interface can handle both read and write operations, requiring additional signals for the handshake process. Similarly to the normal memory, the LIM has been handled with the same protocol.
During the reading phase, the core notifies the memory of its access request by providing a valid address in data_addr_o and setting data_req_o high. The memory responds with data_gnt_i set high as soon as it is ready to satisfy the request. This response may occur in the same cycle as the request or several cycles later. Once a grant is received, the core may change the address in the next cycle. The memory indicates data availability by setting data_rvalid_i high, indicating that data_rdata_i contains valid data. This response may occur one or more cycles after the grant has been received. The read access in the data memory follows the same procedure as for the instruction memory.

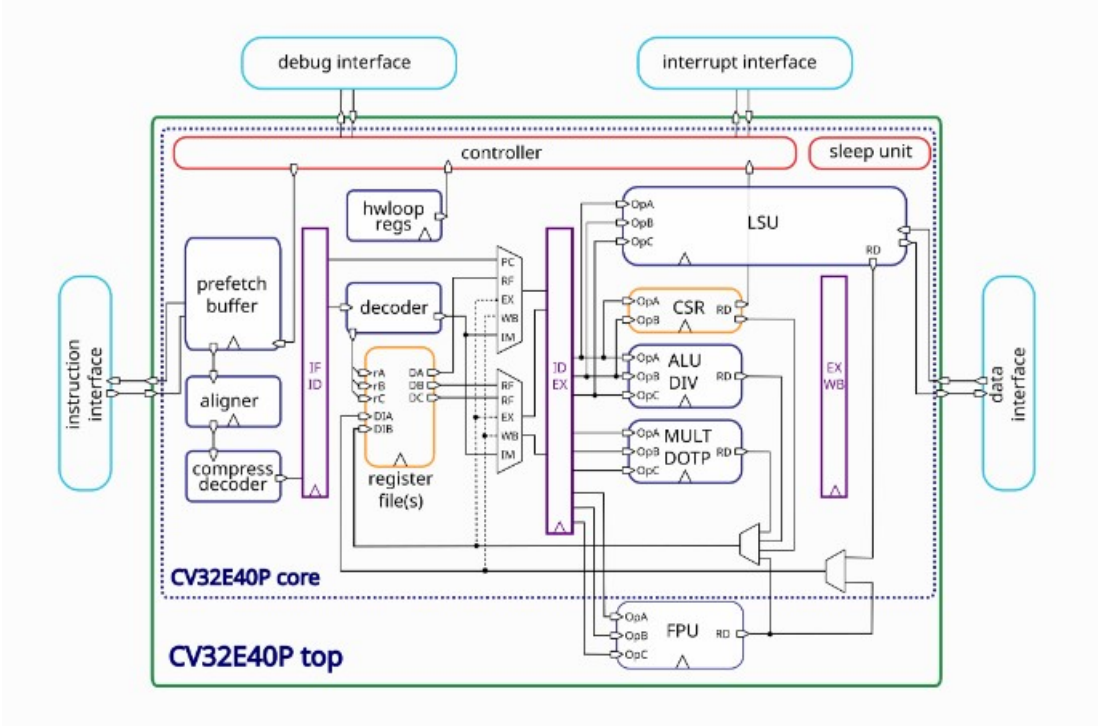


Figura 2.1. Block Diagram of CV32E40P RISC-V Core taken from the manual ris

During the writing phase, the core always initiates a memory access by setting data_addr_o and data_req_o, and it also provides data_wdata_o (the data to be written), data_we_o (write enable), and data_be_o (byte enable that is not used in the LIM). The memory responds with data_gnt_i set high when it is ready to process the request. Once a grant is received, the core may modify the address, data, and other control signals in the next cycle, assuming that the memory has already processed and stored the previous information. It is important to note that the memory must still respond with data_rvalid_i set high even during a write operation, although data_rdata_i has no significance in this case.
In figure 2.2, the used protocol for retrieving data from memory is well-specified, and the signals used are detailed in the table below, taken from the user manual Andreas Traber and Multitherman Lab.

2.2 Ri5cy instruction set

Central to the Instruction Set Architecture (ISA) Goti [2022], as aptly elucidated in the manual [ris], is the fixed length of instructions, set at 32 bits. The ISA encompasses six distinct instruction formats (R/I/S/U/B/J). Notably, the RISC-V ISA maintains consistent placement of source (rs1 and rs2) and destination (rd) registers across all formats,

Signal	Direction	Description
data_req_o	output	Request ready, must stay high until data_gnt_i is high for one cycle
data_addr_o[31:0]	output	Address
data_we_o	output	Write Enable, high for writes, low for reads. Sent together with data_req_o
data_wdata_o[31:0]	output	Data to be written to memory, sent together with data_req_o
data_rdata_i[31:0]	input	Data read from memory
data_rvalid_i	input	data_rdata_is holds valid data when data_rvalid_i is high. This signal will be high for exactly one cycle per request.
data_gnt_i	input	The other side accepted the request. data_addr_o may change in the next cycle

Tabella 2.1. LSU Signals used by memory and Mempa

Figura 2.2. Basic Memory Transaction taken from the manual ris

thereby streamlining the decoding process.

- The R format involves the utilization of two source registers in an instruction, exemplified by an integer register instruction where the combination of the two source registers is dictated by a function specified in the function field. The outcome of this operation is then stored in a designated destination register.
- The I format involves the utilization of an immediate value as the first source and a register as the second source. An example is the JALR instruction, belonging to the unconditional jump instructions. The destination address is obtained by adding the 12-bit signed I-immediate to the rs1 register, followed by setting the least significant bit of the result to zero. The address of the instruction following the jump (PC+4) is then stored in the rd register. In cases where the result is not necessary, the x0 register can be employed as the destination.
- The S format involves one immediate value and two source registers. For example, the store instructions are encoded in the S-type, where the value in register rs2 is stored in memory. B format, akin to the S format, incorporates one immediate value and two source registers. The 12-bit immediate field encodes branch offsets in multiples of 2.
- The U format encompasses a sole immediate value as its only source. The 20-bit immediate value undergoes a left shift by 12 bits.
- The J format, similar to the U type, involves a single immediate value as a source. In this case, the 20-bit immediate value undergoes a left shift by 1 bit.

After the format of the instruction Goti [2022] it is possible to see the various type of instructions:

- **Integer Computational Instructions:**
  Integer arithmetic instructions are encoded as either register-immediate operations

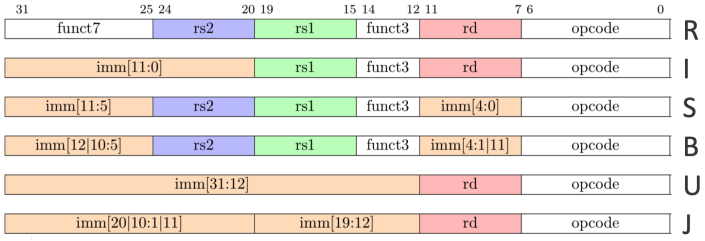


Figura 2.3. Instruction formats ris

employing the I-type format or register-register operations utilizing the R-type format. In both cases, the destination is stored in register rd. It is noteworthy that no integer arithmetic instructions lead to arithmetic exceptions.

- **Control Transfer Instructions:**
  RV32I encompasses two primary categories of control transfer instructions: unconditional jumps and conditional branches. It is noteworthy that control transfer instructions within RV32I do not exhibit architecturally apparent delay slots. In the event of an instruction access-fault or instruction page-fault exception occurring on the target of a jump or a taken branch, the exception is logged with reference to the target instruction rather than the jump or branch instruction.

  Unconditional Branches: The instruction used for both jumping and linking (JAL) utilizes the J-type format, where the J-immediate encodes a signed offset in increments of 2 bytes. This offset undergoes sign-extension and is subsequently added to the address of the jump instruction, thereby culminating in the establishment of the jump target address.

  Conditional Branches: The B-type instruction format is employed for all branching instructions. Within this format, the B-immediate, spanning 12 bits, encodes signed offsets in increments of 2 bytes. After undergoing sign-extension, this offset is combined with the address of the branch instruction to yield the target address. Notably, the range of conditional branches is within ±4 KiB.

- **Load and Store Instructions:**
  Load and store instructions facilitate the transfer of data between registers and memory. Load operations are encoded using the I-type format, whereas store operations utilize the S-type format. The effective memory address is determined by adding the value in register rs1 to the sign-extended 12-bit offset. In the case of load operations, a value is transferred from memory to the designated register rd. Conversely, during store operations, the value in register rs2 is transferred to the memory location specified by the effective address. The LW instruction is responsible for loading a 32-bit value from memory into register rd. For LH, a 16-bit value is retrieved from memory and then sign-extended to 32 bits before being stored in rd. Similarly, LHU loads a 16-bit value from memory and zero-extends it to 32 bits before storing it in

rd. LB and LBU operate similarly for 8-bit values. Conversely, the SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values, respectively, from the lower bits of register rs2 into memory.

- **Memory Ordering Instructions:**
  The FENCE instruction serves the purpose of orchestrating the sequencing of device I/O and memory accesses from the perspective of other RISC-V harts, external devices, or coprocessors. It provides the flexibility to establish an ordering relationship among various combinations of device input (I), device output (O), memory reads (R), and memory writes (W). In simpler terms, no other RISC-V hart or external device can observe any operation in the set that follows a FENCE before any operation in the set that precedes the FENCE.

  Additionally, the FENCE instruction enforces a sequencing of memory reads and writes executed by the hart, as observed from the standpoint of memory reads and writes performed by an external device. However, it's important to note that FENCE does not establish a sequencing of observations of events made by an external device through any alternate signaling mechanism.

Capitolo 3

Mempa model

3.1 Overview

The Mempa architecture is derived from two distinct architectures which have been briefly explained earlier: the PLiM architecture and the CGRA. This synthesis combines the adaptability of the PLiM architecture with the multilevel processing capability of the CGRA [guastamacchia angela [2021]].

The fundamental framework closely resembles that of a PLiM, featuring a memory structure comprised of sophisticated cells capable of internal data processing. Governed by both a uCu and nCu, these cells exhibit the ability to interact with the actual Cu, functioning either as a conventional memory or as a genuine LiM memory.
Mempa draws inspiration from CGRA for its utilization of multiple Single Instruction, Multiple Data (SIMD) units, allowing it to execute diverse instructions for different data sets. The selection of the data set is determined during the prefabrication phase and relies on post-manufacturing programming. Beyond its use of multiple SIMD units, Mempa also incorporates concepts from CGRA regarding the interconnections between rows and columns of various cells. The Smart Block, a cell to be elaborated upon later, is primarily influenced by the flexible nature of the CGRA cell.
To uphold one of the key features inherited from PLiM—the ability to process a wide range of algorithms without necessitating changes to the architecture—the interconnections in Mempa are designed to be as versatile as possible. However, they can be customized to enhance performance.

3.1.1 Mempa Memory structure

The LiM Array is represented by a matrix of memory words[guastamacchia angela [2021]]. This design choice is attributed to the simplicity and versatility it offers in terms of the placement of processing units. This design facilitates data transfer. The interconnections within this grid are more densely packed and can be precisely programmed. Consequently, during algorithm execution, data can flow through a larger number of potential paths across the entire array. This leads to more direct data transfers between different processing elements, resulting in fewer instructions needed to prepare data for upcoming

operations. As a result, the total number of instructions in the programs is reduced. The

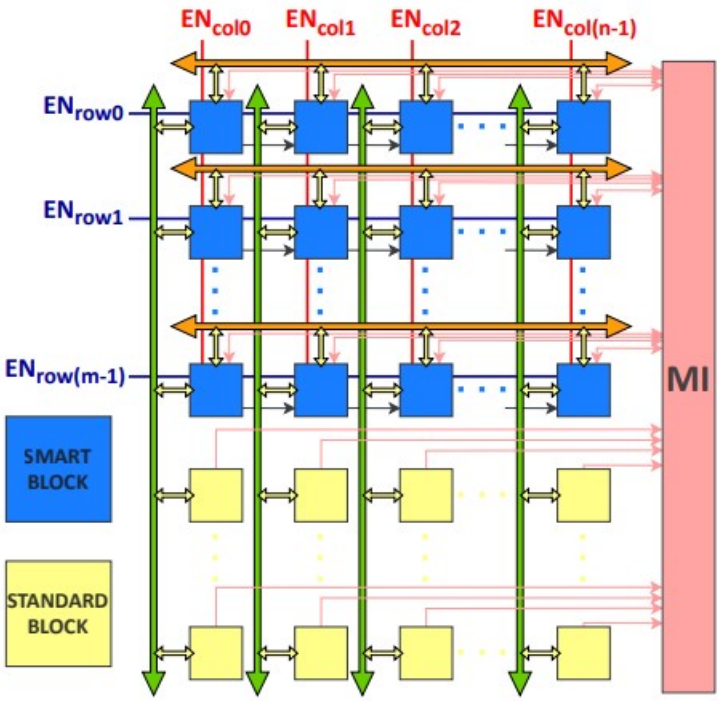


Figura 3.1. Lim Matrix scheme taken from the thesis Mempa: A General Purpose Architectural Model leveraging the Logic-in-Memory Approach guastamacchia angela [2021]

illustration effectively illustrates the segmentation of the different cells: the blue section, referred to as the smart section, contains embedded computing and storage components, known as Smart Blocks. These Smart Blocks are pivotal to the Mempa paradigm, as they are inherently equipped with a default arrangement that enables them to function as miniature, general-purpose processing units. On the other hand, the yellow section, denoted as the standard section, encompasses memory word locations exclusively consisting of storage elements. These storage elements in the standard section will henceforth be identified as Standard Blocks. While the content within the smart blocks remains subject to reading and modification during the LiM processing, the memory positions composing the standard section permit reading by the Smart Blocks. However, the act of writing to these positions is exclusively within the purview of the CPU when accessing the Mempa in standard data memory mode. The interlinkage between all the blocks encompasses two categories of interconnections: Reduction Tree Interconnections and Memory Interconnections. Each row-based interconnection facilitates the movement of data among smart blocks found in the same row. Meanwhile, the singular column-based interconnection extends across the standard section, enabling smart blocks to access data from any block positioned within the same column. Conversely, the Memory Interconnections span the

entirety of the LiM Array, enabling each Smart Block to retrieve the required data from any block through this network.

3.2 Smart Block

The Smart Block is the main component of the LiM, which means it is the component that holds data and performs operations on it. Unlike other LiMs, this particular Smart Block is much more complex and, as a result, much more powerful than the blocks seen in other LiMs. It comes close to emulating a small processor, which allows Mempa to be

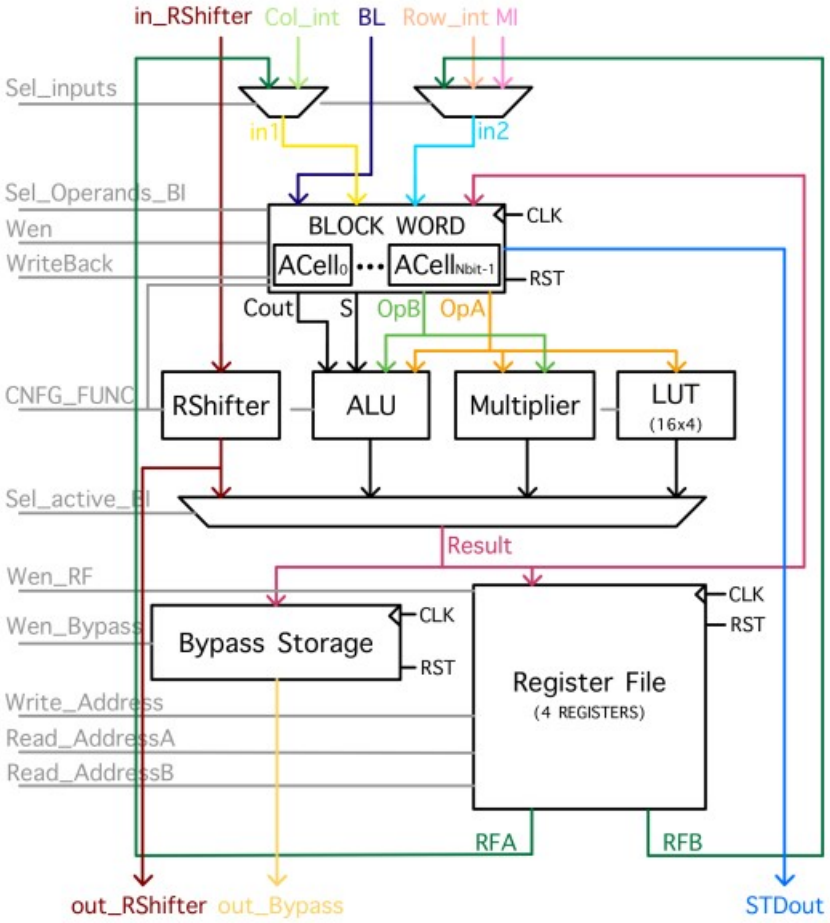


Figura 3.2. Structure of the Smart Block taken from the master thesis of Angela Guastamacchia guastamacchia angela [2021]

general-purpose. In this case, I will provide a brief explanation, while a more in-depth explanation can be found in the thesis dedicated to this architecture.
The fundamental component is the Block Word, where the word associated with that Smart Block is located, and simple bitwise logic functions are computed. However, the

Smart Block also includes additional storage elements designed to store temporary results generated during the LiM program execution. This includes a small Register File with two asynchronous read ports and one synchronous write port, consisting of 4 registers in the default Smart Block version. Additionally, there is a single register known as the Bypass Storage. This register not only serves as a data storage component but also provides data that the Reduction Tree Interconnections need to transmit to other Smart Blocks.
Moreover, the Smart Block's more complex processing capabilities are provided by a set of blocks beneath the Block Word, which function similarly to the Row Interfaces of the PLiM. All these blocks operate in parallel, using the data filtered by the Block Word as operands. The Block Word, in turn, receives part of the data it handles from the output of two multiplexers connected to most of the input signals of the Smart Block. Consequently, in general, the Smart Block can perform operations on the data stored in the Block Word itself, on data from its RF, on the value forwarded by the Memory Interconnections, and on data transmitted by the row and column interconnections and also the content of its Bypass Storage can also be processed.
However, in order to alleviate wire congestion within the Smart Block, there is no direct internal connection between the output of the Bypass Storage and the input multiplexers. To access the value stored in that register, it must be propagated through either the column or the row interconnections and then retrieved from the corresponding signals, Col int or Row int. All the possible destination components where the computation result can be sent include the Block Word itself, the RF, and the Bypass Storage. They are all controlled by the same Result signal output from the central multiplexer, which, for each instruction, appropriately selects one of the signals generated by the logic blocks implementing the more complex functions located beneath the Block Word.
Figure 3.2 illustrates the predefined Smart Block layout, which includes the following components:

- The RShifter block, integrated to enable the execution of a limited set of divisions.
- The ALU component, designed for basic arithmetic operations and comparison functions.
- A 16x4 bits LUT, utilized to customize the Mempa hardware platform even after manufacturing.
- The Multiplier, responsible for calculating the signed product between two data values.

In particular, to manage Smart Block complexity, the instantiated multiplier accepts input signals from only the lower half of OpA and OpB to produce a final result with a bit width equal to the memory parallelism (matching OpA and OpB width). However, this design choice necessitates that the LiM programmer adheres to a constraint to prevent incorrect results; that is, the data values to be multiplied must be representable using a bit width equal to half of the memory parallelism.

3.3 Interconnections

The strong interconnection among the Smart Blocks is what makes the Mempa paradigm powerful. These interconnections enable the acceleration of data movement within the LiM Matrix, allowing the parallel retrieval of data from different Smart Blocks and the corresponding operation, thus reducing the number of instructions. This is also achieved thanks to the reduction tree algorithm, which allows for N operations in log2N clock cycles.
As previously mentioned, in order to facilitate this type of data processing, the reduction tree interconnection method is employed to establish all the row and column connections within the LiM Matrix. Each of these connections retrieves data from the Bypass Storages within the Smart Blocks to which they are linked and then returns them in a modified sequence to the inputs of the corresponding Smart Blocks.

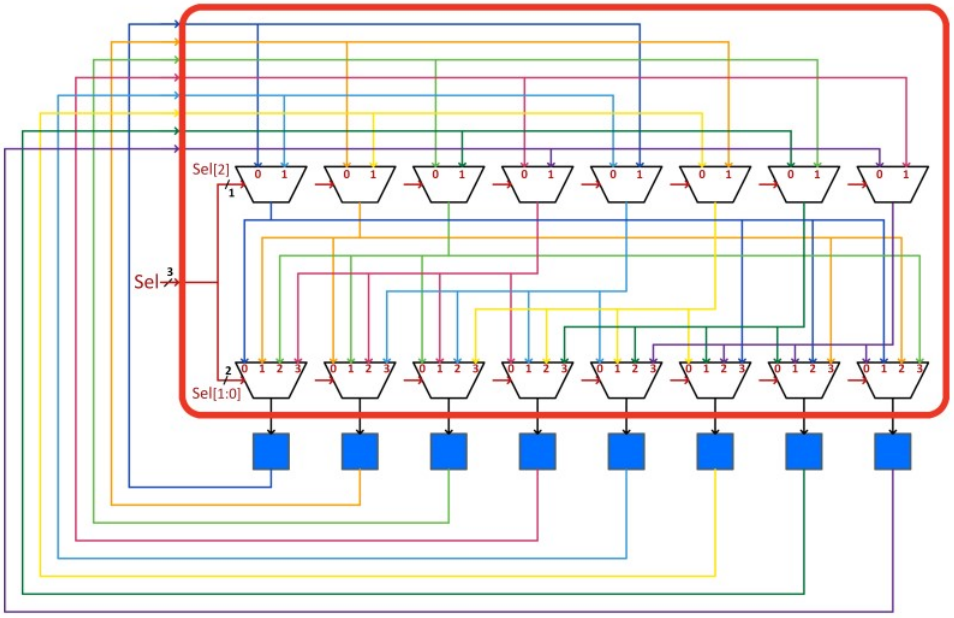


Figura 3.3. Structure of the reduction tree architecture for the row and column interconnection taken from the master thesis of Angela Guastamacchia guastamacchia angela [2021]

3.3.1 Column Interconnection

The way the LiM matrix is structured, when using a column interconnection in an instruction, it is addressed based on the distance between the Smart block being considered and the Smart block that needs to be used. For example, if we take address 0, it corresponds to the same Smart Block, while if the address is 1, it refers to the next block.
To simplify, each Smart Block has access to a multiplexer with N inputs each of M bits,

which is controlled by the same instruction decoder that controls the reference Smart Block. The signals are shifted accordingly based on the Smart Block under consideration. In this case, the multiplexer can become quite complex, which is why an optimization has been chosen, as shown in the figure 3.3.

3.3.2 Row Interconnection

A special case is represented by the row interconnections because they all have the same instruction to execute. Therefore, a multiplexer is used as if it were a downward shifter, in this case, using two levels of multiplexers, to reduce the complexity of the LiM matrix as we can see in the figure 3.3.
Each level consists of a set of multiplexers equal in number to the Smart Blocks to be interconnected. However, each of these multiplexers has a fewer number of inputs. The first level performs a coarse-grained shift of the combined data input, while the second level fine-tunes the final data to align with the specific number of shifts required as indicated by the interconnection configuration signal.
Taking the row interconnection example shown in figure 3.3, the first level of interconnections can shift the data by either 0 or 4 positions, while the second level can further adjust the data previously shifted by the first level by 0, 1, 2, or 3 positions.

3.4 Control Unit

This section provides a more detailed description of the Mempa's control unit.
The control section is divided into two distinct units, each representing one stage of the Mempa pipeline. Initially, there's the uCU (micro Control Unit), responsible for initializing the system in the desired mode, allowing the choice between processing or standard data memory mode, and managing the instruction flow. As such, it interfaces with both the CPU and the IMem (Instruction Memory) and is dedicated to implementing the instruction fetch stage.
Subsequently, the uCU is followed by the nCU (nano Control Unit), which is responsible for instruction decoding. It interacts with the uCU on one side and with the LiM Matrix on the other. The nCU takes a subsection of the instruction, appropriately extracted by the uCU, and processes it to gather the values needed for configuring signals. These signals, in the next clock cycle, enter the LiM Matrix, enabling it to perform the computations specified by the analyzed LiM instruction on the addressed data.

3.4.1 Micro control unit

The uCU in the Mempa system is implemented using a microprogrammed machine. It communicates with both the CPU and the nCU to determine the actions the Mempa Unit should perform.

Within the uCU, there is a small pipeline called "instrQueue," consisting of 5 registers managed by signals for enabling, shifting, and clock control. This pipeline allows the

storage of up to five addresses from the IMem, which stores program instructions that can be executed without CPU assistance. The pipeline facilitates the consecutive execution of program segments or small programs, eliminating the need to rewrite the IMem each time.

Before the Mempa begins operation, both the instrQueue and the IMem are initialized. To ensure the Mempa successfully completes the execution of a macro-program, the CPU must initialize a dedicated register. This involves setting the pipeline enable signal to '1' for one clock cycle and driving the waitADD signal with the address value of the IMem location containing a special "wait" instruction. This "wait" instruction is mandatory and must always be the last instruction before exiting the LiM processing mode.

The core of the uCU consists of a sequencer, comprising two multiplexers, the uPC register, the uRAR register, and a simplified adder. These components manage the instruction flow within the Mempa during LiM program execution. The uPC holds the address of the next LiM instruction to fetch from the IMem, and the sequencer selects the subsequent address to provide to the uPC. The uCU, in combination with the uInstruction, utilizes explicit addressing to dictate the program's progression. Each uInstruction includes a field specifying the address of the next instruction to fetch. Consequently, even jump instructions are implicitly implemented by writing the target instruction's address in that field.

The sequencer is configured to allow the invocation of subroutines using the jump&link instruction. When a jump&link instruction is executed, the uPC input is updated with the next address specified by the instruction. Simultaneously, the uRAR register stores the address of the instruction in the IMem that follows the jump&link. This address is obtained by adding 1 to the address of the currently fetched instruction.

However, it's important to note that this configuration does not support nested subroutine calls. After executing a jump&link instruction, a return from the previous function must have already been called before issuing another jump&link instruction.

3.4.2 Nano control unit

As previously mentioned, the nCU serves as the bridge between the uCU and the LiM Matrix, responsible for managing data transfer and processing within the Mempa, following the M-SIMD computing paradigm. As depicted in figure 3.4, it is a relatively straightforward component primarily containing a set of instruction decoders. The number of these decoders corresponds to the maximum count of distinct operations that the LiM Matrix needs to execute concurrently on various data elements. The individual instruction decoder is a straightforward component primarily responsible for signal mapping. It takes the nInstruction and effectively generates control signals by directly linking, in the majority of instances, specific field bits to their corresponding control signals. This direct association is made possible through a well-structured code assignment for the possible values that each field can take. Essentially, the code values that trigger specific operations correspond directly to the values that configuration signals must adopt to execute those operations

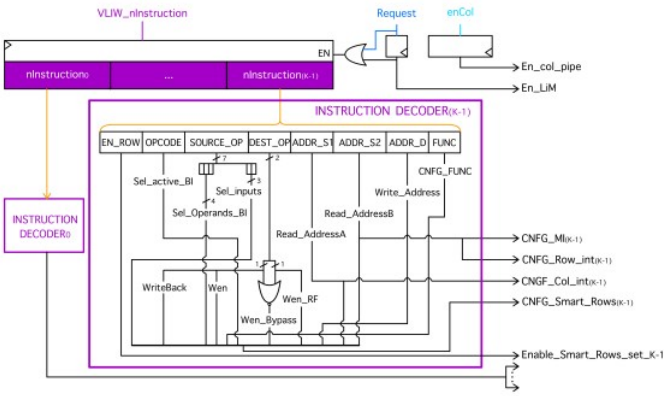


Figura 3.4. Structure of the nano control unit taken from the master thesis of Angela Guastamacchia guastamacchia angela [2021]

within the Smart Block. In this specific structure, we have three nCus, each managing 5 rows, except for the last one, which manages 6 rows.

3.5 Mempa Instructions

The entire LiM instruction used for programming the Mempa is divided into multiple nested fields. At a high level, there are two main slices: The first slice, called "uInstruction," is acquired by the uCU and contains information such as the next instruction to be fetched or when the end of the program is reached. The second slice is the "VLIW nInstruction," which is directly forwarded to the nCU to handle algorithm computations in M-SIMD mode.
The VLIW nInstruction is further composed of K fields, each known as "nInstruction," and each of these fields contains one of the simultaneous instructions that the Mempa can execute in each nCU.
The figure 3.5 describes the complete instruction. Each nCu takes an nInstruction and distributes it to all the smart blocks in the rows it manages. Each nInstruction is composed as follows:

- **Row enable:** This part of the nInstruction is used to enable each row, one bit per row, so there are 5 bits for the first two nCus and 6 bits for the last one.
- **Opcode:** This section specifies the type of instruction that the smart block must execute and, consequently, which element of the smart block should be selected. For example, OP_ALU specifies that it will be a logical operation, and the ALU inside the smart block must be selected.
- **Source Value:** This section indicates from where both operands should be taken—whether from memory, the register file, or if the data should be taken from the interconnections.

- **Destination value:** This section indicates where the data should be stored, whether in the same smart block, directly in the bypass, or if it should be saved in one of the registers of the register file.
- **First source address:** This part of the nInstruction contains the address of the first operand in 5 bits. When necessary, it specifies a valid address within the register file or for column interconnections; otherwise, it is not used.
- **Second source address:** This part of the nInstruction contains the address of the second operand in 9 bits.When necessary, it specifies a valid address within the register file or for row interconnections; otherwise, it is not used.
- **Destination address:** This part of the nInstruction contains the address of the destination in 2 bits Only when the destination value specifies the register file.
- **Function Opcode:** This section is utilized only when needed and in conjunction with the opcode to specify which operation the smart block should perform. For example, for the ALU, it describes whether it should perform a logical operation like OR or an addition, and so on.

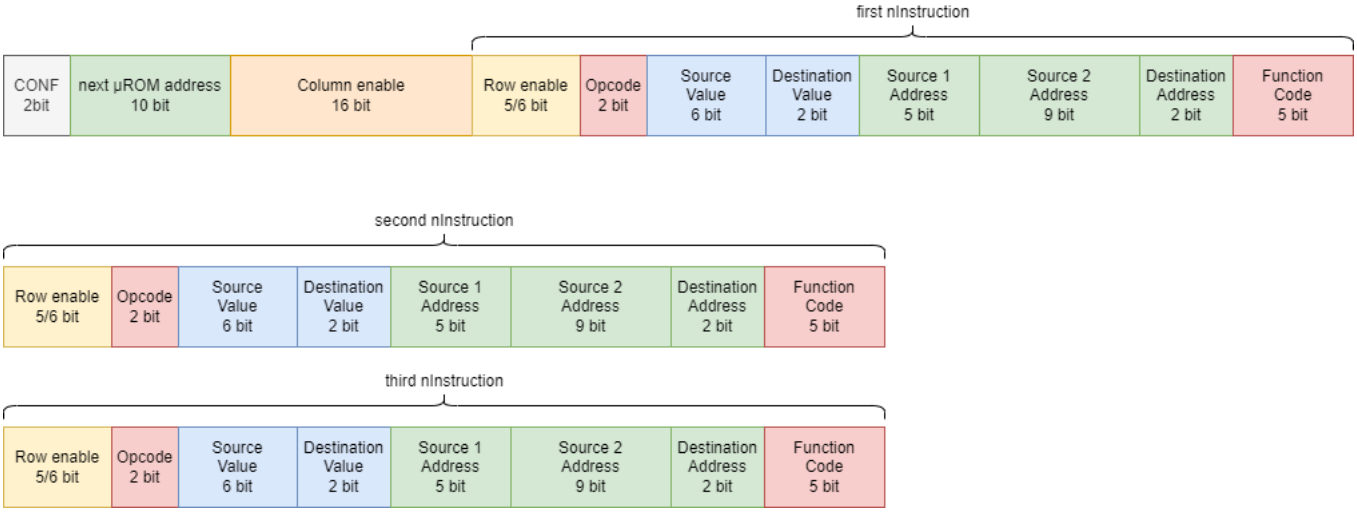


Figura 3.5. Structure of the instructions stored in the µROM

3.5.1 µROM

The µROM is a small memory (in this architecture set to 1KB, but almost never fully utilized) that contains all the instructions for the programs that the Mempa can execute. These programs can be combined and iterated within the control unit for execution without waiting for the signal from the processor. Alongside it, there is a µPC that keeps track of the address of the next instruction to be executed in the program. The PC plays

a fundamental role in controlling the flow of program execution and determines which instruction should be executed next. The Program Counter is crucial for the sequential execution of instructions in a program and provides the primary mechanism for controlling the flow of execution within a processor.

Parte II

Implementation

Capitolo 4

Algorithm for a CNN: Lenet-5

This algorithm was designed to study the behavior of the memory with something more complex, such as a convolutional neural network algorithm. Since the memory is relatively small, the first layer was implemented to avoid overloading the memory with data, which would have become too large.

4.1 CNN

The network structure [Alom et al. [2019]], initially proposed by Fukushima in 1988, faced limited adoption due to hardware constraints. In the 1990s, LeCun and his team applied a gradient-based learning algorithm to CNNs, achieving success in handwritten digit classification. Subsequent research enhanced CNNs, leading to state-of-the-art performance in recognition tasks. CNNs offer advantages over DNNs, resembling human visual processing, optimizing 2D and 3D image processing, and effectively abstracting 2D features. Max-pooling layers are efficient in capturing shape variations. With sparse connections and fewer parameters than fully connected networks, CNNs are trained using gradient-based learning, reducing the vanishing gradient problem and producing highly optimized weights.
The CNN architecture combines three types of layers: Convolution, max-pooling, and classification. In the lower and middle layers of the network, there are two types of layers: Convolutional layers and max-pooling layers. Even-numbered layers are dedicated to convolutions, while odd-numbered layers perform max-pooling operations. The output nodes of the convolution and max-pooling layers are organized into 2D planes known as feature maps. Each plane in a layer is typically derived from one or more planes in the previous layers. Nodes within a plane are connected to a small region in each connected plane from the preceding layer. Each node in the convolution layer extracts features from the input images through convolution operations on the input nodes.

4.2 Lenet-5 architecture

The LeNet-5 design [Lecun et al. [1998]] comprises two sequences of convolutional and average pooling layers. It is succeeded by a convolutional layer that flattens the output, followed by two fully-connected layers, and ultimately, a softmax classifier. In this thesis only the first convolutional layer in effectively translated in Mempa code. The initial stage

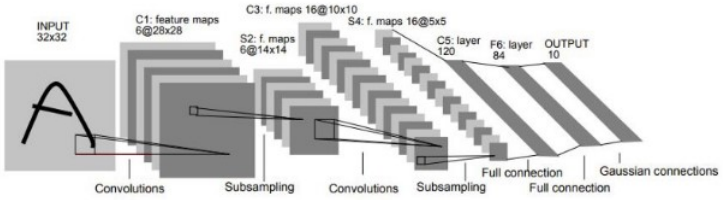


Figura 4.1. Original image published in Lecun et al. [1998]

of the LeNet-5 architecture [Lecun et al. [1998]] involves processing a 32x32 grayscale image. It traverses through the first convolutional layer, composed of 6 feature maps or filters measuring 5x5 in size, with a stride of one. As a result, the image's dimensions shift from 32x32x1 to 28x28x6.
Subsequently, LeNet-5 employs an average pooling layer, also known as a sub-sampling layer, with a filter size of 2x2 and a stride of two. This operation reduces the image dimensions to 14x14x6.
The next step encompasses a second convolutional layer featuring 16 feature maps sized at 5x5, with a stride of 1. Interestingly, only 10 out of these 16 feature maps are connected to the 6 feature maps from the preceding layer. This selective connectivity serves to break the network's symmetry and manage the number of connections within reasonable limits.
Layer four mirrors the second layer in structure, functioning as an average pooling layer with a 2x2 filter size and a stride of 2. However, it features 16 feature maps, leading to an output of 5x5x16.
The fifth layer operates as a fully connected convolutional layer with 120 feature maps, each measuring 1x1 in size. Every one of the 120 units in C5 connects to all 400 nodes in the fourth layer.
Following that, the architecture includes a fully connected layer comprising 84 units.
Finally, the architecture culminates in a fully connected softmax output layer with 10 possible values, corresponding to the digits from 0 to 9.

4.3 Lenet-5 implementation inside the µROM

In this context, the Lenet-5 LiM program includes only the first convolutional layer and it is clearly presented as a rapid guided example. The explanation is structured in such a way that each LiM instruction is explicitly declared, followed by a commentary illustrating

the operations and data transfers implied within the LiM Matrix. Consequently, a comprehensive list of all program instructions is compiled, with each instruction highlighting different Instruction macro-fields, such as uInstruction, nInstruction 0, nInstruction 1, and nInstruction 2, distinguished by corresponding color associations.

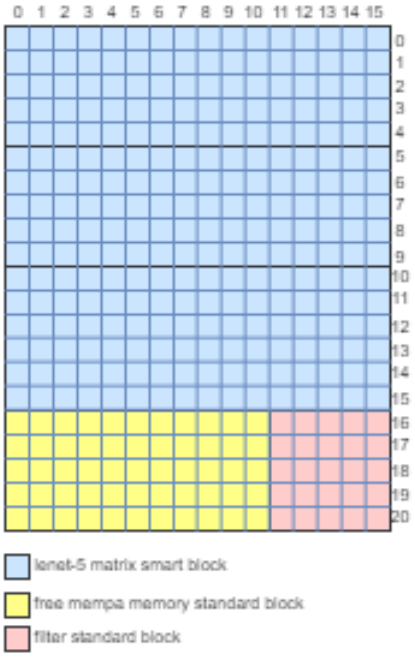


Figura 4.2. Initial values in Mempa before the elaboration of lenet-5

- **Step 1:** Very simple operation save the content of the matrix inside R0 that will not be used.

- **Step 2:** Takes the filter and move it in the first 5x5 elements on the top right that starts the elaboration.

```
init_instructions(2)<=

    "01"&std_logic_vector(to_unsigned(3,SIZE_uROM_Address))&
    '0'&'0'&enable_rows(0)&'1'&

    "11111"&OP_Load&Col_int_col_int&DEST_RF&
    std_logic_vector(to_unsigned(16,SIZE_ADDR_S1))&nullADDR_S2
    &std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&nullFunc&

    "11111"&OP_Load&Col_int_col_int&DEST_RF
    &std_logic_vector(to_unsigned(11,SIZE_ADDR_S1))&nullADDR_S2
    &std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&nullFunc&

    "111110"&OP_Load&Col_int_col_int&DEST_RF
    &std_logic_vector(to_unsigned(6,SIZE_ADDR_S1))&nullADDR_S2
    &std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&nullFunc;
```

This instruction is executed only in the last 5 columns, from 11 to 15, as specified in enable_rows(0)="0000000000011111." In all three sections, the filter located in the standard block is taken and moved to the smart block, which will subsequently use it for processing.
The first bits of each section following the yellow one indicate which rows are selected for the operation. Therefore, all 5 rows for the first two uCUs, while only the first 5 for the last uCU (because the filter is 5x5).
The type of instruction used, OP_LOAD, is a Load because we need to move data from one position in memory to another. The type of connection used is column interconnections, Col_int_col_int.
The first address of each nInstruction indicates how many rows after that the data should be taken (since it's column interconnections). In fact, for each section, this value varies as it gets closer to the standard blocks.
std_logic_vector(to_unsigned(16,SIZE_ADDR_S1)) indicates that there are 16 rows of difference between the first row of the matrix and the first row of the filter. This is because the matrix is 16x16.
The same goes for std_logic_vector(to_unsigned(11,SIZE_ADDR_S1)) and std_logic_vector(to_unsigned(6,SIZE_ADDR_S1)).
While the address of the second parameter is not needed and can be null, the destination address, std_logic_vector(to_unsigned(2,SIZE_ADDR_D)), indicates in which register the data should be saved—in this case, R2.
In this case, all three sections execute the exact same instruction.

- **Step 3:** The filter previously saved in R2 is saved in the bypass register ready to be shifted to all the others smart blocks.

```
init_instructions(3)<=

  "01"&std_logic_vector(to_unsigned(4,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(0)&'1'&

  "11111"&OP_Load&RFA&DEST_Bypass&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&nullFunc&

  "11111"&OP_Load&RFA&DEST_Bypass&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&nullFunc&

  "111110"&OP_Load&RFA&DEST_Bypass&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&nullFunc;
```

The smart blocks in the Mempa containing the filter (last 5 column enable_rows(0)="0000000000011111." and all the 15 rows "111110"), perform a load action as specified in the instructions OP_Load from the register file R2 std_logic_vector(to_unsigned(2,SIZE_ADDR_S1)) with the destination being the Bypass register.

- **Step 4:** Move the filter to the central smart blocks.

```
init_instructions(4)<=

  "01"&std_logic_vector(to_unsigned(5,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(1)&'1'&

  "11111"&OP_Load&Row_int_row_int&DEST_RF&nullADDR_S1&
  std_logic_vector(to_unsigned(5,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&nullFunc&

  "11111"&OP_Load&Row_int_row_int&DEST_RF&nullADDR_S1&
  std_logic_vector(to_unsigned(5,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&nullFunc&

  "111110"&OP_Load&Row_int_row_int&DEST_RF&nullADDR_S1&
  std_logic_vector(to_unsigned(5,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&nullFunc;
```

Now the column selected are different, all the "middle" rows are used enable_rows(1)="0000001111100000" and through the column interconnections Row_int_row_int the filter is shifted from the 5th position calculated for each smart block std_logic_vector(to_unsigned(5,SIZE_ADDR_S2)) to register R2 DEST_RF, std_logic_vector(to_unsigned(2,SIZE_ADDR_D)) .

- **Step 5:** Move the filter to the initial smart blocks.

```
init_instructions(5)<=

  "01"&std_logic_vector(to_unsigned(6,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(2)&'1'&

  "11111"&OP_Load&Row_int_row_int&DEST_RF&nullADDR_S1&
  std_logic_vector(to_unsigned(10,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&nullFunc&

  "11111"&OP_Load&Row_int_row_int&DEST_RF&nullADDR_S1&
  std_logic_vector(to_unsigned(10,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&nullFunc&

  "111110"&OP_Load&Row_int_row_int&DEST_RF&nullADDR_S1&
  std_logic_vector(to_unsigned(10,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&nullFunc;
```

The filter is then shifted, again through the row interconnections Row_int_row_int, from the same rows that previously loaded the filter into the Bypass register to the first 5 rows, always starting from the left, enable_rows(2)="0111110000000000". The specified position for the row interconnection is 10, as indicated in the command std_logic_vector(to_unsigned(10,SIZE_ADDR_S2)). Again, the same instruction is given to all nCu, and the data is saved in register R2. In 4.3 is illustrated how filter is given to all the smart blocks in the matrix.

- **Step 6:** Multiplication in the even column between the smart blocks ans the correspondent element in the filter placed in R2 and save the result into R3.

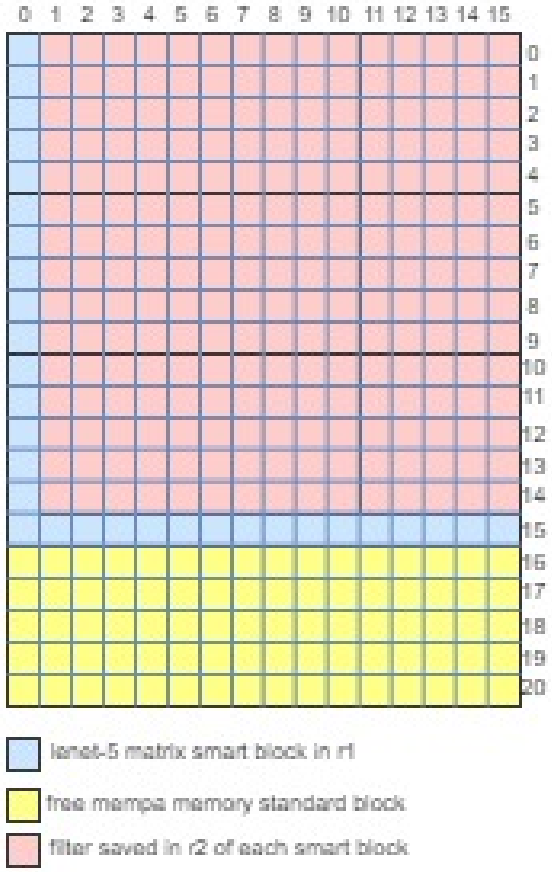


Figura 4.3. The placement of the filter after instruction 5 of the µROM

```
init_instructions(6)<=

  "01"&std_logic_vector(to_unsigned(7,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(4)&'1'&

  "11111"&OP_Multiplier&RFA_Mem&DEST_RF&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&nullFunc&

  "11111"&OP_Multiplier&RFA_Mem&DEST_RF&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&nullFunc&

  "111110"&OP_Multiplier&RFA_Mem&DEST_RF&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&nullFunc;
```

In this instruction, the selected columns enable_rows(4)="0110101101011010" multiply OP_Multiplier the data stored inside the smart block with the filter located in the second register of the register file RFA_Mem, std_logic_vector(to_unsigned(2,SIZE_ADDR_S1)), saving the result in the register file at the third register DEST_RF, std_logic_vector(to_unsigned(3,SIZE_ADDR_D)).

- **Step 7:** Multiplication of the odd column smart blocks with the filter in R2 and save the result into bypass.

```
init_instructions(7)<=

  "01"&std_logic_vector(to_unsigned(8,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(3)&'1'&

  "11111"&OP_Multiplier&RFA_Mem&DEST_Bypass&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&nullFunc&

  "11111"&OP_Multiplier&RFA_Mem&DEST_Bypass&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&nullFunc&

  "111110"&OP_Multiplier&RFA_Mem&DEST_Bypass&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&nullFunc;
```

In this instruction, It is multiplied the smart blocks of the even columns, as specified in the uInstruction part enable_rows(3)="0001010010100101", by the filter, which, as we have seen, is located in the second register of the register file, as also specified in the nInstructions RFA_Mem, std_logic_vector(to_unsigned(2,SIZE_ADDR_S1)). However, this time, it is saved in the Bypass register, as specified in this part of the nInstructions DEST_Bypass, because it will be needed for the next instruction.

- **Step 8:** Sum the columns two by two and save the result into bypass.

```
init_instructions(8)<=

  "01"&std_logic_vector(to_unsigned(9,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(5)&'1'&

  "11111"&OP_Logic_Adder&Row_int_RFA&DEST_Bypass&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S2))&nullADDR_D&sum_op&

  "11111"&OP_Logic_Adder&Row_int_RFA&DEST_Bypass&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S2))&nullADDR_D&sum_op&

  "111110"&OP_Logic_Adder&Row_int_RFA&DEST_Bypass&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S2))&nullADDR_D&sum_op;
```

In this instruction, the even columns are selected enable_rows(5)="0010100101001010" to take, through the row interconnections Row_int_RF, the element to their left, previously saved in the Bypass register with this part of the nInstruction std_logic_vector(to_unsigned(1,SIZE_ADDR_S2)). It indicates how many positions to skip before selecting the data. Once taken, the sum sum_op is performed, and the result is saved in the third register of the register file.

- **Step 9-10:** Sum of the three remaining columns is saved in the first column of each 5x5 sub-matrix. .

```
init_instructions(9)<=

  "01"&std_logic_vector(to_unsigned(10,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(6)&'1'&

  "11111"&OP_Logic_Adder&Row_int_RFA&DEST_RF&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&sum_op&

  "11111"&OP_Logic_Adder&Row_int_RFA&DEST_RF&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&sum_op&

  "111110"&OP_Logic_Adder&Row_int_RFA&DEST_RF&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&sum_op;

init_instructions(10)<=

  "01"&std_logic_vector(to_unsigned(11,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(6)&'1'&

  "11111"&OP_Logic_Adder&Row_int_RFA&DEST_Bypass&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&nullADDR_D&sum_op&

  "11111"&OP_Logic_Adder&Row_int_RFA&DEST_Bypass&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&nullADDR_D&sum_op&

  "111110"&OP_Logic_Adder&Row_int_RFA&DEST_Bypass&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&nullADDR_D&sum_op;
```

In this pair of instructions, the same smart blocks are selected enable_rows(6)="0100001000010000" to essentially perform the same operation but with different interconnections. In the first instruction (number 9), register 3 of the register file is taken, and indeed ADDR_S1=3. Through the row interconnection

Row_int_RFA, the next element is taken to be summed ADDR_S2=1, and the result is saved in the register file in the third register ADDR_D=3. While in the second instruction, the next value is not taken but rather the value from the bypass storage after three positions ADDR_S2=3, and it is saved in the bypass storage DEST_Bypass of the selected smart block. For this reason, no address needs to be specified. After this instructions all the remaining element to sum are collocated in that specific columns, so what it is done now is a sum by row until only one element remains and it is the result.

- **Step 11:** Save the bypass register of the even row into R3.

```
init_instructions(11)<=

  "01"&std_logic_vector(to_unsigned(12,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(6)&'1'&

  "11010"&OP_Load&Row_int_row_int&DEST_RF&
  std_logic_vector(to_unsigned(0,SIZE_ADDR_S1))&nullADDR_S2&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&nullFunc&

  "11010"&OP_Load&Row_int_row_int&DEST_RF&
  std_logic_vector(to_unsigned(0,SIZE_ADDR_S1))&nullADDR_S2&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&nullFunc&

  "110100"&OP_Load&Row_int_row_int&DEST_RF&
  std_logic_vector(to_unsigned(0,SIZE_ADDR_S1))&nullADDR_S2&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&nullFunc;
```

In this instruction the even row are used as we can see from the beginnig left part of all the nInstructions "11010", only the last one has always a 0 left cause of the different size. This case is a particular one: the row interconnection takes the value of the positions to jump to reach the correct value, which in this case is ADDR_S1=0. This means that the value to be taken is the bypass register itself, and it needs to be saved in the register file at the third register DEST_RF, ADDR_D=3.

- **Step 12:** Sum the even rows to the odd rows and save the result into register r3.

```
init_instructions(12)<=

  "01"&std_logic_vector(to_unsigned(13,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(6)&'1'&

  "01010"&OP_Logic_Adder&Col_int_RFB&DEST_Bypass&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&nullADDR_D&sum_op&

  "01010"&OP_Logic_Adder&Col_int_RFB&DEST_Bypass&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&nullADDR_D&sum_op&

  "010100"&OP_Logic_Adder&Col_int_RFB&DEST_Bypass&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&nullADDR_D&sum_op;
```

This instruction is very similar to the sum of the columns done before, but here, as it can be seen, the columns don't change. Instead, the enabling of the rows is always different; in this case, "01010" so the odd rows are selected and taken by the column interconnections Col_int_RFB the successive value since the position to take is ADDR_S1=1. Then, it performs a sum operation, enabling the ALU of each smart block interested OP_Logic_Adder, sum_op between the value chosen by the column interconnection and the register file's third register ADDR_S2=3. Finally, save the value inside the Bypass register.

- **Step 13-14:** Sum of the three remaining rows is saved in the first row of each 5x5 sub-matrix.

```
init_instructions(13)<=

  "01"&std_logic_vector(to_unsigned(14,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(6)&'1'&

  "10000"&OP_Logic_Adder&Col_int_RFB&DEST_RF&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&sum_op&

  "10000"&OP_Logic_Adder&Col_int_RFB&DEST_RF&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&sum_op&

  "100000"&OP_Logic_Adder&Col_int_RFB&DEST_RF&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&sum_op;

init_instructions(14)<=

  "01"&std_logic_vector(to_unsigned(15,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(6)&'1'&

  "10000"&OP_Logic_Adder&Col_int_RFB&DEST_RF&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&sum_op&

  "10000"&OP_Logic_Adder&Col_int_RFB&DEST_RF&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&sum_op&

  "100000"&OP_Logic_Adder&Col_int_RFB&DEST_RF&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&sum_op;
```

These two instructions are very similar: both of them use the third register of the register file as the operand ADDR_S2=3 and as the destination ADDR_D=3. The

other operand is taken by the column interconnection, and in this case, the two instructions differ. The first one takes the element distant by 1 position ADDR_S1=1, the 14th instruction takes the element distant 3 positions ADDR_S1=3. Both of them perform a sum operation sum_op and of course all this instruction is performed on the same row and column "10000" enable_rows(6)="0100001000010000".

- **Step 15:** Save the result element of all this instruction in the register R1 where it can be read in another time.

```
init_instructions(15)<=

  "01"&std_logic_vector(to_unsigned(16,SIZE_uROM_Address))&
  '0'&'0'&enable_rows(6)&'1'&

  "10000"&OP_Load&RFA_RFA&DEST_RF&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&nullADDR_S2&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&nullFunc&

  "10000"&OP_Load&RFA_RFA&DEST_RF&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&nullADDR_S2&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&nullFunc&

  "100000"&OP_Load&RFA_RFA&DEST_RF&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&nullADDR_S2&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&nullFunc;
```

The last instruction of this cycle is a simple load instruction OP_Load performed on the columns enable_rows(6) and rows "10000" that has the final value of the result matrix. In this case, the value is taken from the third register of the register file ADDR_S1=3 and saved into the first register ADDR_D=1 in such a way to be carried to the end of the algorithm.

At the end of a cycle of steps, we will have 9 elements of the final matrix as it is illustrate in 4.4. All these steps are repeated 20 times, shifting the filter both by rows and columns (as the enables of the rows and columns) until the complete processing of the final matrix is achieved, resulting in 317 instructions 4.5.

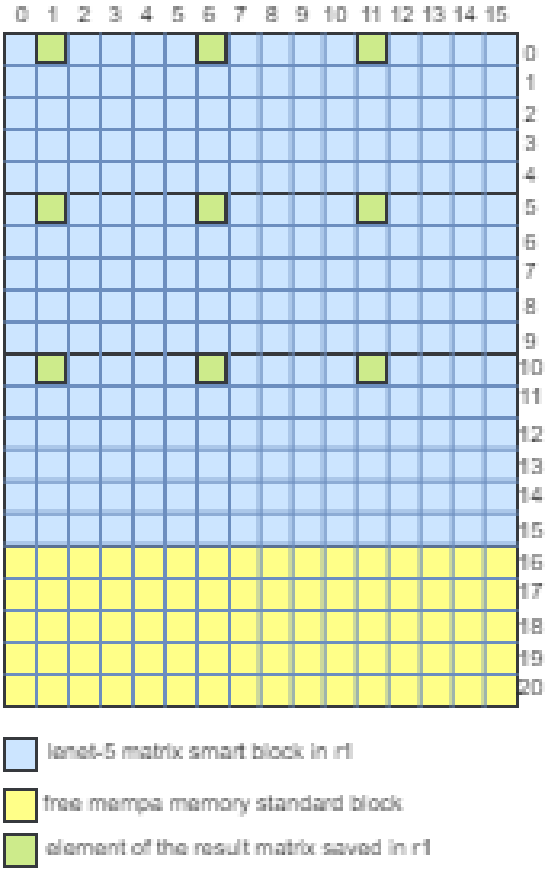


Figura 4.4. State of the matrix after the first cycle of the LeNet-5 algorithm.

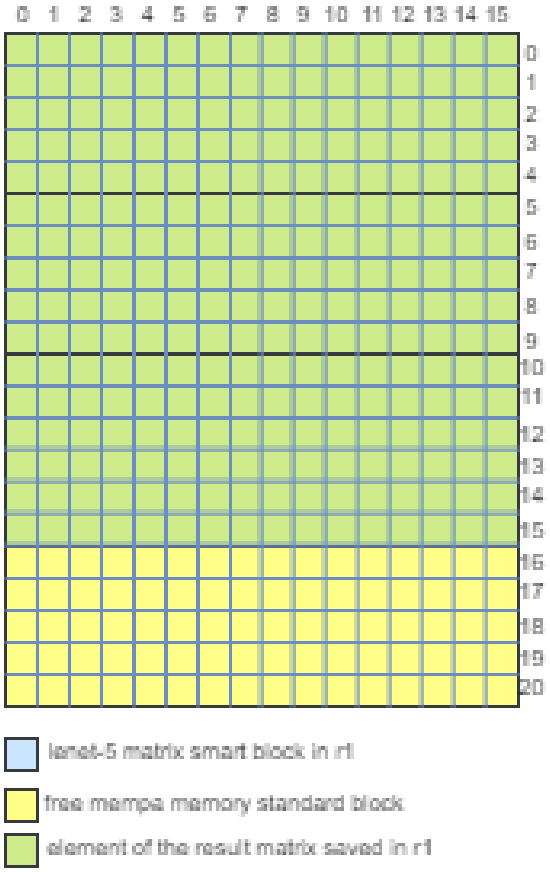


Figura 4.5. Final values in Mempa after the elaboration of lenet-5

Capitolo 5

Communication between Ri5scy and Mempa

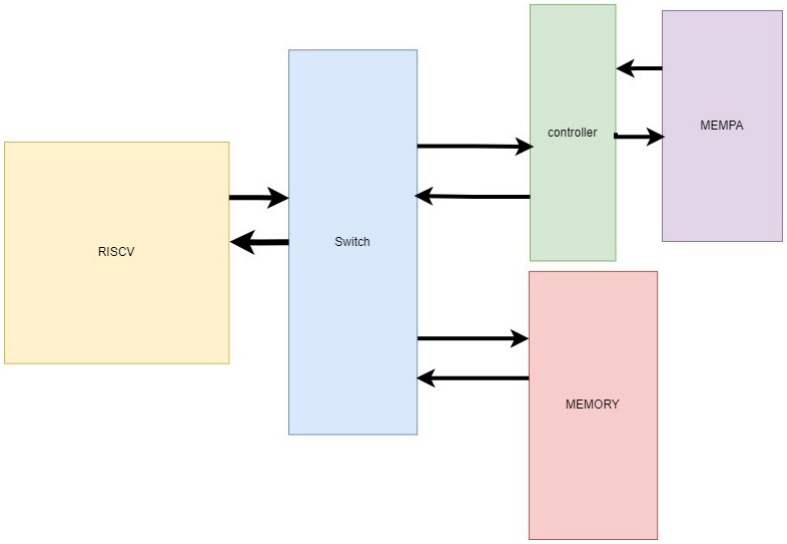


Figura 5.1. General scheme of the architecture

The bulk of the thesis has been dedicated to the communication between Ri5cy and Mempa. In this scenario, Mempa and Ri5cy employ two distinct protocols. To facilitate their communication, two components have been integrated, which will be described in more detail later: A controller to effectively enable the interaction between the two components and a databus to route data to Mempa and RAM (figure 5.1).

5.1 Memory range

When introducing a new memory module, the initial task is to define the specific address range where it will be allocated within the system. In the context of Ri5cy, the processor strategically operates on four addresses simultaneously. This design choice is rooted in the Ri5cy architecture philosophy, characteristic of the entire Ri5cy processor family. This architecture allows the processor to access every individual byte within the RAM, and each block of RAM comprises a word, composed of four bytes.
However, the scenario is different for Mempa, as each smart block possesses its unique address. This means that unlike the contiguous and evenly distributed memory access in Ri5cy, Mempa's memory structure is more fragmented, with each smart block having its designated address.
The disparity in addressing mechanisms becomes evident when examining Figure 5.2. On the Ri5cy side, the perception is that of a larger memory space compared to the conventional setup. This disparity arises due to the way Ri5cy processes memory in contiguous blocks, whereas Mempa assigns a distinct address to each smart block.
The challenge at hand involves reconciling these divergent approaches in memory management. The subsequent discussion will delve into two blocks that offer solutions to this disparity and shed light on how the system addresses the nuanced complexities of memory access and organization in both Ri5cy and Mempa.

Figura 5.2. Memory Range difference inside the Ri5cy and in real architecture

5.2 DataBus switch

The databus (figure 5.3) operates as a switch based on the address of the data address sent by Ri5cy:

- If the address is less than FFE0C, the request is for the memory so all the signals sended by the Ri5cy are passed to the RAM
- If the address is greater than FFE0C, the request is for the Mempa (used as memory) so all the signals are sended to the controller.
- If the address is 100600, the request is for the activation of the Mempa so the signals Lim_a is activated.

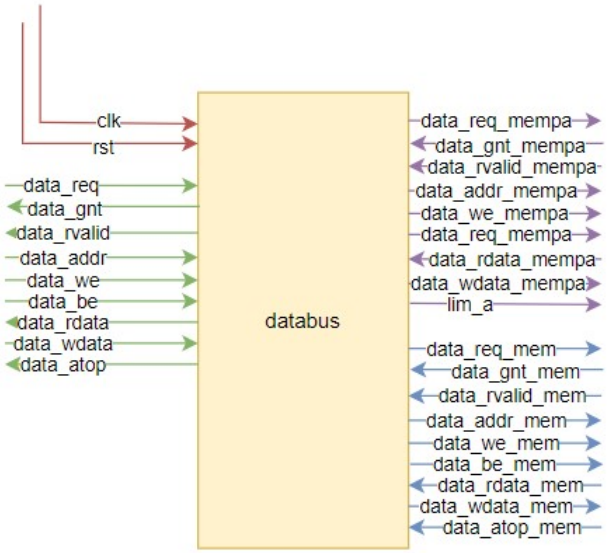


Figura 5.3. General scheme of the DataBus

A more detailed description of the databus can be seen in Figure 5.4, where it is clearly shown how various signals are generated.
Most of the outputs are selected by a multiplexer, with the control signal being a condition on the address sent by Ri5cy.
The signals from memory, namely data_req_mem, data_addr_mem, data_we_mem, data_be_mem, and data_wdata_mem, are selected based on a double condition: that the sent address does not belong to the portion of memory dedicated to Mempa and that it is not the address dedicated to activating LiM. When this condition is true, the signals from Ri5cy (the corresponding signal for each memory signal) are selected; otherwise, all signals are connected to ground.
The signals from Mempa, data_req_Mempa, data_addr_Mempa, data_we_Mempa, data_be_Mempa, and data_wdata_Mempa, are selected through a multiplexer by a single logic, namely the condition that checks whether the address is valid and falls within the memory range dedicated to Mempa. This specific use is given by lim_a, activated by a

different control signal, specifically in the condition that the address dedicated to activating Mempa is selected by Ri5cy5.
A specific mention is made for data_rdata, which, unlike other signals, is a direct output to Ri5cy and can therefore have three values based on three conditions: no request has been made by Ri5cy, a read request has been made to Mempa, and Mempa returns the requested data, and a request has been made to the main memory, and the main memory returns the requested data. In this case, since there are three conditions, the multiplexer is wider and has two control signals: one based on data_rvalid_mem (indicating when the data is available in the main memory output) and the other based on data_rvalid_Mempa, which signals the same thing but from the Mempa block. Therefore, the multiplexer has four possible values:

- "00": No request presented, the signal is connected to ground.
- "01": data_rvalid_mem arrives, and the data_rdata signal will have the value brought by data_rdata_mem.
- "10": data_rvalid_Mempa arrives, and the data_rdata signal will have the value brought by data_rdata_Mempa.
- "11": Condition not possible because we cannot have two synchronous requests to Mempa and mem.

Two other signals that are outputs to Ri5cy, namely data_gnd and data_rvalid, are a simple OR between the two Mempa and mem signals.

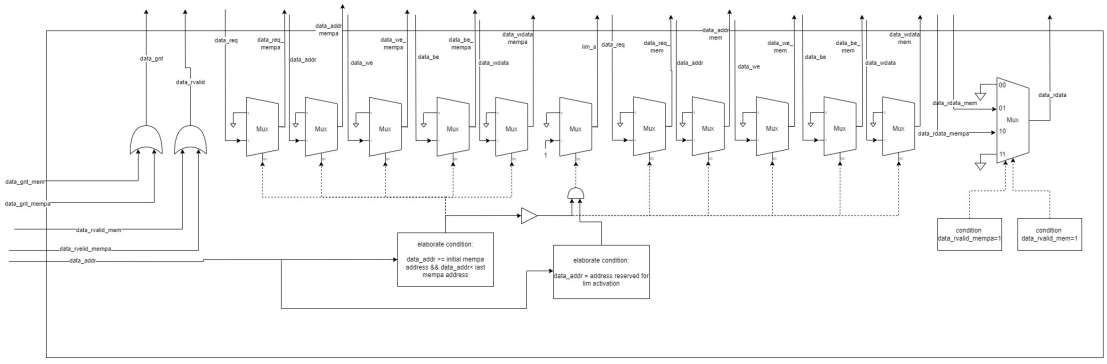


Figura 5.4. Structural scheme of databus

To calculate the previous addresses the size of both the memories are important. Here now we considering a RAM of 1MB and a Mempa of 1KB.

5.3 Controller and protocol description

As mentioned earlier, any data request from Ri5cy is triggered by the data_req signal. When this signal is raised, the processor also sends the address to which the data request is

referring (data_addr signal), If it is a write operation it raises the data_we signal,include a possible data mask through the data_be signal, and for a write operation, it also sends the data associated with it via the data_wdata signal. The processor then waits for the data_gnt signal, which is ideally sent by the memory when it has accepted the request (2.2). The Mempa has a significantly different access protocol, as shown in Figure 5.5. For

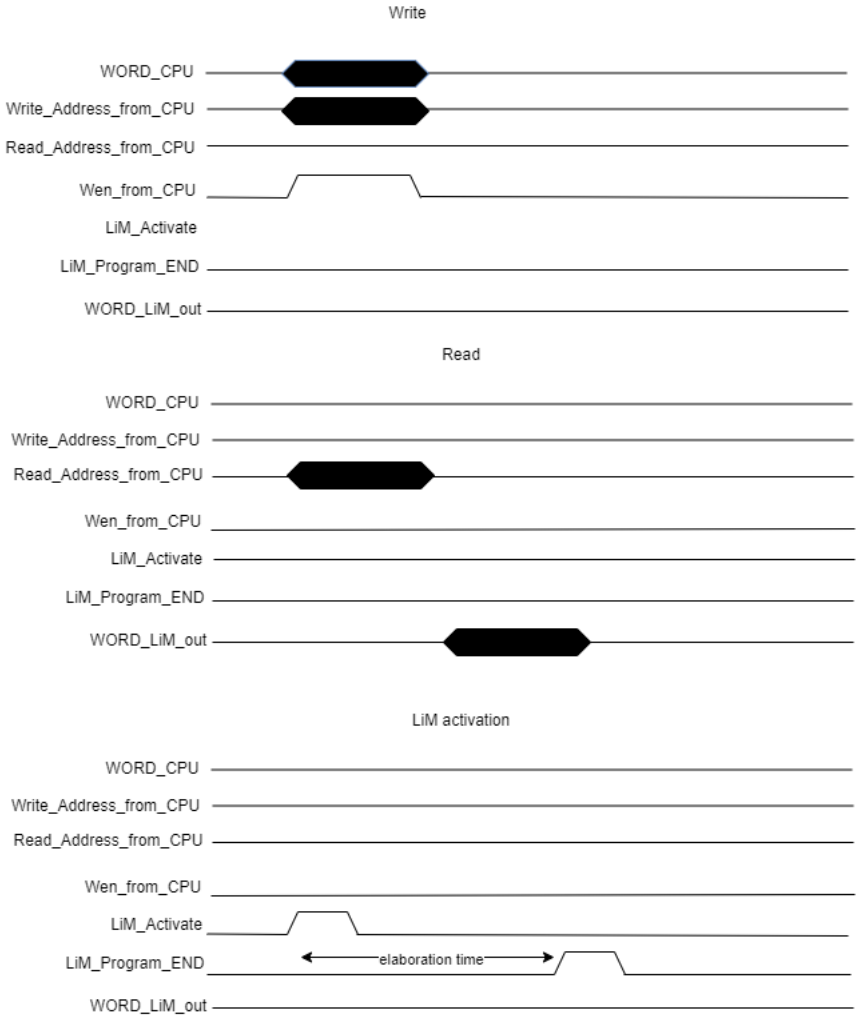


Figura 5.5. Mempa Access Protocol for read, write, and LiM activation.

instance, it does not require the data_gnt signal, nor does it use any signal to confirm the data reception or to indicate the presence of data on the bus. Another distinction in the Mempa is the differentiation between the two signals used for address transmission, which are distinct for read and write operations (write_address_from_CPU and read_address_from_CPU).

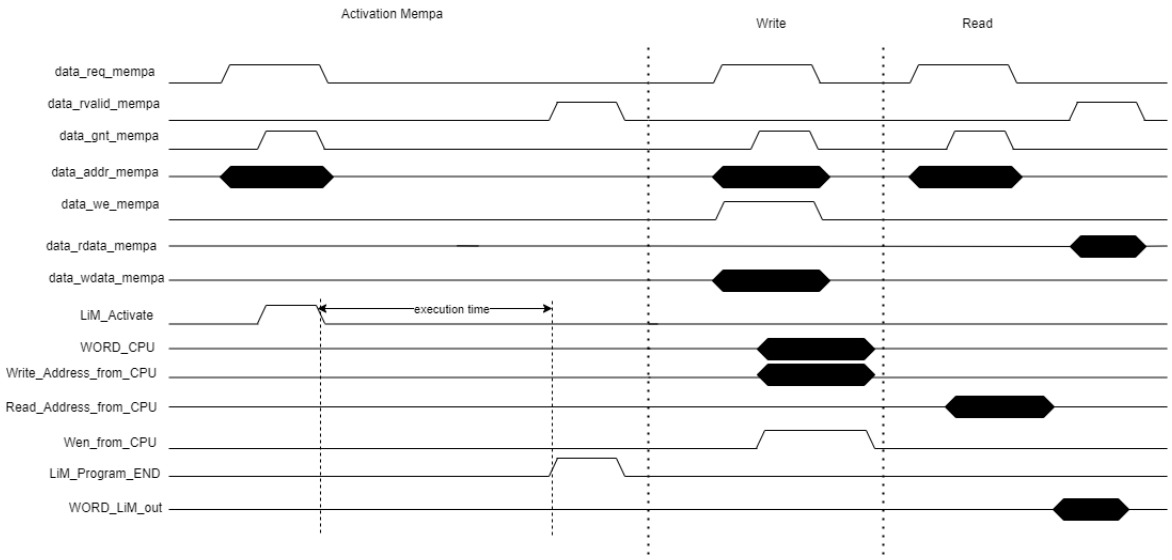


Figura 5.6. Protocol translation done by Controller.

One of the responsibilities of the controller is to translate between one protocol and another. The steps are depicted in Figure 5.6 and are detailed below:

1. It verifies the validity of the address; if valid, it raises the data_gnt signal to release the Ri5cy, which is meanwhile awaiting a response.

2. Upon receiving a request, it checks the data_we signal; based on its value, it transfers the address to either the read_address_from_CPU or the write_address_from_CPU signal.

3. Once the data_gnt signal is asserted, it checks whether the address corresponds to a read or write request. If affirmative, it raises the data_rvalid signal; otherwise, it waits for the LiM to complete the algorithm execution and assert the LiM_program_end signal before raising data_rvalid.

Another responsibility is the initialization of the Mempa to use it as a LiM. The Mempa, containing an address queue, needs to be initialized with the starting addresses of the code based on the selected µROM. For this reason, once the controller receives the LiM activation signal, it initializes the queue with the code's starting address and subsequently propagates the LiM_activate signal.

The last Controller action is address translation. The Ri5scy sends addresses based on RAM memory and its storage of words within, thereby providing addresses in multiples of four. However, the Mempa, comprised of smart blocks, references them with subsequent addresses. The controller translates the addresses given by the Ri5scy by shifting them by two to send the data to the correct Mempa address. Without this action, only a quarter of the Mempa could be utilized.

A more specific description of the controller is illustrated inside the Figure 5.8.

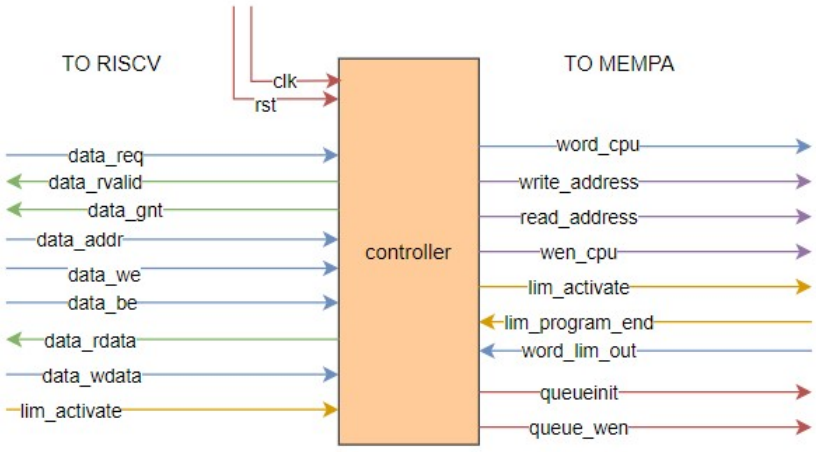


Figura 5.7. Controller General scheme

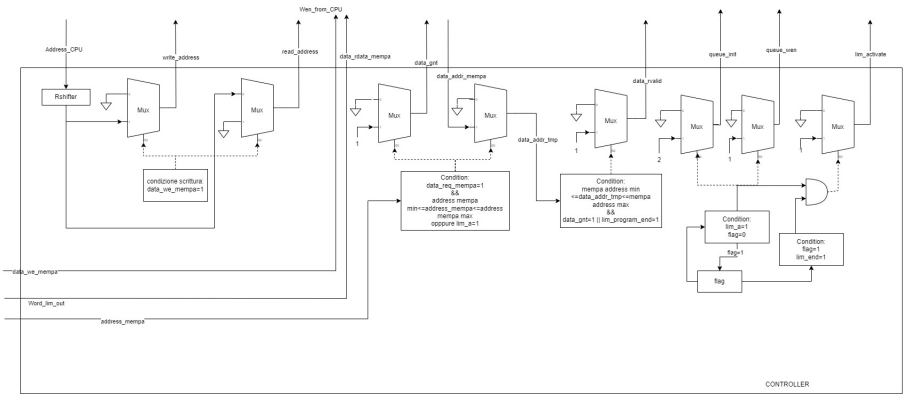


Figura 5.8. Controller Structure

The signals `write_address`, `read_address`, `lim_activate`, `queueinit`, and `queue_wen` are outputs from the controller to the memory that need to be controlled by logic; all of them are selected through a multiplexer controlled by conditions based on input signals. `write_address` and `read_address` are complementary and depend on the same control on a specific signal: `data_we`. If `data_we` is asserted, it means that Ri5cy is requesting a write, and then the `data_addr` signal (which is shifted right by 4) is assigned to `write_address`; vice versa for `read_address`.
`Queueinit` and `queue_wen` also depend on the same and only condition. This is because both signals are used to initialize the memory. The condition is that there is no program currently being executed by the memory (this is where the variable `flag` comes into play; in fact, `flag` must be 0) and that there is a request from Ri5cy to activate the memory

through the signals `lim_a`. If this condition is met, the correct values of the two signals are set, and the flag is set to 1; otherwise, both are set to ground.
`Lim_activate` (which may seem like a simple connection of two identical signals) has a more complex condition because it must wait for `lim` to be initialized and is therefore delayed by one clock cycle. To achieve this, it has a control with a flag. When the flag is 1 (indicating that the previous initialization step has been executed), `lim_activate` can be asserted; otherwise, it is connected to ground.
Two other signals, `wen_cpu` and `word_cpu`, are output signals to the memory, but having no conditions, they are directly connected to the input signals `data_we` and `data_wdata`. Regarding the output signals to Ri5cy, `data_rvalid` and `data_gnt` also depend on timing conditions, while `data_rdata`, having no conditions, is directly connected to `word_lim_out`.
`Data_gnt` has the normal control condition, i.e., if a request has been activated by Ri5cy and if the requested address is correct. Meanwhile, `data_rvalid` has a slightly time-dependent condition because it must differ from `data_gnt` by one clock cycle. For this reason, a variable `data_addr_tmp` is required, which, to be set, needs the same condition used by `data_gnt`. If this condition is true, it takes the value of the input address signal `data_addr`; otherwise, it is set to ground. This variable is used in another condition to check that the address is valid and that `data_gnt` is set to 1. Additionally, when the Lim_program_end signal is raised, if these two cases occur, `data_rvalid` is set to 1; otherwise, it is connected to ground.

Capitolo 6

Activation LiM in C code

To enable the LiM functionality, a unique address beyond the range of the primary memory and Mempa has been specified. To be more precise, this address is calculated based on the sizes of both the RAM and Mempa, as elucidated in a preceding section. In this specific instance, the hexadecimal representation of this address is `0x00100600`. Executing any operation at this address initiates the execution of the program stored in the µROM by Mempa.

Examining the provided code snippet, we observe a write operation to this particular address. However, before any operation can be carried out successfully, it is imperative for Mempa to already contain the relevant values for processing. In the given context, the preceding step involves the complete writing of data to Mempa. Alternatively, users may choose to write only to specific addresses of interest.

Prior to engaging in any operation, the Mempa must be populated with the necessary values. In this case, a comprehensive writing operation is demonstrated in the preceding steps. This ensures that Mempa holds the requisite data for subsequent processing. It is worth noting that users also have the flexibility to selectively write only to the specific addresses relevant to their processing requirements. An essential consideration, albeit one typically discouraged in C programming practices, involves the specification of the initial addresses for both the main memory and the activation address for program execution. This caution arises from the fact that the processor lacks awareness of any inherent difference between conventional memory and Mempa; it treats them as a singular, unified memory entity.

This cautionary approach ensures a more robust and predictable system behavior, preventing unintended consequences that may arise from the processor's inability to discern between the two memory types. By adhering to standard memory management practices, developers can leverage the inherent capabilities of the system without introducing unnecessary complications related to manual address assignments for memory and execution activation.

```
int * addressmem= 0x001000;
int * addressMempa= 0x0FFE00;
int * addresslim= 0x00100600;

        int i,j;
        //fill the Mempa
        for(i=0;i<ROW;i++){
                for(j=0;j<COL;j++){
                        addressMempa[i*COL+j]=i+j;
                }
        }
        //activate the LiM
        addresslim[0]=1;
```

To facilitate the processing of any given value, a critical prerequisite is to verify the presence of the intended program within the µROM before commencing the simulation. Unlike certain automated processes, the loading of the µROM demands a manual intervention due to its inability to be automated.
This manual loading procedure entails the careful and deliberate insertion of the required program into the µROM, ensuring that it aligns with the specific processing needs. This step is crucial in guaranteeing the availability of the relevant instructions and data within the µROM, setting the stage for a successful and accurate simulation.
The manual nature of this process also serves as a safeguard against inadvertent errors or oversights, allowing developers to meticulously review and verify the contents of the µROM before simulation initiation. While automated procedures can streamline certain aspects of development, the manual loading of the µROM remains a deliberate and essential step to ensure precision and reliability in the subsequent processing of values during simulation.

Capitolo 7

Description of others µROM algorithms

For the various simulations, different algorithms were used; some of them were already available in C code and only needed to be transformed into a series of instructions for the uROM, while others required the opposite process.
In this chapter, we take a brief look at how various C codes for the Ri5cy with a few instructions are transformed into a series of instructions to be executed by Mempa.
We can observe, of course, that these algorithms are much lighter and use less memory and resources within the smart block itself compared to the lenet-5 algorithm, and it is also for this reason that the instructions appear simpler.

7.1 Aes-128 AddRoundKey

"Add Round Key" is a step in the AES (Advanced Encryption Standard) algorithm during its encryption process, specifically in the SubBytes, ShiftRows, MixColumns, and AddRoundKey operations. AES operates on blocks of data, and the "Add Round Key" step involves bitwise XOR (exclusive OR) of the block with a round key derived from the original encryption key.

In the case of AES-128, which uses a 128-bit key, there are 10 rounds of processing. The "Add Round Key" operation is performed in each round, where the block of data is XORed with a portion of the expanded key schedule. The key schedule is derived from the original 128-bit key and is expanded to provide a unique round key for each round of the encryption process.

The purpose of the "Add Round Key" step is to introduce the secret key into the encryption process and ensure that each round of encryption uses a different portion of the expanded key. This operation adds a layer of complexity and security to the encryption algorithm by incorporating the unique properties of the secret key into each round of

processing, making it more resistant to various cryptographic attacks.
Below, we display the C code that needs to be transformed into Mempa code.

```
/* Add around key */
   for (i=0; i<4; i++) {
       for (j=0; j<4; j++) {
           (*states)[i][j] = (*states)[i][j] ^ (*key)[i][j];
       }
   }
```

The data for this algorithm consists of two matrices that are placed in the initial smart blocks of Mempa, as described in Figure 7.1.

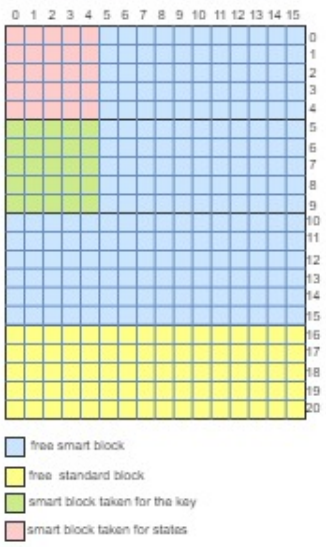


Figura 7.1. Initial Values in Mempa for AES 128 algorithm

The translation of this code is done in 3 steps. The first step involves preserving the content of the memory by saving all the data in register R0, which will not be used from this point onward. The following two steps, which are the most interesting, will be shown next.

- **Step 1:** Very simple operation save the content of the matrix inside R0 that will not be used.

- **Step 2:** Add to Bypass all the element of the key.

```
init_instructions(2)<=

  "01"&std_logic_vector(to_unsigned(3,SIZE_uROM_Address))&
  '0'&'0'&four_col_enabled&'1'&

  "00001"&OP_Load&Mem&DEST_Bypass&nullADDR_S1&nullADDR_S2&
  nullADDR_D&nullFunc&

  "11100"&OP_Load&Mem&DEST_Bypass&nullADDR_S1&nullADDR_S2&
  nullADDR_D&nullFunc&

  Disable_instr_dec_last;
```

In this step, we prepare the Mempa for the next instruction by loading the current values of the smart blocks used for the Key matrix into rows 4, 5, 6, and 7 (as specified in the row enable "00001" and "11100") and columns 0, 1, 2, 3, and 4 as specified by the value four_col_enabled="1111100000000000".

- **Step 3:** Computation of the XOR between source and key .

```
init_instructions(3)<=

  "01"&std_logic_vector(to_unsigned(4,SIZE_uROM_Address))&
  '0'&'0'&four_col_enabled&'1'&

  "11110"&OP_Logic_Adder&Col_int_Mem&DEST_Mem&
  std_logic_vector(to_unsigned(4,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&XOR_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

In this final step, only the first nCU is used to select the smart blocks of the source. Through the column interconnections Col_int_Mem, the equivalent values of the key are taken, discarding 4 positions for each smart block, respectively ADDR_S1=4. The XOR operation XOR_OP is computed, enabling the OP_Logic_Adder operation, which sends the enable signal to each individual ALU present in each smart block. Once the calculation is done, it saves everything in the memory of the smart block.

The result of this algorithm will be visible at the same address as the source.

7.2 Bitwise

Bitwise is a type of operation performed on each bit of a binary representation of data. These operation involves direct manipulations of bits within a binary data, and are often used in programming to perform bitwise-level operations.

The main bitwise operations that are performed in the algorithm are:

- Bitwise AND (&): At the bit level, if both bits are 1, the result is 1. In C, the bitwise AND operator is represented by &.
- Bitwise OR (|): At the bit level, if at least one of the bits is 1, the result is 1. In C, the bitwise OR operator is represented by |.
- Bitwise XOR (^): At the bit level, if the bits are different, the result is 1; if they are the same, the result is 0. In C, the bitwise XOR operator is represented by ^.

These operations are often used to perform manipulations on binary data, control or set specific bits in a binary representation, and carry out other advanced bitwise-level operations.

In the block below, the code used to perform the Bitwise operation is shown. It's important to note that the vector "vector" is a 15-element vector that is initialized beforehand.

```
/* OR operation */
mask_or = 0xF1;
for(i=0; i<N; i++){
        vector[i] = vector[i] | mask_or;
}
*stand_alone = *stand_alone | mask_or;

/* AND operation */
mask_and = vector[N-1] & 0x8F;
for(i=0; i<N; i++){
        vector[i] = vector[i] & mask_and;
}
*stand_alone = *stand_alone & mask_and;

/* XOR operation */
mask_xor = vector[N-2] ^ 0xF0;
for(i=0; i<N; i++){
        vector[i] = vector[i] ^ mask_xor;
}
*stand_alone = *stand_alone ^ mask_xor;

*final_result = ~vector[N-3] + ~(*stand_alone);
```

For this Bitwise algorithm, the first step before executing the program in memory was to fill it with the necessary data. Below is an image showing the placement of the variables that are then used in the processing 7.2. Inside the memory, there will be the 'vector' array, the "stand_alone" variable, and the three masks. In total, the algorithm consists of 9 instructions, of which, as is now known, the first is used to preserve the value of the matrix.

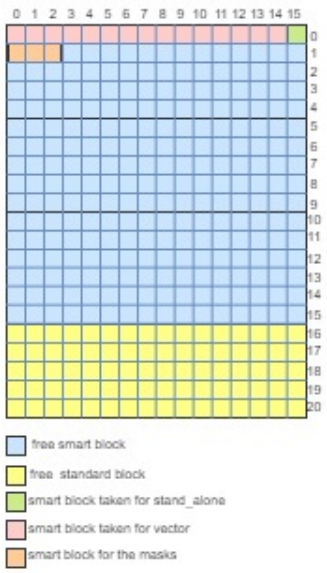


Figura 7.2. Initial Values in Mempa for Bitwise algorithm

- **Step 1:** Very simple operation save the content of the matrix inside R0 that will not be used.

- **Step 2:** Move the three masks into the bypass register of the smart block.

```
init_instructions(2)<=

  "01"&std_logic_vector(to_unsigned(3,SIZE_uROM_Address))&
  '0'&'0'&three_col_enabled&'1'&

  "01000"&OP_Load&Mem&DEST_Bypass&nullADDR_S1&nullADDR_S2&
  nullADDR_D&OR_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

It is selected only the first nCU because it is the only one involved in this algorithm, and from its section, the first row is taken as specified by `"01000"` and the first three columns (`three_col_enabled="1110000000000000"`). These are the three masks used in this algorithm, and them are loaded into the same bypass register `DEST_Bypass` . This step will be used later to retrieve the masks.

- **Step 3:**Make to OR operation between the vector and the OR-mask.

```
init_instructions(3)<=

  "01"&std_logic_vector(to_unsigned(4,SIZE_uROM_Address))&
  '0'&'0'&All_Col_Enabled&'1'&

  "10000"&OP_Logic_Adder&Mem_MI_int&DEST_Mem&nullADDR_S1&
  std_logic_vector(to_unsigned(16,SIZE_ADDR_S2))&nullADDR_D&OR_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

The smart blocks of the vector and the unique variable 'stand_alone' are selected, and through memory interconnection, a single address is chosen:
`std_logic_vector(to_unsigned(16,SIZE_ADDR_S2))` . This address will be used to perform the OR operation on all the selected smart blocks, i.e., all the columns of the first row. The result is then saved in the same smart block.

- **Step 4:** The AND mask and the last element of the vector perform an AND.

```
init_instructions(4)<=

  "01"&std_logic_vector(to_unsigned(5,SIZE_uROM_Address))&
  '0'&'0'&second_col_enabled&'1'&

  "10000"&OP_Logic_Adder&Mem_MI_int&DEST_Bypass&nullADDR_S1&
  std_logic_vector(to_unsigned(14,SIZE_ADDR_S2))&nullADDR_D&AND_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

In this instruction, the AND-mask is computed. Only its smart block is selected, as can be seen from the row and column enables (`second_col_enabled="0100000000000000"`). Through memory interconnection, it takes the last value of the vector available at address `ADDR_S2=14`, computes the AND operation, and saves the value in the bypass register to be used in the next instruction.

- **Step 5:** AND operation.

```
init_instructions(5)<=

  "01"&std_logic_vector(to_unsigned(6,SIZE_uROM_Address))&
  '0'&'0'&All_Col_Enabled&'1'&

  "10000"&OP_Logic_Adder&Mem_MI_int&DEST_Mem&nullADDR_S1&
  std_logic_vector(to_unsigned(17,SIZE_ADDR_S2))&nullADDR_D&AND_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

The just-computed mask is retrieved through memory interconnection at address `ADDR_S2=17`, and the AND operation (`AND_OP`) is performed on every smart block that saved the vector and the variable, covering the entire column of the first row.

- **Step 6:** Update of the XOR-mask.

```
init_instructions(6)<=

  "01"&std_logic_vector(to_unsigned(7,SIZE_uROM_Address))&
  '0'&'0'&third_col_enabled&'1'&

  "10000"&OP_Logic_Adder&Mem_MI_int&DEST_Bypass&nullADDR_S1&
  std_logic_vector(to_unsigned(14,SIZE_ADDR_S2))&nullADDR_D&XOR_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

In this case, like the AND mask, the XOR mask is also updated. Its smart block is selected, and by taking the element at address ADDR_S2=14 (the last element of the vector), the ALU is used for XOR (XOR_OP), and the result is saved in the bypass register DEST_Bypass .

- **Step 7:** XOR operation.

```
init_instructions(7)<=

  "01"&std_logic_vector(to_unsigned(8,SIZE_uROM_Address))&
  '0'&'0'&All_Col_Enabled&'1'&

  "10000"&OP_Logic_Adder&Mem_MI_int&DEST_Mem&nullADDR_S1&
  std_logic_vector(to_unsigned(18,SIZE_ADDR_S2))&nullADDR_D&XOR_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

Same as the previous step but for XOR: all the columns of the first row are selected, and through the memory interconnection, the smart block of the XOR mask is chosen. After performing the XOR, everything is saved in memory.

- **Step 8:** NOt su due elementi del vettore

```
init_instructions(8)<=

  "01"&std_logic_vector(to_unsigned(9,SIZE_uROM_Address))&
  '0'&'0'&eleven_last_col_enabled&'1'&

  "10000"&OP_Logic_Adder&Mem&DEST_Bypass&nullADDR_S1&
  nulllADDR_S2&nullADDR_D&NOT_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

This penultimate instruction prepares the final result of this algorithm by taking the last element and the eleventh element of the vector, as specified by the row enables "10000" and the column enables eleven_last_col_enabled="0000000000010001". It performs a NOT operation (NOT_OP) on both, and then saves them in the bypass register DEST_Bypass .

- **Step 9:** Perform the sum of the two values of the vector and find the result.

```
init_instructions(9)<=

  "01"&std_logic_vector(to_unsigned(10,SIZE_uROM_Address))&
  '0'&'0'&eleven_col_enabled&'1'&

  "10000"&OP_Logic_Adder&Row_int_Col_int&DEST_RF&
  std_logic_vector(to_unsigned(0,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(4,SIZE_ADDR_S2))&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_D))&SUM_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

This is the last operation that leads to the final result of the bitwise. Both values are taken from the bypass through row and column connections. Through the column connection, the value of the selected smart block's bypass register is taken since the position difference is 0 (ADDR_S1=0). Meanwhile, through the row interconnections, the last value of the vector is taken (ADDR_S2=4),a sum is performed and the result is saved in the register file at the third register (ADDR_D=3).

In addition to the bitwise operation, there is also code for bitwise negation, where operations of NAND, NOR, and XNOR are performed on the same vector. These operations won't be explained here as the instructions for the Mempa are almost identical.

7.3 Bitmap research

Bitmap indexing or bitmap search refers to a data structure and search algorithm that uses bitmaps to represent sets of data efficiently. This technique is often employed in databases and information retrieval systems.
In a bitmap index, each distinct value in a dataset is associated with a bitmap. The bitmap is a binary representation where each bit corresponds to a row or record in the dataset. If a particular value is present in a row, the corresponding bit in the bitmap is set to 1; otherwise, it's set to 0.
To search for data, you perform logical operations (AND, OR, XOR, etc.) on these bitmaps. For example, to find rows where multiple conditions are satisfied, you perform a bitwise AND operation on the corresponding bitmaps. Bitmap search is particularly efficient when dealing with low cardinality columns (columns with a small number of distinct values).
Below is a search for two queries: males older than 19 years and individuals older than 18 years.

```
    /* Initialize bitmap */
    int *v_age16   = 0x30000;
    int *v_age17   = 0x30018;
    int *v_age18   = 0x30030;
    int *v_age19   = 0x30048;
    int *v_age20   = 0x30060;
    int *v_genderM = 0x30078;
    int *v_genderF = 0x30090;

/* Initialise results to 0 */
    for(i=0; i<6; i++) {
        result_M_over19[i] = 0;
        result_over18[i]   = 0;
    }

    /* Query: identify male people that are 19 or 20 */
    for(i=0; i<6; i++) {
        result_M_over19[i] = v_genderM[i] & (v_age19[i] | v_age20[i]);
    }

    /* Query: identify people that are older than 18 */
    for(i=0; i<6; i++) {
        result_over18[i] = ~v_age16[i] & ~v_age17[i] ;
    }
```

In this algorithm, a total of 7 bitmaps are used, as shown in the figure below 7.3. In addition to the 7 bitmaps, we also have two result vectors. The total instructions are 8 and in the following table they are well explained.

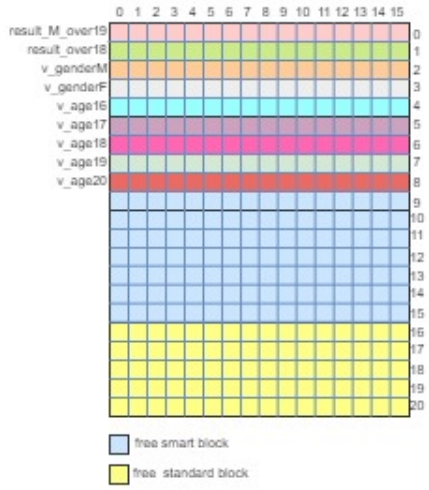


Figura 7.3. Initial Values in Mempa for Bitmap research algorithm

- **Step 1:** Very simple operation save the content of the matrix inside R0 that will not be used.
- **Step 2:** Move the vector v_age_20 in Bypass .

```
init_instructions(2)<=

  "01"&std_logic_vector(to_unsigned(3,SIZE_uROM_Address))&
  '0'&'0'&five_col_enabled&'1'&

  Disable_instr_dec&

  "00010"&OP_Load&Mem&DEST_Bypass&nullADDR_S1&nullADDR_S2&
  nullADDR_D&nullFunc&

  Disable_instr_dec_last;
```

In this instruction, on line 8, " "00010" " is activated, causing two out of the three nCU to be disabled. The first five columns are then put into the Bypass DEST_Bypass , and for this reason, no address needs to be specified.

- **Step 3:** Performing OR operation between v_age_19 and v_age_20 .

```
init_instructions(3)<=

  "01"&std_logic_vector(to_unsigned(4,SIZE_uROM_Address))&
  '0'&'0'&five_col_enabled&'1'&

  Disable_instr_dec&

  "00100"&OP_Logic_Adder&Col_int_Mem&DEST_Bypass&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&OR_OP&

  Disable_instr_dec_last;
```

In this case, only one line is considered, and it is line 7 that contains v_age_19. It takes the next row via column connections with ADDR_S1=1 and performs a bitwise OR operation OR_OP, saving the result in the Bypass register.

- **Step 4:** The row of V_genderM takes the precedent results and perform an AND.

```
init_instructions(4)<=

  "01"&std_logic_vector(to_unsigned(5,SIZE_uROM_Address))&
  '0'&'0'&five_col_enabled&'1'&

  "00100"&OP_Logic_Adder&Col_int_Mem&DEST_Bypass&
  std_logic_vector(to_unsigned(5,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&AND_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

The penultimate step of the first query is the selection of the first 5 columns five_col_enabled="1111100000000000" of the third row, i.e., the smart blocks that store the vector v_genderM. Through column interconnection, the bypass register of the 5 columns of the seventh row is selected and used as operands for the AND_OP operation. This value will then be saved in the Bypass register of the selected smart blocks.

- **Step 5:** Save the results into the vector of the query.

```
init_instructions(5)<=

  "01"&std_logic_vector(to_unsigned(6,SIZE_uROM_Address))&
  '0'&'0'&five_col_enabled&'1'&

  "10000"&OP_Load&Col_int_Mem&DEST_Mem&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&nullFunc&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

This is the final step of the first query. The only purpose of the instruction is to retrieve, through row interconnection Col_int_Mem, the row located two positions below (ADDR_S1=2), which corresponds to the result of the query. The retrieved result is then stored in the smart blocks of the result vector.

- **Step 6:** Not operation of v_age_16 and v_age_17.

```
init_instructions(6)<=

  "01"&std_logic_vector(to_unsigned(7,SIZE_uROM_Address))&
  '0'&'0'&five_col_enabled&'1'&

  "00001"&OP_Logic_Adder&Mem&DEST_Bypass&nullADDR_S1&
  nulllADDR_S2&nullADDR_D&NOT_OP&

  "10000"&OP_Logic_Adder&Mem&DEST_Bypass&nullADDR_S1&
  nulllADDR_S2&nullADDR_D&NOT_OP&

  Disable_instr_dec_last;
```

The second query starts with the perform of the NOT operation NOT_OP of two rows of the Mempa that contains the vector v_age_16 "00001" (the fifth column) and the one containing the vector v_age_17 "10000" (the sixth row) the value negated is then saved inside the Bypass register.

- **Step 7:** Saving the vector v_age_17 inside the result vector.

```
init_instructions(7)<=

  "01"&std_logic_vector(to_unsigned(8,SIZE_uROM_Address))&
  '0'&'0'&five_col_enabled&'1'&

  "01000"&OP_Load&col_int_mem&DEST_mem&
  std_logic_vector(to_unsigned(4,SIZE_ADDR_S1))&
  nullADDR_S2&nullADDR_D&&nullFunc&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

The value taken by column interconnections col_int_mem 4 position under the smart block enabled ADDR_S1=4 is saved inside the memory of the smart block it-selves ready to the next and last step.

- **Step 8:** Perform the AND between the v_age_16 and v_age_17 negated.

```
init_instructions(8)<=

  "01"&std_logic_vector(to_unsigned(9,SIZE_uROM_Address))&
  '0'&'0'&five_col_enabled&'1'&

  "01000"&OP_Logic_Adder&col_int_mem&DEST_mem&
  std_logic_vector(to_unsigned(3,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&&AND_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

In this last step the value of v_age_16 negated that previously has been saved inside the bypass register of the first five smart block in the fourth row is taken ADDR_S1=3 by the smart block that needs to contain the results of the second query (and now contains the value of v_age_17) that perform an and operation AND_OP and than is saved inside the smart block.

7.4 Xnor-Net

XNOR-Net is a binary neural network variant designed to efficiently perform convolution operations using binary operations like XNOR, with the addition of scaling factors to improve accuracy.
Binary convolution is performed using XNOR and pop-count operations. The XNOR operation is similar to XOR but produces a result of 1 only when both operands are 1.
To address the precision loss associated with binarization, XNOR-Net introduces full-precision (e.g., 32-bit) scaling factors. These scaling factors are multiplied with the results of binary convolution to enhance the accuracy of the network.

In this specific algorithm that describes the computation of the xor-network, there is only one channel, so the iteration over the matrix is done only once. The image matrix and the offmap matrix are both 16x16, while the filter is 2x2.

```
for(c = 0; c < n_channels; c++)
{
for(j = 0; j < w_out; j++)
{
    for(i = 0; i < w_out; i++)
    {
        for(m = 0; m < wf; m++)
        {
            for(t = 0; t < wf; t++)
            {
                A = j+m+j*(stride-1);
                B = i+t+i*(stride-1);
                (*ofmap)[j][i] = ((*image)[A][B]) | ((*ofmap)[j][i] << 1);
                if (flag == 0)
                {
                    bWeight = ( (*weight)[m][t] ) | (bWeight << 1);
                }
            }
        }
        //xor bitwise between the ofmap content and the binary weight
        (*ofmap)[j][i] = (*ofmap)[j][i] ^ bWeight;
        flag = 1;
    }
}
```

Initially, the matrices are stored in the memory map (Mempa), and the result will always be a 16x16 matrix that will replace the image matrix. Below is the description of all the steps used for the implementation of the algorithm. In total, there are 17 instructions, and the result is then saved inside the smart blocks. It can be visualized using a read operation. As we can see, Step 1 is always the same for all algorithms.

- **Step 1:** Very simple operation save the content of the matrix inside R0 that will

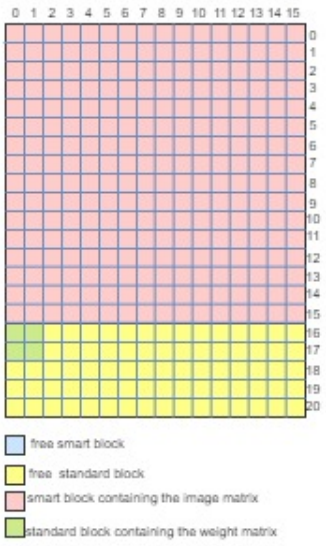


Figura 7.4. Initial Values in Mempa for Xnor-net algorithm

not be used.

- **Step 2:** Move the matrix image in the register 1.

```
init_instructions(2)<=

  "01"&std_logic_vector(to_unsigned(3,SIZE_uROM_Address))&
  '0'&'0'&All_Col_Enabled&'1'&

  "11111"&OP_Load&Mem&DEST_RF&nullADDR_S1&nullADDR_S2&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&nullFunc&

  "11111"&OP_Load&Mem&DEST_RF&nullADDR_S1&nullADDR_S2&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&nullFunc&

  "111111"&OP_Load&Mem&DEST_RF&nullADDR_S1&nullADDR_S2&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_D))&nullFunc&
```

To enable the calculation of bweight, all elements go through an OP_Load from the memory of the smart block Mem to the register file DEST_RF, the first register ADDR_D=1, and from there, they will be retrieved later.

- **Step 3:** Computation of the bWeight taking the first element of weight and making OR to bweight.

```
init_instructions(3)<=

  "01"&std_logic_vector(to_unsigned(4,SIZE_uROM_Address))&
  '0'&'0'&first_one_col_enabled&'1'&

  "10000"&OP_Logic_Adder&Mem_Col_int&DEST_Mem&
  std_logic_vector(to_unsigned(16,SIZE_ADDR_S1))&
  nullADDR_S2&nullADDR_D&OR_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

In this case, only one smart block is selected, namely the first smart block (row 0, column 0), for the computation of bWeight. The first element of the matrix takes, through the column interconnection, the first element of the weight matrix located 16 rows away ADDR_S1=16. An OR operation is performed between the element in the smart block's memory and the retrieved element, and everything is saved in memory.

- **Step 4:** Shifting the bweight.

```
init_instructions(4)<=

  "01"&std_logic_vector(to_unsigned(5,SIZE_uROM_Address))&
  '0'&'0'&first_one_col_enabled&'1'&

  "10000"&OP_RShifter&Mem_mem&DEST_Mem&nullADDR_S1&
  nullADDR_S2&nullADDR_D&Logic_RShift_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

As described in the code, between one computation and the next, the value of bWeight must be shifted one position to the right. In this case, the same enables as the previous instruction are used, so only the first element of the memory is selected, and the shift operation OP_RShifter is chosen. For this operation, two types of

shifting can be selected: logical or arithmetic. In this case, the logical shift will be used Logic_RShift_OP. The shift will be applied to the element stored in the smart block's memory Mem_mem and saved in the same location DEST_Mem.

- **Step 5:** Or operation between first element of the matrix and second of the weight matrix.

```
init_instructions(5)<=

  "01"&std_logic_vector(to_unsigned(6,SIZE_uROM_Address))&
  '0'&'0'&first_one_col_enabled&'1'&

  "10000"&OP_Logic_Adder&Mem_MI_int&DEST_Mem&nullADDR_S1&
  std_logic_vector(to_unsigned(256,SIZE_ADDR_S2))&nullADDR_D&OR_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

In this case, since the second element of the weight matrix is not in the same column, the column interconnection cannot be used. Instead, the memory interconnection is used by taking the address of the smart block where the weight element is stored (i.e., the second column of row 16) ADDR_S2=256. Once the element is obtained and the OR operation OR_OP is performed, the result is saved in the smart block.

- **Step 6:** As for the step 4 a Shift of the first element of the matrix is performed.
- **Step 7:** Or performed between third element of matrix weight and bweight_tmp.

```
init_instructions(7)<=

  "01"&std_logic_vector(to_unsigned(8,SIZE_uROM_Address))&
  '0'&'0'&first_one_col_enabled&'1'&

  "10000"&OP_Logic_Adder&Mem_Col_int&DEST_Mem&
  std_logic_vector(to_unsigned(17,SIZE_ADDR_S1))&nullADDR_S2&
  nullADDR_D&OR_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

This operation is very similar to step 3; in this case as well, to reach element 3 of the weight matrix, which is located in the first element of the last row of the map, we use the column interconnection by taking the first element (since the first element is selected by first_one_col_enabled) of the row at a distance of 17 positions as specified in the nInstruction ADDR_S1=17 . Also in this case, the logical operation is an OR, and the element is saved in the smart block.

- **Step 8:** As for the step 4 a Shift of the first element of the matrix is performed.

- **Step 9:** Computation between the last element of weight and bweight.

```
init_instructions(9)<=

  "01"&std_logic_vector(to_unsigned(10,SIZE_uROM_Address))&
  '0'&'0'&first_one_col_enabled&'1'&

  "10000"&OP_Logic_Adder&Mem_MI_int&DEST_Bypass&nullADDR_S1&
  std_logic_vector(to_unsigned(272,SIZE_ADDR_S2))&nullADDR_D&OR_OP&

  Disable_instr_dec&

  Disable_instr_dec_last;
```

This instruction, like one of the previous steps, takes the last element of the weight matrix, which in this case is at address 272 ADDR_S2=272 through memory interconnection, and performs the OR operation OR_OP , saving it in the register Bypass of the smart block DEST_Bypass .
With this step, the computation for the bweight value is complete.

- At this point, a loop follows consisting of two instructions between OR and shift computations. This loop is repeated 3 times, and in the end, we will have the result matrix of the XNOR-Net.

  Step 10: Or between the offmap matrix and image matrix.

  Step 11: Shift to the right of the offmap matrix.

```
init_instructions(10)<=

  "01"&std_logic_vector(to_unsigned(11,SIZE_uROM_Address))&
  '0'&'0'&All_Col_Enabled&'1'&

  "11111"&OP_Logic_Adder&RFA_RFB&DEST_Mem&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S2))&nullADDR_D&OR_OP&

  "11111"&OP_Logic_Adder&RFA_RFB&DEST_Mem&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S2))&nullADDR_D&OR_OP&

  "111111"&OP_Logic_Adder&RFA_RFB&DEST_Mem&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S2))&nullADDR_D&OR_OP;

init_instructions(11)<=

  "01"&std_logic_vector(to_unsigned(12,SIZE_uROM_Address))&
  '0'&'0'&All_Col_Enabled&'1'&

  "11111"&OP_RShifter&Mem_mem&DEST_RF&nullADDR_S1&
  nullADDR_S2&std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&
  Logic_RShift_OP;

  "11111"&OP_RShifter&Mem_mem&DEST_RF&nullADDR_S1&
  nullADDR_S2&std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&
  Logic_RShift_OP;

  "111111"&OP_RShifter&Mem_mem&DEST_RF&nullADDR_S1&
  nullADDR_S2&std_logic_vector(to_unsigned(2,SIZE_ADDR_D))&
  Logic_RShift_OP;
```

Step 10 performs the OR operation OR_OP on all smart blocks in memory, as required by the algorithm. In this case, the operation is performed between the image matrix, which is stored in register r1 ADDR_S1=1, and the offmap matrix (i.e., the result matrix), which will be stored in register r2 ADDR_S2=2, and saves the result in memory DEST_Mem.

In step 11, the result matrix from the previous step is right-shifted using OP_RShifter

with a logical shift (Logic_RShift_OP), and it is saved in the register file at register 2 (ADDR_D=2), ready to be used in the next iteration of the loop.

- **Step 16:** Or operation between the matrix image and offmap.

```
init_instructions(16)<=

  "01"&std_logic_vector(to_unsigned(17,SIZE_uROM_Address))&
  '0'&'0'&All_Col_Enabled&'1'&

  "11111"&OP_Logic_Adder&RFA_RFB&DEST_Mem&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S2))&nullADDR_D&OR_OP&

  "11111"&OP_Logic_Adder&RFA_RFB&DEST_Mem&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S2))&nullADDR_D&OR_OP&

  "111111"&OP_Logic_Adder&RFA_RFB&DEST_Mem&
  std_logic_vector(to_unsigned(1,SIZE_ADDR_S1))&
  std_logic_vector(to_unsigned(2,SIZE_ADDR_S2))&nullADDR_D&OR_OP&
```

This last operation is identical to step 10.

- **Step 17:** Or operation between the matrix offmap and the value bweight .

```
init_instructions(17)<=

  "01"&std_logic_vector(to_unsigned(18,SIZE_uROM_Address))&
  '0'&'0'&All_Col_Enabled&'1'&

  "11111"&OP_Logic_Adder&Mem_MI_int&DEST_Mem&nullADDR_S1&
  std_logic_vector(to_unsigned(0,SIZE_ADDR_S2))&nullADDR_D&OR_OP&

  "11111"&OP_Logic_Adder&Mem_MI_int&DEST_Mem&nullADDR_S1&
  std_logic_vector(to_unsigned(0,SIZE_ADDR_S2))&nullADDR_D&OR_OP&

  "111111"&OP_Logic_Adder&Mem_MI_int&DEST_Mem&nullADDR_S1&
  std_logic_vector(to_unsigned(0,SIZE_ADDR_S2))&nullADDR_D&OR_OP&
```

This is the final step of the algorithm, an OR (OR_OP) operation between the value of bweight, which was calculated at the beginning and placed in the bypass register of the first smart block of the matrix (and never modified), and all the elements of the resulting matrix offmap. Through memory connections, the address of bweight is selected (ADDR_S2=0) and an OR operation is performed with the memory Mem_MI_int . The final resulting matrix is then saved in memory DEST_Mem , ready to be read.

Parte III

Results

Capitolo 8

Simulation

The simulations were conducted on diverse algorithmic approaches that leverage varying amounts of data, exhibiting different levels of complexity. Certain simulations involve 16x16 matrices, with evaluations on these matrices proving to be highly advantageous for Mempa, given its intended use. Conversely, other algorithms utilize vectors containing only five elements, and in such cases, the observed differences in performance tend to be minimal.

Simulation times were computed based on two distinct benchmarks: the standard time required for loading Mempa and initializing it, and the effective processing time of the algorithm. This was then compared to the execution time of the algorithm when run exclusively on Ri5cy, facilitated by a C program.

Multiple makefile recipes were employed for the simulation process.

- One recipe was designed for the compilation and execution of the architecture with Mempa, and although this remained constant across various algorithms, the uROM component varied. It needed to be manually included among the RTL (Register-Transfer Level) sources.

  ```
  .PHONY: custom-vsim-run-start
  custom-vsim-run-start: vsim-all custom/start.hex
  custom-vsim-run-start: ALL_VSIM_FLAGS += "+firmware=custom/start.hex" \
  +verbose
  custom-vsim-run-start: vsim-run
  ```

  This recipe compiles all the files related to Mempa and Ri5cy, running vsim with the start.hex code, which is the hexadecimal representation of the C code seen earlier that fills the Mempa and activate the Mempa as a LiM. This C code is first transformed into ".elf" code and then into ".hex" code before being executed in this recipe.

- Regarding the execution with only Ri5cy, there are multiple recipes, one for each hexadecimal code.

```
.PHONY: custom-vsim-run-gui-knn
custom-vsim-run-gui-knn: vsim-all-noMempa custom/knn.hex
custom-vsim-run-gui-knn: ALL_VSIM_FLAGS += \
"+firmware=custom/knn.hex" +verbose
custom-vsim-run-gui-knn: vsim-run-gui-noMempa
```

In this case, let's take the example of the recipe for running the K-NN (K-Nearest Neighbors) code via vsim-gui. Only the RTL (Register-Transfer Level) files for Ri5cy are compiled, and the knn.hex file is executed.

The methodology for determining the results has been:

- Creating the C program and compiling it using the existing RISC-V compiler produces the <program>.hex file, which contains the machine language instructions OR fill the uROM with the instruction based on the C program attempting to optimize the process through parallel executions of the memory map.
- Compiling and initiating the simulation on the RISC-V system with the Mempa. The Mempa is utilized as a LiM in this scenario. The simulation outcomes will serve as the baseline for comparing with the non-LiM-specific instructions.
- Compiling and initiating the simulation on the RISC-V system without the Mempa, executing the C program that perform the same operation described in the uROM.
- Comparison of the two results for each algorithm.

8.1 Performance Evaluation

Algorithm simulation and subsequent comparison of results is essential for assessing the performance and efficacy of various strategies. This process involves modeling specific algorithms in a controlled environment and analyzing the outcomes to determine which strategy performs better under specific conditions.
Simulations were conducted, comparing the developed architecture with the architecture consisting solely of RI5CY and RAM. Benchmark tests previously employed in other architectures, such as PLiM and GPLiMA—recognized for their efficient handling of large datasets and substantial data movement—were utilized. Additionally, smaller-scale algorithms viable within RI5CY, like maximum and minimum value searches, were included. The goal was to acquire a deeper comprehension of how the memory-processor unit could provide advantages within the architecture, contingent upon the specific computational demands. Each test necessitates initializing the memory before usage, along with the initialization of all variables essential across various tests, followed by processing the results. The algorithms used encompass a variety and will be briefly outlined below.

- KNN(K-Nearest Neighbors Algorithm):It is a learning method used for classification and regression. It operates by identifying the category to which a data point belongs by analyzing examples in its nearest "neighborhood" within the dataset. In practice,

it finds the "K" nearest data points to the one being classified and uses their labels to determine its class. Widely applied for classification problems, it can also be utilized in regression problems to estimate numerical values.

- DFT(Discrete Fourier Transform):It is a technique that converts signals from the time domain to the frequency domain. It's used to analyze the frequency content of discrete signals. It transforms a signal into a series of sinusoids of different frequencies and amplitudes.

- Mean and Variance: The mean, typically referred to as the average, represents the central tendency of a dataset. It is calculated by summing all the values in a dataset and then dividing by the number of values. It provides a single value that is representative of the entire dataset. Variance, on the other hand, measures the dispersion or spread of the data points around the mean. It indicates how much the data points deviate from the mean. The higher the variance, the more spread out the values are within the dataset.

- MVM(Minimum Variance Matching algorithm):It is a financial method used to create a portfolio with the least possible variance. It aims to minimize risk while achieving a desired rate of return. By analyzing expected returns, variances, and correlations between assets, it constructs a balanced portfolio with minimized risk exposure.

- K-MEAN:It is a method used in unsupervised machine learning to cluster data points. It aims to partition a dataset into "K" clusters, where each data point belongs to the cluster with the nearest mean (centroid). It iteratively assigns data points to clusters and updates the centroids until convergence, minimizing the sum of distances between data points and their respective cluster centroids. This process enables the identification of natural groupings or clusters within the data.

- Lenet5 layer 1: As seen before, it is an algorithm for convolutional neural network primarily used for handwritten character recognition. Comprising convolutional, pooling, and fully connected layers, it processes 32x32-pixel grayscale performing the first layer of convolutional network.

Here are two tables that differ in terms of data size and complexity. The first table compares the larger and more complex algorithms, while the second one compares the simpler ones.
The charts are for illustrative purposes. From these data, we can observe that the number of clock cycles in the memory map (Mempa) remains consistently close to 2,700 clock cycles regardless of the number of instructions contained in the uROM. This suggests that the majority of the time is spent in the data transfer between the RAM and the memory map.

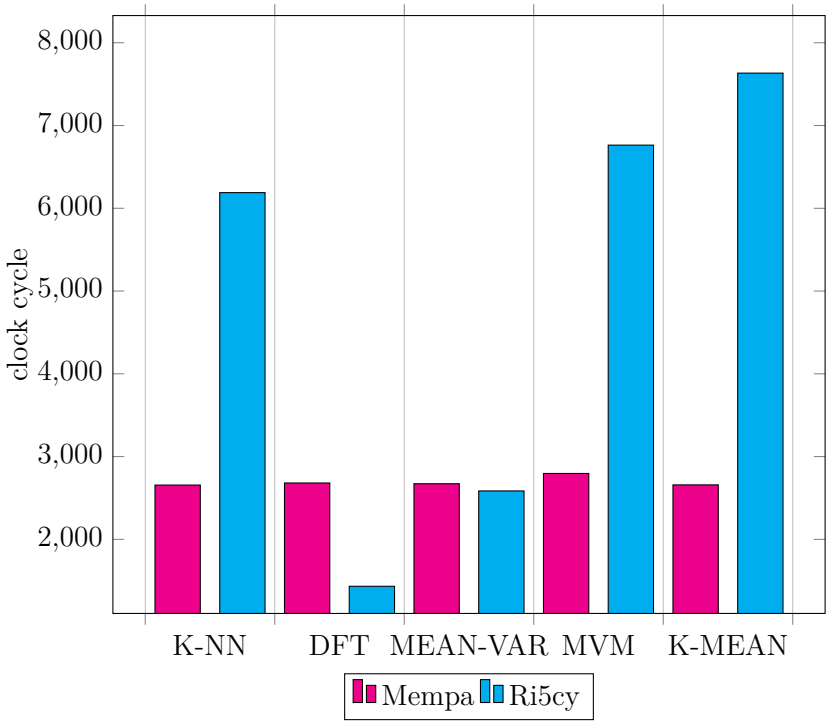
The gain in using the uROM depends on the quantity of instructions (as seen in Lenet5), but not only that; it also depends on how these instructions utilize the memory map. For example, in the case of the "mean var" algorithm, it involves relatively simple calculations for calculating the mean and variance. On the other hand, algorithms like MVM, with only 22 instructions, show a gain of 58,7%, utilizing the memory map more

Name Algorithm	uROM Instruc-tions	Clock Cycle with Ri5cy+Mempa[cc]	Clock Cycle with Ri5cy [cc]	Percentage of Profit [%]
KNN	7	2.656	6.189	57
DFT	46	2.681	1.433	-87
MEAN-VAR	82	2.672	2.585	-3,3
MVM	22	2.796	6.763	58,7
K-MEAN	23	2.658	7.633	65,2
LENET5 layer 1	318	2.953	62.445	95,3

Tabella 8.1. Comparison between the algorithm executed in Mempa and Ri5cy measured in clock cycles

efficiently in a matrix-based manner.

In general, the utilization of the memory map for algorithms with substantial data sizes is beneficial, especially when their complexity is N^2, allowing the memory map to compress multiple instruction cycles into a single instruction.

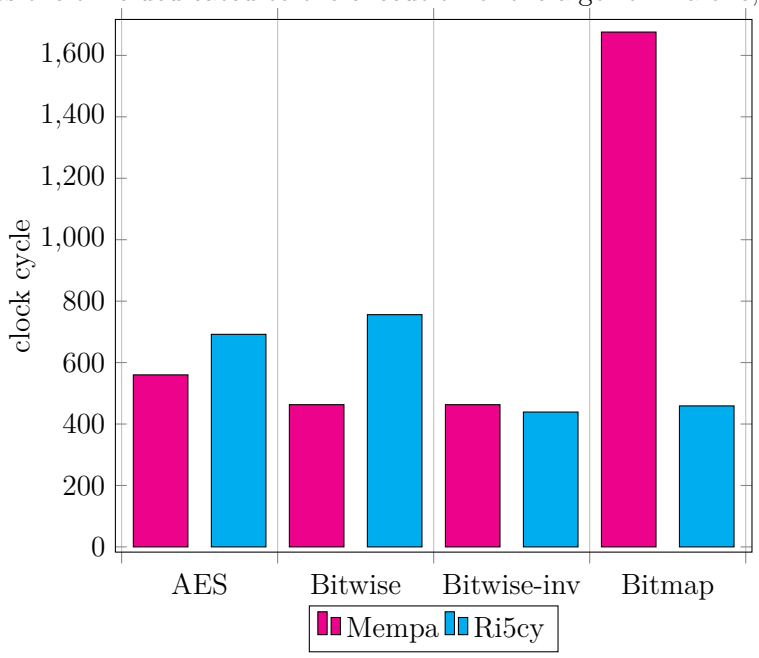


Name Algorithm	uROM Instructions	Clock Cycle with Ri5cy+Mempa[cc]	Clock Cycle with Ri5cy [cc]	Percentage of Profit [%]
AES-AddRoundKey	3	560	691	18,9
Bitwise	9	463	492	5,9
Bitwise-inverted	9	463	439	5,1
Bitmap research	8	1.676	459	-267,5
XOR-Net	17	3.215	19.747	83,7

Tabella 8.2. Comparison between the algorithm executed in Mempa and Ri5cy measured in clock cycles

In these algorithms, the utilization of the memory map is partial. As we have seen from the previous section, not all nCU (computational units) are used together. Consequently, the gain is nearly limited, if not null. For instance, the bitmap operation is a search operation with complexity N, and if performed in the memory map, it has minimal complexity, requiring only 8 instructions instead of 25 in C code (for 5 elements). However, the data loading significantly reduces the gain, sometimes even making it negative.

On the other hand, for the XOR net, the situation is different. Since it involves a different data size, data loading occurs for all 256 positions of the memory map. As seen, this significantly increases the execution time. However, the algorithm has a complexity of $4xN^2$ for each channel, which is compacted into only 17 operations in the uROM. This, in turn, reduces the time dedicated to the execution of the algorithm alone, resulting in a positive gain.



Bibliografia

riscv-isa-manual.

Md Zahangir Alom, Tarek M. Taha, Chris Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Mahmudul Hasan, Brian C. Van Essen, Abdul A. S. Awwal, and Vijayan K. Asari. A state-of-the-art survey on deep learning theory and architectures. *Electronics*, 8(3), 2019. ISSN 2079-9292. doi: 10.3390/electronics8030292. URL https://www.mdpi.com/2079-9292/8/3/292.

Pasquale Davide Schiavone Arjan Bink Paul Zavalney Pascal Gouédo Micrel Lab Andreas Traber, Michael Gautschi and Integrated Systems Lab ETH Zürich Multitherman Lab, University of Bologna. cv32e40p user manual. URL https://docs.openhwgroup.org/projects/cv32e40p-user-manual/en/latest/intro.html#blockdiagram.

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

Gianluca Goti. Application of the logic-in-memory approach to a risc-v processor using emerging technologies., 2022.

guastamacchia angela. Gp-lima: A general purpose architectural model leveraging the logic-in-memory approach, 2021.

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

Giulia Santoro. Exploring new computing paradigms for data-intensive applications. *Politecnico di Torino: Torino, Italy*, 2019.