POLITECNICO DI TORINO

Master Degree in Electornic Engineering

Master Degree Thesis

Application of the Logic-in-Memory approach to a RISC-V processor using emerging technologies

Academic Supervisors
Prof. Fabrizio RIENTE
Prof. Marco VACCA
Ing. Andrea COLUCCIO

Candidate
Gianluca GOTI

October 2022

Abstract

In recent years, many efforts have been spent on research on emerging technologies applied to memories. Many different devices are present on the market, each of them with its peculiarity but in general, as a rule of thumb, if the storage capacity increases the speed decreases. Furthermore, the standard technology suffers from physical and technological limitations.
Novel technologies try to overcome such limitations, Racetrack technology seems to be a good candidate to satisfy at the same time the requirements of speed and storage capacity. Racetrack is essentially a ferromagnetic wire where bits are retained by exploiting the magnetization direction. Bits are accessed and modified only through dedicated ports, thus this requires bit alignment to the access ports by means of shift operations along the ferromagnetic structure. Studies on this technology showed interesting performances in both access latency and storage capacity.
Another very important issue in modern Electronics is the so called Memory-wall. Today's architectures are capable of astonishing performances but the exchange with the memory slows down the overall performance. It is in this context that the Logic-in-Memory paradigm takes place. The idea is to create completely new architectures able to partially or even totally perform computations directly in memory embedding logic elements within the memory. This limits the exchange of data back and for increasing the system performance.
This Thesis work focused on the application of the Logic-in-Memory paradigm to the emerging Racetrack technology. This new architecture was applied to an open computing system named RI5CY already provided with a LiM architecture.

The proposed architecture aims to offer an open and configurable Logic-in-Memory platform in which multiple types of memory can be tested. In addition, this work proposes a working RTL model of a Racetrack memory with LiM capabilities with performances comparable to the original LiM system. Furthermore, this Thesis tries to give some ideas on new possible internal organizations for the Racetrack memory.
Simulations showed that the new Racetrack memory seems to be a good candidate to replace the standard technology. Combining Racetrack with the Logic-in-Memory paradigm should be an interesting solution to overcome the memory-wall problem and limitations due to standard memory technologies.

Keywords: Logic-in-Memory, Racetrack, memory-wall, RISC-V

Contents

List of Tables

List of Figures

Chapter 1

Motivations and background

Modern Electronic is based on a common framework, data and instructions are stored in external memories and CPUs retrieve information from them. Astonishing performances have been reached during the years but one of the main problems is the so called Von-Neuman bottleneck, research's efforts are concentrated on finding valid solutions to this problem. As the architectures' computing speed increases the continuous exchange of information back and forth from the memory implies a huge waste of time and resources.
One of the most promising candidates to solve this problem is the concept of Logic-in-Memory (LiM), where memory is enhanced with internal (and/or external) logic making possible to perform simple or even more complex in-place computations. This solution could provide a performance speed-up and a reduction of power consumption reducing data transfer on memory links.
The first goal of this Thesis is to develop a fully customizable LiM framework for Pulpino RISC-V, then it will be adapted to a novel memory, based on an emerging technology called Racetrack. The last part of this work is focused on the analysis of several algorithms to understand benefits and drawbacks of this architecture.

1.1 Memory hierarchy

All computing systems, from the old ones to the new ones, exploit memories to store information and retrieve data to carry out tasks.
Ideally, these systems would need large and fast memories, unfortunately technology cannot provide such systems, the solution is *memory hierarchy*. There exists many different types of memories with their own specific properties, combining them together could satisfy all the different needs of the CPU. This is exactly what memory hierarchy does, combining different memory systems gives the impression to the core of having a large memory, that acts as a backing store, but also fast so as not to block the exchange of information.
In general computing systems are provided with small and fast (and expensive) memories placed in proximity of the CPU, increasing the distance from this one, memories increase in size but, unfortunately, decrease in speed, Figure 1.1 gives an idea of the different properties at each level of the memory hierarchy.

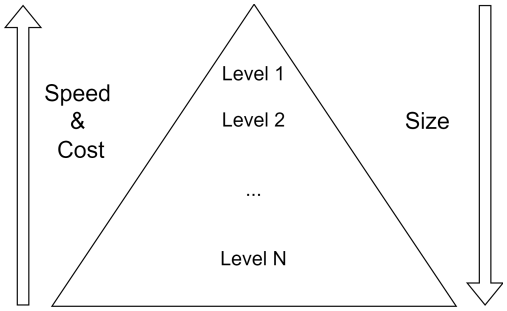

Figure 1.1. Memory hierarchy

The principle of memory hierarchy is based on the way in which software is written, this argument is widely treated in [16].
Designers tend to solve problems step-by-step and this reflects directly on memory accesses, which tend to be non-random and predictable. Starting from this fact it is possible to define the concept of *Locality of References*, which is divided in:

- *Temporal Locality*: during the execution of a program, it is very likely to access the same resource more than once;
- *Spatial Locality*: during the execution of a program, it is very likely that the next accesses will be "physically" in proximity to the last one;

These two properties have shaped the design of modern memory systems because they suggest that it is not necessary to have the whole memory to run a program and furthermore "in proximity" accesses are very common during the execution of tasks.
Thus, modern computing systems embeds three different memory types, a small and fast memory (i.e. Cache), an intermediate one large enough to run several tasks but not as fast as the first one (i.e. Main memory) and a final memory stage capable of storing all the set of data (i.e. HDD).

1.2 Memory Systems

In the following, an overview of the main memory systems will be given, the main features and properties will be highlighted as well as their controllers, that are essential to ensure their correct functioning. This review will follow an ideal path starting from the memory stage close to the CPU then moving away more and more up to the non-volatile memory stage.
It is possible to find a common structure among all the different types of memory. Due to technological and fabrication reasons, in general memory arrays are squared or rectangular, this improves the array regularity. Assuming n address bits, a fully generic memory array is composed by 2^{n-k} rows composed by one or more words, each row contains 2^k column. In general the word is addressed exploiting Row and Column Decoders, Column Circuitry is useful to read stored data and other functionalities depending on the analyzed memory. A bit-line conditioning could be present or not, this depends on the mechanism on which the memory relies, Figure 1.2 depicts a generic memory array.

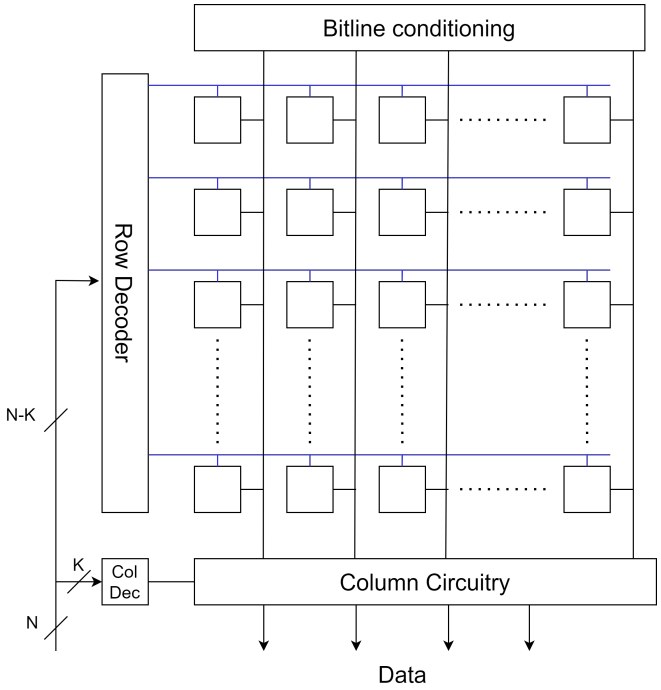

Figure 1.2. Basic structure of a memory array

Starting from the computing unit, the first memory system that is found is the *Cache*, a cache is a generic name indicating that a memory is masking the access latency of another one. As matter of fact, the main memory acts as a cache for the backing store and the "true" cache acts as a cache for the main memory, but in modern computer architecture the cache is the memory placed between the main memory and the CPU. The main task of this component is to be at least as fast as the processor, to not slow down the execution flow, for this reason caches are commonly designed using SRAM technology.

1.2.1 SRAM

SRAM stands for Static Random Access Memory, it is an asynchronous memory adopted whenever speed is the main requirement, in fact it is used for cache systems, register files and CPU's internal registers. The words *Static* and *Random* highlights two main features of this technology, memory is able to retain data without need of a refresh system and it is possible to address every single cell within the array.
During years, many different SRAM implementations have been proposed, but the most adopted and reliable one is the 6T (6-Transistor) cell. Information is retained in the cross coupled inverters that can be accessed by two pass-transistors activated by a wor-dline. Two complementary bit-lines are exploited for read and write operations. Figure 1.3 depicts the schematic of a 6T-cell, the read and write circuitry is also present.

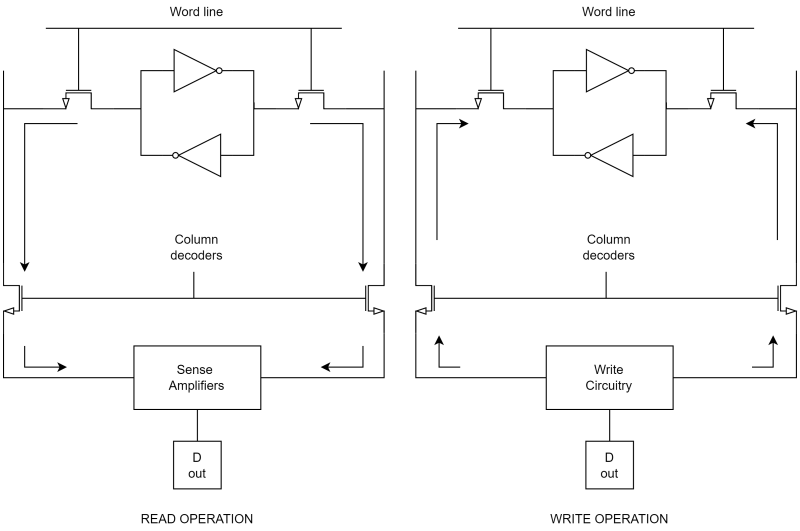

Figure 1.3. 1-bit 6T SRAM cell

A read operation consists in very simple steps:

- Precharge bit-lines to Vdd;
- Activate word-lines (this connects the 6T-cell to word-lines);
- Sense data with Sense amplifiers (to speed-up read operation);

The write operation, is even simpler:

- Apply a differential voltage to bit-lines;
- Activate word-lines;

Once word-lines are activated, if the applied voltage is correct, the internal bit of the cell is forced to the value imposed by the user.
The complete block diagram of a SRAM memory is composed by one or more arrays, in addition to the circuitry presented in Section 1.2. Figure 1.4 shows a complete SRAM architecture, among the already mentioned building blocks such as sense amplifiers, column and row decoders, bit-line

conditioning circuitry etc., the scheme shows a block organization of the memory array, activated by a specific decoder. Additional circuitry is required for the correct behaviour of the memory, like the start-up circuitry that is necessary to generate a start signal and the timing circuitry, which coordinates all the signals within the memory.

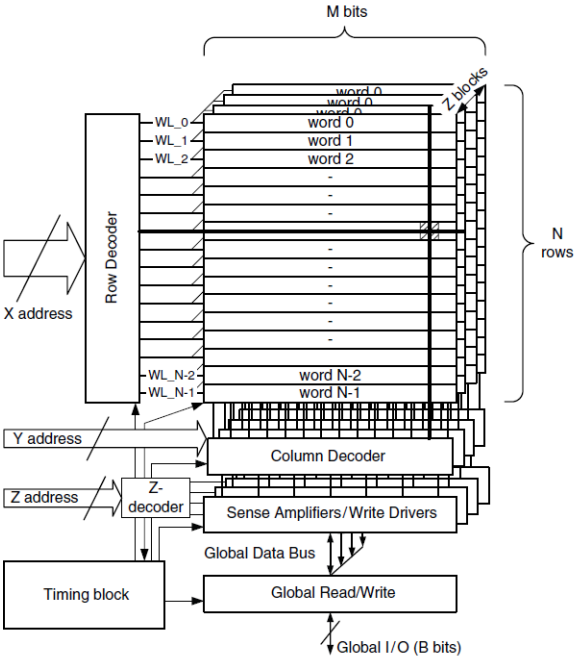

Figure 1.4. SRAM architecture [23]

The SRAM has an asynchronous interface composed by few signals:

- $\overline{CE}$: activates the communication with the memory core;
- $\overline{WE}$: enables write operation;
- $\overline{OE}$: activates 3-state buffers during read operation;
- Addr: address port;
- Data: bidirectional data port;

Figure 1.5 and Table 1.1 show respectively a generic SRAM interface and its truth table, it's clearly visible how the memory array acts like a combinational circuit, setting specific signal combinations leads to different results.

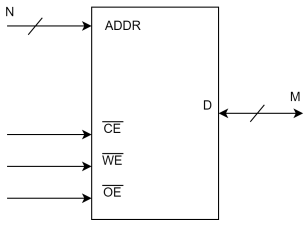

Figure 1.5. SRAM interface

$\overline{CE}$	$\overline{WE}$	$\overline{OE}$	Access Type
0	0	0	Write
0	1	0	Read
1	X	X	Disable

Table 1.1. SRAM truth table

In Figure 1.6 the timing diagram of read and write operations is depicted.
There is not a specific order for applying signals, a possible reliable solution is to adopt $\overline{CE}$ as a control signal to define a safe window in which other signals can be asserted.
The Read operation starts enabling $\overline{CE}$, then if the address is stable, asserting $\overline{OE}$ gives in output the requested data, here $\overline{WE}$ is a don't care. The write operation is similar, once input data and the address are stable, $\overline{CE}$ is activated, in this case $\overline{WE}$ is active, while $\overline{OE}$ is a don't care, once this sequence of signals is set, after a while the input data is sampled. Since in this case $\overline{CE}$ is used to create a "window", input data is sampled on the "low-high" transition of this signal.

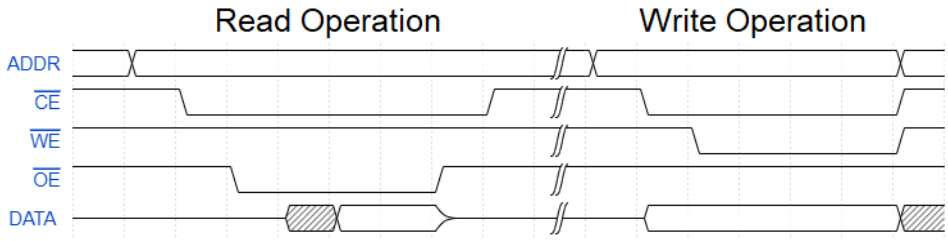

Figure 1.6. SRAM timing diagram

As said previously, this memory acts like a combinational circuit, so by definition it is asynchronous. As the execution speed increases it is more and more difficult to properly generate all the internal signals with correct delays, timing tolerances will lead to errors, a solution to such problem is to rely on a synchronous architecture.
SSRAM, Synchronous Static Random Memory, employs standard SRAM technology improved with some pipeline stages. Different types of SSRAMs have been presented during years, one example is the *pipelined* SSRAM, that implements an input and an output pipeline stage to improve performance, Figure 1.7 depicts a possible block diagram. Of course the drawback of this solution is the increase of latency and an higher cost in terms of power and area due to the additional circuitry.

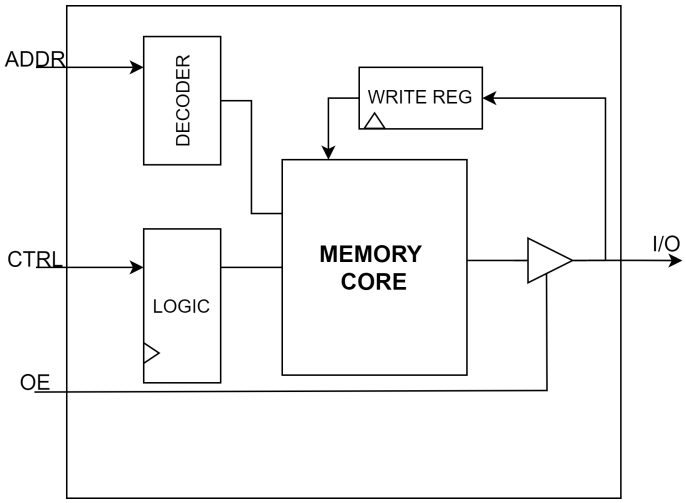

Figure 1.7. SSRAM architecture

Memories are devices controlled by external entities, in general the computing unit does not directly communicate with the memory and an intermediate device act as an mediator between the two, this is the Memory Controller. These devices provide an easy interfacing with memories, CPU sends addresses, operation type (read or write) and eventually the data to store, then the controller will provide to set the specific signals to the memory core and to retrieve data from memory.
A possible SRAM controller is depicted in Figure 1.8, it has a very simple structure, it contains a FSM to control all the different states, a decoder to decode input addresses and several registers to store information and to provide a synchronous interface to the microprocessor.

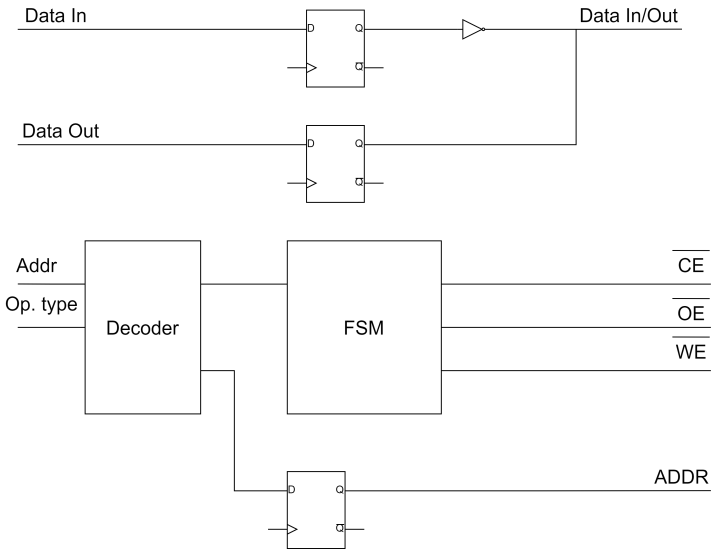

Figure 1.8. SRAM controller architecture

1.2.2 Cache Systems

During years, the increasing demand of computational speed have led to faster and faster microprocessors, this evolution have not been followed by memory systems for multiple reasons. As addressed before, this is one of the main problems in modern computer architectures, researchers have investigated many different solutions and technologies to overcome this problem, one of this is to exploit *memory hierarchy*. Following this path, cache systems have been introduced, Figure 1.9 depicts a typical processor-memory organization that exploits a cache to speed-up the execution.

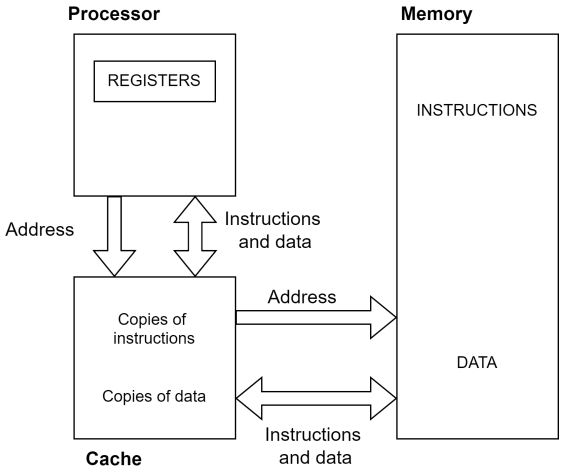

Figure 1.9. Typical processor-memory organization with cache system

Caches are small and fast memories placed in between of the computational system and the main memory, they try to mask the access penalty by storing often-accessed or likely-accessed data, relying heavily on the well known concepts of *temporal locality* and *spatial locality*.
Generally, a Cache is transparent, meaning that the processor acts normally and it is not aware of the Cache system. When data is present in the Cache, there is a HIT and the execution speeds-up, otherwise when data is not present there is a MISS and a normal memory access is performed, paying the correspondent access penalty.
The *line* is the basic Cache memory uni, its counterpart in the Main memory is called *block*. In this insight only transparent-addressed Caches will be investigated, this means, as said before, that the Cache is transparent to the system, so the same addresses as in the main memory are used.
From the technological point of view, Caches are generally based on SRAM technology, this choice has several advantages like high transfer speed and compatibility with standard CPU's fabrication processes. The architecture is very similar to a standard SRAMs, the memory core is identical to the ones showed in 1.2.1, additional circuitry is required i.e. Tag Comparison circuitry in addition to a different organization depending on the *mapping* policy. Possibly, a Control Unit is needed to control the *replacement* and *writing* policies.
The first parameter that distinguishes all the different Cache structures is the mapping, it represents how data is mapped within the cache.
There are three possible solutions:

- Direct mapped: data is store only in a unique location identified by means of the modulo operation. Since the Cache is much smaller than the main memory, many different addresses can be mapped in the same Cache line, this leads to *contention*. The address in divided in

three fields, Tag, Set index and Offset. Set index is basically the modulo operation between the block number and the overall number of lines, the result gives the line number where to map the data. Offset field (LSBs) indexes the words within the line, while the Tag field is used to check whether a line is present or not in the Cache. Tag field is basically a set of MSBs of the address that uniquely identifies data mapped within the cache.
This is a very simple and effective implementation, the main limitation is the line contention, since multiple blocks can be mapped in the same cache line, this problem is called *trashing* and produces cache misses;

- Fully Associative: data can be stored in any line of the cache, it overcomes the problem of trashing, addressed in the previous implementation.
  Tag circuitry is replicated for every cache line, the behaviour is similar to a CAM plus a RAM, the tag-check is fully parallel. The main limitations of this implementation are the high dynamic power and the very high cost due to the redundancy of the tag circuitry, for this reason to keep low these costs the storage capacity is limited;

- Set-Associative: this is an hybrid solution that embeds some characteristics of the previous two implementations. Cache is organized in *set* or *ways*, synonyms (cache lines with the same set index) can be mapped in one of the available set. Tag field allows to identify the requested data among all the different sets. Tag circuitry is replicated for each set, this solution is less expensive with respect to the direct mapped one, furthermore it reduces the cache misses due to the same cache line contention;

In the following, Figures 1.10, 1.11 and 1.12 shows respectively the architecture of a Direct-Mapped Cache, a Set-Associative Cache and a Fully-Associative Cache.

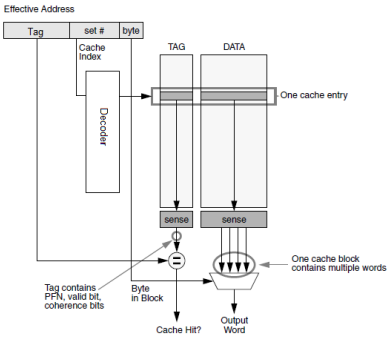

Figure 1.10. Direct-mapped cache [16]

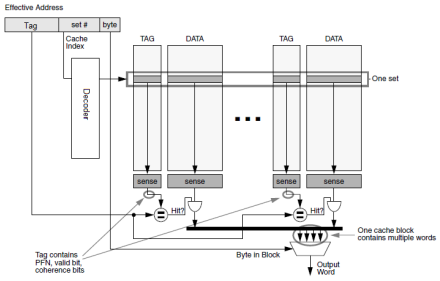

Figure 1.11. Set-associative cache [16]

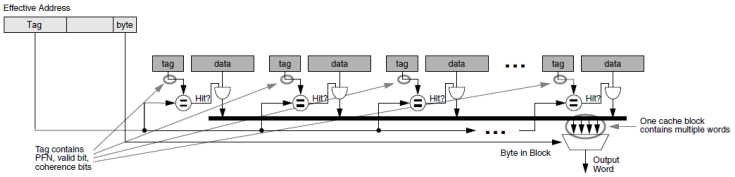

Figure 1.12. Fully-associative cache [16]

Cache systems differ also from writing and replacement policies.
The first one regards how consistency between cache and main memory is handled. The second one controls the data replacement in the cache whenever a miss occurs.
The main replacement policies are

- Random: cache line to replace is randomly chosen;
- LRU (Least Recently Used): replaces the least recently used line;
- FIFO (First In First Out): replaces the first line entered in the cache;
- LFU (Least Frequently Used): replaces the least frequently used line;

The most adopted write policies are:

- Write-through: data is written both in the Cache and in the main memory;
- Write-back: data is written or updated only in the Cache, only in specific moments it is also written in main memory i.e. replacement phases;

Literature doesn't provide a clear view of how Cache controllers are designed, while the book *Computer Organization adn Design RISC-V Edition* [22] offers a possible design solution. Here a behavioural description of the different controller's states is proposed, Figure 1.13 shows a possible implementation of the state diagram. Here, all main features are developed, tag comparison, replacement and write policies are handled by a simple FSM that controls all the task of the cache.

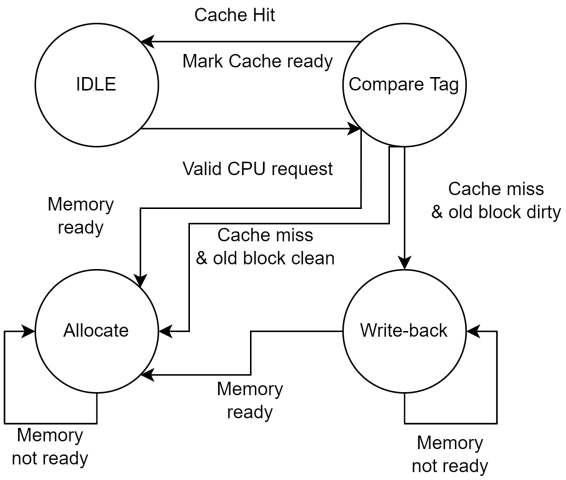

Figure 1.13. State diagram of a simple cache controller [22]

1.2.3 DRAM

DRAM memory is what is usually called *main memory* inside a computing system. Retention mechanism is completely different with respect to the cross-coupled inverters of the SRAM. Data is stored in a parasitic capacitance of a MOS transistor that acts also as a pass-transistor for the memory cell, Figure 1.14 shows the basic 1-bit cell. This mechanism gives a very high storage density at a very low cost, as drawback the charge stored in the capacitor slowly leaks through the pass transistor. This slow discharge corrupts the retained data, for this reason DRAM needs

to be periodically *refreshed* and this type of memory is typically is referred as *dynamic*. In this technology read and write operations are typically much slower with respect to SRAMs.

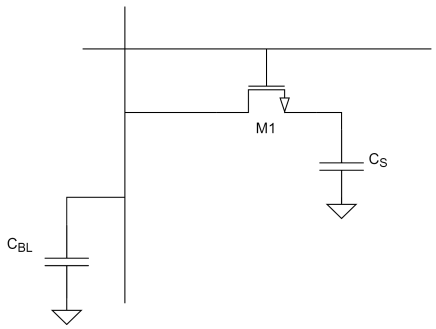

Figure 1.14. 1-bit DRAM cell

For historical reasons, the address bus in these structures is multiplexed, row address and column address are set in two different phases.
The very first version of this memory was asynchronous, then progress pushed the architecture towards the synchronous world, in fact today only synchronous DDR DRAMs are used.
The internal architecture, recalls deeply the SRAM structure, basically it embeds the same main components plus additional circuitry to handle the refresh phase. Recalling Figure 1.14, data is stored in the parasitic capacitor C_S, the access is granted by the pass transistor M_1 that connects the inner memory cell with the global bit-line. This latter connects all the memory cells along its path, thus it has a huge parasitic capacitance named C_{BL}. The presence of this bit-line capacitance makes more tough the read operation on the memory cell.
In the following, the three main cell operations are described:

- Write operation: the bit-line is set to '1' or '0', then the word-line is activated and the voltage V_{BL} across C_{BL} is transferred into C_S, other cells along the bit-line are simply refreshed;

- Read operation: the bit-line is pre-charged at $\frac{Vdd}{2}$, then the wor-dline is activated, C_S and C_{BL} start sharing charges. This has two effects, from one side, voltage across C_{BL} changes, making possible to read the stored data, on the other side the voltage across C_S is completely corrupted since the storage capacitance is much smaller with respect to the bit-line one. Voltage perturbation Δ_{VB} across C_{BL} is sensed with a Sense Amplifier that speeds up the read operation. The memory cell is then refreshed with a positive feedback mechanism;

- Refresh operation: this task is periodically carried out with a simple dummy read operation;

At system level, the access to a DRAM cell requires very precise steps that the designer should follow to ensure correct accesses, they consist in two different commands, *Activate* and *Pre-charge*. Figure 1.15 depicts the different phases involved in a read/write operation.

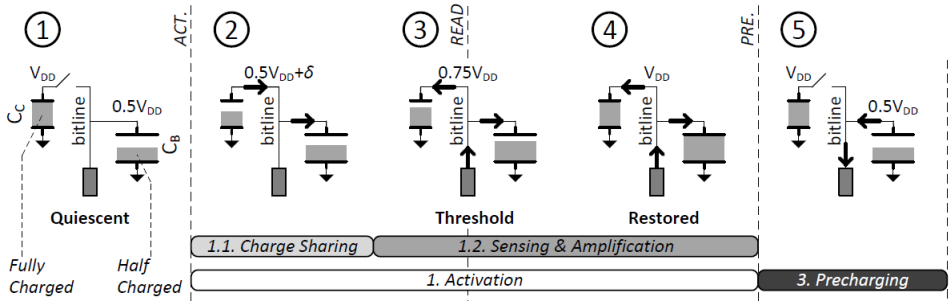

Figure 1.15. DRAM access phases [18]

The cell is initially in the quiescent (1) state, here the pass-transistor is open and the bit-lines are at $\frac{Vdd}{2}$. Then, *Activate* command closes the pass-transistor and connects the Cell capacitor with the bit-line capacitor, this leads to a charge sharing (2). Sense amplifiers at the end of the bi-tline sense the voltage perturbation across the bit-line capacitor (3) , at this point a Read or Write command can be issued. Eventually, in case of a read operation, the original value can be restored (4). Finally, *Pre–charge* command (5) closes the access cycle restoring the original quiescent condition, thus pre-charging the bit-line to $\frac{Vdd}{2}$. These are the main steps to follow during an access cycle, every read or write operation should follow this scheme.
DRAMs have a hierarchical organization, they are divided in several structures, unfortunately literature does not provide a common nomenclature, the proposed one is reported in the following (Figure 1.16):

- Column: it is the smalles addressable unit;
- Row: group of memory cells connected by the same row, also called *Page*;
- Bank: array of memory cells that responds to same commands;
- Rank: collection of one or more banks, it can cooperate with other ranks;
- Channel: collection of ranks which shares the same physical link, it can operate independently;

DRAMs are different from SRAMs, they are not just a piece of combinational logic, since the address bus is multiplexed, column and row addresses need to be saved in internal latches, this reflects deeply on the memory access interface. First versions of this memory were still asynchronous, in the following a set of possible interface signals are reported (Figure 1.17):

- $\overline{RAS}$: Row Address Strobe;
- $\overline{CAS}$: Column Address Strobe;
- $\overline{WR}$: write enable;
- Addr: multiplexed address port;
- Data: bidirectional data port;

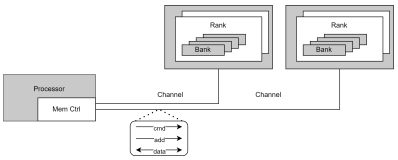

Figure 1.16. DRAM organization

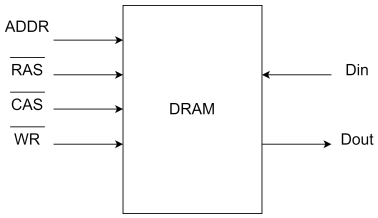

Figure 1.17. DRAM asynchronous interface

The internal architecture, Figure 1.18, is very similar to the SRAM's one. The core of the memory array is composed by common building blocks present also in other memories such as row and column decoders, write and sense amplifiers, bitline conditioning circuits and many others. Since DRAM is volatile, it needs a specific circuitry that periodically perform a refresh of the entire array, this is carried out automatically by an internal FSM designed specifically for this task. As said before, addresses are multiplexed, this design solution was taken for historical reason to save pins because they had a very huge impact on the final cost of the memory. Addresses are given in two different phases and they need to be saved somewhere, generally internal latches are adopted for this task.

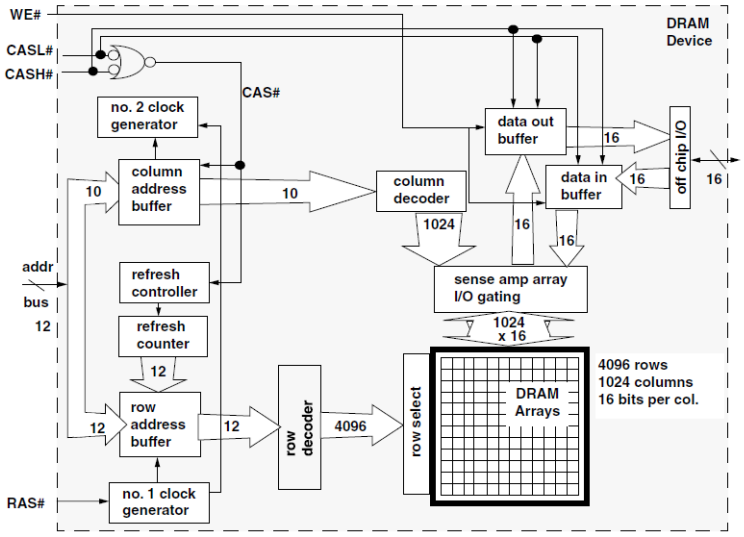

Figure 1.18. DRAM internal architecture [16]

Timing diagrams of read and write operations are very similar, these operations share the same access phase and they differ just in their final steps. Figure 1.19 show a typical read and write access, in both cases the access starts with an activation phase that consists of asserting $\overline{RAS}$ once the Row Address is stable.
The first memory cycle is a read operation, after $\overline{RAS}$ assertion, $\overline{WE}$ should not be asserted to notify that it is a read operation. Then, once the column address is stable, $\overline{CAS}$ should be activated to sample it. $\overline{RAS}$ and $\overline{CAS}$ should be kept stable for the whole cycle, once $\overline{OE}$ is

asserted after a while the required data is available on the output port.
Write operation is a little bit different, once the row address is sampled with $\overline{RAS}$, $\overline{WE}$ should be set before $\overline{CAS}$ is asserted. Then, data is written into the memory after $\overline{CAS}$ assertion.
In both cases a memory cycle terminates once $\overline{RAS}$ and $\overline{CAS}$ return in their original positions, before a new access cycle, a precharge cycle should be carried out.

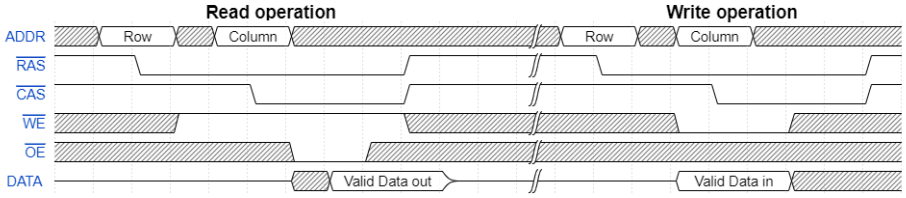

Figure 1.19. DRAM Read and Write timing diagram

The evolution of DRAMs follows the one described for SRAMs, memory cycles require very precise time constraints and control signals should be generated with proper delays. These can be achieved easily at low operative frequency but as the clock frequency increases it is way more difficult to provide signals with te correct combination, thus a synchronous system is required. The internal organization becomes more complex, a FSM controls all the internal signals and the user can program it by means of the *mode register* that should be programmed at the switch-on. Internal latches are no more controlled directly by $\overline{RAS}$ and $\overline{CAS}$, they do not act as strobes but now they are sampled on the rising edge of the clock, Figure 1.21. A new Chip Select ($\overline{CS}$) signal, controls the communication with the memory chip, Figure 1.20 shows the external interface and the internal architecture of a DRAM array.

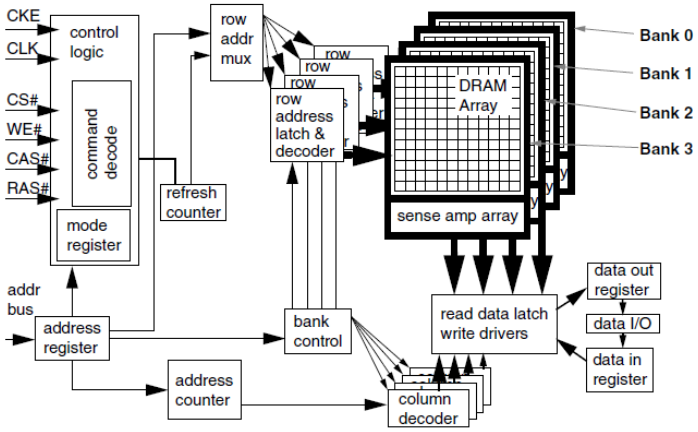

Figure 1.20. DRAM interface and internal architecture [16]

In SDRAMs $\overline{RAS}$, $\overline{CAS}$ and $\overline{WE}$ compose a *command word* that is used to control the operation of the memory through the FSM. Modern DRAMs implement a Double Data Read (DDR) feature, data is sent both on the rising and falling edge of the clock, in this condition memory also provides a strobe named DQS, used by the receiver for sampling data with the correct timing, Figure 1.22.

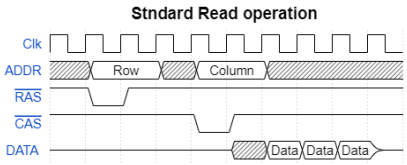

Figure 1.21. SDRAM standard read operation

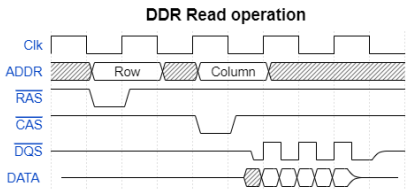

Figure 1.22. SDRAM DDR read operation

This complex structure requires a memory controller able to menage all the different operations. First of all, it has to carry out the initial configuration of the memory programming through the mode register. Then, it should activate all the specific commands to perform read and write operations i.e. assert Activate and Pre-charge commands. It should also provide all the features of a standard memory controller like address decoding, data and memory request buffering and many others. Since all these operations are very complex, memory controller can be implemented with a FSM which controls all the different memory states and phases. In [3] a general DRAM controller is proposed, 1.23 shows its schematic. The proposed structure is provided with a memory mapping unit useful to translate the addresses into block, row and column numbers, an arbiter is also present and it schedules memory transactions in all the available banks. This controller implements also a set of input and output buffers to speed up the execution.

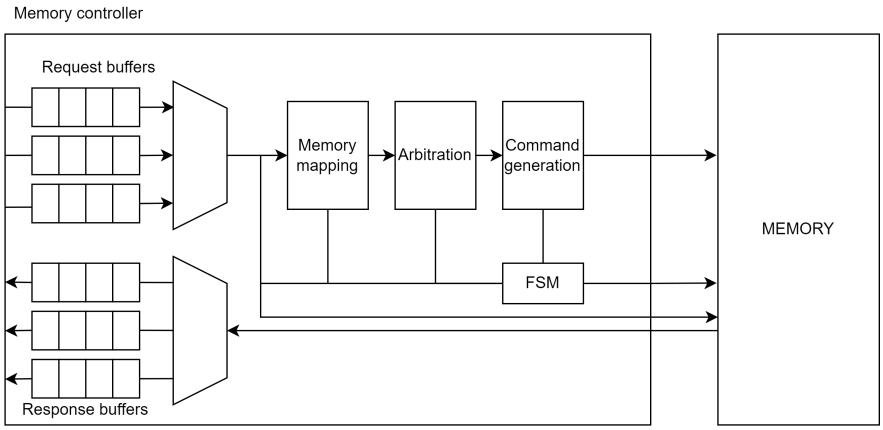

Figure 1.23. DRAM controller [3]

In [28], Xilinx provides a DDR SDRAM controller, Figure 1.24. The main building blocks are similar to the previous example, a controller menages all the different phases of the memory, then some latches sample the input addresses, a DLL is implemented to provide a stable clock reference to the memory, furthermore some counters (i.e. burst counter, latency counter etc.) are implemented.

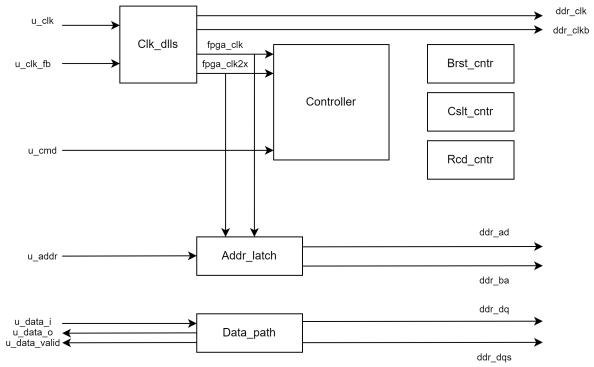

Figure 1.24. SDRAM controller [28]

1.2.4 MRAM

Magnetoresistive Random Access Memories (MRAMs) are the first non volatile memories of this overview. Their peculiarity is that their are based on emerging technologies such as magnetic materials and Magnetic Tunneling Junctions (MTJs). They store data as a stable state of magnetic devices, then information is read measuring resistance to estimate the magnetic state. MRAMs behave like resistive memories during read operations, while they differ in the writing operations based on the mechanism adopted by the specific MRAM type.
The main features of these emerging technologies are:

- zero stand-by leakage;
- high read/write speed;
- CMOS BEOL fabrication processes compatibility;
- scalability;
- integration density;

All these feature make MRAM technology a good candidate to replace standard memories, studies are still ongoing.
The fundamental component for a 1-bit MRAM cell is the MTJ. In its most straightforward configuration, Figure 1.25, it is composed by three layers:

- Free layer (FL): where information is stored;
- Insulating layer (IL): allows to switch and read the FL
- Reference layer (RL): provides a stable magnetic reference;

The basic cell comprises one or more access transistors.

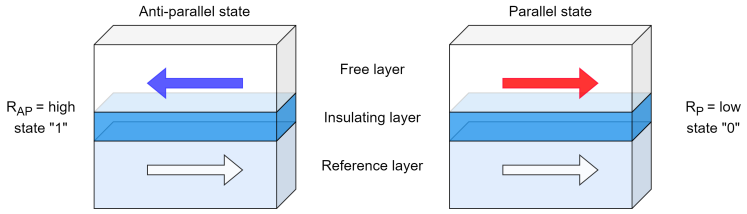

Figure 1.25. 1-bit MRAM cell with MTJ

Data is stored as a stable magnetic state obtained through the relative orientation of the magnetization of the two ferromagnetic layers. Furthermore, this condition determines the resistive behaviour of the device,this is the so called Tunneling MagnetoResistance (TMR) effect . For most materials, the resistance is low when the layers' magnetization is parallel, conversely the resistance is high when magnetization is anti-parallel, these two states are exploited to encode logic '0s' and '1s'.
Read operations exploits a specific circuitry which compares the resistance of the cell with a reference value provided by the memory array to determine the cell's state. Tunneling MagnetoResistance ratio is an important parameter for MTJs, it is defined as following:

$$TMR = \frac{R_{AP} - R_P}{R_P}$$

It basically shows the relative resistance change, R_P and R_{AP} are respectively the resistance in the parallel state and the resistance in the anti-parallel state.
Different types of MRAMs rely on the same reading mechanism, what distinguish them is the writing mechanism, two types of the most promising MRAMs are:

- STT-MRAM
- SOT-MRAM

STT MRAMs take their name from Spin Transfer Torque effect, when a current is passed through the MTJ, this exerts a torque on the FL's magnetization. If the current is large enough this will result in the switching of the magnetization state of the FL and of the resistive value as well. Furthermore, current polarity determines the parallel or the anti-parallel magnetization state. The basic 1-bit STT-MRAM cell is a two terminal structure, 1.26, a word-line activated transistor determines the current flow or not. Read operation implies the MOS activation and the application of a read voltage across the two port device, the read current is then sensed and compared with a reference value by a Sense Amplifier, which determines the resistive value of the cell.

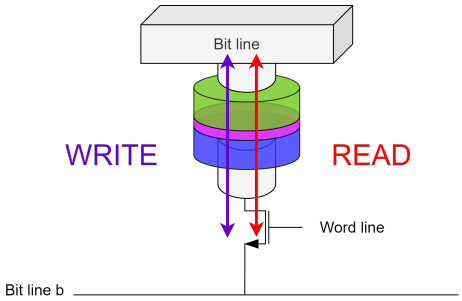

Figure 1.26. STT-MRAM cell

Multiple cells can be arranged together to create arrays like in standard CMOS memories. Array's internal architecture is similar to a DRAM's one, [8] proposes a simple STT-MRAM internal schematic reported in Figure 1.27. Differently from DRAMs, in this case data should not be written back after read operations, so Sense Amplifiers can be shared among multiple bit-lines, determining a low energy consumption and area occupation. A row buffer stores output data disconnecting SAs, this limits wasted power after data sensing. Then, architecture contains building common blocks to each memory like SAs, column and row decoders, registers/latches, timing circuitry and others. As known, read operations are carried out comparing the current flowing through the MTJ with a reference value, this requires a current generator to provide a stable reference source for SAs.

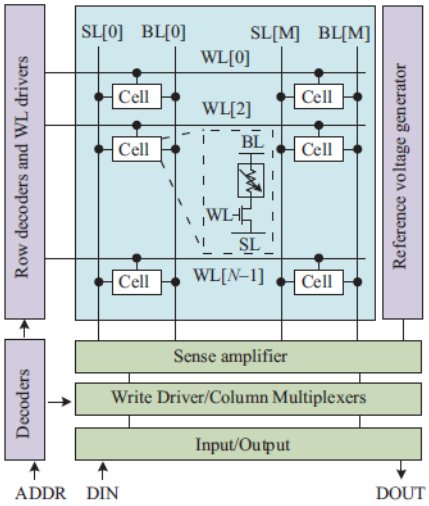

Figure 1.27. STT-MRAM array internal architecture [8]

In 2018, Everspin Technologies proposed a 256MB DDR3 STT-MRAM [1], as stated previously the overall architecture resembles deeply a DRAM as shown in Figure 1.28.

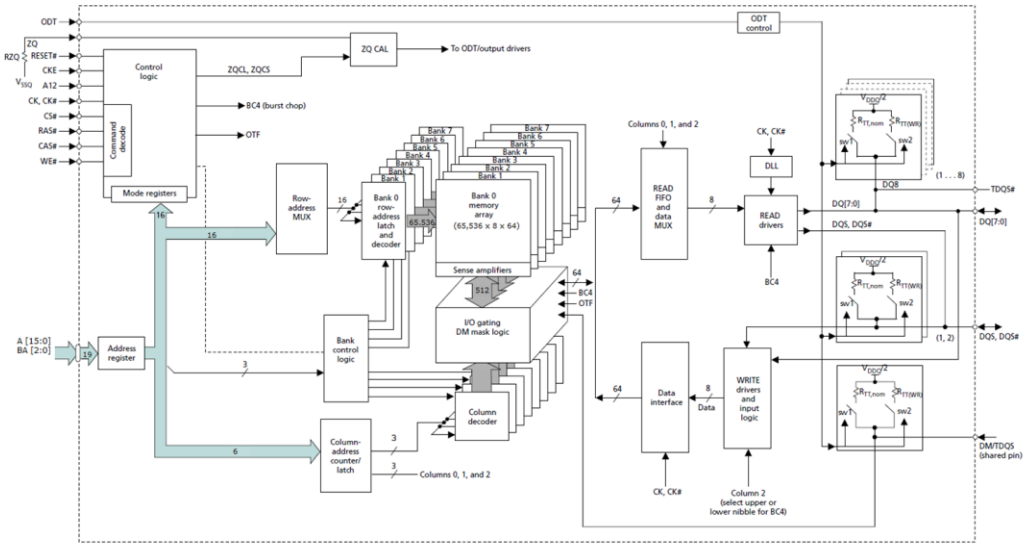

Figure 1.28. 32MB x 8 STT-MRAM architecture by Everspin [1]

Everspin's datasheet provides also several timing diagrams, Figure 1.29 depicts the read cycle timing diagram, it is very similar to the DRAM's one. Before any read or write operation a row should be activated through the *activate* command (not shown in the diagram), then the read/write command can be issued, notice that addresses are issued with the read/write operation command.

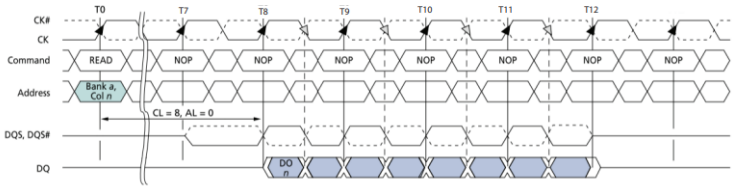

Figure 1.29. STT-MRAM read operation timing[1]

STT technology implies that read and write currents pass along the same path, this leads to some drawbacks. Fast switching requires a large current flow through the MTJ insulating layer, this speeds-up aging of the barrier leading to a lower reliability. This is why during the year researchers have moved to other MRAM types such as SOT-based MRAMs.
SOT-MRAMs are based on the so called Spin-Orbit Effect, from which they take the name. The 1-bit cell architecture, 1.30, is modified to decouple the read and write current path, this solution fixes one of the most limiting issues of STT-MRAMs.

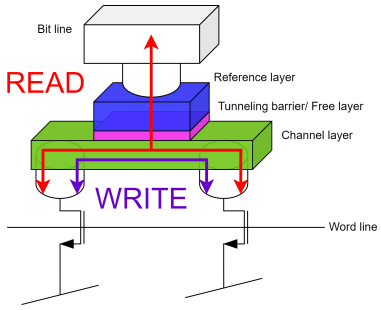

Figure 1.30. SOT-MRAM cell

The MTJ's Free Layer is replaced by a Channel Layer composed by a heavy metal. An in-plane current flowing in this latter induces a spin torque through the Free layer, which is able to switch the magnetization state of the cell. Thus, read and write current paths are separated, this reduces the Isolation Layer aging increasing the reliability of the cell. Furthermore, SOT-MRAMs result in a lower writing current with respect to STT ones.
The drawback of this technology is the larger area overhead due to the additional transistor involved in the write operation. In any case SOT technology is a good candidate for overcoming STT-based MRAMs.
Regarding MRAMS's controllers, in 2015 Northwest Logic and Everspin Technologies Inc. announced a "MRAM Controller IP compatible with Everspin's STT-MRAM" [21], unfortunately any further information is provided by this two companies. In any case the main modules required by the controller are similar to the ones involved in the DRAM Controller (i.e. FSM, decoders, memory mapping modules, buffers and additional logic), so it is safe to suppose that the controller's architecture is not so different from the previous ones.

1.2.5 FLASH

FLASH memories are non-volatile devices, they adopt a more classic mechanism with respect to MRAM technology and they are adopted in modern SSDs, USB drives and many other devices. Unlike standard HDDs, that adopt a mechanical tip to read and write magnetic disks, these memories adopt a fully electronic mechanism for read and write operations. This type of memory relies on the Floating Gate MOS transistor, Figure 1.31, a particular device provided with two different gates.

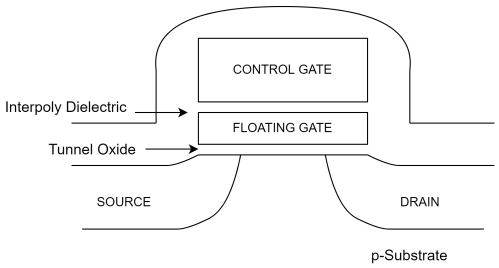

Figure 1.31. Floating gate MOS structure

The first one, named *Control gate*, acts normally like in a standard MOS system, the second

one called *Floating gate*, is completely surrounded by the gate dielectric. Thanks to this particular configuration, this MOS can be programmed. It is possible to inject electrons in the floating gate, this changes electrical characteristics and thus the behavior of the transistor. In a standard situation the floating gate is empty and the system acts as a normal MOS, so V_{GS} vs V_{DS} characteristic is standard, this state is called *Not-programmed*. The *programmed* state is basically opposite, applying specific voltages it is possible to trap negative charges in the floating gate. This modifies the MOS characteristics rising the threshold voltage as depicted in Figure 1.32. By programming the MOS it is possible to set an higher threshold voltage required to switch-on the transistor, in this way logic '1s' and '0s' can be encoded.

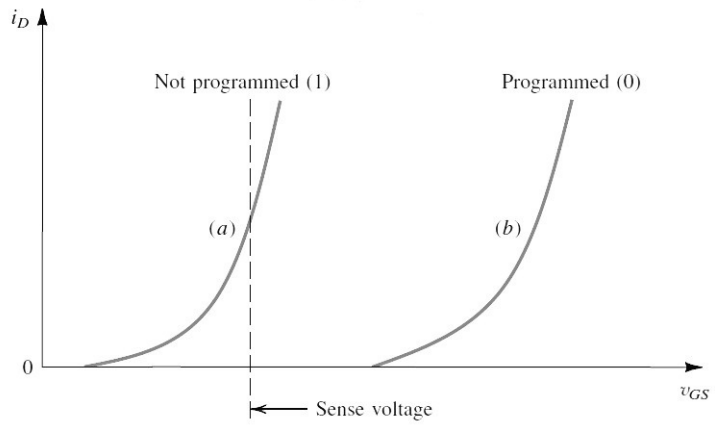

Figure 1.32. Floating gate MOS characteristics

In FLASH memories the *page* is the minimum readable unit, while the *block* is the minimum erasable unit. During years two different types of FLASH memories have been proposed, based on NAND and NOR technology respectively. The differences between these two architectures are shown in Figure 1.33. NOR architecture is based on a NOR-type logic where each cell can drive the bit-line, while in the NAND case the bit-line can be driven only through a chain of transistors. These two structures have their own advantages and drawbacks:

- NOR
  - Random accesses
  - Slow write and erase operations
  - Suitable to store instructions
  - No multiplexed bus
- NAND
  - Slow random accesses (page access)
  - Fast write and erase operations
  - Suitable to store data
  - Multiplexed bus
  - Requires less area

– Requires ECC

Due to technological and economical reasons, NAND Flash memories have become the standard technology for non-volatile memory devices. Modern memories can achieve a very high storage density, this comes directly from the NAND architecture. Bits are stored in long transistor chains, this reflects in a smaller area occupation space since some metallizzation are shared, thus for the same amount of cells, area occupation is minimized. Unfortunately the drawback is a more complex read operation since data is degraded along transistors chain, so ECC is required.

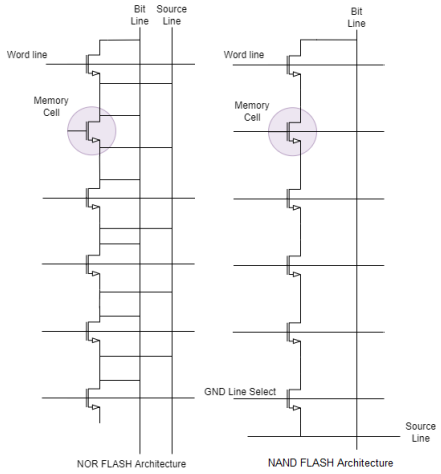

Figure 1.33. NOR and NAND Flash architectures

Internal architecture of these memories is similar to the previous ones but at the same time requires additional complex building blocks. There are three main operations: write, read and *erase*. Read operation is similar to previous memories, once bit-lines and word-lines are activated, data is read by Sense amplifiers (SAs). Write and erase operations require more complex tasks because the floating gate need to be filled or emptied with negative charges. Thus, very specific voltages are required for such operations, internal architecture is provided with specific FSMs to handle all the different steps and many modules to generate and apply such voltages.
First versions of these memories were asynchronous, Figure 1.34 depicts the interfaces for NOR and NAND Flash memories, in the following their interface signals are listed:

- NAND:
  - CLE: Command Latch Enable
  - ALE: Address Latch Enable
  - $\overline{CE}$: Chip enable
  - $\overline{WE}$: Write enable
  - $\overline{WP}$: Write protect
  - $\overline{RE}$: Read enable
  - RD/BY: Ready/Busy flag
  - I/O: Input/Output bus
- NOR:
  - ADD: Address bus

– $\overline{CE}$: Chip enable

– $\overline{WE}$: Write enable

– $\overline{WP}$: Write protect

– $\overline{RE}$: Read enable

– $\overline{OE}$: Output enable

– DATA: Data bus

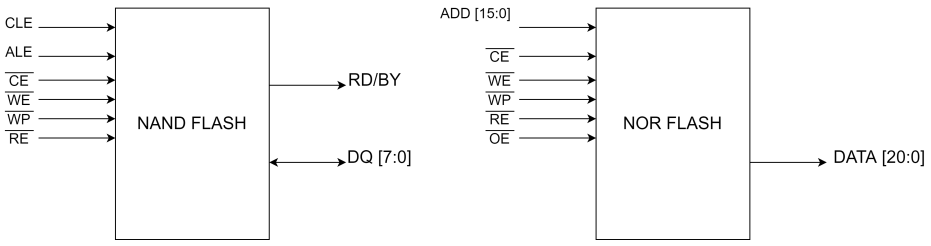

Figure 1.34. NOR and NAND Flash interfaces

Modern Flash memories are based on NAND technology due to an higher density capability, NOR Flashes have a very small applications, for this reason a brief insight of this Flash type will be given in the following.
NAND memories are divided in blocks, made of several pages. They can be programmed page by page and erased by blocks, random reads are slower with respect to NOR Flashes. As shown in the previous schematic, this configuration leads to an higher integration, but additional pass transistors are required (i.e. top and bottom ones). As for the program operation, pages are the minimum readable entity, while the *string* is the basic entity of the memory array (column of transistors).
Like in NAND-based logic, read operations are carried out through chains of transistors, this degrades read data, indeed ECC is required to check data integrity. Furthermore, for historical reasons this memory is provided with a multiplexed I/O bus in which addresses, commands and data are transferred.
Internal architecture can change depending on size, manufacturer and design choices, one possible internal schematic is the one depicted in Figure 1.35. Addresses and commands (read, write or erase) are sampled by internal latches, these are controlled by CLE and ALE signals. Since the internal mechanisms are very complex, a FSM is required to control all the tasks. Memory is provided with a command register like in DRAMs with which it is possible to program its behaviour. Due to the small I/O bandwidth, read pages are sent in a page buffer, then data is sent out like in a shift register, this increases the throughput of the memory. Vice versa, during write operation, data is written in the page buffer and then the internal circuitry will take care to write the correct page. The erase operation is applied on the whole block.

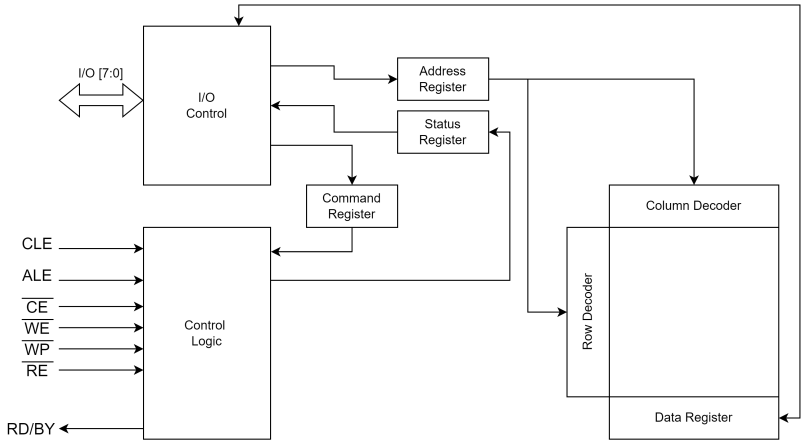

Figure 1.35. NAND Flash internal architecture

All these operations are represented in Figure 1.36, notice that part of pages is reserved for ECC bits. In NAND Flash memories, different operations are characterized by different timings, program and erase are in the range of hundreds of μs, page read in the range of tens of μs, while the communication with the output buffer is in the range of tens of ns.

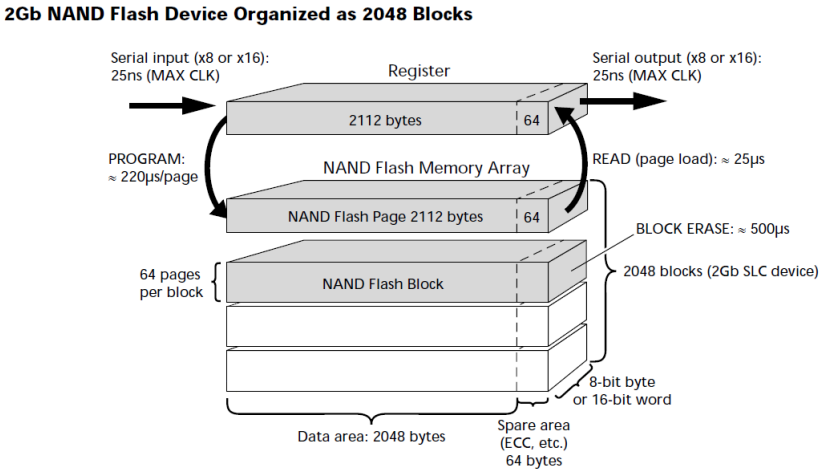

Figure 1.36. NAND Flash internal data organization [12]

In Figure 1.37 and 1.38 read and write timing are depicted. In the read operation, at the beginning the address acquisition is started sampling the initial command with CLE assertion. Then, since the I/O bus is multiplexed, multiple address cycles need to be issued to send the complete address, ALE signal is used to sample them. $\overline{WE}$ is used as a strobe to sample input data. Finally CLE is activated and thus read command is sampled, after a while data is provided to the output.
Write operation is similar to read one, the initial transition is started with the assertion of the CLE, then addresses are sent to the memory in multiple cycles. Then input data is sent through

the I/O bus, again $\overline{WE}$ is used as strobe to sample input data. At the end, write command is sampled with CLE assertion.
Erase operation is very similar to a write one but in this case input data is not required. At the end of both operations is necessary to read the Status Register, this contains useful information about the status of such operations.

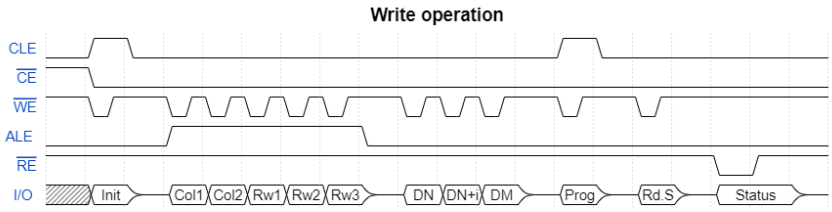

Figure 1.37. Flash write operation

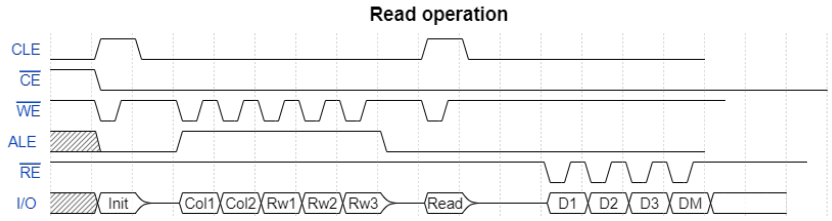

Figure 1.38. Flash read operation

As for previous memories, an asynchronous interface represents a bottleneck for data exchanges, also in this case there have been an evolution towards the synchronous word. During years Flash memory manufacturers defined a standard named ONFI, nowadays it is the common standard for modern Flash systems. This implies some modifications to the memory structure:

- Synchronous interface (Figure 1.39)
- DQS strobe for data sampling at high speed
- DDR data transfer
- $\overline{WE}$ turned into a free running clock
- $\overline{RE}$ becomes $\overline{W/R}$ signal

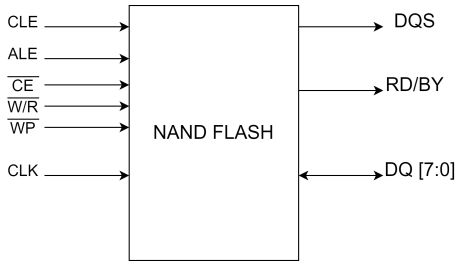

Figure 1.39. ONFI Flash interface

Controllers for these memories have a common structure to the ones analyzed before. They provide a synchronous interface for command and data exchange, they embed decoders and they implement multiple FSM routines for handling different memory operations. As case study is proposed, a Flash NAND controller by Lattice Semiconductor Corporation [20] is reported in the following. Figure 1.40 depicts the proposed architecture, it is possible to identify all the main module such as ECC logic, buffers, control FSM, Timing FSM and others.

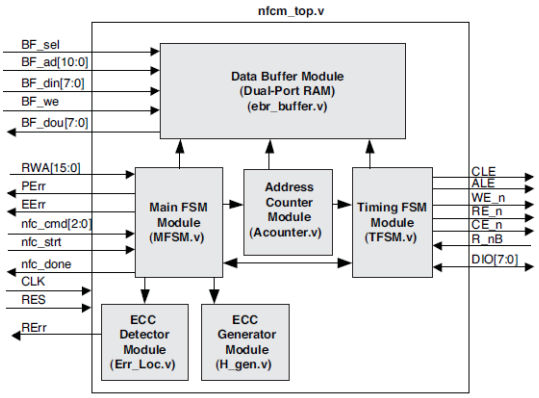

Figure 1.40. NAND Flash Controller [20]

1.2.6 Memory & memory controller overview

The initial goal of this Thesis work was to develop a generic memory controller able to adapt its functionalities to a set of different memory types and technologies. Thus, it is important to highlight common building blocks within memory and their controllers to find an initial basic structure from which it is possible to start the design.
In the following, Table 1.2 summarizes all internal building block of each memory.

Table 1.2. Memory building blocks overview

Building blocks	SRAM	SSRAM	CACHE	DRAM	SDRAM	MRAM	FLASH
Bit-line conditioning	✓	✓	✓	✓	✓		
Write circuitry & Drivers	✓	✓	✓	✓	✓	✓	✓
Sense Amplifiers	✓	✓	✓	✓	✓	✓	✓
Column decoder & Drivers	✓	✓	✓	✓	✓	✓	✓
Row decoder & Drivers	✓	✓	✓	✓	✓	✓	✓
Array (block) decoder	✓	✓	✓	✓	✓	✓	✓
Start-up circuitry	✓	✓	✓	✓	✓	✓	✓
Timing circuitry	✓	✓	✓	✓	✓	✓	✓
Column Mux	✓	✓	✓	✓	✓	✓	✓
TAG circuitry			✓				
Internal latches/registers		✓		✓	✓	✓	✓
Refresh circuitry				✓	✓		
Internal control logic (FSM)			✓	✓	✓	✓	✓
Row buffer						✓	✓
Reference current generator						✓	
Program/Erase circuitry							✓

It is possible to notice that a huge set of blocks are common between all the memories.
In the following all the building blocks inside the different memory controllers are summarized. As clearly visible from Table 1.3, Registers, FSM, Decoders, Memory mapping and Buffers are common to all the different controllers.

Table 1.3. Memory controllers building blocks overview

Building blocks	SRAM	SSRAM	CACHE	DRAM	SDRAM	MRAM	FLASH
Registers	✓	✓	✓	✓	✓	✓	✓
FSM	✓	✓	✓	✓	✓	✓	✓
Decoders	✓	✓	✓	✓	✓	✓	✓
Replacement policy circ.			✓				
Writing policy circ.			✓				
Memory mapping	✓	✓	✓	✓	✓	✓	✓
Arbiter					✓		
ECC logic						✓	✓
Buffers	✓	✓	✓	✓	✓	✓	✓

In the perspective to design a generic memory controller able to adapt itself to a large variety of memories, this study helped to understand the different needs of each of them.

1.3 L2 Cache

Once highlighted a set of possible building blocks which represents the initial point of the memory controller design, the next task was to decide which memory level is the best candidate to enhance its capabilities with Logic in Memory paradigm. It was decided to focus on L2 cache memory systems, because the perspective to perform logic operations at this level could bring an huge improvement in terms of computational speed.
As said at the beginning of this overview, modern architectures exploit the concept of memory hierarchy, this concept can be stressed out and this leads to the concept of multi-level caches, Figure 1.41 represents a typical computing system organized with a multi-level cache system. This is a direct application of memory hierarchy concept to cache systems, the goal is to reduce the miss penalty masking the generated overhead.

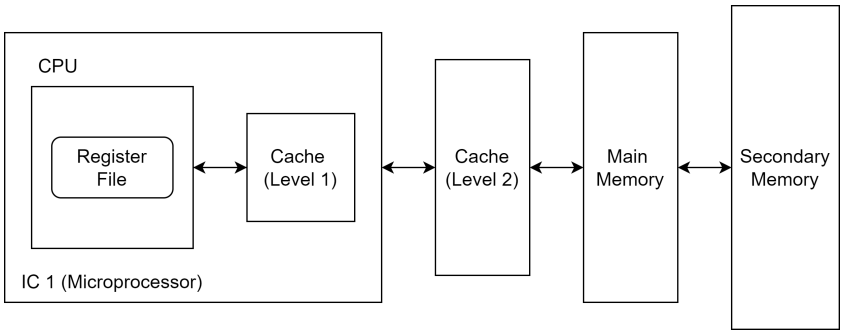

Figure 1.41. Computing systems with memory hierarchy

L1 caches are very fast and they are placed very close to the microprocessor, this ensure small latencies, the huge drawback is the small size. L2 caches instead are placed a little bit far with respect to the first ones and the act as real cache systems for L1 ones. Their behaviour is quite simple, when a miss in L1 cache occurs, the required data is searched in L2 cache and if it is present it will be provided to the system, if a miss occurs also at this level, data is retrieved from main memory and loaded in L2 cache. As consequence, they are larger at least one order of magnitude, to contain also data not present in L1 caches, furthermore data transfer speed is slower for three main reasons [17]:

- Longer critical path: a larger memory array and a more complex circuitry have a direct impact on the critical path;
- Off-chip accesses: unlike on-chip accesses, these are slower due to physical limitations;
- Bandwidth: the number of I/O pins is limited due to size, cost and design choices;

To describe better the improvements brought by a multi-level cache system it is possible to introduce the concepts of Global Miss Rate and Local Miss Rate [9]:

- Global Miss Rate (GMR): cache misses divided by CPU accesses;
- Local Miss Rate (LMR): cache misses divided by cache accesses;

In the following GMRs and LMRs for L1 and L2 caches are reported:

- L1:
  - $GMR_{L1} = MR_{L1}$

  - $GMR_{L2} = MR_{L1} \cdot MR_{L2}$

- L2:
  - $GMR_{L1} = MR_{L1}$
  - $GMR_{L2} = MR_{L2}$

Then, it is possible to define the Average Access Time to Memory (AMAT), which represents the average access time to memory taking into account the improvements brought by the memory hierarchy. This parameter can be defined as follows:

$$AMAT = HT + MR \cdot MP \tag{1.1}$$

Considering a system provided with only a L1 cache, the AMAT can be written in this way:

$$AMAT = HT_{L1} + MR_{L1} \cdot MP_{L1} \tag{1.2}$$

Assuming a system provided with both L1 and L2 cache systems, the AMAT is defined as follows:

$$AMAT = HT_{L1} + MR_{L1} \cdot (HT_{L2} + MR_{L2} \cdot MP_{L2}) \tag{1.3}$$

Legend:

- MR=Miss Rate
- HT=Hit time
- MP=Miss Penalty

As clearly visible from the previous equations, Miss Penalty in L1 can be re-defined in terms of L2 properties (HT, MP and MR), this leads to a reduction of MP_{L1} and thus to a speed-up of the system. It is possible to characterize L2 Caches with several properties, in the following a brief overview of the main characteristics is shown [31],[17]:

- Inclusion vs Exclusion:
  - Multi-level inclusion: L1 data always present in L2, eviction in L2 affects also L1 but eviction in L1 does not affect L2. When a L1 miss occurs, if data is present in L2, it will be fetched in L1. In general the degree of associativity in L2 is larger or at least equal to the L1 one, this characteristic is the same also for the number of sets;
  - Multi-level exclusion: L1 data never present in L2, if a required data is present in L2, then it is moved to L1, thus L2 is populated only with L1 evicted data. On L1 and L2 miss, new data is stored only in L1. For all these reasons, L2 behaves as a victim cache for the system;

- Split vs Unified: this property refers to the data type contained in the cache. A cache is said unified if it contains both data and instruction, on the contrary it is called Split if they are stored in two different caches. A split cache could give higher bandwidth but unified ones give flexibility. In [17] it is suggested to adopt a L1 cache combined with one or more L2 caches for good performance.

- Write policy: as in a standard cache, different writing policies could affect the system behaviour. In [17] it is suggested to adopt a write back policy in addition to a write allocate one.

- Associativity: as explained in the previous section, there are several policies that could affect system performance.

As briefly described, handling a L2 cache system is very complex and unfortunately literature does not provide enough information to design a reliable model for the pursuance of this Thesis. For this reason, it was decided to leave the L2 cache system design.
Since this Thesis is relies on a previous work based on a RISC-V framework, the new objective, will be to search a RISC-V framework already provided with a L1 cache system, then once found it, L1 cache will be enhanced with the LiM paradigm.

1.4 RISC-V frameworks

As seen in the previous Section, handling a full L2 Cache system is very complex and Literature does not provide any useful model for academic purposes.
During years RISC-V framework has grown in popularity in the academic world and many different versions have been developed. In [15] there is a very complete insight of the available open-source RISC-V processors state of the art. This paper reports (Table 1.42) a list of the open source RISC-V IP Cores providing many different information like the type of instruction set, number of pipeline stages, HDL description language and especially the presence or not of a Cache system.

Name of processor	Instruct. Set Architecture and Data Width	No.of Pipeline Stages	Bus Architecture	MMU	FPU	HDL	Compiler	Debug Support	License	Last Update	Multi-Core	In Order	Cache	JTAG	Peripherals Included
Amber	ARM v2a, 32 bit	5	WB	N	N	Verilog	GCC	Y	LGPL	2017	N	Y	Y	N	Y
Lattice Mico 32	LatticeMico32, 32 bit	6	WB	N	N	Verilog	GCC	Y	GPL	2017	N	Y	Y	Y	Y
openrisc	ORBIS, 32 bit	5	WB	Y	Y	Verilog	GCC	Y	LGPL	2019	N	Y	Y	N	Y
Leon3	SPARC V8, 32 bit	7	AHB	Y	Y	VHDL	GCC	Y	GPL	2018	Y	Y	Y	Y	Y
freedom	RISC-V, 32 bit	5	TL/AXI	N	Y	Chisel	GCC	Y	BSD	2018	N	Y	Y	Y	Y
ORCA	RISC-V, 32 bit	5	WB/AXI	N	N	VHDL	GCC	N	BSD	2019	N	Y	Y	N	N
RI5CY	RISC-V, 32 bit	4	AXI	N	Y	Verilog	GCC	Y	Solderpad	2018	N	Y	N	Y	Y
zero-riscy	RISC-V, 32 bit	2	AXI	N	N	Verilog	GCC	Y	Solderpad	2018	N	Y	N	N	Y
OPenV	RISC-V, 32 bit	3	AXI	N	N	Verilog	GCC	N	MIT	2018	N	Y	N	N	Y
VexRiscv	RISC-V, 32 bit	5	AXI	Y	N	SpinalHDL	GCC	Y	MIT	2019	N	Y	Y	Y	Y
Roa Logic RV12	RISC-V, 32 bit	6	AHB/WB	N	N	Verilog	GCC	Y	Non	2018	N	Y	Y	N	N
SCR1	RISC-V, 32 bit	4	AXI	N	N	Verilog	GCC	Y	Solderpad	2019	N	Y	N	Y	N
Hummingbird E200	RISC-V, 32 bit	2	AXI	N	N	Verilog	GCC	Y	Apache	2019	N	Y	Y	Y	Y
Shakti	RISC-V, 32 bit	3	AXI	N	N	Bluespec	GCC	N	BSD	2019	N	Y	Y	N	Y
ReonV	RISC-V, 32 bit	7	AHB	Y	Y	VHDL	GCC	Y	GPL v3	2018	Y	Y	Y	Y	Y
PicoRV32	RISC-V, 32 bit	0	AXI	N	N	Verilog	GCC	N	ISC	2018	N	Y	N	N	Y
SweRV EH1	RISC-V, 32 bit	9	AXI	N	N	Verilog	GCC	Y	Apache	2019	N	Y	Y	Y	N
Taiga	RISC-V, 32 bit	3±	AXI	Y	N	Verilog	GCC	N	Apache	2018	N	Y	Y	N	N
potato	RISC-V, 32 bit	5	WB	N	N	VHDL	GCC	N	BSD	2018	N	Y	Y	N	Y

Figure 1.42. List of RISC-V open-source cores [15]

The starting point of this Thesis is based on [5], here the RI5CY "Pulpino" Core developed by the PULP Platform (now mantained by OPENHW Group) have been adopted, unfortunately it is not provided with a Cache system. The objective is to find a similar Core but provided with a Cache system, based on the properties of RI5CY, two different Cores have been selected from the previous list, ORCA and Taiga processors.

- ORCA: this is a FPGA-optimized RISC-V core implementing a RV32I ISA and optional AXI3/4 data and instruction caches.

- CVA5 (formerly Taiga): this is still a FPGA-optimized RISC-V core, it implements a RV32IMA ISA and it is designed to support parallel and variable-latency execution units. This core is fully configurable and gives the possibility to adopt a local memory.

Other possible solutions have been found in Literature, in the following various examples are reported. OPENHW Group supports also CVA6 core [13], this is a 6-pipeline stage that CPU implements a 64-bit RISC-V instruction set. It implements two separate Data and Instruction L1 caches. In [33], a novel interleaved LiM architecture is proposed, named MISK. This work is based on OpenRISC CPU [14], it is an open-source CPU implementing a 32-bit RISC architecture with 5 pipeline stages with Data and Instruction L1 caches. Many different available RISC-V frameworks implementing L1 Cache systems have been found, unfortunately these solutions are very different from the adopted RI5CY core and they would have required an intensive study of the internal architecture before adopting them.
For this reason the research of a RISC-V framework implementing a L1 cache system has been left, deciding to adopt the same core and its memory system used in [5].

1.5 Configurable memory controller

Once decided to maintain the actual memory system of the RI5CY "Pulpino" RISC-V core, the new objective is to create an open-source configurable RISC-V framework which supports the LiM paradigm, like a sort of sandbox in which everyone can implement their own LiM architectures. This configurable framework should be fully generic, like sort of generic memory controller which should be able to support any kind of memory instantiating the correct hardware blocks. The memory arrays will be surrounded by additional logic to fulfil the LiM requirements, thus this special controller acts as an interface between the Core and the memory arrays adapting itself to the chosen memory type. Figure 1.43 shows this configurable architecture, several ‘define compiler directives triggers the HDL compiler to instantiate specific hardware blocks to interface the surrounding controller model with the chosen memory array.
This configurable framework will support, in addition to a classic CMOS memory array (referred here in after with the adjective *standard*), also an array implemented with an innovative technology like Racetrack. Furthermore, it will be possible to select the plain versions of these memories or the ones which implement LiM features. In all the cases hardware is tailored on the memory array, this limits area consumption and wasted power.
This configurable controller supports the following memories:

- Standard memory array (No LiM);
- LiM standard memory array;
- Racetrack memory array (No LiM)
- LiM Racetrack memory array;

In this Thesis the word *configurable* is adopted multiple times with different meanings, it is used in this Section to indicate the capability if the memory controller to instantiate different types of memory (in terms of types technologies and LiM functionalities). In the next Chapter the word *configurable* is linked to the fact that the LiM functionalities supported by the Core can be expanded and supported with multiple hardware implementations.
In [5] two main limitations were the insertion of new LiM instructions by hand directly in the .hex file and the limited available bits for programming the memory. In the next Chapter will be shown how the RISC-V GNU/GCC Compiler and the system have been modified to support new LiM feature,s which can be implemented in multiple different ways even different from the actual

implementation. Thus, *configurable* stands either for the capability to select the wanted memory configuration and at the same time also for the possibility to add new LiM features to the system. In Chapter 2 a brief description of the current LiM memory array model will be given, furthermore all the hardware and software modifications required to extend the LiM functionalities will be described. Chapter 3 presents the Racetrack memory array design and its integration with the RISC-V Core. Chapter 4 will be focused on the comparison of the different structures which are supported by the configurable memory controller.

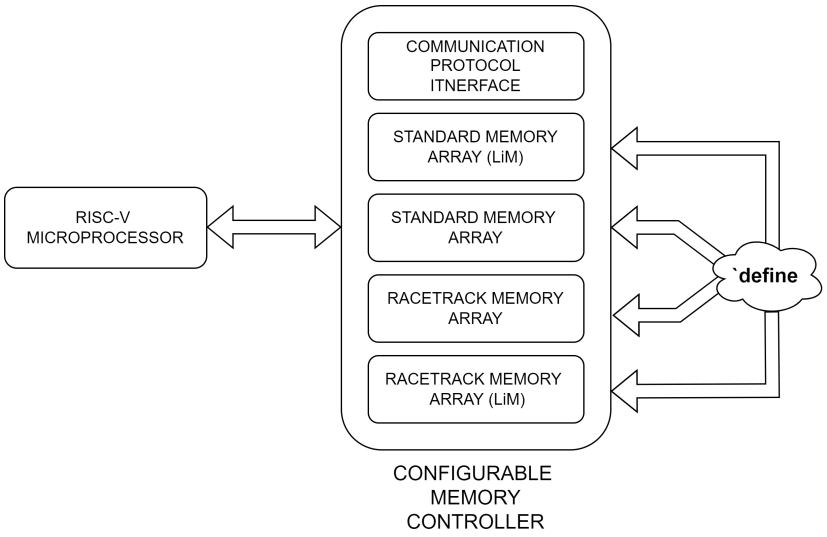

Figure 1.43. Configurable dual-port memory interface model

Chapter 2

LiM RISC-V configurable Framework

2.1 LiM framework starting point

This project is based on the already available structure presented in [5], in this work the architecture of the ID stage was modified to support LiM operations. Then, also the memory model was integrated with additional logic to support in-memory operations like bit-wise operations and max/min research, these functionalities will be described in the following. The proposed LiM architecture, shown Figure 2.1, is capable to perform standard and special operations:

- Standard *load* and *store*: as in a classic RISC-V architecture, these instructions are able to serve all the memory requests fetched by the core;
- Bit-wise oeprations: each cell of the memory array has a set of built-in logic gates which enable in-place logic operations like AND, OR, XOR in just one clock cycle. In the case of a *logic store*, memory content is overwritten by new computed values, while for a *logic load* the selected value is not corrupted but it is served to the Core once computed the logic operation;
- Range operations: an external hardware logic supports range operations during store logic instructions and max/min research;
- Max/Min research: a special instruction triggers the max/min logic which computes, in a fully parallel way, the max or the min value among a set of values specified by the user in just 33 clock cycles;

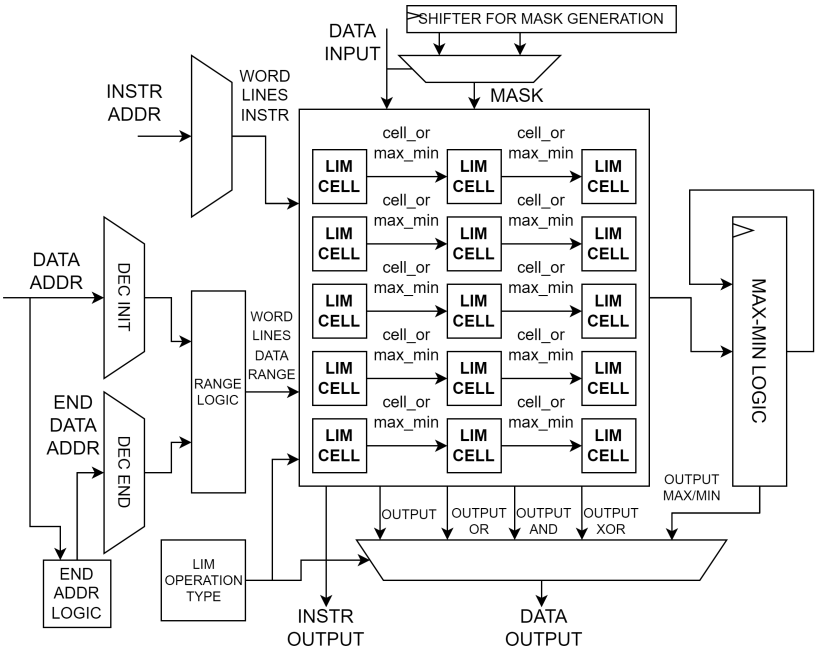

Figure 2.1. Dual-port Logic-in-Memory architecture 2.1

Figure 2.2 depicts a 1-bit LiM cell architecture, the output of the memory cell is used to compute the three logic operations and the wired-or output necessary for max/min computation. The input value is chosen with an external signal which allows to select the bit to write in the memory cell between a value provided from outside or one of the feedback logic output. All the logic operations are performed with a *mask* provided by the user, thus this structure resembles a *vector processor*.

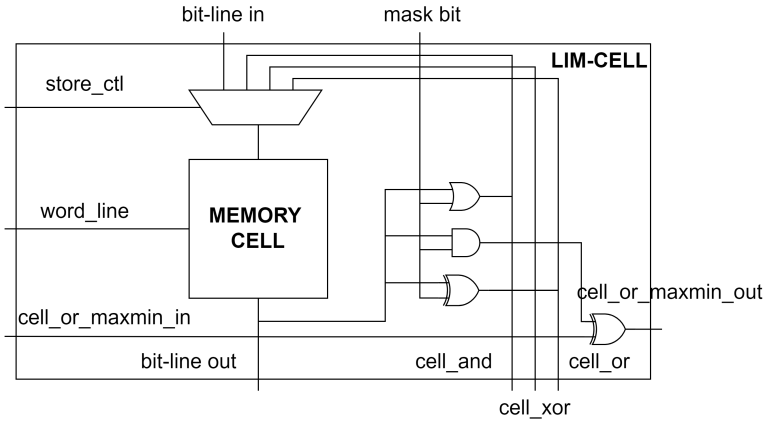

Figure 2.2. Logic-in-Memory bit cell 2.1

2.2 LiM functionalities state of the art

Unfortunately, this structure presents a limited set of logic operations, with the objective of supporting new in-memory operations, literature was investigated to understand the most adopted ones. A brief review of the most interesting LiM architecture is listed in the following:

- Hybrid-SIMD [11]: this architecture is composed by a stack of standard memory rows and enhanced rows called *Smart rows*. Standard rows are fundamental, they hold data used by smart rows during computations. Each smart row is composed by multiple cells, which contains a storage element, two XOR gates and a full adder.
  In this Single Instruction Multiple Data (SIMD) architecture each smart-row's bit-cell supports different logic operations like OR, AND, XOR, $\overline{XOR}$ and even sum operations by means of the full adder. Furthermore, a row interface offers support to complex operations like multiplications, tailored on the mapped algorithm;

- CLiMA [27]: this PhD Thesis proposes a configurable hybrid-in-memory and near-memory architecture. This structure is composed by LiM cells and eventually it is surrounded by near-memory logic to enhance the structure's capabilities and overcome the memory bottleneck problem. This architecture can exploit the LiM approach but it is able to support also other degree of in-memory computing if it is necessary. Simple logic operations, such as configurable AND, OR, XOR or even inter-row addition are computed directly in each bit-cell, while more complex operations are carried out by means of additional external logic. Furthermore, thanks to the built-in full adder, fixing one or more inputs, it is possible to obtain more complex logic operations;

- DRC2 [4]: this work presents a peripheral SRAM accelerator circuitry, which offers ALU-like capabilities. This architecture exploits the concept of logic-near-memory because the memory block is fabricated with standard 10T SRAM technology. The peripheral ALU-like circuitry supports logic operations such as XOR, OR, AND (also their inverted forms) and more complex operations like addition/subtraction, shift, increment and decrement by 1 and grater/less computation;

- CRAM [30]: an hybrid-in-/near-memory bit-serial SRAM architecture called CRAM is proposed. Data element of two different words to be manipulated are placed in the same word-line, then peripheral bit-line logic performs the required operation. Data elements of the same bit-line are manipulated one bit at the time, bits of the two operands involved in the computation, need to be activated, then peripheral circuitry will perform the required computation and finally result is stored back in another cell.
  This architecture offers a wide range of operations like bit-wise operations (AND, NOR, XNOR and inverted ones), addition/subtraction, multiplication and other listed in Table 2.1;

- MISK [33]: standard 6T SRAM cells are interleaved with logic layers, then at the end of each standard-logic stack a latch layer is placed to store intermediate and final results. Each logic layer is capable to manipulate bits of the surrounding data layers, this architecture supports XOR operation and a programmable 2-bit LUT;

In Table 2.1 all the analyzed architecture with their supported instructions are reported.

Table 2.1. In-memory/near-memory supported instructions

Instructions	Standard-LiM	Hybrid-SIMD	CLiMA	DRC2	CRAM	MISK
AND	✓	✓	✓	✓	✓	
NAND		✓		✓	✓	
OR	✓	✓	✓	✓	✓	
NOR		✓		✓	✓	
XOR	✓	✓	✓	✓	✓	✓
XNOR		✓	✓	✓	✓	
MAX/MIN	✓					
SUM		✓	✓(RCA)	✓	✓+FP	
MULT			✓(Array Mult.)	✓	✓+FP	
DIV					✓(UNS) +FP	
GREATER/LESS	✓			✓	✓	
INC/DEC by 1				✓		
EQUAL					✓	
SEARCH					✓	
LUT						✓

N.B. Standard-LiM is the initial structure described in Section 1.5.

This Table shows how these structures support a wide range of logic operation, ranging from simple ones (i.e. bit-wise operations) to complex ones (i.e. multiplications and divisions) which require expensive additional logic. In order to keep the structure similar to the original one, for the sake of simplicity it was decided to implement the following operations:

- NAND
- NOR
- XNOR

This implies small modifications to the already provided LiM RAM model, but at the same time this could bring several improvements in terms of logic functionalities.

2.3 LiM Instructions

The available LiM architecture is capable to perform different logic operations, but this structure needs to be controlled in some way by the RISC-V. For this reason, two custom-LiM instructions were designed, in the following they will be briefly described.

2.3.1 Store_active_logic instruction

This new instruction is defined to program the LiM architecture, it is possible to specify the operation type to perform in memory and the range on which such operation will be applied. Figure 2.3 shows the instruction format, which is composed by:

- opcode: new opcode generated from scratch;
- rsN: contains the address of the Register File's location where the range information is stored;

- funct3: this field specifies the LiM operation;
- rs1: source register;
- imm: immediate field, together with the content pointed by rs1 builds the target address;

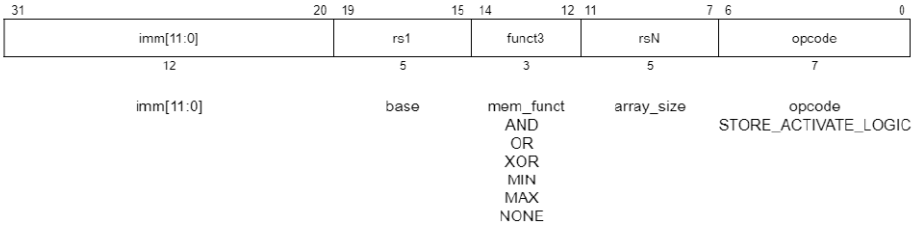

Figure 2.3. store_active_logic instruction

This instruction has the task to program the memory, once the range size is retrieved from the Register File, it is merged together with the funct3 bits to create the 32-bit program word. Then, the RISC-V operates a store in the memory and it writes the program word in the special programming address, Figure 2.4 represents a schematic view of the instruction's work-flow.
Once store_active_logic is completed, LiM memory is capable to perform logic operations.

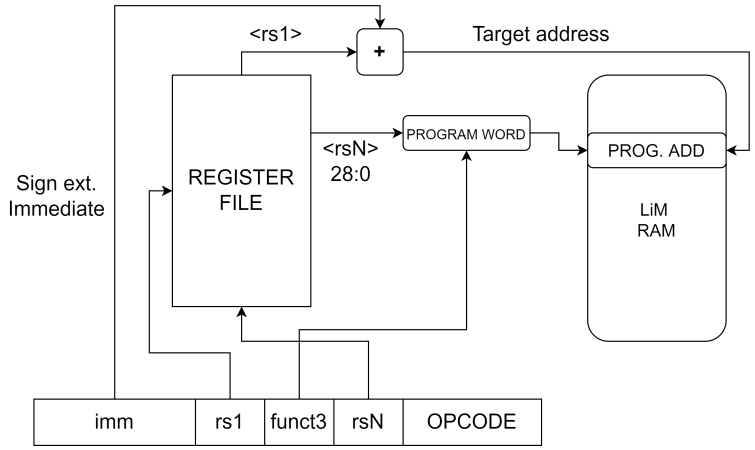

Figure 2.4. store_active_logic work-flow

2.3.2 Load_mask instruction

The standard RISC-V load instruction is not suited to perform a logic load, for this reason a new instruction was defined. Load_mask instruction sends to memory, through the data bus, the mask value read by means the source register rs2. Then the logic value is computed and sent to the RISC-V core, Figure 2.6 represents the work-flow of the instruction. Differently from a logic store, the logic load performs logic operations only on single values.
The format of this instruction (Figure 2.5) is the following:

- opcode: new opcode generated from scratch;

- rd: destination register for load operation;
- funct3: specifies the word width;
- rs1: base register;
- rs2: source register, points to RF's location containing mask value;
- imm: immediate field, together with the content pointed by rs1 builds the target address;

Figure 2.5. load_mask instruction

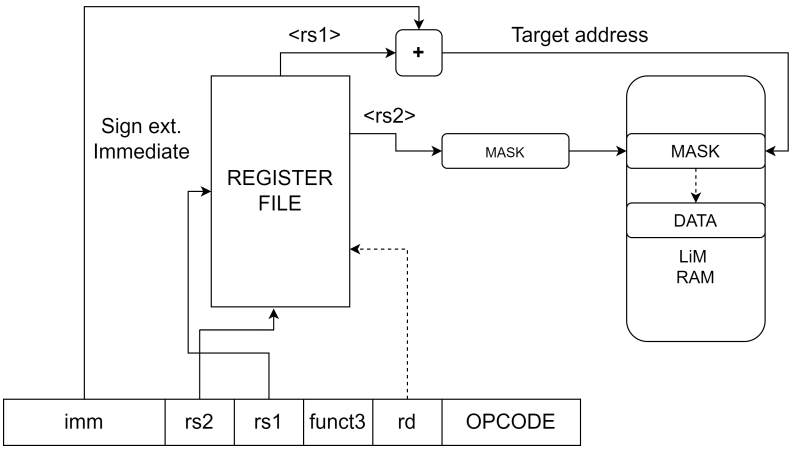

Figure 2.6. load_mask work-flow

2.3.3 Store

This is not a new instruction, once memory is programmed, a store instruction acts differently from a standard RISC-V store (sw). This instruction exploits the programming of the LiM structure and perform the corresponding logic operation directly in memory. The value specified by the source register rs2 is interpreted in this case as the mask value for the logic operation;

2.4 LiM instruction set extension

2.4.1 Store_active_logic redefinition

The 32-bit LiM program word (Figure 2.7) issued by store_active_logic is a combination of two information:

- funct3: 3-bit field, specifies the LiM operation (AND, OR, XOR, MAX/MIN, NONE);
- operation size: 29-bit field coming from the Register File, gives the information of the size of the range operation to perform in memory;

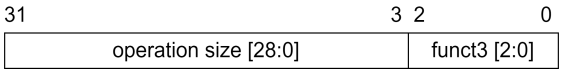

Figure 2.7. Program word format

Table 2.2 reports the encoding of the funct3 field, as clearly visible only one combination (3'b111) is free, this narrows down the possibility to expand the set of supported instruction of the LiM structure. This is clearly a limitation in terms of supporting new functionalities and with a view to turn this structure in a LiM-sandbox, the available function field has to be expanded.

Table 2.2. funct3 field encoding

Function	Code
NONE	000
XOR	001
AND	010
OR	011
MIN	101
MAX	110

It was decided to redesign the store_active_logic instruction to expand the funct field of the program word. The new instruction is reported in Figure 2.8, in which the following fields were modified:

- imm: immediate field reduced from 12-bits to 7-bits as in the load_mask instruction;
- funct: new 5-bits function field, which expands the available codes for new instructions;

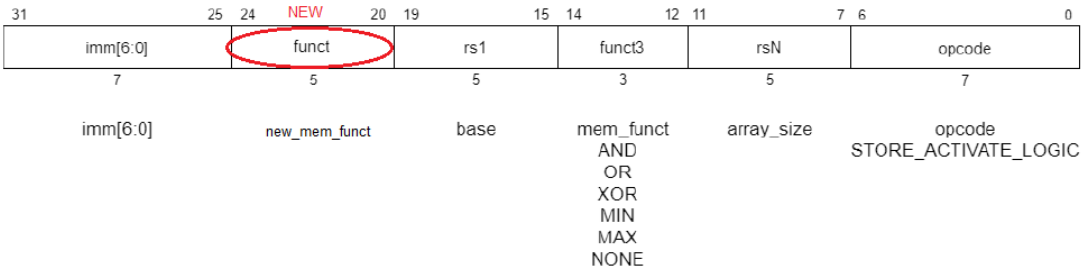

Figure 2.8. New store_active_logic format

The new program word structure is depicted in Figure 2.9, here the function field is composed by 8-bits enabling the possibility to support up to $2^8 - 1$ different LiM operations. Operand size field is reduced to 24-bits, this is not a limitation because the RISC-V's RAM address width is on 22-bits, so it is fully possible to support a range operation on the whole memory.

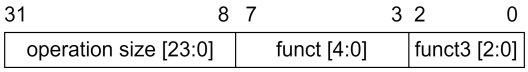

Figure 2.9. New program word format

This design choice implies small modifications to:

- RISC-V Core: modify ID-stage and decoder module to support new store_active_logic;
- LiM RAM: modify program word decoding and add inverting logic operations support;
- RISC-V Compiler: modify instruction definition and define new instructions;

The benefits of this solution choice are several, first of all a huge set of new function encodings are unlocked, this expand dramatically the LiM capabilities. The program word decoding undergoes small modifications since the new function field is composed by funct3 and the new defined funct field, so only a re-size of some signals is required. Considering to support also inverting versions of the already available logic instructions, the new function codes result in the ones reported in Table 2.3.

Table 2.3. New funct field encoding

Function	Code
NONE	0000000
XOR	0000001
AND	0000010
OR	0000011
XNOR	0001001
NAND	0001010
NOR	0001011
MIN	0000101
MAX	0000110

2.4.2 RISC-V modifications

RISC-V ID-stage hosts the modifications to build the LiM program word and to decode corretly the load_mask 7-bits immediate field. These modifications are deeply described in [5], but it is worth to recall them. The Immediate Sign Extension Block was modified to host a new output named *imm_logmem_type* which sign-extends the immediate field of the load_mask isntruction. Then the program word is sent to the LiM structure through alu_operand_c mux which takes a new special input. This new input takes 28-bits operation size information from the RF and 3-bits funct3 field directly from the instruction.
To support the new store_active_logic several modifications are needed:

- Immediate selection in Immediate Sign Extension Block takes *imm_logmem_type* as in a load_mask instruction;
- logic_in_mem_funct input of alu_operand_c mux is built also with the new portion of bits [24:20] (new funct field) of the input instruction (Figure 2.10);

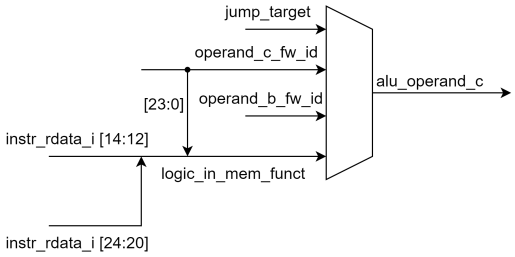

Figure 2.10. alu_operand_c mux modifications

These modifications are necessary to build the new LiM program word according to the new store_active_logic format.

2.4.3 LiM RAM modifications

The LiM structure in the RAM model needs only few modifications, only the way in which the program word is decoded has been modified. The operation size signal has been reduced while the function signal has been enlarged. Then, to support the new inverting instructions (NAND, NOR,

XNOR), additional logic has been inserted. The feedback write mux has been enlarged to host the inverting logic operations, which are generated just adopting NOT gates as shown in Figure 2.11.

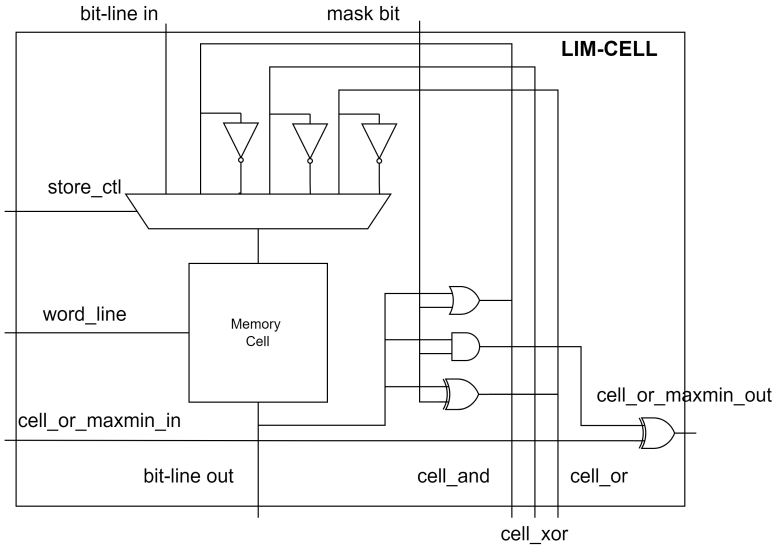

Figure 2.11. Modified LiM cell

The output logic has been modified as well in the same way, as depicted in Figure 2.12.

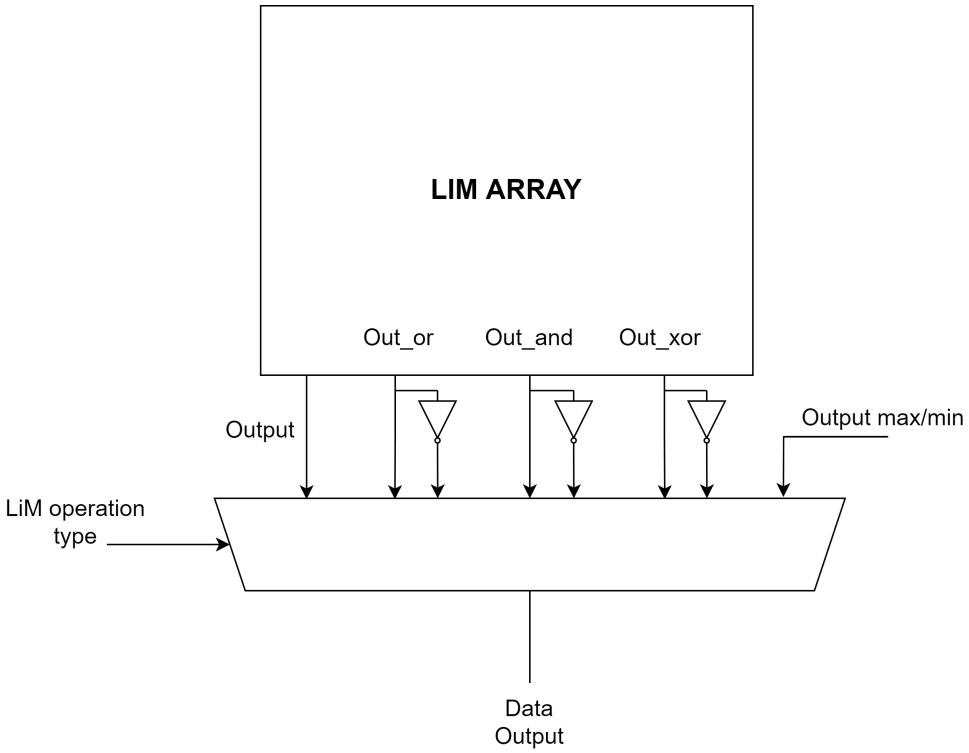

Figure 2.12. Modified output LiM output logic

2.4.4 RISC-V GNU-GCC toolchain modifications

Hardware modifications are necessary to support this new extended LiM framework, but this requires also software modifications. In the original work, LiM instructions (i.e. store_active_logic and load_mask) were inserted by hand directly in the .hex file. This is not easy and requires a deep understanding of the assembly code, for this reason it was decided to modify the RISC-V GNU-GCC Toolchain to define new LiM-custom instructions.
According to [26], modifications have been applied to RISC-V GNU-GCC Binutils, two files have been modified:

- riscv-opc.h: #define (Figure 2.13) and DECLARE_INS() (Figure 2.14) of new instructions have been inserted;

- riscv-opc.c: all new instructions have been declared with their new parameters (Figure 2.15);

```
/* Instruction opcode macros.  */
#define MATCH_SW_ACTIVE_OR 0x303b
#define MASK_SW_ACTIVE_OR  0x1f0707f
#define MATCH_SW_ACTIVE_AND 0x203b
#define MASK_SW_ACTIVE_AND  0x1f0707f
#define MATCH_SW_ACTIVE_XOR 0x103b
#define MASK_SW_ACTIVE_XOR  0x1f0707f
#define MATCH_SW_ACTIVE_NONE 0x3b
#define MASK_SW_ACTIVE_NONE 0x1f0707f
#define MATCH_SW_ACTIVE_MIN 0x503b
#define MASK_SW_ACTIVE_MIN  0x1f0707f
#define MATCH_SW_ACTIVE_MAX 0x603b
#define MASK_SW_ACTIVE_MAX 0x1f0707f
#define MATCH_SW_ACTIVE_NOR 0x10303b
#define MASK_SW_ACTIVE_NOR  0x1f0707f
#define MATCH_SW_ACTIVE_NAND 0x10203b
#define MASK_SW_ACTIVE_NAND  0x1f0707f
#define MATCH_SW_ACTIVE_XNOR 0x10103b
#define MASK_SW_ACTIVE_XNOR  0x1f0707f
#define MATCH_LW_MASK 0x201b
#define MASK_LW_MASK  0x707f
```

Figure 2.13. riscv_opc.h #define for new instructions

```
DECLARE_INSN(sw_active_or, MATCH_SW_ACTIVE_OR, MASK_SW_ACTIVE_OR)
DECLARE_INSN(sw_active_nor, MATCH_SW_ACTIVE_NOR, MASK_SW_ACTIVE_NOR)
DECLARE_INSN(sw_active_and, MATCH_SW_ACTIVE_AND, MASK_SW_ACTIVE_AND)
DECLARE_INSN(sw_active_nand, MATCH_SW_ACTIVE_NAND, MASK_SW_ACTIVE_NAND)
DECLARE_INSN(sw_active_xor, MATCH_SW_ACTIVE_XOR, MASK_SW_ACTIVE_XOR)
DECLARE_INSN(sw_active_xnor, MATCH_SW_ACTIVE_XNOR, MASK_SW_ACTIVE_XNOR)
DECLARE_INSN(sw_active_none, MATCH_SW_ACTIVE_NONE, MASK_SW_ACTIVE_NONE)
DECLARE_INSN(sw_active_min, MATCH_SW_ACTIVE_MIN, MASK_SW_ACTIVE_MIN)
DECLARE_INSN(sw_active_max, MATCH_SW_ACTIVE_MAX, MASK_SW_ACTIVE_MAX)
DECLARE_INSN(lw_mask, MATCH_LW_MASK, MASK_LW_MASK)
```

Figure 2.14. riscv_opc.h DECLARE_INS() for new instructions

```
const struct riscv_opcode riscv_opcodes[] =
{
/* name,     xlen, isa,   operands, match, mask, match_func, pinfo.  */
{"sw_active_or",          0, INSN_CLASS_I, "d,s,j",     MATCH_SW_ACTIVE_OR, MASK_SW_ACTIVE_OR, match_opcode, 0 },
{"sw_active_and",          0, INSN_CLASS_I, "d,s,j",     MATCH_SW_ACTIVE_AND, MASK_SW_ACTIVE_AND, match_opcode, 0 },
{"sw_active_xor",          0, INSN_CLASS_I, "d,s,j",     MATCH_SW_ACTIVE_XOR, MASK_SW_ACTIVE_XOR, match_opcode, 0 },
{"sw_active_none",          0, INSN_CLASS_I, "d,s,j",     MATCH_SW_ACTIVE_NONE, MASK_SW_ACTIVE_NONE, match_opcode, 0 },
{"sw_active_max",          0, INSN_CLASS_I, "d,s,j",     MATCH_SW_ACTIVE_MAX, MASK_SW_ACTIVE_MAX, match_opcode, 0 },
{"sw_active_min",          0, INSN_CLASS_I, "d,s,j",     MATCH_SW_ACTIVE_MIN, MASK_SW_ACTIVE_MIN, match_opcode, 0 },
{"sw_active_nor",          0, INSN_CLASS_I, "d,s,j",     MATCH_SW_ACTIVE_NOR, MASK_SW_ACTIVE_NOR, match_opcode, 0 },
{"sw_active_nand",          0, INSN_CLASS_I, "d,s,j",     MATCH_SW_ACTIVE_NAND, MASK_SW_ACTIVE_NAND, match_opcode, 0 },
{"sw_active_xnor",          0, INSN_CLASS_I, "d,s,j",     MATCH_SW_ACTIVE_XNOR, MASK_SW_ACTIVE_XNOR, match_opcode, 0 },
{"lw_mask",                0, INSN_CLASS_I, "d,s,t,j",     MATCH_LW_MASK, MASK_LW_MASK, match_opcode, 0 },
```

Figure 2.15. riscv_opc.c declaration for new isntructions

After applied these modifications and re-built the compiler, the new Toolchain is able to support LiM instructions. Differently from the original work, now it is possible to define in-line assembly code and exploit new custom instructions.

Chapter 3

LiM Racetrack memory model

3.1 Racetrack technology overview

Once defined a configurable LiM Framework, the new goal is to decline the current architecture to a novel memory technology named *Racetrack*. Before discussing the solutions adopted in the Racetrack memory design, it is worth to briefly review what is Racetrack technology.
During recent years, a lot of studies have been focused on emerging technology, it is exactly in this scope that Racetrack technology is located [7]. Standard memory technologies suffer of physical and technological limitations, modern HDDs have reached an astonishing storage capability but they lack in speed and data retention, while modern technologies like MRAMs are still to far from being competitive both on the cost side as well as in terms of integration or storage capability. Racetrack technology tries to solve both problems promising a low-power and high-storage capability.
A Racetrack is basically a ferromagnetic track where data is stored as the magnetization state of different areas called *Magnetic Domains*. The magnetic nature of this memory implies immediately a low-power behaviour because, as mentioned before, bits are represented by the magnetization direction of each Magnetic Domain, so there is no wasted power involved in data retention. Furthermore, this technology is not limited by 2-D structures, it is possible to design horizontal Racetracks as well as vertical ones, this increases dramatically the integration capabilities. Data is accessed by means of *access ports*, which are composed by transistors and MTJs. To access a specific bit along the track, unless it is already aligned with an access port, it should be shifted through a current pulse (other methods relying on different effects can be exploited), electrons interact with the magnetization of Magnetic Domains, resulting into a shifting of magnetization states along the track. Data is then sensed through a read current which evaluates the magnetization state of the cell, it is also possible to switch the magnetization state forcing a high value current. Generally, access ports are much larger than Domains, for this reason the number of ports is always lower than the overall number of Domains within a Racetrack. Furthermore, multiple Racetracks can share the same access port, this increases are occupation efficiency.
In this Thesis, SOT access ports are used, as explained for MRAM technology in Subsection 1.2.4, this technology implies that Read and Write current paths are decoupled.

3.2 pNML NAND-NOR gate

The fundamental part of the proposed Racetrack architecture is based on the *Nano Magnetic Logic* (NML) technology. In this context it is possible to distinguish two different types of NML, iNML and pNML, in particular for this Thesis the pNML type is adopted. The acronym pNML stands for *perpendicular Nano Magnetic Logic*, this implies that the magnetization direction is always

perpendicular to the magnetic material. The peculiarity of NML technology is the possibility to arrange multiple magnetic elements and exploit the interaction of their magnetization direction to create logic functions.
Declining the Racetrack technology in a LiM context implies to adopt special pNML cells able to perform in-place computations. In this regard a programmable NAND-NOR pNML gate is proposed in [6] and itwas be the starting element for the LiM Racetrack design. This gate, as shown in Figure 3.1, has a multi-layer structure. The top layer hosts the track of control gates which program the bottom central cells, turning the structure into a 2-port NAND gate or a 2-port NOR gate. The bottom layer is composed by multiple cells, central ones are arranged as a Racetrack and they will generate the logic values, sides cells represent the inputs of the logic gate. These latter cells can be also arranged in a Racetrack context, thus this makes possible to compute bit-wise operations directly in memory. Thus, the final Racetrack is arranged as a multi-layer structure composed by multiple Racetrack with different tasks.

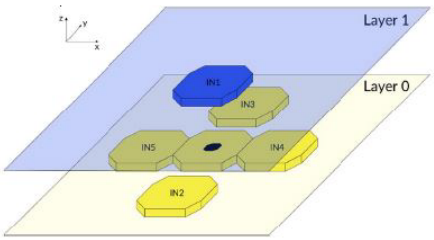

Figure 3.1. pNML NAND-NOR gate [6]

3.3 HDL models of Racetrack cells

The new LiM Racetrack memory model will be fully described in *SystemVerilog*, thus each physical cell will be translated into the correspondent HDL model trying to mimic as much as possible the real behaviour. As described in the previous sections this memory embeds in-place computing cells (i.e. pNML NAND NOR gates), standard Racetrack cells for shifting values within the Racetrack lines and also read/write cells which act as access ports.
All the adopted HDL models, used as starting point, were proposed in [6].

3.3.1 pNML NAND/NOR gate model

This is the fundamental cell for the Racetrack logic part, this cell is responsible of computing NAND/NOR operations. Its physical counterpart has been briefly described in the previous section, the corresponding model was proposed in [6] and it is tailored on the expected cell's real behaviour.
The HDL model is the one depicted in Figure 3.2, it is important to highlight that this cell is integrated in a Racetrack-based context, thus it is replicated multiple times and interconnected with the neighboring replicas. The cell switches in two different situations, the simplest one is during shift operations, where a shift current enables the shifting of bits stored in the Racetrack, thus the gate can host bits coming from surrounding cells. The second situation, is during the computation of a NAND or a NOR operation, where the gate switches only if the applied magnetic field's sign is opposite to the magnetization of the cell. Both shift current and magnetic field are expressed in terms of magnitude and sign.

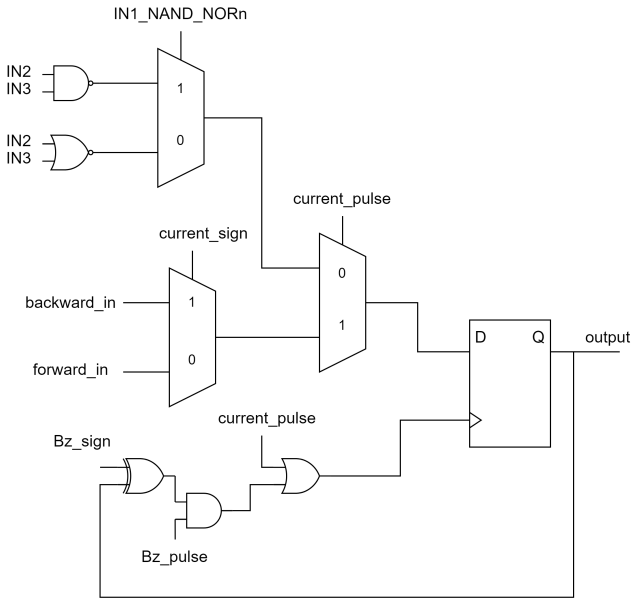

Figure 3.2. pNML NAND-NOR gate HDL model

The model is very simple, in the upper part a NAND and a NOR gates computes the logic operation, then a mux forwards the result based on the cell's programming. The shift current sign selects the right or the left value during a shift operation, the presence or not of the shift current pulse acts as a control signal for the central mux. In this way, if a shift operation is present, bits coming from left or right cells are forwarded, if the current pulse is not present the computed logical result is selected. The storage part of the cell is modeled as a Flip-Flop triggered by a signal generated by several logic gates, the switching pulse is generated only in those situations described previously.

3.3.2 pNML NAND/NOR SOT gate model

As will be clear in next Sections, the cell presented previously is not responsible for read and write operations. Access cells are based on a different technology named SOT. Actually, the current cell model is not capable to write external values, but this is not a problem since the central Racetrack is only responsible for logic computations. The only modification with respect to a standard pNML NAND-NOR gate is the ability to read the stored value, this feature has been modeled using an additional Flip-Flop which takes as input the output signal of the storage element and it is triggered by a read signal provided from the external. These cells are used for modeling read ports of the Racetrack, the resulting HDL model is reported in Figure 3.3.

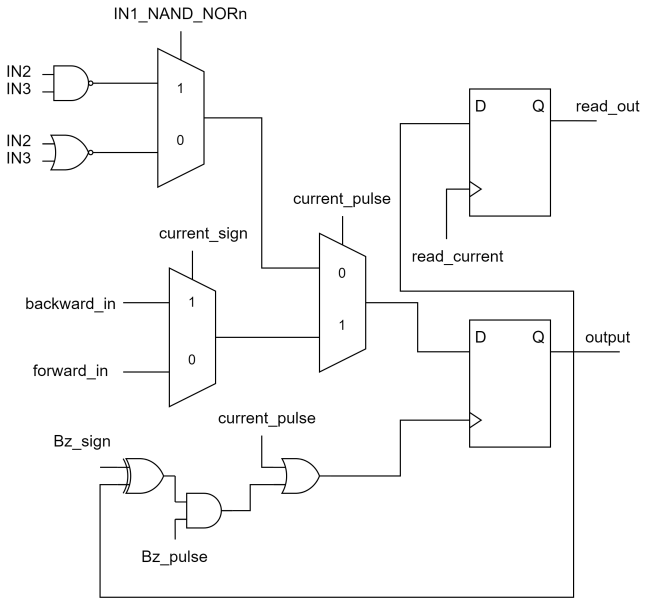

Figure 3.3. pNML NAND-NOR SOT gate HDL model

3.3.3 Standard Racetrack cell model

Previously, the basic pNML NAND-NOR gate structure has been described, the computational part is the central cell, while surrounding cells have different tasks like programming the logic function or providing the inputs. All these cells are modeled with standard Racetrack cells which are capable only of storing and shifting data. The HDL model presented in Figure 3.4 is very straightforward, a Flip-Flop triggered by the shift current pulse acts as memory element, the input is selected based on the current pulse sign.

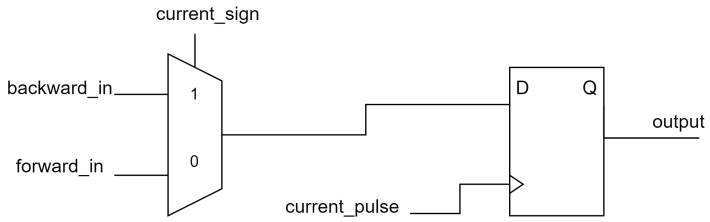

Figure 3.4. Standard Racetrack cell

3.3.4 Racetrack read-write SOT model

Standard Racetrack cells are not enough for designing a complete Racetrack model, two important features are missing. Data need to be written and stored values should be read, special SOT cells are adopted for these purposes. Such cells are used as read and write ports. The read feature of the SOT cell is modeled adding a Flip-Flop triggered by a read current while two multiplexers select the input value for the storage Flip-Flop (Figure 3.5).

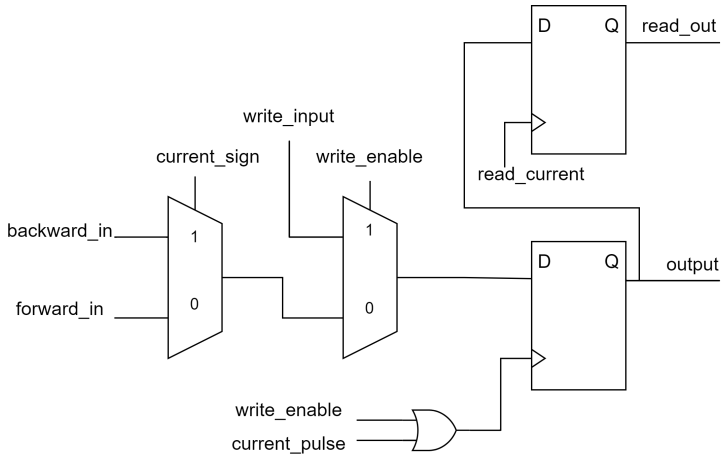

Figure 3.5. Standard Read-Write SOT Racetrack cell

3.4 Racetrack memory array design

The next step is to design the whole memory array starting form the proposed basic cells. It was decided to design the new memory with two different modes. The first one, named *LiM Mode* is exploited in a LiM context providing enhanced functionalities (i.e. NAND-NOR operations), while the second one, named *Memory Mode* exploits the same Racetrack structure but it is adopted as a standard memory. The presence of a standard *Memory Mode* is justified by the exploitation as much as possible of the available hardware. In this way, the current Racetrack composed by 4 single Racetracks can be exploited, increasing the memory capacity by a 3x factor as explained in the following.

3.4.1 Design parameters

The new Racetrack memory has a hierarchical structure, all the decisions taken during the architectural design are based on [34]. In this work, an elementary block named *Macro Unit* (MU) is presented, each of them embeds multiple pNML Racetracks.
The MU has three main design parameters:

- Number of Racetracks (Nr);
- Number of access ports (Np);
- Number of domain per Racetrack (Nb);

Many different solution are suggested, based on [34] it was decided to adopt a configuration suitable for low-power and low-area applications like IoT ones. It was decided to adopt MU-32-08-04 layout which includes 32 domains, 8 access ports and 4 Racetrack per MU.

3.4.2 Head management policies

Once decided the layout of the basic building block, it is important to highlight the management policy for access ports. Having a number of access ports equal to the number of domain within a Racetrack would be impossible because read/write cells occupies more area and consume more

energy with respect to standard cells. For this reason the number of these elements is limited, this implies shift operations to align required data to the access port. Head management policies have a huge impact on the wasted power during accesses, so it is important to find a good trade-off between efficiency and wasted power.
There are two main aspects to take into account [29]:

- Head selection: this policy selects the appropriate "head" for data access, the selection could be *static* or *dynamic*. In the first case each part of the Racetrack is assigned statically to an access port, in the second one the selected port is the nearest to the required data;
- Head update: this property decides what to do with a port after an access. There are three possibilities: *Eager*, *Lazy* and *Pre-shifting*. In the first one, access ports are restored in their initial positions, this simplifies the shift protocol and also the shift logic. In the second case the access ports are not restored, this policy tries to exploit the spatial locality of access to minimize the required shift for the next access. In this case the logic involved in the computation of the number of required shifts is much more complex because it is necessary to take into account the current position of each port. The last possibility tries to guess the next likely accessed data performing in advance a certain number of shifts, this require a prediction algorithm and an appropriate control logic, it is the most complex one.

Fort this work it was decided to adopt the combination Static Selection and Eager Update identified with the name *Static-Eager* (SE) to reduce the design complexity.

3.4.3 LiM Racetrack

In previous sections, it was defined the Racetrack MU parameters (i.e. Number of access ports and number of bits per line) and the policies which govern the access port management, Figure 3.6 shows the resulting layout of a single Racetrack within a MU. These four separate Racetracks interact together to give the possibility to compute in-place NAND and NOR operation.
It is important to clarify the meaning of the name *LiM Racetrack*, hereinafter with this name, the whole structure made of four different Racetracks will identified. In all the other cases, when it is necessary to refer to a specific Racetrack within the *LiM Racetrack*, a meaningful name will be used, such as *Program Racetrack*, *Data Racetrack*, *Mask Racetrack*, *Logic Racetrack*, these names refer to the internal separate Racetracks.

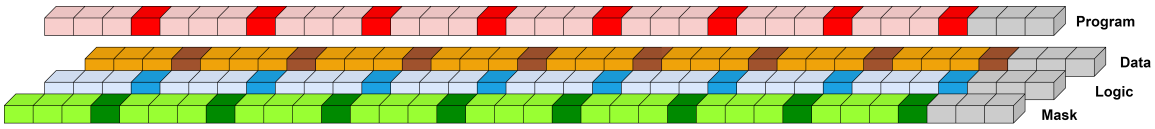

Figure 3.6. *LiM Racetrack* structure

The Figure shows the multi-layer structure described in Section 3.2, the red top Racetrack, named *Program Racetrack*, is responsible for the programming of the whole structure (it selects the NAND or NOR operation), while the bottom layer is composed by three separated Racetracks. The central blue Racetrack is the *Logic Racetrack* and it is the logic part which computes the NAND/NOR operation while peripheral Racetracks are respectively the *Mask Racetrack* (Green) and the *Data Racetrack* (Orange).
During an access (read or write operations) data should be shifted to align correctly ports to the accessed location. Each Racetrack could be thought as a long shift register where bits are shifted

back and forth, to prevent data loss during shift operations some overhead cells are placed at the beginning of each Racetrack (gray cells), these are built with standard Racetrack cells able only to shift data.
Data parallelism differs considering different operational modes.During the *LiM Mode*, Figures 3.7 and 3.8, a single Data bit or Logic bit is stored in a single *LiM Racetrack*, so data parallelism is 1-bit.

Figure 3.7. *LiM Mode* - Data parallelism

Figure 3.8. *LiM Mode* - Logic data parallelism

Considering the *Memory mode*, data parallelism is completely different because *Data*, *Program* and *Mask Racetracks* are used to store three bits. Thus, a single *LiM Racetrack* is actually storing a single bit of three different words, Figure 3.8 gives an idea of the configuration, thick lines of different colors highlights the bits of the three words. In this Mode, *Logic Racetrack* is not exploited to store data, unfortunately further studies are required to understand if it is possible to exploit it for storing data, for this reason in this work it was decided to adopt a conservative solution, so *Logic Racetrack* is not used in *Memory Mode*. *Data Racetrack*, *Program Racetrack* and *Mask Racetrack* can be controlled separately, each of them can store three different bits of three different words.
Note that in these Figures the *Program Racetrack* and the *Logic Racetrack* are not shown respectively in Figures 3.7-3.8 and Figure 3.8 to simplify the images understanding.

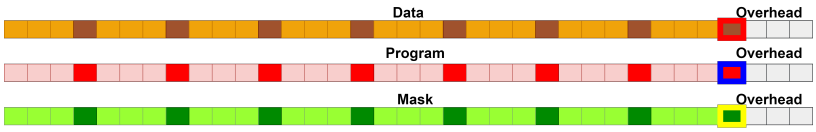

Figure 3.9. *Memory Mode* - Data parallelism

3.4.4 Macro Unit

Previously, it was defined the concept of *Macro Unit* (MU), which is the basic building block of the Racetrack memory array. It was decided to adopt the following design parameters:

- Nb = 32;
- Nr = 4;
- Np = 8;

The resulting Macro Unit structure is depicted in Figure 3.10, all the four *LiM Racetrack* work together. Data parallelism is 4-bits in *LiM Mode* (Logic Data and Data), while is 12-bits in *Memory Mode* (4-bits for each of the 3 different words).

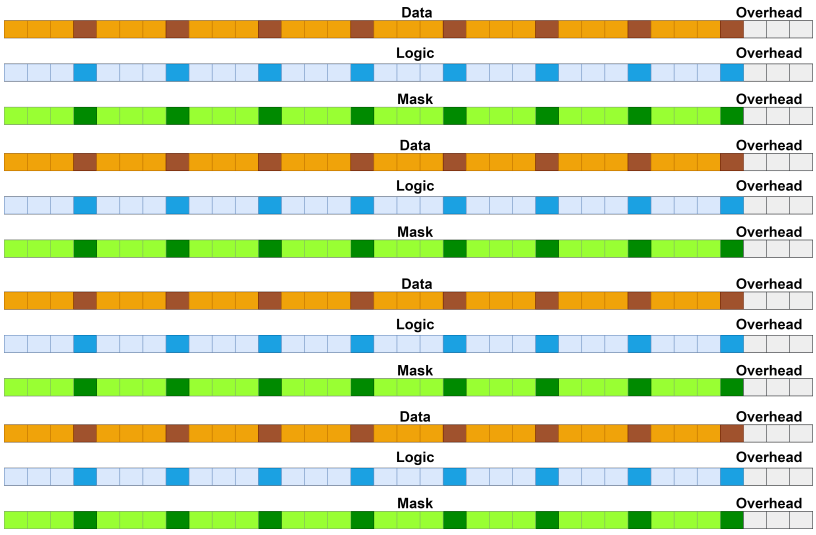

Figure 3.10. Macro Unit structure

3.4.5 Racetrack memory *Block*

A Macro Unit has a Data parallelism of 4-bits in *LiM Mode*, since the main focus of this Thesis is to use the LiM paradigm to speed-up execution, it was decided to adopt this parallelism as the standard one for the MU. Since the RISC-V core works with words made of 32-bits, a new structure composed by 8 MUs, named *Block*, was defined. A Block *Block* has 32-bits parallelism, equal to the original memory model, so it can store a full 32-bits word of the RISC-V. When memory operates in *Memory mode* the parallelism increases of a 3x factor, thus three words can be stored in a single word-line. This comes basically for free because, *Data Racetrack*, *Program Racetrack* and *Mask Racetrack* have separated control signals, so each group of Racetracks is able to store a word. Multiple *Blocks* can be grouped together to satisfy the required memory size, in this Thesis it was decided to adopt the size of typical L1 Cache, it was supposed to use a 256kB size. Thus considering that each *Block* contains 32 words, 2000 *Blocks* are required to satisfy the chosen memory size.

3.4.6 Racetrack waveforms

The Racetrack behaviour is governed by many different waveforms, which are reported in Figure 3.11. As explained in previous Sections, data should be aligned with access ports during a read or write operation, to do so a shift current is required. Signal clk_m models the module of the waveform, the sign is generated by the FSM based on the shift direction.
Read and write operations require a current flow through the Racetrack cell structure, as seen in the MRAM review. These currents are modeled as pulses of 1ns width, note that the write pulse is coming later with respect to the read one, this helps during byte-write operations because in this way stored data is first read and then not-selected bytes are re-written as they are. The last signal is the magnetic field B_Z which is necessary for in-memory computations.
Periods and width of each waveform are based on the results presented in [6], [29], [25], considering a 100 MHz clock system, the following parameters have been selected:

- clk_system = 100 MHz frequency;
- clk_m = 500 MHz frequency, this corresponds to a 1ns width high signal, in [25] shift operations are performed with a 1ns width pulse;
- Bz_S = 50 MHz frequency, results presented in [25], the output logic value is computed after 7/8 ns, for this reason a 20ns period signal was selected;
- Read_pulse: 1ns width pulse, this depends deeply on the sensing circuit, it could be even lower;
- Write_pulse: 1ns width pulse;

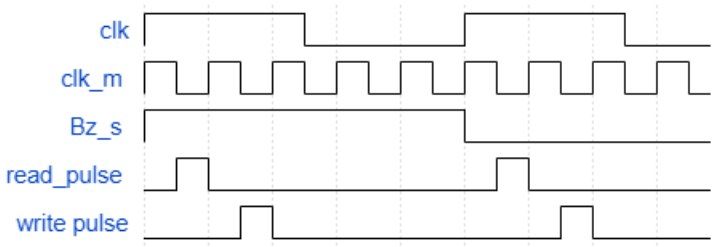

Figure 3.11. Racetrack memory waveforms

3.5 Racetrack Memory architecture

The whole memory array is composed by multiple modules, additional surrounding logic is required to complete the Racetrack memory and assuring the correct behaviour. Figure 3.12 shows the high-level architecture of the Racetrack memory array. This schematic represents only the memory, all the additional logic such as word-line activation logic, hand-shaking logic and other important components will be highlighted in the next Section.
The memory array is very simple, the Racetrack array is controlled by the FSM, which acts as a memory controller, activating all the useful control signals. The shift generator sends the correct number shift pulses based on the location of the accesses. Additional logic is then required to compute logic store and load operations, muxes are exploited to choose the correct computation in both read and write operations.
A more relevant view on the building blocks will be given in the following.

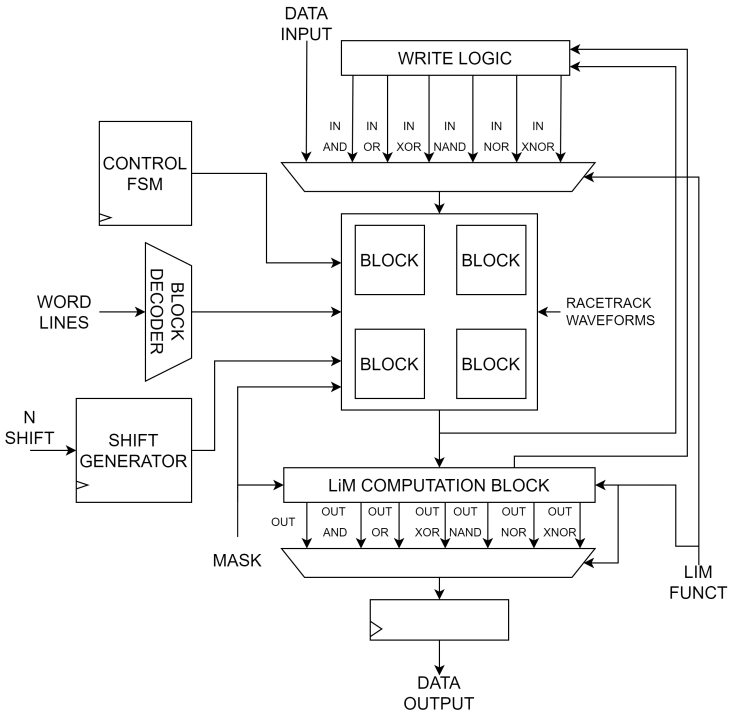

Figure 3.12. Racetrack memory array architecture

3.5.1 Block decoder

This simple logic module, shown in Figure 3.13, activates the correct *Block* based on the memory access position. It simply groups together in a OR operation a set of 32 word-lines, because each *Block* contains 32 words.

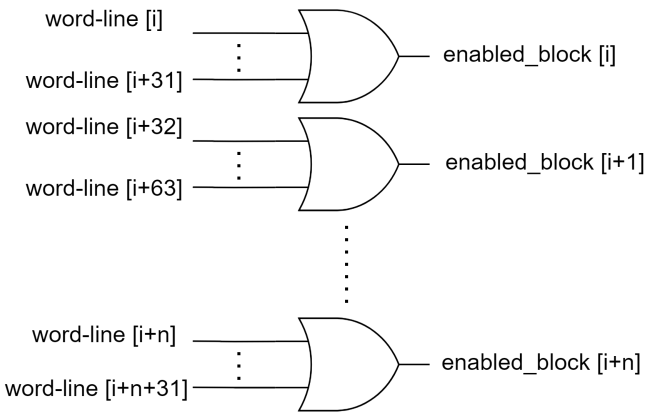

Figure 3.13. Block decoder structure

3.5.2 Output Logic

The Output logic is composed by the LiM Computation Block, a multiplexer and a register. The LiM Computation block, shown in Figure 3.14, computes all the logic results starting from the Racetrack output, as known the Racetrack is capable of in-place NAND/NOR computations, for all the other logic functions additional gates are required. Then, based on the LiM operation type, the correct result is selected and sampled by the output register which has the unique role of stabilizing the output data.

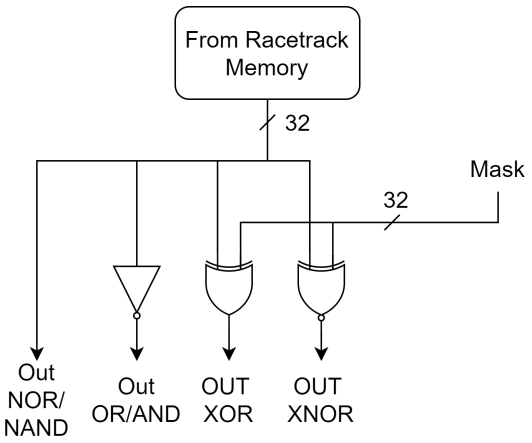

Figure 3.14. LiM computation block structure

3.5.3 Input Logic

The input Logic selects data to write inside the Racetrack memory, based on the operation type it is possible to select between an external value or a feedback logic value computed by the output logic. The original structure had the possibility to perform logic byte-writes during no-LiM operations, it was decided to keep this feature, this required few additional logic. The Racetrack does not provide

natively the capability to perform byte-writes, an additional logic is adopted to sample forward in output not-selected bytes and overwrite selected bytes with new input data. In a standard write operation, data is first read, then selected-bytes are overwritten using the external data input.
As mentioned before, during LiM store operations, data computed by the LiM computation block is reused.

3.5.4 Shifter

This component is fundamental for the correct behaviour of the Racrtrack memory. As explained before, access ports are displaced along each Racetrack, data should be shifted and aligned to the correct access port depending on the adopted head management policy. The shift generator generates the correct number of shift pulses based on the access performed in memory. Once the generation is completed the modules freezes, it sends a "done" signal to the FSM and waits for a new shift request. This component is reset by a control signal issued by the FSM and this blocks the system to start a new transaction just after the port reset. To speed-up the memory accesses, two shift generators work in couple (Figure 3.15), one serves the *set* shifts (port alignment) and the second one serves the *reset* shifts (port reset). This configuration allows to start a new memory access just after the port reset completion. A register samples the shift number for the reset shift generator, in this way the address could change even before the completion of the port reset operation.

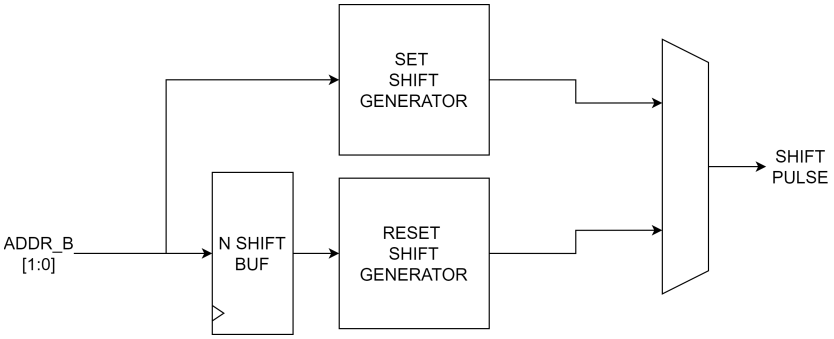

Figure 3.15. Dual shifter architecture

3.5.5 Finite State Machine

The Racetrack behaviour requires a Finite State Machine (FSM) to control all the different actions, in Figure 3.16, the algorithm implemented by the FSM is reported. The first operation carried out by the FSM is the port set, here the shift generator is activated and then a "done" signal is waited, once the port alignment is completed the operations can resume.
Then, the algorithm discriminates between standard operations (no-LiM) , LiM operations which could be executed directly in-place (i.e. NAND/NOR and AND/OR, these latter require additional external logic) and LiM operations which cannot be executed directly in-memory (i.e. XOR/XNOR) and which require external logic. Standard or not-in-place LiM operations are carried out like normal memory accesses. For in-place LiM accesses, the algorithm takes a special path where the first task is to write the mask inside the *Mask Racetrack*, this value is written only in the moment in which a LiM access is performed, this prevent wasting of time. Furthermore, during this state, depending on the type of performed operation, also the program bit is written into the *Program Racetrack*, this is the reason why there are two "write_mask" states, one for each operation type.

Then, the programming magnetic field B_z is applied and the FSM waits a number of clock cycles (which can be set by the user) for the end of the programming phase, finally a read or a write operation is performed. The access cycle terminates with a port reset, when this task is completed and if a new memory request is present, it is possible to jump directly to a new port set operation.

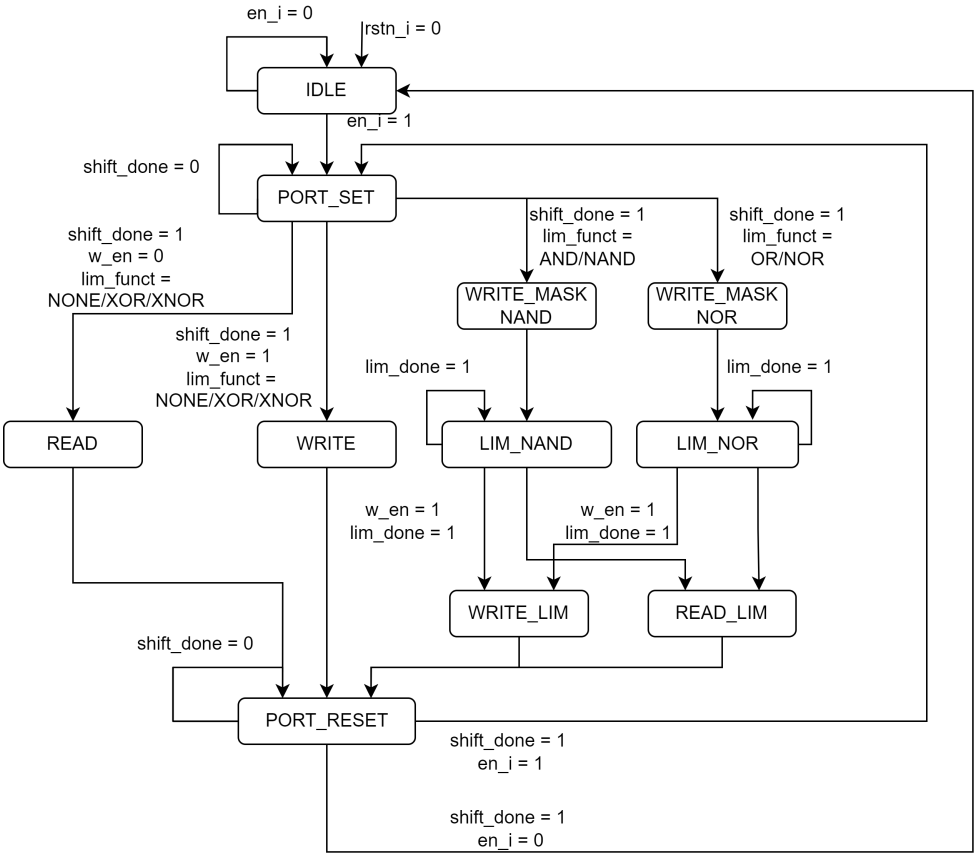

Figure 3.16. FSM algortihm

3.6 Integration in RISC-V memory model

The Racetrack memory array as it is, is not able to work directly with the pre-existing dual port memory model. First of all, this model will be used only as data memory, this requires a special routine in the top module of the testbench which performs the writing of the .hex file of the programs. Unfortunately the memory adopted by the RISC-V does not have a clear separation between instructions and data, thus whole content of the .hex file should be written in the Racetrack.
This routine is Racetrack-specific, only in the case in which a Racetrack memory is adopted, the memory initialization routine is performed, in all the other cases a simple "readmemh" statement is adopted.
From the architectural point of view, additional efforts are required to integrate the new Racetrack memory model inside the RISC-V dual-port memory model, Figure 3.17 shows the resulting high-level architecture. In the following a brief description of the most interesting building blocks will be given.

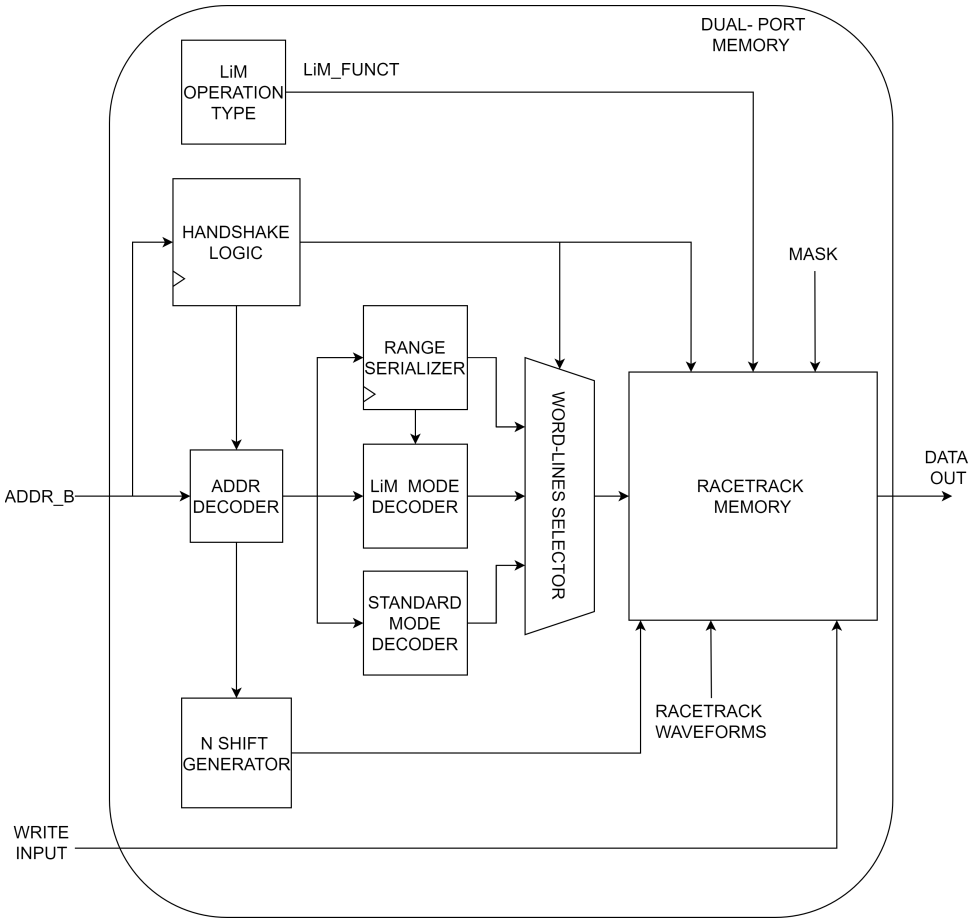

Figure 3.17. Dual-port memory architecture with Racetrack memory integration

3.6.1 Handshake logic

The handshake logic is responsible of keeping stable all the signals required during a memory access, it is basically a register triggered by the memory request signal or by the memory valid signal. This module is very important because the Racetrack memory includes additional latency during accesses, thus it is important to keep stable all the signals coming from the RISC-V core.

3.6.2 Address and mode decoders

Depending on the access type, addresses are decoded in different ways. The most straightforward situation is a single access adopting the *LiM mode*, in this case the word-lines are activated decoding directly the incoming address. The word-lines generation is quite more complex during a range operation, in this case addresses are serialized by the Range serializer (explained in the next Subsection). Based on the range_active signal, which is high during range operations, a multiplexer selects between one of the two decoders.
The picture is completely different adopting the *Standard mode*, here each row within the Racetrack stores three different words, this means that three addresses are mapped in the same Racetrack row. This implies that addresses are divided by a factor 3. At this point, an intermediate word-lines generation is applied using the decoded address and then word-lines are grouped three by three with an OR gate. The new generated word-lines will be called here in after *triplets*, Figure 3.18 shows the *triplets* generation.
During this memory access, three words are addressed, to understand which of the three words is the selected one, it is possible to see which is the active intermediate word-line. Each ot the three intermediate word-lines are used as control signals for the word selection within the same Racetrack row.

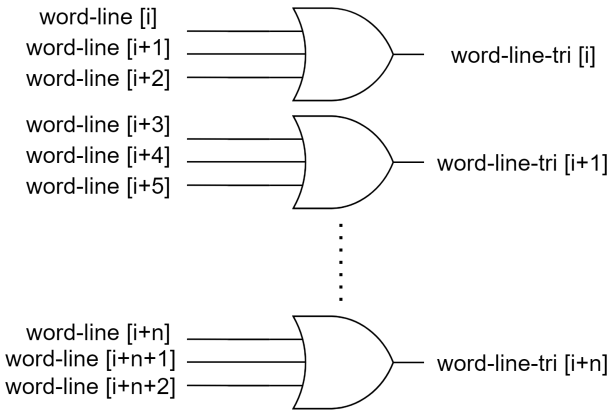

Figure 3.18. Memory mode word-lines generation

3.6.3 Range serializer

This component operates only during LiM operations, as explained in the previous chapter, the actual LiM architecture is able to perform range operations on the memory array. This feature should be preserved also with the Racetrack memory implementation, but since only single accesses are possible, a parallel range access on N adjacent elements is serialized in N distinct accesses. This block, starting from the initial address, generates all the following address required for the range operation. A load signal is adopted yo inject the starting address of the range operation, then every

time the Racetrack completes an access, the valid signal is used as an enable signal to increase the address value. A comparator checks the actual address with the final range address, then it generates a done signal to stop the range routine, Figure 3.19 depicts the high-level schematic of the range serialized.

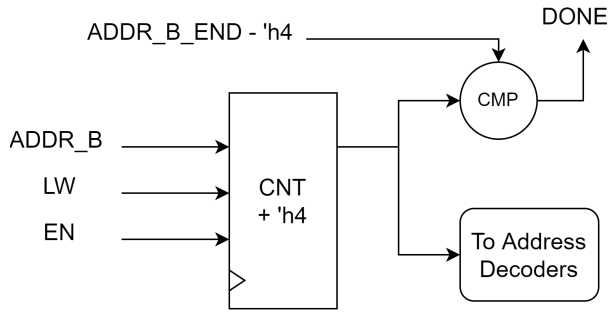

Figure 3.19. Range serializer high level structure

3.6.4 Shift number generator

Thanks to the port configuration the maximum number of required shift for port alignment is three. The reason can be understood analyzing Figures present in Subsection 3.4.3. It is possible to see that each port is assigned to 4 cells, if it is a lucky situation, the required data is already aligned with the access port, otherwise a number of shift between 1 and 3 is required. Furthermore, this particular configuration simplify the computation of the required number of shifts. Analyzing the binary representation of Addresses, it is possible to observe that the last two LSBs give directly the required information, this avoids any additional logic, in fact these two lines are directly forwarded in the Shifter.

3.6.5 Implementation considerations

This is the most straightforward Racetrack memory architecture, it is capable of a single word access and every parallel store operation is serialized by means of the additional Range serializer block. From a performance point of view this slows down range operations adding a not negligible latency, in fact a parallel access on N memory locations is split in N separated memory accesses. Another problem is the working frequency of the Racetrack, the first idea was to set the Racetrack's control FSM at the same frequency of the clock system (100MHz). This implies that standard accesses or LiM accesses are performed with a number of clock cycles always higher than an access performed with the original memory.

3.7 Parallel implementation

The remarkable limitation shown in the previous Section has led an additional step in the architectural design. Word organization inside each block has been completely re-designed.
In particular, the idea is to exploit the high number of *blocks* to perform parallel accesses during logic range store operation. In the initial implementation words were stored one after the other, thus block 0 contained word 0 to word 31. In this new implementation words are displaced all over the available blocks, e.g block 0 takes word 0 at line 0, block 1 takes word 1 at line 1 and so

on, all the other memory locations are filled in the same way, Figure 3.20 gives an idea of the new organization.

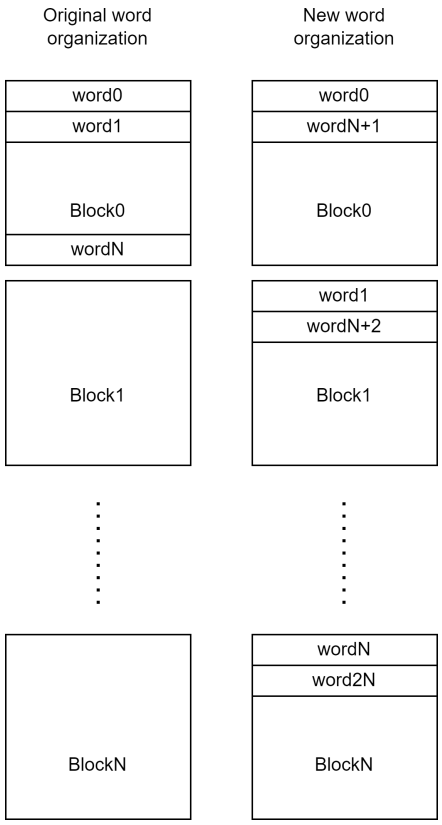

Figure 3.20. Old and new memory organization

Since Racetrack commands arrive at every single blocks, these are exploited to perform a parallel memory access on B (number of available blocks) words in parallel. Considering the selected size for the Racetrack memory, the available blocks are 2000, so as the number of words involved in a parallel logic store access.
This new organization required small design modifications, the range serializer was completely removed and replaced with the range decoder explained deeply in [5]. This special decoder activates multiple word-lines whenever a LiM range operation is required. Word-lines for *LiM mode* and *Memory mode* are generated in parallel, MEM_MODE parameter forwards the correct word-lines to the routing block which has the only aim of routing word-lines to the correct *Blocks* with the logic explained before.
A detailed analysis of Simulation results of this implementation is reported in the next Chapter.

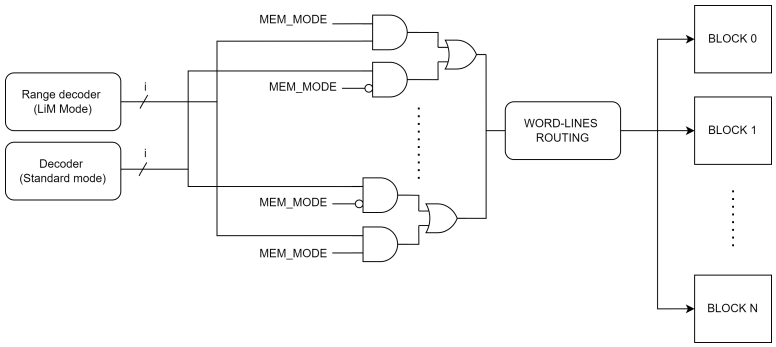

Figure 3.21. Word-line generation with parallel implementation

3.8 Core compliant implementation

In [5], Synthesis highlighted that the minimum clock period for the RISC-V Core is $\simeq 37/38ns$ for the three different implementations. Considering a $40ns$ clock period for the RISC-V Core (compliant with the minimum period highlighted by the synthesis) and still considering a $100MHz$ frequency for the Racetrack, it is possible to perform accesses in one or at most in two system clock cycles, depending on the operation. A standard access or a XOR/XNOR LiM access takes 1 system clock period to be completed, while all the others takes 2 system clock cycles, an access latency almost equal to the pre-existing memory. The new waveforms format is shown in Figure 3.22.
The final design is based on this implementations and embeds the parallel feature as well as being compliant with the true working frequency of the RISC-V Core. This new implementation required small modifications to the handshaking protocol.
A detailed analysis of Simulation results of this implementation is reported in the next Chapter.

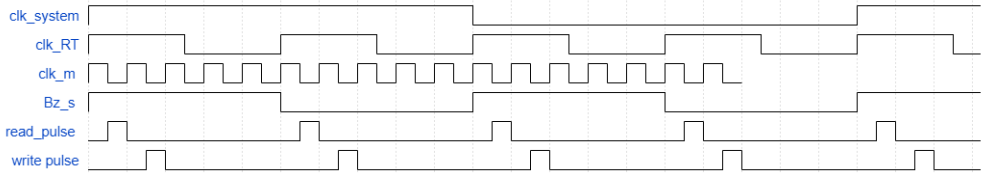

Figure 3.22. New waveform format

Chapter 4

Simulations

4.1 Tools

The configurable LiM Framework which supports three different types of memory, as well as the RISC-V core, were designed using SystemVerilog. Simulations were performed adopting Modelsim Questa Sim 2020.4 and Synopsys VCS 2021.09.
All the following simulations refer to the final Racetrack version , simulation results of the three different implementations will be analyzed in Section 4.4.

4.2 Simulation of custom programs

The aim of this Thesis is to design a configurable LiM Framework able to instantiate different types of memory which mimic the adoption of two different types of technology. Using different compiler directives, it is possible to implement a standard memory with or without LiM functionalities and a Racetrack memory with or without LiM functionalities. As a matter of fact, the first two models are much ideal, in fact the memory core is modeled as an array of registers in which data is read or written in just one clock cycle. The Racetrack memory model have an higher latency in some situations with respect of a standard memory model, thus the execution time in terms of spent clock cycles will be probably worse for some algorithms.
In the work presented in [5], LiM operations were inserted by hand directly in the .hex file, thanks to the Compiler modifications this is no more required. As explained in Chapter 2, RISC-V GNU-GCC Compiler was modified to support LiM instructions (i.e. store_active_logic and load_mask). To test both old bitwise operations and new inverting ones, two custom programs were designed. Note that the Racetrack memory implementation does not support the max/min logic, thus only tests supported by all the memories were performed.
The first program, *bitwise.c*, is basically an already available program designed in the previous work, in which in-line LiM assembly instructions have been inserted. The second one, *bitwise_inv.c* has the same structure but implements inverting operations instead of the original ones, this program is useful to test new LiM functionalities presented in Section 2.4.

4.2.1 Bitwise

In the following, Listing 4.1 shows the original *bitwise.c* code. In this example a vector of N elements and a stand-alone variable are defined, then several logic operations are applied. Vector computations are carried out with for loops, this increases dramatically the size of the code, because for loops are un-rolled by the Compiler and the inner code is replicated N times.

Listing 4.1. *bitwise.c* code

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    /* variable declaration */
    int N = 5, i, mask_or, mask_and, mask_xor;
    int *vector = 0x030000, *stand_alone = 0x30040, *final_result = 0x30044;

    /* fill vector */
    for(i=0; i<N; i++){
      vector[i] = i*13467;
    }
    *stand_alone = vector[1]+0x768;

    /* 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);

    return EXIT_SUCCESS;
}
```

Thanks to the Compiler modifications it is possible to define in-line assembly portions of code, the new LiM-bitwise program is reported in Listing 4.2. The Compiler recognizes these in-line pieces of code and replaces them with the corresponding assembly instruction.

Listing 4.2. *bitwise.c* code with new compiler

```
/*Bitwise custom program*/
//Compute different bitwise logic operations on a user-defined vector

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

    int mask_or, mask_and, mask_xor, N = 5, i, sum_a = 0xFFFFFFFF, sum_b = 0
    xFFFFFFFF;
    volatile int (*vector)[N];
```

```
12      volatile int (*stand_alone);
13      volatile int (*final_result);
14
15      register unsigned int x0 asm("x0");
16
17      //define variables' addresses
18      vector = (volatile int(*)[N]) 0x030000,
19      stand_alone = (volatile int(*))0x30040,
20      final_result= (volatile int(*))0x30044;
21
22      //configuration address, where the config of the memory is stored.
23      int cnfAddress = 0x1fffc;
24      //configure vector[N-1] address
25      int andAddress = 0x030010;
26      //configure vector[N-2] address
27      int xorAddress  = 0x03000C;
28      //configure vector[N-3] address
29      int opAddress  = 0x30008;
30
31
32      //initialize mask values
33      mask_and  = 0x8F;
34      mask_or   = 0xF1;
35      mask_xor  = 0xF0;
36
37
38      /* fill vector */
39      for(i=0; i<N; i++){
40        (*vector)[i] = i*13467;
41      }
42
43      (*stand_alone) = (*vector)[1]+0x768;
44
45
46      /* OR operation */
47
48      //program LiM for range operation
49      asm volatile("sw_active_or %[result], %[input_i], 0"
50      : [result] "=r" (N)
51      : [input_i] "r" (cnfAddress), "[result]" (N)
52      );
53
54      //sw operation to active OR LiM
55      (*vector)[0] = mask_or;
56
57      //program LiM for stand-alone operation
58      asm volatile("sw_active_or %[result], %[input_i], 0"
59      : [result] "=r" (x0)
60      : [input_i] "r" (cnfAddress), "[result]" (x0)
61      );
62    (*stand_alone) = mask_or;
63
64      /* AND operation */
65
66      //program LiM for stand-alone operation
67      asm volatile("sw_active_and %[result], %[input_i], 0"
68      : [result] "=r" (x0)
69      : [input_i] "r" (cnfAddress), "[result]" (x0)
70      );
71
```

```
72
73      //lw_mask operation for mask_and computation
74      asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
75      : [result] "=r" (mask_and)
76      : [input_s] "r" (andAddress), [input_t] "r"  (mask_and), "[result]" (
        mask_and)
77      );
78
79      //program LiM for range operation
80      asm volatile("sw_active_and %[result], %[input_i], 0"
81      : [result] "=r" (N)
82      : [input_i] "r" (cnfAddress), "[result]" (N)
83      );
84
85      //sw operation to active AND LiM
86      (*vector)[0] = mask_and;
87
88      //program LiM for stand-alone operation
89      asm volatile("sw_active_and %[result], %[input_i], 0"
90      : [result] "=r" (x0)
91      : [input_i] "r" (cnfAddress), "[result]" (x0)
92      );
93
94    (*stand_alone) = mask_and;
95
96
97      /* XOR operation*/
98
99      //program LiM for stand-alone operation
100     asm volatile("sw_active_xor %[result], %[input_i], 0"
101     : [result] "=r" (x0)
102     : [input_i] "r" (cnfAddress), "[result]" (x0)
103     );
104
105
106     //lw_mask operation for mask_xor computation
107     asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
108     : [result] "=r" (mask_xor)
109     : [input_s] "r" (xorAddress), [input_t] "r"  (mask_xor), "[result]" (
        mask_xor)
110     );
111
112
113     //program LiM for range operation
114     asm volatile("sw_active_xor %[result], %[input_i], 0"
115     : [result] "=r" (N)
116     : [input_i] "r" (cnfAddress), "[result]" (N)
117     );
118
119     (*vector)[0] = mask_xor;
120
121     //program LiM for stand-alone operation
122     asm volatile("sw_active_xor %[result], %[input_i], 0"
123     : [result] "=r" (x0)
124     : [input_i] "r" (cnfAddress), "[result]" (x0)
125     );
126
127     (*stand_alone) = mask_xor;
128
129     //lw_mask operation for ~(*vector)[N-3] computation exploting xor
```

```
130     asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
131     : [result] "=r" (sum_a)
132     : [input_s] "r" (opAddress), [input_t] "r"  (sum_a), "[result]" (sum_a)
133     );
134
135     //lw_mask operation for ~(*stand_alone) computation exploting xor
136     asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
137     : [result] "=r" (sum_b)
138     : [input_s] "r" (&(*stand_alone)), [input_t] "r"  (sum_b), "[result]" (
        sum_b)
139     );
140
141
142     //restore standard operations
143     asm volatile("sw_active_none %[result], %[input_i], 0"
144     : [result] "=r" (x0)
145     : [input_i] "r" (cnfAddress), "[result]" (x0)
146     );
147
148     (*final_result) = sum_a + sum_b;
149
150
151     return EXIT_SUCCESS;
152 }
```

Even if the new code appears longer (in terms of code lines), the resulting code is actually smaller since for loops are avoided because the Compiler replaces each in-line assembly code with a single LiM range instruction (store_active_logic). In this way, code replications applied in for loops are avoided. Furthermore, logic operations are carried out directly in memory, this avoids a lot of computations inside the Core. Once a logic operation needs to be performed in memory, this should be programmed with an appropriate store_active_logic. Then, it is possible to perform a logic store simply assigning to the target element the mask value, memory will perform a logic operations on elements specified by the operation size.
It is also possible to perform a logic load, where a data is loaded from memory and at the same time a logic operation specified by the LiM programming is applied. In this example, lw_mask instruction is exploited to compute new mask values and the inverted operands necessary for the computation of the final result.
To estimate the execution performance, it is possible to take into account the Execution Time in terms of number of clock cycles (cc) and assuming N as the size of the vector. For this estimation, only meaningful parts of the code have been taken into account, initialization of the vector and mask computations have not included. The following equations show the Execution time considering the standard memory and the Racetrack memory both in normal and LiM configurations.

$$Execution\_time_{std\_mem} \approx 3[3N(vector\_element) + 3(single\_element)] + 6(final\_result) \tag{4.1}$$

$$\begin{aligned} Execution\_time_{std\_LiM\_mem} \approx 3[1(mem\_active) + 1(vector\_element) + 1(mem\_active) \\ +1(single\_element)] + 4(final\_result) + 1(mem\_active) \end{aligned} \tag{4.2}$$

$$Execution\_time_{std\_mem} \approx 3[3N(vector\_element) + 3(single\_element)] + 6(final\_result) \tag{4.3}$$

$$Execution\_time_{RT\_LiM\_mem} \approx 2[1(mem\_active) + 2(vector\_element) + 1(mem\_active) + 2(single\_element)] + [(1(mem\_active) + vector\_element) + 1(mem\_active) + 1(single\_element)] + 4(final\_result) + 1(mem\_active) \quad (4.4)$$

Equation 4.1, shows the Execution time in the case in which a standard memory is adopted, here any kind of LiM operation can be carried out. The first part of the equation refers to the bitwise computations, in fact the factor 3 refers to the three different logic operations. For operations involving vector elements or the stand-alone element, 3 clock cycles are required, because data is loaded in the Core, it is manipulated and then finally it is stored-back in memory. The computation of the final result took six clock cycles because two different data need to be loaded in the Core, then they are inverted, added together and in the last clock cycle the result is stored in memory. This equation has a linear dependency on N.
In the case of adopting a standard memory with the LiM functionalities, the execution time can be expressed as Equation 4.2. Here, logic store operations can be carried out in parallel, in fact all the vector operations are carried out in just two cycles, one for setting-up the memory and another one for computing and storing the results. Final result requires one clock cycle for setting-up the memory and four additional clock cycles. Two for loading data in the Core and at the same time performing the NOT operation, one for adding them together and the last one for storing data in memory. Here, there is not a dependency from parameter N because thanks to range logic operations, it is possible to perform parallel in-memory computations.
For the Racetrack memory case the estimation is quite different. Thanks to the FSM algorithm shown in the previous Chapter it is possible to count the amount of clock cycles for standard memory access and LiM memory accesses:

- Standard type: 1 system clock cycles are required;
- LiM type: 2 clock cycles are required;

If the Racetrack is exploited as a standard memory, the performance is exactly equal to the starting point, a normal memory access requires one clock cycle and the Execution time equation reported in 4.3 is exactly equal to the standard memory one.
Equation 4.4 describes Execution Time using the Racetrack with LiM funxtionalities, the first part with the factor 2 is the computation of OR and AND operations which use built-in logic computation (NAND/NOR). The XOR part has the same latency of a standard access, because the logic computation is carried out by means of external logic, as soon as data is available outside the Racetrack memory array. The final part is the computation of the final result, sum operands are loaded from memory and at the same time a XOR operation is performed. Once operands are loaded in the Core, they are added and then stored back. The XOR operations is useful to invert operands directly in memory, for this operation a mask with all ones was adopted.
Figure 4.1 depicts the behaviour of the previous formulas adopting a different values of N. For the standard LiM memory case and the LiM Racetrack memory case, the behaviour is constant because regardless the size of the vector, all the logic operations are carried out in parallel. For the other cases, the curves have a linear behaviour. The Racetrack case has an higher offset due to the large number of clock cycles required for the built-in NAND/NOR computations.
Comparing the standard memory implementation and the Racetrack one, it is possible to observe a similar behaviour in terms of performance. Without adopting any LiM functionality the Execution time is exactly the same in both memories, while for the LiM case the Racetrack has a small performance degradation ($+ \simeq 23{,}5\%$s).

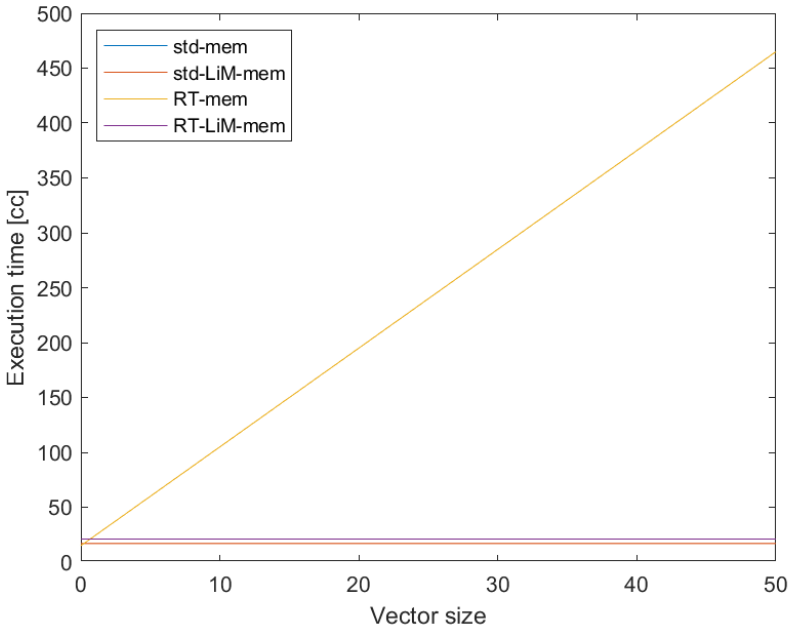

Figure 4.1. Execution time estimation for *bitwise.c* with different vector size N

All the simulations have been carried out with Synopsys VCS, this tool produces a log file with all the executed instructions. This is very useful to understand the differences of the execution in all the cases, in the following some extracts will be presented, only meaningful parts will be shown and loops will be highlighted with blank spaces.
With a not-LiM configuration, the system performs several for loops to compute all the bitwise results, an example is given in log 4.3. Adopting a LiM configuration with still a standard memory, all loops are converted in single range operation, this can be seen in log 4.4, here it is also possible to see the insertion of sw_active and lw_mask instructions. Results for Racetrack memory implementations are not shown since they are exactly equal to the ones presented here, the only difference is the Execution time which is much larger due to the initial memory initialization and due to the additional memory latency.

Listing 4.3. Extract of simulation log of *bitwise.c* - standard configuration

```
 Time      Cycles PC    Instr    Mnemonic
1896ns     186 00000244 04e7a023 sw       x14, 64(x15)        x14:00003c03 x15:00030000  PA:00030040
1906ns     187 00000248 0007a023 sw       x0, 0(x15)          x15:00030000  PA:00030000
1916ns     188 0000024c 00d7a823 sw       x13, 16(x15)        x13:0000d26c x15:00030000  PA:00030010
1926ns     189 00000250 00030737 lui      x14, 0x30000        x14=00030000
1936ns     190 00000254 01478613 addi     x12, x15, 20        x12=00030014 x15:00030000
1946ns     191 00000258 0007a683 lw       x13, 0(x15)         x13=00000000 x15:00030000  PA:00030000
1956ns     192 0000025c 00478793 addi     x15, x15, 4         x15=00030004 x15:00030000
1966ns     193 00000260 0f16e693 ori      x13, x13, 241       x13=000000f1 x13:00000000
1976ns     194 00000264 fed7ae23 sw       x13, -4(x15)        x13:000000f1 x15:00030004  PA:00030000
1986ns     195 00000268 fec798e3 bne      x15, x12, -16       x15:00030004 x12:00030014

2266ns     223 00000268 fec798e3 bne      x15, x12, -16       x15:00030014 x12:00030014
2276ns     224 0000026c 04072783 lw       x15, 64(x14)        x15=00003c03 x14:00030000  PA:00030040
2286ns     225 00000270 000306b7 lui      x13, 0x30000        x13=00030000
2296ns     226 00000274 0f17e793 ori      x15, x15, 241       x15=00003cf3 x15:00003c03
2306ns     227 00000278 04f72023 sw       x15, 64(x14)        x15:00003cf3 x14:00030000  PA:00030040
2316ns     228 0000027c 01072703 lw       x14, 16(x14)        x14=0000d2fd x14:00030000  PA:00030010
2326ns     229 00000280 000307b7 lui      x15, 0x30000        x15=00030000
2336ns     230 00000284 01478593 addi     x11, x15, 20        x11=00030014 x15:00030000
2346ns     231 00000288 08f77713 andi     x14, x14, 143       x14=0000008d x14:0000d2fd
2356ns     232 0000028c 0007a603 lw       x12, 0(x15)         x12=000000f1 x15:00030000  PA:00030000
2366ns     233 00000290 00478793 addi     x15, x15, 4         x15=00030004 x15:00030000
2376ns     234 00000294 00e67633 and      x12, x12, x14       x12=00000081 x12:000000f1 x14:0000008d
2386ns     235 00000298 fec7ae23 sw       x12, -4(x15)        x12:00000081 x15:00030004  PA:00030000
2396ns     236 0000029c feb798e3 bne      x15, x11, -16       x15:00030004 x11:00030014
```

```
2666ns     263 00000298 fec7ae23 sw       x12, -4(x15)        x12:0000008d x15:00030014  PA:00030010
2676ns     264 0000029c feb798e3 bne      x15, x11, -16       x15:00030014 x11:00030014
2686ns     265 000002a0 0406a783 lw       x15, 64(x13)        x15=00003cf3 x13:00030000  PA:00030040
2706ns     267 000002a4 00e7f7b3 and      x15, x15, x14       x15=00000081 x15:00003cf3 x14:0000008d
2716ns     268 000002a8 04f6a023 sw       x15, 64(x13)        x15:00000081 x13:00030000  PA:00030040
2726ns     269 000002ac 00c6a783 lw       x15, 12(x13)        x15=00000081 x13:00030000  PA:0003000c
2736ns     270 000002b0 00030737 lui      x14, 0x30000        x14=00030000
2746ns     271 000002b4 000306b7 lui      x13, 0x30000        x13=00030000
2756ns     272 000002b8 0f07c793 xori     x15, x15, 240       x15=00000071 x15:00000081
2766ns     273 000002bc 01470593 addi     x11, x14, 20        x11=00030014 x14:00030000
2776ns     274 000002c0 00072603 lw       x12, 0(x14)         x12=00000081 x14:00030000  PA:00030000
2786ns     275 000002c4 00470713 addi     x14, x14, 4         x14=00030004 x14:00030000
2796ns     276 000002c8 00f64633 xor      x12, x12, x15       x12=000000f0 x12:00000081 x15:00000071
2806ns     277 000002cc fec72e23 sw       x12, -4(x14)        x12:000000f0 x14:00030004  PA:00030000
2816ns     278 000002d0 feb718e3 bne      x14, x11, -16       x14:00030004 x11:00030014

3096ns     306 000002d0 feb718e3 bne      x14, x11, -16       x14:00030014 x11:00030014
3106ns     307 000002d4 0406a703 lw       x14, 64(x13)        x14=00000081 x13:00030000  PA:00030040
3116ns     308 000002d8 00000513 addi     x10, x0, 0          x10=00000000
3126ns     309 000002dc 00e7c7b3 xor      x15, x15, x14       x15=000000f0 x15:00000071 x14:00000081
3136ns     310 000002e0 0086a703 lw       x14, 8(x13)         x14=000000f4 x13:00030000  PA:00030008
3146ns     311 000002e4 04f6a023 sw       x15, 64(x13)        x15:000000f0 x13:00030000  PA:00030040
3156ns     312 000002e8 fff7c793 xori     x15, x15, -1        x15=ffffff0f x15:000000f0
3166ns     313 000002ec fff74713 xori     x14, x14, -1        x14=ffffff0b x14:000000f4
3176ns     314 000002f0 00f707b3 add      x15, x14, x15       x15=fffffe1a x14:ffffff0b x15:ffffff0f
3186ns     315 000002f4 08f6a023 sw       x15, 128(x13)       x15:fffffe1a x13:00030000  PA:00030080
3196ns     316 000002f8 00008067 jalr     x0, x1, 0            x1:000001d8
```

Listing 4.4. Extract of simulation log of *bitwise.c* - LiM configuration with new compiler

```
Time          Cycles PC       Instr  Mnemonic
173821ns      4341 00000244 00e7a823 sw              x14, 16(x15)        x14:0000d26c x15:00030000  PA:00030010
173861ns      4342 00000248 0047a703 lw              x14, 4(x15)         x14=0000349b x15:00030000  PA:00030004
173901ns      4343 0000024c 04078593 addi            x11, x15, 64        x11=00030040 x15:00030000
173941ns      4344 00000250 00500693 addi            x13, x0, 5          x13=00000005
173981ns      4345 00000254 76870713 addi            x14, x14, 1896      x14=00003c03 x14:0000349b
174021ns      4346 00000258 04e7a023 sw              x14, 64(x15)        x14:00003c03 x15:00030000  PA:00030040
174061ns      4347 0000025c 00020737 lui             x14, 0x20000        x14=00020000
174101ns      4348 00000260 ffc70713 addi            x14, x14, -4        x14=0001fffc x14:00020000
174141ns      4349 00000264 000736bb sw_active_or    Nx13 0(x14)         x14:0001fffc x13:00000005  PA:0001fffc
174181ns      4350 00000268 0f100613 addi            x12, x0, 241        x12=000000f1
174221ns      4351 0000026c 00c7a023 sw              x12, 0(x15)         x12:000000f1 x15:00030000  PA:00030000
174261ns      4352 00000270 0007303b sw_active_or    Nx0 0(x14)          x14:0001fffc  PA:0001fffc
174341ns      4354 00000274 04c7a023 sw              x12, 64(x15)        x12:000000f1 x15:00030000  PA:00030040
174381ns      4355 00000278 0007203b sw_active_and   Nx0 0(x14)          x14:0001fffc  PA:0001fffc
174461ns      4357 0000027c 08f00613 addi            x12, x0, 143        x12=0000008f
174501ns      4358 00000280 01078513 addi            x10, x15, 16        x10=00030010 x15:00030000
174541ns      4359 00000284 00c5261b lw_mask         x12, x12, 0(x10)    x12=0000008d x12:0000008f x10:00030010  PA:00030010
174581ns      4360 00000288 000726bb sw_active_and   Nx13 0(x14)         x14:0001fffc x13:00000005  PA:0001fffc
174661ns      4362 0000028c 00c7a023 sw              x12, 0(x15)         x12:0000008d x15:00030000  PA:00030000
174701ns      4363 00000290 0007203b sw_active_and   Nx0 0(x14)          x14:0001fffc  PA:0001fffc
174781ns      4365 00000294 04c7a023 sw              x12, 64(x15)        x12:0000008d x15:00030000  PA:00030040
174821ns      4366 00000298 0007103b sw_active_xor   Nx0 0(x14)          x14:0001fffc  PA:0001fffc
174901ns      4368 0000029c 0f000613 addi            x12, x0, 240        x12=000000f0
174941ns      4369 000002a0 00c78513 addi            x10, x15, 12        x10=0003000c x15:00030000
174981ns      4370 000002a4 00c5261b lw_mask         x12, x12, 0(x10)    x12=00000071 x12:000000f0 x10:0003000c  PA:0003000c
175021ns      4371 000002a8 000716bb sw_active_xor   Nx13 0(x14)         x14:0001fffc x13:00000005  PA:0001fffc
175061ns      4372 000002ac 00c7a023 sw              x12, 0(x15)         x12:00000071 x15:00030000  PA:00030000
175101ns      4373 000002b0 0007103b sw_active_xor   Nx0 0(x14)          x14:0001fffc  PA:0001fffc
175141ns      4374 000002b4 fff00693 addi            x13, x0, -1         x13=ffffffff
175181ns      4375 000002b8 04c7a023 sw              x12, 64(x15)        x12:00000071 x15:00030000  PA:00030040
175221ns      4376 000002bc 00878513 addi            x10, x15, 8         x10=00030008 x15:00030000
175261ns      4377 000002c0 00068613 addi            x12, x13, 0         x12=ffffffff x13:ffffffff
175301ns      4378 000002c4 00d5261b lw_mask         x12, x13, 0(x10)    x12=ffffff0b x13:ffffffff x10:00030008  PA:00030008
175341ns      4379 000002c8 00d5a69b lw_mask         x13, x13, 0(x11)    x13=ffffff0f x13:ffffffff x11:00030040  PA:00030040
175381ns      4380 000002cc 0007003b sw_active_none  Nx0 0(x14)          x14:0001fffc  PA:0001fffc
175421ns      4381 000002d0 00d60633 add             x12, x12, x13       x12=fffffe1a x12:ffffff0b x13:ffffff0f
175461ns      4382 000002d4 04c7a223 sw              x12, 68(x15)        x12:fffffe1a x15:00030000  PA:00030044
175501ns      4383 000002d8 00000513 addi            x10, x0, 0          x10=00000000
175541ns      4384 000002dc 00008067 jalr            x0, x1, 0            x1:000001d8
```

4.2.2 Inverted-bitwise

Example code *bitwise_inv.c*, reported in Listing 4.5, has basically the same structure as *bitwise.c*, but instead of the original logic operations, these are replaced with inverting ones (NAND, NOR, XNOR). This test is interesting because it is possible to analyze also the new inverting logic operations, results are coherent with the expected ones. The modified C code with in-line assembly instructions is reported in Listing 4.6

Listing 4.5. *bitwise_inv.c* code

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char *argv[])
```

```
{
    /* variable declaration */
    int N = 5, i, mask_or, mask_and, mask_xor;
    int *vector = 0x030000, *stand_alone = 0x30040, *final_result = 0x30080;

    /* fill vector */
    for(i=0; i<N; i++){
      vector[i] = i*13467;
    }
    *stand_alone = vector[1]+0x768;

    /* 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);

    return EXIT_SUCCESS;
}
```

Listing 4.6. *bitwise_inv.c* code with new compiler

```
/*Bitwise-inv custom program*/
//Compute different inverting bitwise logic operations on a user-defined vector

#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{

    int mask_nor, mask_nand, mask_xnor, N = 5, i, sum_a = 0x0, sum_b = 0x0;
    volatile int (*vector)[N];
    volatile int (*stand_alone);
    volatile int (*final_result);

    //define variables' addresses
    vector = (volatile int(*)[N]) 0x030000,
    stand_alone = (volatile int(*))0x30040,
    final_result= (volatile int(*))0x30044;

    register unsigned int x0 asm("x0");

```

```
22      //configuration address, where the config of the memory is stored.
23      int cnfAddress = 0x1fffc;
24    //configure vector[N-1] address
25    int andAddress = 0x030010;
26    //configure vector[N-2] address
27    int xorAddress  = 0x03000C;
28      //configure vector[N-3] address
29    int opAddress  = 0x30008;
30
31      //initialize mask values
32      mask_nand  = 0x8F;
33      mask_nor   = 0xF1;
34      mask_xnor  = 0xF0;
35
36
37
38      /* fill vector */
39      for(i=0; i<N; i++){
40        (*vector)[i] = i*13467;
41      }
42
43      (*stand_alone) = (*vector)[1]+0x768;
44
45
46
47
48
49      /* NOR operation */
50
51      //program LiM for range operation
52      asm volatile("sw_active_nor %[result], %[input_i], 0"
53      : [result] "=r" (N)
54      : [input_i] "r" (cnfAddress), "[result]" (N)
55      );
56
57      //sw operation to active NOR LiM
58      (*vector)[0] = mask_nor;
59
60      //program LiM for stand-alone operation
61      asm volatile("sw_active_nor %[result], %[input_i], 0"
62      : [result] "=r" (x0)
63      : [input_i] "r" (cnfAddress), "[result]" (x0)
64      );
65
66      (*stand_alone) = mask_nor;
67
68      /* NAND operation */
69
70
71      //program LiM for stand-alone operation
72      asm volatile("sw_active_nand %[result], %[input_i], 0"
73      : [result] "=r" (x0)
74      : [input_i] "r" (cnfAddress), "[result]" (x0)
75      );
76
77
78      //lw_mask operation for mask_nand computation
79      asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
80      : [result] "=r" (mask_nand)
```

```
81      : [input_s] "r" (andAddress), [input_t] "r"  (mask_nand), "[result]" (
        mask_nand)
82      );
83
84      //program LiM for range operation
85      asm volatile("sw_active_nand %[result], %[input_i], 0"
86      : [result] "=r" (N)
87      : [input_i] "r" (cnfAddress), "[result]" (N)
88      );
89
90      //sw operation to active NAND LiM
91      (*vector)[0] = mask_nand;
92
93      //program LiM for stand-alone operation
94      asm volatile("sw_active_nand %[result], %[input_i], 0"
95      : [result] "=r" (x0)
96      : [input_i] "r" (cnfAddress), "[result]" (x0)
97      );
98
99      (*stand_alone) = mask_nand;
100
101
102     /* XNOR operation*/
103
104     //program LiM for stand-alone operation
105     asm volatile("sw_active_xnor %[result], %[input_i], 0"
106     : [result] "=r" (x0)
107     : [input_i] "r" (cnfAddress), "[result]" (x0)
108     );
109
110
111     //lw_mask operation for mask_xnor computation
112     asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
113     : [result] "=r" (mask_xnor)
114     : [input_s] "r" (xorAddress), [input_t] "r"  (mask_xnor), "[result]" (
        mask_xnor)
115     );
116
117
118     //program LiM for range operation
119     asm volatile("sw_active_xnor %[result], %[input_i], 0"
120     : [result] "=r" (N)
121     : [input_i] "r" (cnfAddress), "[result]" (N)
122     );
123
124     (*vector)[0] = mask_xnor;
125
126     //program LiM for stand-alone operation
127     asm volatile("sw_active_xnor %[result], %[input_i], 0"
128     : [result] "=r" (x0)
129     : [input_i] "r" (cnfAddress), "[result]" (x0)
130     );
131
132     (*stand_alone) = mask_xnor;
133
134
135     //lw_mask operation for ~(*vector)[N-3] computation exploting xnor
136     asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
137     : [result] "=r" (sum_a)
138     : [input_s] "r" (opAddress), [input_t] "r"  (sum_a), "[result]" (sum_a)
```

```
139     );
140
141     //lw_mask operation for ~(*stand_alone) computation exploting xnor
142     asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
143     : [result] "=r" (sum_b)
144     : [input_s] "r" (&(*stand_alone)), [input_t] "r"  (sum_b), "[result]" (
        sum_b)
145     );
146
147
148     //restore standard operations
149     asm volatile("sw_active_none %[result], %[input_i], 0"
150     : [result] "=r" (x0)
151     : [input_i] "r" (cnfAddress), "[result]" (x0)
152     );
153
154     (*final_result) = sum_a + sum_b;
155
156
157
158     return EXIT_SUCCESS;
159 }
```

In the following, Equations for Execution Time estimation are reported, as in the previous code, variable N is assumed as vector size.

$$Execution\_time_{std\_mem} \approx 3[3N(vector\_element) + 3(single\_element)] + 6(final\_result) \tag{4.5}$$

$$\begin{aligned} Execution\_time_{std\_LiM\_mem} \approx 3[1(mem\_active) + 1(vector\_element) + 1(mem\_active) \\ +1(single\_element)] + 4(final\_result) + 1(mem\_active) \end{aligned} \tag{4.6}$$

$$Execution\_time_{RT\_mem} \approx 3[3N(vector\_element) + 3(single\_element)] + 6(final\_result) \tag{4.7}$$

$$\begin{aligned} Execution\_time_{RT\_LiM\_mem} \approx 2[1(mem\_active) + 2(vector\_element) + 1(mem\_active) \\ +2(single\_element)] + [(vector\_element) + 1(single\_element) \\ +1(mem\_active)] + 4(final\_result) + 1(mem\_active) \end{aligned} \tag{4.8}$$

This program exhibits the same results in terms of Execution time because the program's structure is identical to *bitwise.c* unless for the inverted operations. Figure 4.2, reports the plot of the Execution time for the three different cases. Curves are the same because only inverting operation have been added inside the computations but these do not require additional clock cycles because they are performed directly in memory, thus the same results are obtained.

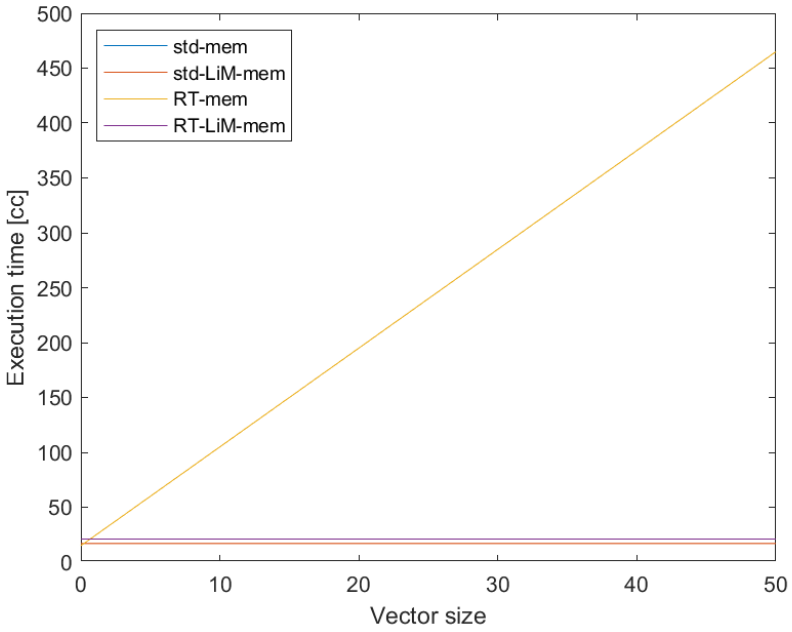

Figure 4.2. Execution time estimation for *bitwise_inv.c* with different vector size N

In log 4.7, the most interesting part of the Simulation Log is reported. Opcodes and *funct* fields of new logic inverting operations are very similar to old ones, unfortunately the Compiler is not capable to recognize new instructions. For this reason the simulation log reports wrong names for logic store instructions (i.e. sw_active_or instead of sw_active_nor), this is an intrinsic limitation of the compiler, in any case results are correct.

Listing 4.7. Extract of simulation log of *bitwise_inv.c* - LiM configuration with new compiler

```
Time         Cycles PC     Instr    Mnemonic
173821ns     4341 00000244 00e7a823 sw              x14, 16(x15)        x14:0000d26c x15:00030000  PA:00030010
173861ns     4342 00000248 0047a703 lw              x14, 4(x15)         x14=0000349b x15:00030000  PA:00030004
173901ns     4343 0000024c 04078593 addi            x11, x15, 64        x11=00030040 x15:00030000
173941ns     4344 00000250 00500693 addi            x13, x0, 5          x13=00000005
173981ns     4345 00000254 76870713 addi            x14, x14, 1896      x14=00003c03 x14:0000349b
174021ns     4346 00000258 04e7a023 sw              x14, 64(x15)        x14:00003c03 x15:00030000  PA:00030040
174061ns     4347 0000025c 00020737 lui             x14, 0x20000        x14=00020000
174101ns     4348 00000260 ffc70713 addi            x14, x14, -4        x14=0001fffc x14:00020000
174141ns     4349 00000264 001736bb sw_active_or    Nx13 1(x14)         x14:0001fffc x13:00000005  PA:0001fffc
174181ns     4350 00000268 0f100613 addi            x12, x0, 241        x12=000000f1
174221ns     4351 0000026c 00c7a023 sw              x12, 0(x15)         x12:000000f1 x15:00030000  PA:00030000
174261ns     4352 00000270 0017303b sw_active_or    Nx0 1(x14)          x14:0001fffc  PA:0001fffc
174341ns     4354 00000274 04c7a023 sw              x12, 64(x15)        x12:000000f1 x15:00030000  PA:00030040
174381ns     4355 00000278 0017203b sw_active_and   Nx0 1(x14)          x14:0001fffc  PA:0001fffc
174461ns     4357 0000027c 08f00613 addi            x12, x0, 143        x12=0000008f
174501ns     4358 00000280 01078513 addi            x10, x15, 16        x10=00030010 x15:00030000
174541ns     4359 00000284 00c5261b lw_mask         x12, x12, 0(x10)    x12=ffffffd x12:0000008f x10:00030010  PA:00030010
174581ns     4360 00000288 001726bb sw_active_and   Nx13 1(x14)         x14:0001fffc x13:00000005  PA:0001fffc
174661ns     4362 0000028c 00c7a023 sw              x12, 0(x15)         x12:ffffffd x15:00030000  PA:00030000
174701ns     4363 00000290 0017203b sw_active_and   Nx0 1(x14)          x14:0001fffc  PA:0001fffc
174781ns     4365 00000294 04c7a023 sw              x12, 64(x15)        x12:ffffffd x15:00030000  PA:00030040
174821ns     4366 00000298 0017103b sw_active_xor   Nx0 1(x14)          x14:0001fffc  PA:0001fffc
174901ns     4368 0000029c 0f000613 addi            x12, x0, 240        x12=000000f0
174941ns     4369 000002a0 00c78513 addi            x10, x15, 12        x10=0003000c x15:00030000
174981ns     4370 000002a4 00c5261b lw_mask         x12, x12, 0(x10)    x12=ffff62fc x12:000000f0 x10:0003000c  PA:0003000c
175021ns     4371 000002a8 001716bb sw_active_xor   Nx13 1(x14)         x14:0001fffc x13:00000005  PA:0001fffc
175061ns     4372 000002ac 00c7a023 sw              x12, 0(x15)         x12:ffff62fc x15:00030000  PA:00030000
175101ns     4373 000002b0 0017103b sw_active_xor   Nx0 1(x14)          x14:0001fffc  PA:0001fffc
175141ns     4374 000002b4 00000693 addi            x13, x0, 0          x13=00000000
175181ns     4375 000002b8 04c7a023 sw              x12, 64(x15)        x12:ffff62fc x15:00030000  PA:00030040
175221ns     4376 000002bc 00878513 addi            x10, x15, 8         x10=00030008 x15:00030000
175261ns     4377 000002c0 00068613 addi            x12, x13, 0         x12=00000000 x13:00000000
175301ns     4378 000002c4 00d5261b lw_mask         x12, x13, 0(x10)    x12=ffff0b0b x13:00000000 x10:00030008  PA:00030008
175341ns     4379 000002c8 00d5a69b lw_mask         x13, x13, 0(x11)    x13=ffff5e0f x13:00000000 x11:00030040  PA:00030040
175381ns     4380 000002cc 0007003b sw_active_none  Nx0 0(x14)          x14:0001fffc  PA:0001fffc
175421ns     4381 000002d0 00d60633 add             x12, x12, x13       x12=fffe691a x12:ffff0b0b x13:ffff5e0f
175461ns     4382 000002d4 04c7a223 sw              x12, 68(x15)        x12:fffe691a x15:00030000  PA:00030044
```

```
175501ns     4383 000002d8 00000513 addi            x10, x0, 0          x10=00000000
175541ns     4384 000002dc 00008067 jalr            x0, x1, 0            x1:000001d8
175621ns     4386 000001d8 0080006f jal             x0, 8
```

4.3 Simulation with standard programs

In [5], a set of real world algorithms, usually referred as standard algorithms, are analyzed. The aim of this Section is to estimate the performances of all the four types of memories and understand the possible improvements brought by the Racetrack implementation. The first two tests are also performed in [5], the last one is completely new.

4.3.1 Database search with Bitmap Indexes algorithm

Bitmap Indexing is a particular type of database indexing which adopts Bitmaps, this technique is used in huge databases and it could speed-up data processing [32].
The idea is to map a specific property with a binary value (1 or 0), whether the property is satisfied or not by the stored data. The original Code, shown in Listing 4.8, uses this algorithm to search some properties on a set of high school students. Gender and range age are considered as properties.
The two queries are:

- Which students are older than 19 and male?
- Which students are older than 18?

The Dataset is composed by 192 students, properties are stored in 6 words of 32-bits because LiM operations are carried out on a full 32-bits word. Listing 4.8, reports the standard version of the program, here as in a standard architecture, logic operations are carried out completely by the Core.

Listing 4.8. *bitmap_search.c* code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char* argv[])
5  {
6      /* variable declaration */
7      int i;
8      int *result_M_over19=0x300B0;
9      int *result_over18  =0x300D0;
10
11     /* Initialize bitmap */
12     int *v_age16   = 0x30000;
13     int *v_age17   = 0x30018;
14     int *v_age18   = 0x30030;
15     int *v_age19   = 0x30048;
16     int *v_age20   = 0x30060;
17     int *v_genderM = 0x30078;
18     int *v_genderF = 0x30090;
19
20     v_genderM[0]=0x00000000; v_genderM[1]=0x00000000; v_genderM[2]=0x00000000;
       v_genderM[3]=0xFFFFFFFF; v_genderM[4]=0xFFFFFFFF; v_genderM[5]=0xFFFFFFFF;
21     v_genderF[0]=0xFFFFFFFF; v_genderF[1]=0xFFFFFFFF; v_genderF[2]=0xFFFFFFFF;
       v_genderF[3]=0x00000000; v_genderF[4]=0x00000000; v_genderF[5]=0x00000000;
22     v_age16[0]  =0x00000000; v_age16[1]  =0x00000000; v_age16[2]  =0x0000FFFF;
       v_age16[3]  =0x00000000; v_age16[4]  =0x00000000; v_age16[5]  =0x0000FFFF;
```

```
23      v_age17[0]  =0x00000000; v_age17[1]  =0x00000000; v_age17[2]  =0xFFFF0000;
        v_age17[3]  =0x00000000; v_age17[4]  =0x00000000; v_age17[5]  =0xFFFF0000;
24      v_age18[0]  =0x00000000; v_age18[1]  =0x0000FFFF; v_age18[2]  =0x00000000;
        v_age18[3]  =0x00000000; v_age18[4]  =0x0000FFFF; v_age18[5]  =0x00000000;
25      v_age19[0]  =0x00000000; v_age19[1]  =0xFFFF0000; v_age19[2]  =0x00000000;
        v_age19[3]  =0x00000000; v_age19[4]  =0xFFFF0000; v_age19[5]  =0x00000000;
26      v_age20[0]  =0xFFFFFFFF; v_age20[1]  =0x00000000; v_age20[2]  =0x00000000;
        v_age20[3]  =0xFFFFFFFF; v_age20[4]  =0x00000000; v_age20[5]  =0x00000000;
27
28      /* Initialise results to 0 */
29      for(i=0; i<6; i++) {
30          result_M_over19[i] = 0;
31          result_over18[i]   = 0;
32      }
33
34      /* Query: identify male people that are 19 or 20 */
35      for(i=0; i<6; i++) {
36          result_M_over19[i] = v_genderM[i] & (v_age19[i] | v_age20[i]);
37      }
38
39      /* Query: identify people that are older than 18 */
40      for(i=0; i<6; i++) {
41          result_over18[i] = ~v_age16[i] & ~v_age17[i] ;
42      }
43
44      return EXIT_SUCCESS;
45  }
```

The program was modified to integrate new Compiler capabilities, Listing 4.9 shows the code with new LiM operations (store_active and load_mask). To optimize the LiM functionalities, it was decided to adopt a single in-memory operation, in this case only OR operations are performed in memory. Since, once programmed the memory, this will behave differently as in standard situations, the mask value was selected as 0, this allows to load variables into the Core without changing their values. Also memory locations which will host final results should be zeroed, in this way the logic store will not corrupt the results, in fact the result of the query will be used as mask value, thus by means of the OR operation the final result could be stored in the final memory destination.
Note that in the second Query, the famous De Morgan's Law is adopted to simplify the calculation, in fact $(\sim A)\&(\sim B) = \sim (A|B)$. In this specific algorithm, range operations cannot be used because operands which are needed to compute the final result need to be manipulated and loaded into the Core. This limits the possible improvement brought by the LiM approach. In the following, Listing 4.9 reports the simulation log of the modified C program.

Listing 4.9. *bitmap_search.c* code with new compiler

```
1   /* Bitmap search program*/
2   //This program emulates the bitmap search algorithm, students' features are
        distributed over 6 integer vectors.
3   //In this program two queries are perfomed
4   // 1. Which students are male and older than 19?
5   // 2. Which students are older than 18?
6
7   #include <stdio.h>
8   #include <stdlib.h>
9
10  int main(int argc, char* argv[])
11  {
12      /* variable declaration */
13      int i, N=0, mask=0, partial, res, operand;
```

```
14      int Nr = 7; //number of rows
15      volatile int (*result_M_over19)[Nr];    //declare memory variables as
        volatile int
16      volatile int (*result_over18)[Nr];
17      volatile int (*v_age16)[Nr];
18      volatile int (*v_age17)[Nr];
19      volatile int (*v_age18)[Nr];
20      volatile int (*v_age19)[Nr];
21      volatile int (*v_age20)[Nr];
22      volatile int (*v_genderM)[Nr];
23      volatile int (*v_genderF)[Nr];
24
25      register unsigned int x0 asm("x0");
26
27      //configuration address, where the config of the memory is stored.
28      int cnfAddress = 0x1fffc;
29
30      //Define variable addresses
31      result_M_over19 = (volatile int(*)[Nr]) 0x300B0;
32      result_over18   = (volatile int(*)[Nr]) 0x300D0;
33      v_age16         = (volatile int(*)[Nr]) 0x30000;
34      v_age17         = (volatile int(*)[Nr]) 0x30018;
35      v_age18         = (volatile int(*)[Nr]) 0x30030;
36      v_age19         = (volatile int(*)[Nr]) 0x30048;
37      v_age20         = (volatile int(*)[Nr]) 0x30060;
38      v_genderM       = (volatile int(*)[Nr]) 0x30078;
39      v_genderF       = (volatile int(*)[Nr]) 0x30090;
40
41      /* Initialize bitmap */
42      (*v_genderM)[0]=0x00000000; (*v_genderM)[1]=0x00000000; (*v_genderM)[2]=0
        x00000000; (*v_genderM)[3]=0xFFFFFFFF; (*v_genderM)[4]=0xFFFFFFFF; (*
        v_genderM)[5]=0xFFFFFFFF;
43      (*v_genderF)[0]=0xFFFFFFFF; (*v_genderF)[1]=0xFFFFFFFF; (*v_genderF)[2]=0
        xFFFFFFFF; (*v_genderF)[3]=0x00000000; (*v_genderF)[4]=0x00000000; (*
        v_genderF)[5]=0x00000000;
44      (*v_age16)[0]   =0x00000000; (*v_age16)[1]   =0x00000000; (*v_age16)[2]   =0
        x0000FFFF; (*v_age16)[3]   =0x00000000; (*v_age16)[4]   =0x00000000; (*
        v_age16)[5]   =0x0000FFFF;
45      (*v_age17)[0]   =0x00000000; (*v_age17)[1]   =0x00000000; (*v_age17)[2]   =0
        xFFFF0000; (*v_age17)[3]   =0x00000000; (*v_age17)[4]   =0x00000000; (*
        v_age17)[5]   =0xFFFF0000;
46      (*v_age18)[0]   =0x00000000; (*v_age18)[1]   =0x0000FFFF; (*v_age18)[2]   =0
        x00000000; (*v_age18)[3]   =0x00000000; (*v_age18)[4]   =0x0000FFFF; (*
        v_age18)[5]   =0x00000000;
47      (*v_age19)[0]   =0x00000000; (*v_age19)[1]   =0xFFFF0000; (*v_age19)[2]   =0
        x00000000; (*v_age19)[3]   =0x00000000; (*v_age19)[4]   =0xFFFF0000; (*
        v_age19)[5]   =0x00000000;
48      (*v_age20)[0]   =0xFFFFFFFF; (*v_age20)[1]   =0x00000000; (*v_age20)[2]   =0
        x00000000; (*v_age20)[3]   =0xFFFFFFFF; (*v_age20)[4]   =0x00000000; (*
        v_age20)[5]   =0x00000000;
49
50      // Initialize results to 0 (useful for neutral lw_or operation)
51      for(i=0; i<Nr-1; i++) {
52          (*result_M_over19)[i] = 0;
53          (*result_over18)[i]   = 0;
54      }
55
56      //program LiM for stand-alone OR operation
57      asm volatile("sw_active_or %[result], %[input_i], 0"
58      : [result] "=r" (x0)
```

```
59      : [input_i] "r" (cnfAddress), "[result]" (N)
60      );
61
62      /* Query: identify male people that are 19 or 20 */
63      for(i=0; i<Nr-1; i++) {
64
65          //load first operand with neutral load
66          asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
67          : [result] "=r" (partial)
68          : [input_s] "r" (&(*v_age20)[i]), [input_t] "r"  (x0), "[result]" (
        partial)
69          );
70
71          //lw_mask operation for OR computation, use previous operand as mask (
        equivalent to v_age19[i] | v_age20[i])
72          asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
73          : [result] "=r" (partial)
74          : [input_s] "r" (&(*v_age19)[i]), [input_t] "r"  (partial), "[result]"
        (partial)
75          );
76
77          //load last operand with neutral load
78          asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
79          : [result] "=r" (operand)
80          : [input_s] "r" (&(*v_genderM)[i]), [input_t] "r"  (x0), "[result]" (
        operand)
81          );
82          res = operand & partial; //compute AND operation inside the core
83
84          //sw operation to store results (neutral store)
85          (*result_M_over19)[i] = res;
86      }
87
88      /* Query: identify people that are older than 18 */
89     for(i=0; i<Nr-1; i++) {
90
91          //lload first operand with neutral load
92          asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
93          : [result] "=r" (partial)
94          : [input_s] "r" (&(*v_age17)[i]), [input_t] "r"  (x0), "[result]" (
        partial)
95          );
96
97          //lw_mask operation for OR computation, use previous operand as mask (
        equivalent to v_age17[i] | v_age16[i])
98          asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
99          : [result] "=r" (partial)
100         : [input_s] "r" (&(*v_age16)[i]), [input_t] "r"  (partial), "[result]"
        (partial)
101         );
102         res = ~ partial; //compute AND operation inside the core
103
104        //sw operation to store results (neutral store)
105      (*result_over18)[i] = res;
106    }
107
108
109     //restore standard operations
110     asm volatile("sw_active_none %[result], %[input_i], 0"
111     : [result] "=r" (x0)
```

```
112     : [input_i] "r" (cnfAddress), "[result]" (x0)
113     );
114
115
116     return EXIT_SUCCESS;
117 }
```

In the following the Execution time is estimated for all the four memory configurations. In these formulas, N is considered as $N = \frac{N_{indexes}}{32}$ because operations are executed on 32-bits.

$$Execution\_time_{std\_mem} \approx 6N(result\_M\_over19) + 5N(result\_M\_over18) \tag{4.9}$$

$$Execution\_time_{std\_LiM\_mem} \approx 1(mem\_active) + 5N(result\_M\_over19) + 4N(result\_over18) + 1(mem\_active) \tag{4.10}$$

$$Execution\_time_{std\_mem} \approx 6N(result\_M\_over19) + 5N(result\_M\_over18) \tag{4.11}$$

$$Execution\_time_{RT\_LiM\_mem} \approx 1(mem\_active) + 9N[result\_M\_over19] + 7N[result\_over18] + 1(mem\_active) \tag{4.12}$$

The factor 6 in Equation 4.9 comes to the fact that three clock cycles are required to load all the operands in the Core, two clock cycles are requested to perform the logic computations and the last one is used to store-back the final result. The considerations are similar also for the second query. Equation 4.10 shows the possible improvement brought by the LiM approach, two clock cycles are wasted to program the memory, but queries computations are performed in less clock cycles. The standard Racetrack case, Equation 4.11, has the same Execution time of the standard memory case. While, the LiM Racetrack case reported in Equation 4.12, embeds an higher latency due to the internal built-in NOR feature exploitation. Remember that all the memory operations are performed keeping the memory programmed to compute in-memory OR operations. Neutral memory accesses are treated as in-place computations using as mask the zero value. For this reason, this implementation results in worse performance with respect to all the other cases, even worse than not adopting the LiM feature.
Figure 4.3, shows the Execution time estimation with different N indexes, as anticipated, the LiM Racetrack case has a much steeper curve due to the additional latency brought by the internal NOR computations.

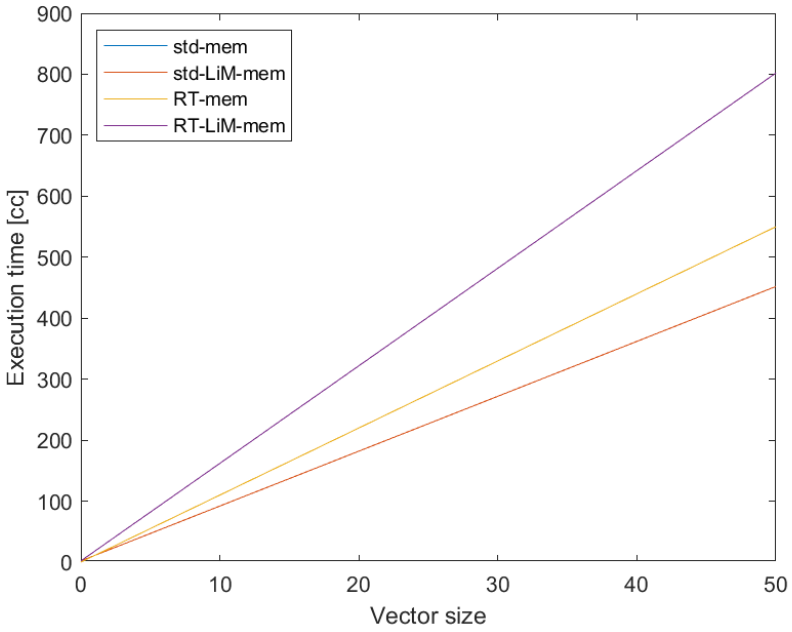

Figure 4.3. Execution time estimation for *bitmap_search.c* with different vector size *N*

Log 4.11 shows how some operations are partially replaced by in-memory computations. Simulations were taken adopting N=6 (N_indexes=192).

Listing 4.10. Extract of simulation log of *bitmap_search.c* - standard configuration with new compiler

```
Time        Cycles PC    Instr    Mnemonic
2366ns      233 00000300 09078793 addi       x15, x15, 144       x15=00030090 x15:00030000
2376ns      234 00000304 fe872603 lw         x12, -24(x14)       x12=ffffffff x14:00030078  PA:00030060
2386ns      235 00000308 fd072683 lw         x13, -48(x14)       x13=00000000 x14:00030078  PA:00030048
2396ns      236 0000030c 00470713 addi       x14, x14, 4         x14=0003007c x14:00030078
2406ns      237 00000310 00c6e6b3 or         x13, x13, x12       x13=ffffffff x13:00000000 x12:ffffffff
2416ns      238 00000314 ffc72603 lw         x12, -4(x14)        x12=00000000 x14:0003007c  PA:00030078
2436ns      240 00000318 00c6f6b3 and        x13, x13, x12       x13=00000000 x13:ffffffff x12:00000000
2446ns      241 0000031c 02d72a23 sw         x13, 52(x14)        x13:00000000 x14:0003007c  PA:000300b0
2456ns      242 00000320 fef712e3 bne        x14, x15, -28       x14:0003007c x15:00030090

2896ns      286 00000320 fef712e3 bne        x14, x15, -28       x14:0003008c x15:00030090
2926ns      289 00000304 fe872603 lw         x12, -24(x14)       x12=00000000 x14:0003008c  PA:00030074
2936ns      290 00000308 fd072683 lw         x13, -48(x14)       x13=00000000 x14:0003008c  PA:0003005c
2946ns      291 0000030c 00470713 addi       x14, x14, 4         x14=00030090 x14:0003008c
2956ns      292 00000310 00c6e6b3 or         x13, x13, x12       x13=00000000 x13:00000000 x12:00000000
2966ns      293 00000314 ffc72603 lw         x12, -4(x14)        x12=ffffffff x14:00030090  PA:0003008c
2986ns      295 00000318 00c6f6b3 and        x13, x13, x12       x13=00000000 x13:00000000 x12:ffffffff
2996ns      296 0000031c 02d72a23 sw         x13, 52(x14)        x13:00000000 x14:00030090  PA:000300c4
3006ns      297 00000320 fef712e3 bne        x14, x15, -28       x14:00030090 x15:00030090

3486ns      345 0000032c 0187a703 lw         x14, 24(x15)        x14=ffff0000 x15:00030014  PA:0003002c
3496ns      346 00000330 0007a603 lw         x12, 0(x15)         x12=0000ffff x15:00030014  PA:00030014
3506ns      347 00000334 00478793 addi       x15, x15, 4         x15=00030018 x15:00030014
3516ns      348 00000338 00c76733 or         x14, x14, x12       x14=ffffffff x14:ffff0000 x12:0000ffff
3526ns      349 0000033c fff74713 xori       x14, x14, -1        x14=00000000 x14:ffffffff
3536ns      350 00000340 0ce7a623 sw         x14, 204(x15)       x14:00000000 x15:00030018  PA:000300e4
3546ns      351 00000344 fed794e3 bne        x15, x13, -24       x15:00030018 x13:00030018
3556ns      352 00000348 00000513 addi       x10, x0, 0          x10=00000000
3566ns      353 0000034c 00008067 jalr       x0, x1, 0            x1:000001d8
```

Listing 4.11. Extract of simulation log of *bitmap_search.c* - LiM configuration with new compiler

```
Time        Cycles PC    Instr    Mnemonic
177501ns    4433 00000300 ffc68693 addi             x13, x13, -4        x13=0001fffc x13:00020000
177541ns    4434 00000304 0006b03b sw_active_or     Nx0 0(x13)          x13:0001fffc  PA:0001fffc
177581ns    4435 00000308 000306b7 lui              x13, 0x30000        x13=00030000
177621ns    4436 0000030c 06078793 addi             x15, x15, 96        x15=00030060 x15:00030000
177661ns    4437 00000310 00000613 addi             x12, x0, 0          x12=00000000
```

```
177701ns     4438 00000314 0b068313 addi            x6, x13, 176         x6=000300b0 x13:00030000
177741ns     4439 00000318 00600513 addi            x10, x0, 6           x10=00000006
177781ns     4440 0000031c 0007a71b lw_mask         x14, x0, 0(x15)      x14=ffffffff x15:00030060  PA:00030060
177821ns     4441 00000320 fe878593 addi            x11, x15, -24        x11=00030048 x15:00030060
177901ns     4443 00000324 00e5a71b lw_mask         x14, x14, 0(x11)     x14=ffffffff x14:ffffffff x11:00030048  PA:00030048
177941ns     4444 00000328 01878593 addi            x11, x15, 24         x11=00030078 x15:00030060
178021ns     4446 0000032c 0005a81b lw_mask         x16, x0, 0(x11)      x16=00000000 x11:00030078  PA:00030078
178061ns     4447 00000330 00261593 slli            x11, x12, 0x2        x11=00000000 x12:00000000
178141ns     4449 00000334 010778b3 and             x17, x14, x16        x17=00000000 x14:ffffffff x16:00000000
178181ns     4450 00000338 00b305b3 add             x11, x6, x11         x11=000300b0  x6:000300b0 x11:00000000
178221ns     4451 0000033c 0115a023 sw              x17, 0(x11)          x17:00000000 x11:000300b0  PA:000300b0
178261ns     4452 00000340 00160613 addi            x12, x12, 1          x12=00000001 x12:00000000
178341ns     4454 00000344 00478793 addi            x15, x15, 4          x15=00030064 x15:00030060
178381ns     4455 00000348 fca61ae3 bne             x12, x10, -44        x12:00000001 x10:00000006

182021ns     4546 0000034c 000305b7 lui             x11, 0x30000         x11=00030000
182061ns     4547 00000350 01868793 addi            x15, x13, 24         x15=00030018 x13:00030000
182101ns     4548 00000354 0d058593 addi            x11, x11, 208        x11=000300d0 x11:00030000
182141ns     4549 00000358 00000693 addi            x13, x0, 0           x13=00000000
182181ns     4550 0000035c 00600513 addi            x10, x0, 6           x10=00000006
182221ns     4551 00000360 0007a71b lw_mask         x14, x0, 0(x15)      x14=00000000 x15:00030018  PA:00030018
182261ns     4552 00000364 fe878613 addi            x12, x15, -24        x12=00030000 x15:00030018
182341ns     4554 00000368 00e6271b lw_mask         x14, x14, 0(x12)     x14=00000000 x14:00000000 x12:00030000  PA:00030000
182381ns     4555 0000036c 00269613 slli            x12, x13, 0x2        x12=00000000 x13:00000000
182461ns     4557 00000370 fff74813 xori            x16, x14, -1         x16=ffffffff x14:00000000
182501ns     4558 00000374 00c58633 add             x12, x11, x12        x12=000300d0 x11:000300d0 x12:00000000
182541ns     4559 00000378 01062023 sw              x16, 0(x12)          x16:ffffffff x12:000300d0  PA:000300d0
182581ns     4560 0000037c 00168693 addi            x13, x13, 1          x13=00000001 x13:00000000
182661ns     4562 00000380 00478793 addi            x15, x15, 4          x15=0003001c x15:00030018
182701ns     4563 00000384 fca69ee3 bne             x13, x10, -36        x13:00000001 x10:00000006

185101ns     4623 00000384 fca69ee3 bne             x13, x10, -36        x13:00000005 x10:00000006
185221ns     4626 00000360 0007a71b lw_mask         x14, x0, 0(x15)      x14=ffff0000 x15:0003002c  PA:0003002c
185261ns     4627 00000364 fe878613 addi            x12, x15, -24        x12=00030014 x15:0003002c
185341ns     4629 00000368 00e6271b lw_mask         x14, x14, 0(x12)     x14=ffffffff x14:ffff0000 x12:00030014  PA:00030014
185381ns     4630 0000036c 00269613 slli            x12, x13, 0x2        x12=00000014 x13:00000005
185461ns     4632 00000370 fff74813 xori            x16, x14, -1         x16=00000000 x14:ffffffff
185501ns     4633 00000374 00c58633 add             x12, x11, x12        x12=000300e4 x11:000300d0 x12:00000014
185541ns     4634 00000378 01062023 sw              x16, 0(x12)          x16:00000000 x12:000300e4  PA:000300e4
185581ns     4635 0000037c 00168693 addi            x13, x13, 1          x13=00000006 x13:00000005
185661ns     4637 00000380 00478793 addi            x15, x15, 4          x15=00030030 x15:0003002c
185701ns     4638 00000384 fca69ee3 bne             x13, x10, -36        x13:00000006 x10:00000006
185741ns     4639 00000388 000207b7 lui             x15, 0x20000         x15=00020000
185781ns     4640 0000038c ffc78793 addi            x15, x15, -4         x15=0001fffc x15:00020000
185821ns     4641 00000390 0007803b sw_active_none  Nx0 0(x15)           x15:0001fffc  PA:0001fffc
185861ns     4642 00000394 00000513 addi            x10, x0, 0           x10=00000000
185901ns     4643 00000398 00008067 jalr            x0, x1, 0             x1:000001d8
```

4.3.2 AES Addroundkey algorithm

The Advanced Encryprion Standard (AES) algorithm is very important in the security of data transmission [35], [19]. The algorithm encrypts sets of 128-bits data organized in a 4x4 matrix identified as *states*. In the case of AES-128, data is manipulated by means of a 128-bit *key*, still organizes in a 4x4 matrix. During the algorithm execution, the *AddRoundKey* step is performed 11 times, it consists in XOR operations, element by element, between data of the *states* matrix and data of the *key* matrix. The LiM approach could speed-up the algorithm execution because XOR operations can be executed directly in memory.
The Code reported in Listing 4.12 is quite simple, matrix data and *key* data are stored in specific memory locations, then a double loop computes all the results element by element.

Listing 4.12. *aes128_addroundkey.c* code

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(int argc, char* argv[])
5  {
6      /* input variables declaration */
7      int (*states)[4][4] = 0x30000;
8      (*states)[0][0]=0x32; (*states)[0][1]=0x88; (*states)[0][2]=0x31; (*states)
       [0][3]=0xE0;
9      (*states)[1][0]=0x43; (*states)[1][1]=0x54; (*states)[1][2]=0x31; (*states)
       [1][3]=0x37;
10     (*states)[2][0]=0xF6; (*states)[2][1]=0x30; (*states)[2][2]=0x98; (*states)
       [2][3]=0x07;
11     (*states)[3][0]=0xA8; (*states)[3][1]=0x8D; (*states)[3][2]=0xA2; (*states)
       [3][3]=0x34;
```

```
12
13      int (*key)[4][4] = 0x30200;
14      (*key)[0][0]=0x00; (*key)[0][1]=0xA5; (*key)[0][2]=0xA8; (*key)[0][3]=0xA0;
15      (*key)[1][0]=0xE9; (*key)[1][1]=0x09; (*key)[1][2]=0xBB; (*key)[1][3]=0x2A;
16      (*key)[2][0]=0xC9; (*key)[2][1]=0xD4; (*key)[2][2]=0xB7; (*key)[2][3]=0xAB;
17      (*key)[3][0]=0xF2; (*key)[3][1]=0xE8; (*key)[3][2]=0x60; (*key)[3][3]=0x08;
18
19    /* Others */
20    int i, j;
21
22    /* Add around key */
23    for (i=0; i<4; i++) {
24        for (j=0; j<4; j++) {
25            (*states)[i][j] = (*states)[i][j] ^ (*key)[i][j];
26        }
27    }
28
29    return EXIT_SUCCESS;
30 }
```

Listing 4.15 shows the code modified with new LiM instructions, once the memory is programmed for XOR operation, the load of the first operand requires a mask equal to 0, in this way data loaded in the Core is not corrupted. Once *key* data is loaded, it could be used for the logic store and for computing in-memory XOR operations.

Listing 4.13. *aes128_addroundkey.c* code with new compiler

```
1  /* AES128 Addroundkey program*/
2  //Compute AES128 Addroundkey, the algorithm encrypts chunks of 128-bit data
       organized in a 4x4 matrix named 'states'.
3  //Data is transformed with a XOR operation using a 4x4 matrix named 'key'.
4
5  #include <stdio.h>
6  #include <stdlib.h>
7
8  int main(int argc, char* argv[])
9  {
10
11      /* input variables declaration */
12      volatile int (*states)[4][4];
13      volatile int (*key)[4][4];
14
15      states = (volatile int(*)[4][4]) 0x30000;     //define states matrix
        starting address
16      key    = (volatile int(*)[4][4]) 0x30200;     //define key matrix starting
        address
17
18      //configuration address, where the config of the memory is stored.
19      int cnfAddress = 0x1fffc;
20
21      register unsigned int x0 asm("x0");
22
23      //Initialize states matrix
24      (*states)[0][0]=0x32; (*states)[0][1]=0x88; (*states)[0][2]=0x31; (*states)
        [0][3]=0xE0;
25      (*states)[1][0]=0x43; (*states)[1][1]=0x54; (*states)[1][2]=0x31; (*states)
        [1][3]=0x37;
26      (*states)[2][0]=0xF6; (*states)[2][1]=0x30; (*states)[2][2]=0x98; (*states)
        [2][3]=0x07;
27      (*states)[3][0]=0xA8; (*states)[3][1]=0x8D; (*states)[3][2]=0xA2; (*states)
        [3][3]=0x34;
```

```
28
29      //Initialize key matrix
30      (*key)[0][0]=0x00; (*key)[0][1]=0xA5; (*key)[0][2]=0xA8; (*key)[0][3]=0xA0;
31      (*key)[1][0]=0xE9; (*key)[1][1]=0x09; (*key)[1][2]=0xBB; (*key)[1][3]=0x2A;
32      (*key)[2][0]=0xC9; (*key)[2][1]=0xD4; (*key)[2][2]=0xB7; (*key)[2][3]=0xAB;
33      (*key)[3][0]=0xF2; (*key)[3][1]=0xE8; (*key)[3][2]=0x60; (*key)[3][3]=0x08;
34
35    /* Other variables */
36    int i, j, N = 1, opK;
37
38    //Program memory for XOR operations
39    asm volatile("sw_active_xor %[result], %[input_i], 0"
40     : [result] "=r" (x0)
41     : [input_i] "r" (cnfAddress), "[result]" (N)
42     );
43
44
45    /* Add around key */
46    for (i=0; i<4; i++) {
47        for (j=0; j<4; j++) {
48
49            //lw key[i][j] inside the core
50            asm volatile("lw_mask %[result], %[input_s], %[input_t], 0 "
51            : [result] "=r" (opK)
52            : [input_s] "r" (&(*key)[i][j]), [input_t] "r"  (x0), "[result]" (
      opK)
53            );
54
55            //sw operation to activate sw_xor, compute XOR oepration between
      states[i][j] and key[i][j]
56            (*states)[i][j] = opK; //use key as mask
57        }
58    }
59
60     //restore standard operations
61     asm volatile("sw_active_none %[result], %[input_i], 0"
62     : [result] "=r" (x0)
63     : [input_i] "r" (cnfAddress), "[result]" (x0)
64     );
65
66
67    return EXIT_SUCCESS;
68 }
```

The Execution time estimation for this algorithm does not depend on any parameter because it requires a fixed number of elements equal to N = 16, all the following computations are performed assuming this value.

$$Execution\_time_{std\_mem} \approx 4N = 64cc \tag{4.13}$$

$$Execution\_time_{std\_LiM\_mem} \approx 1(mem\_active) + 2N + 1(mem\_active) = 34cc \tag{4.14}$$

$$Execution\_time_{std\_mem} \approx 4N = 64cc \tag{4.15}$$

$$Execution\_time_{std\_LiM\_mem} \approx 1(mem\_active) + 2N + 1(mem\_active) = 34cc \qquad (4.16)$$

The factor 4 in front of Equation 4.13 comes to the fact that *key* and *states* bits should be loaded in the Core, then the XOR operation is performed and then finally the result is stored-back. Adopting the LiM configuration, the logic computations takes two clock cycles because the *key* element should be loaded in memory and then it is used as mask for the logic store, note that two additional cycles are required for memory programming. Note that, in this algorithm it is not possible to exploit logic range operations because every iteration requires a different set of *key* and state bits. Racetrack implementations, Equations 4.15 and 4.16, show the same behaviour in terms of Execution time with respect to their counterparts.
Racetrack memory with LiM functionalities computes XOR operations with no additional required latency, for this reason the performance is equal to the standard memory LiM implementation.
For both types of memories, speed-up improvements are remarkable, they are near 50%, Figure 4.4 shows an histogram of the Execution time.

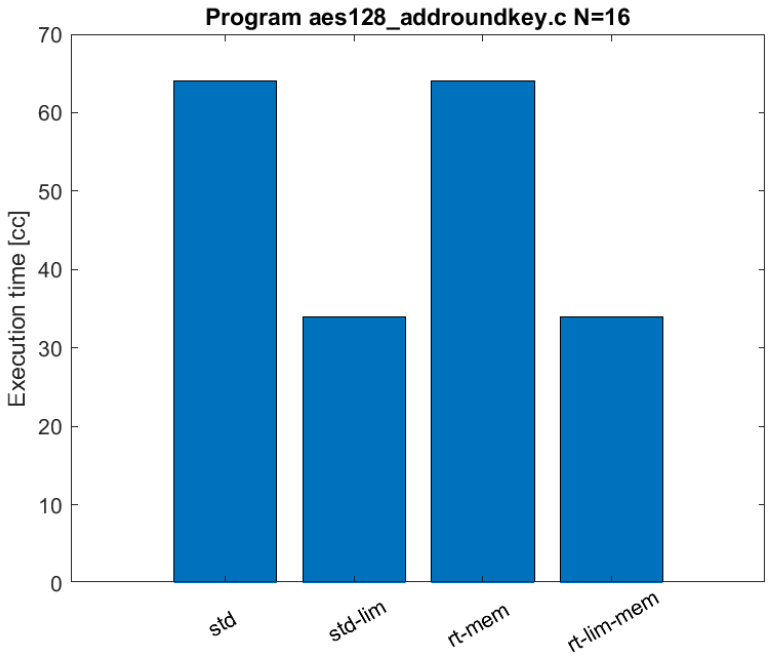

Figure 4.4. Execution time estimation for *aes128_addroundkey.c* with N=16

Simulations results are reported in simulation logs 4.14 and 4.15, it is clearly visible how the result computation requires less instructions than in the normal configuration.

Listing 4.14. Extract of simulation log of *aes128_addroundkey.c* - standard configuration with new compiler

```
Time       Cycles PC    Instr    Mnemonic
2486ns     245 00000310 000308b7 lui       x17, 0x30000        x17=00030000
2496ns     246 00000314 00400313 addi      x6, x0, 4            x6=00000004
2506ns     247 00000318 00000693 addi      x13, x0, 0          x13=00000000
2516ns     248 0000031c 00261813 slli      x16, x12, 0x2       x16=00000000 x12:00000000
2526ns     249 00000320 00d807b3 add       x15, x16, x13       x15=00000000 x16:00000000 x13:00000000
2536ns     250 00000324 00279793 slli      x15, x15, 0x2       x15=00000000 x15:00000000
2546ns     251 00000328 00f88533 add       x10, x17, x15       x10=00030000 x17:00030000 x15:00000000
```

```
2556ns     252 0000032c 00f707b3 add       x15, x14, x15       x15=00030200 x14:00030200 x15:00000000
2566ns     253 00000330 00052583 lw        x11, 0(x10)         x11=00000032 x10:00030000  PA:00030000
2576ns     254 00000334 0007a783 lw        x15, 0(x15)         x15=00000000 x15:00030200  PA:00030200
2586ns     255 00000338 00168693 addi      x13, x13, 1         x13=00000001 x13:00000000
2596ns     256 0000033c 00f5c7b3 xor       x15, x11, x15       x15=00000032 x11:00000032 x15:00000000
2606ns     257 00000340 00f52023 sw        x15, 0(x10)         x15:00000032 x10:00030000  PA:00030000
2616ns     258 00000344 fc669ee3 bne       x13, x6, -36        x13:00000001  x6:00000004

4486ns     445 00000330 00052583 lw        x11, 0(x10)         x11=00000034 x10:0003003c  PA:0003003c
4496ns     446 00000334 0007a783 lw        x15, 0(x15)         x15=00000008 x15:0003023c  PA:0003023c
4506ns     447 00000338 00168693 addi      x13, x13, 1         x13=00000004 x13:00000003
4516ns     448 0000033c 00f5c7b3 xor       x15, x11, x15       x15=0000003c x11:00000034 x15:00000008
4526ns     449 00000340 00f52023 sw        x15, 0(x10)         x15:0000003c x10:0003003c  PA:0003003c
4536ns     450 00000344 fc669ee3 bne       x13, x6, -36        x13:00000004  x6:00000004
4546ns     451 00000348 00160613 addi      x12, x12, 1         x12=00000004 x12:00000003
4556ns     452 0000034c fcd616e3 bne       x12, x13, -52       x12:00000004 x13:00000004
4566ns     453 00000350 00000513 addi      x10, x0, 0          x10=00000000
4576ns     454 00000354 00008067 jalr      x0, x1, 0            x1:000001d8
```

Listing 4.15. Extract of simulation log of *aes128_addroundkey.c* - LiM configuration with new compiler

```
Time          Cycles PC     Instr    Mnemonic
176021ns      4396 00000320 00030837 lui             x16, 0x30000        x16=00030000
176061ns      4397 00000324 00400593 addi            x11, x0, 4          x11=00000004
176101ns      4398 00000328 0007a69b lw_mask         x13, x0, 0(x15)     x13=00000000 x15:00030200  PA:00030200
176141ns      4399 0000032c 00461713 slli            x14, x12, 0x4       x14=00000000 x12:00000000
176181ns      4400 00000330 00e80733 add             x14, x16, x14       x14=00030000 x16:00030000 x14:00000000
176221ns      4401 00000334 00d72023 sw              x13, 0(x14)         x13:00000000 x14:00030000  PA:00030000
176261ns      4402 00000338 00478513 addi            x10, x15, 4         x10=00030204 x15:00030200
176301ns      4403 0000033c 0005269b lw_mask         x13, x0, 0(x10)     x13=000000a5 x10:00030204  PA:00030204
176381ns      4405 00000340 00d72223 sw              x13, 4(x14)         x13:000000a5 x14:00030000  PA:00030004
176421ns      4406 00000344 00878513 addi            x10, x15, 8         x10=00030208 x15:00030200
176461ns      4407 00000348 0005269b lw_mask         x13, x0, 0(x10)     x13=000000a8 x10:00030208  PA:00030208
176541ns      4409 0000034c 00d72423 sw              x13, 8(x14)         x13:000000a8 x14:00030000  PA:00030008
176581ns      4410 00000350 00c78513 addi            x10, x15, 12        x10=0003020c x15:00030200
176621ns      4411 00000354 0005269b lw_mask         x13, x0, 0(x10)     x13=000000a0 x10:0003020c  PA:0003020c
176701ns      4413 00000358 00d72623 sw              x13, 12(x14)        x13:000000a0 x14:00030000  PA:0003000c
176741ns      4414 0000035c 00160613 addi            x12, x12, 1         x12=00000001 x12:00000000
176781ns      4415 00000360 01078793 addi            x15, x15, 16        x15=00030210 x15:00030200
176821ns      4416 00000364 fcb612e3 bne             x12, x11, -60       x12:00000001 x11:00000004

179101ns      4473 00000350 00c78513 addi            x10, x15, 12        x10=0003023c x15:00030230
179141ns      4474 00000354 0005269b lw_mask         x13, x0, 0(x10)     x13=00000008 x10:0003023c  PA:0003023c
179221ns      4476 00000358 00d72623 sw              x13, 12(x14)        x13:00000008 x14:00030030  PA:0003003c
179261ns      4477 0000035c 00160613 addi            x12, x12, 1         x12=00000004 x12:00000003
179301ns      4478 00000360 01078793 addi            x15, x15, 16        x15=00030240 x15:00030230
179341ns      4479 00000364 fcb612e3 bne             x12, x11, -60       x12:00000004 x11:00000004
179381ns      4480 00000368 000207b7 lui             x15, 0x20000        x15=00020000
179421ns      4481 0000036c ffc78793 addi            x15, x15, -4        x15=0001fffc x15:00020000
179461ns      4482 00000370 0007803b sw_active_none  Nx0 0(x15)          x15:0001fffc  PA:0001fffc
179501ns      4483 00000374 00000513 addi            x10, x0, 0          x10=00000000
179541ns      4484 00000378 00008067 jalr            x0, x1, 0            x1:000001d8
179621ns      4486 000001d8 0080006f jal             x0, 8
```

4.3.3 Binary Neural Network

A Neural Network (NN) is a structure capable of very complex tasks, it is composed by *neurons*[2], these basic building blocks cooperate together to take decisions. As shown in Figure 4.5, neurons are composed by two parts, the *net* part performs the weighted computations, while *f(net)* is basically an activation function applied to the output and in general it is a non-linear function. The formula of the net part is the following:

$$net = \sum_{i=0}^{N} X_i \times W_i + Bias \tag{4.17}$$

Each input is multiplied by the corresponding weight, possibly an additional bias term can be applied. Weights are adjusted during the *training* procedure, in this way the net's behaviour is tailored on the required goals.

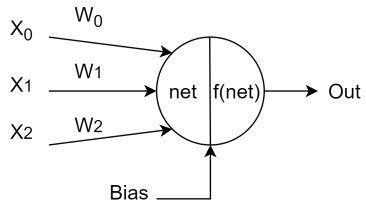

Figure 4.5. Neuron structure

A Neural Network is composed by multiple layers formed by multiple neurons, a very common structure of Neural Network is the Multi-Layer Perceptron, which is the one used for this test. The network is composed by multiple layers with different tasks:

- Convolutional layer: performs the convolution between the input values and the weights;
- Pooling layer: similar to convolutional layers, but performs maximum or the average of the selected inputs returning only one value, it perform the sub-sampling operation;
- FC layer: it is composed by fully-interconnected neurons and it performs classification operations;

Since NNs are very complex, a Binary approximation was introduced, this leads to a reduction of complexity as well as a reduction of the energy consumption of the algorithm. The approximation used for this test is the XNOR-Net, where all wights and inputs are in binary format [24].
The XNOR-net is used as a case study, the main part of the algorithm is the computation of the XNOR products and the pop-counting [10]. The proposed architecture will compute internally the XNOR products while pop-counting is performed outside the memory. In the proposed test program, shown in Listing 4.16, an input feature map (IFMAP) is convolved with a set of weights called kernel. When the first convolution is finished, the kernel windows is moved by a position defined by the *stride* parameter [10].
In *xnor_net.c* code 4.16, XNOR products are computed, several for-loops binaryzes the input and then for each element of ofmap matrix a XNOR operation is performed between the selected ofmap element and the computed bWeight. Note that the program implementation is a little bit different from the theory, in fact XOR products are computed and then outside pop-counting is performed on zeros. This version is completely equivalent to the original algorithm, it has the advantage that XOR operations requires less time to be executed inside the core, thus this approach was adopted.
A LiM architecture could speed up the computation, in 4.17 the code was modified to exploit a range XNOR operation, final matrix is computed with a single logic store operation. The final computation is performed as many times the number of channels, which is a single one in this case. In this way, a lot of instructions are avoided because this computation is performed directly, this leads also to a reduction of the size of the code because XNOR computation is not replicated every time in the inner for-loops but it is replicated only n_channels times.

Listing 4.16. *xnor_net.c* code

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <time.h>
4 #define N 28
5 #define W_F 2
6
7 int sign_function(int x)
8 {
```

```
9       if(x > 0)
10      {
11          return 1;
12      }
13      else
14      {
15          return 0;
16      }
17  }
18
19
20
21  int main()
22  {
23      // //initialize srand function
24      // srand(time(NULL));
25      //weight matrix
26      volatile int (*weight)[W_F][W_F];
27      //image matrix
28      volatile int (*image)[N][N];
29      //store location
30      weight =  (volatile int(*)[W_F][W_F]) 0x3000;
31      image =  (volatile int(*)[N][N]) 0x30800;
32      //the of-map is stored from 0x20000 address and so on.
33      volatile int (*ofmap)[N][N];
34      ofmap = (volatile int(*)[N][N]) 0x20000;
35      int zero = 0;
36      //configuration address, where the config of the memory is stored.
37      int cnfAddress = 0x1fffc;
38
39      //indexes definition.
40      int i,j,c,m,t;
41
42      for(i = 0; i < W_F; i++)
43      {
44          for(j = 0; j < W_F; j++)
45          {
46              (*weight)[i][j] = sign_function(0);
47          }
48      }
49      for(i = 0; i < N; i++)
50      {
51          for(j = 0; j < N; j++)
52          {
53              (*image)[i][j] = sign_function(0);
54              (*ofmap)[i][j] = 0;
55          }
56      }
57
58      //number of channels
59      int n_channels = 1;
60      //stride
61      int stride = 1;
62      //size of the kernel
63      int wf = W_F;
64      //dimension of the output
65      int w_out = (N-wf)/stride + 1;
66      //dimension squared of the output
67      int w_out2 = w_out*w_out + 1;
68      //index A and B
```

```
69      int A = 0;
70      int B = 0;
71      //flag indicating if the weight has been already binarized or not.
72      int flag = 0;
73      //binarized weight
74      unsigned int bWeight = 0;
75      //counting zeros
76      int countZeros;
77      for(c = 0; c < n_channels; c++)
78      {
79         for(j = 0; j < w_out; j++)
80         {
81            for(i = 0; i < w_out; i++)
82            {
83               for(m = 0; m < wf; m++)
84               {
85                  for(t = 0; t < wf; t++)
86                  {
87                     A = j+m+j*(stride-1);
88                     B = i+t+i*(stride-1);
89                     (*ofmap)[j][i] = ((*image)[A][B]) | ((*ofmap)[j][i] << 1);
90                     if (flag == 0)
91                     {
92                         bWeight = ( (*weight)[m][t] ) | (bWeight << 1);
93                     }
94                  }
95               }
96               //xor bitwise between the ofmap content and the binary weight
97               (*ofmap)[j][i] = (*ofmap)[j][i] ^ bWeight;
98               flag = 1;
99            }
100        }
101
102     }
103
104     return EXIT_SUCCESS;
105 }
```

Listing 4.17. *xnor_net_lim.c* code with new Compiler

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <time.h>
4  #define N 28
5  #define W_F 2
6
7  int sign_function(int x)
8  {
9      if(x > 0)
10     {
11         return 1;
12     }
13     else
14     {
15         return 0;
16     }
17 }
18
19
20
```

```
21 int main()
22 {
23     // //initialize srand function
24     // srand(time(NULL));
25     //weight matrix
26     volatile int (*weight)[W_F][W_F];
27     //image matrix
28     volatile int (*image)[N][N];
29     //store location
30     weight =  (volatile int(*)[W_F][W_F]) 0x3000;
31     image =  (volatile int(*)[N][N]) 0x30800;
32     //the of-map is stored from 0x20000 address and so on.
33     volatile int (*ofmap)[N][N];
34     ofmap = (volatile int(*)[N][N]) 0x20000;
35     int zero = 0;
36     //configuration address, where the config of the memory is stored.
37     int cnfAddress = 0x1fffc;
38 
39     //indexes definition.
40     int i,j,c,m,t;
41 
42     for(i = 0; i < W_F; i++)
43     {
44         for(j = 0; j < W_F; j++)
45         {
46             (*weight)[i][j] = sign_function(0);
47         }
48     }
49     for(i = 0; i < N; i++)
50     {
51         for(j = 0; j < N; j++)
52         {
53             (*image)[i][j] = sign_function(0);
54             (*ofmap)[i][j] = 0;
55         }
56     }
57 
58     //number of channels
59     int n_channels = 1;
60     //stride
61     int stride = 1;
62     //size of the kernel
63     int wf = W_F;
64     //dimension of the output
65     int w_out = (N-wf)/stride + 1;
66     //dimension squared of the output
67     int w_out2 = w_out*w_out + 1;
68     //index A and B
69     int A = 0;
70     int B = 0;
71     //flag indicating if the weight has been already binarized or not.
72     int flag = 0;
73     //binarized weight
74     unsigned int bWeight = 0;
75     //counting zeros
76     int countZeros;
77     for(c = 0; c < n_channels; c++)
78     {
79        for(j = 0; j < w_out; j++)
80        {
```

```
81            for(i = 0; i < w_out; i++)
82            {
83               for(m = 0; m < wf; m++)
84               {
85                  for(t = 0; t < wf; t++)
86                  {
87                     A = j+m+j*(stride-1);
88                     B = i+t+i*(stride-1);
89                     (*ofmap)[j][i] = ((*image)[A][B]) | ((*ofmap)[j][i] << 1);
90                     if (flag == 0)
91                     {
92                         bWeight = ( (*weight)[m][t] ) | (bWeight << 1);
93                     }
94                  }
95               }
96               flag = 1;
97            }
98         }
99          //activate the xor operation on W_FxW_F rows
100         asm volatile("sw_active_xor %[result], %[input_i], 0"
101         : [result] "=r" (w_out2)
102         : [input_i] "r" (cnfAddress), "[result]" (w_out2)
103         );
104         //store operation to run the xor in-memory
105         (*ofmap)[0][0] = bWeight;
106         //restore the normal function of the memory.
107         asm volatile("sw_active_none %[result], %[input_i], 0"
108         : [result] "=r" (zero)
109         : [input_i] "r" (cnfAddress), "[result]" (zero)
110         );
111
112     }
113
114     return EXIT_SUCCESS;
115 }
```

In the following a tentative Execution Time estimation is proposed. For the estimation only operations and operands involved in the LiM XNOR computation are taken into account, all the other computations are considered as an offset. Multiple parameters are involved in the Equations, for sake of simplicity it was assumed only N as variable parameters all the others are fixed:

- wf = 2;
- n_channel = 1;

Note that parameter w_out is a function of N and it is expressed as $w_{out} = \frac{(N - wf)}{stride} + 1$. After the weights bynarization the line of code regarding bWeight is no more executed, this is taken into account in the equation with a factor wf^2.

$$Execution\_time_{std\_mem} \approx n_{channel} \cdot w_{out}^2[wf^2(fixed) + 3(ofmap)] + wf^2(fixed\_bWeight) \tag{4.18}$$

$$Execution\_time_{std\_LiM\_mem} \approx n_{channel} \cdot w_{out}^2[wf^2(fixed)] + n_{channel}[1(mem\_active) + 1(ofmap) + 1(mem\_active)] + wf^2(fixed\_bWeight) \tag{4.19}$$

$$Execution\_time_{rt\_mem} \approx n_{channel} \cdot w_{out}^2[wf^2(fixed) + 3(ofmap)] + wf^2(fixed\_bWeight) \tag{4.20}$$

$$Execution\_time_{RT\_LiM\_mem} \approx n_{channel} \cdot w_{out}^2[wf^2(fixed)] + n_{channel}[1(mem\_active) + 1(ofmap) + 1(mem\_active)] + wf^2(fixed\_bWeight) \tag{4.21}$$

Equations 4.19 and 4.21 shows that the logic computation is carried out in a fully parallel way, this results in a constant Execution time with any value N, note that two additional clock cycles are required for memory programming. In this case Racetrack memory has the same behaviour because the involved operation is a XOR one, and as known it is performed with the same latency of a standard memory access.
The plot shows that Execution times are overlapped for both LiM and not-LiM versions, thus the behaviour in terms of performances is exactly the same. Standard memory and Raetrack memory without LiM functionalities show a quadratic behaviour due to the factor w_{out}^2 involved in the output computation.
Code that does not change between the four memory versions is considered as *fixed* and it is not included as offset in the plot reported in Figure 4.6.

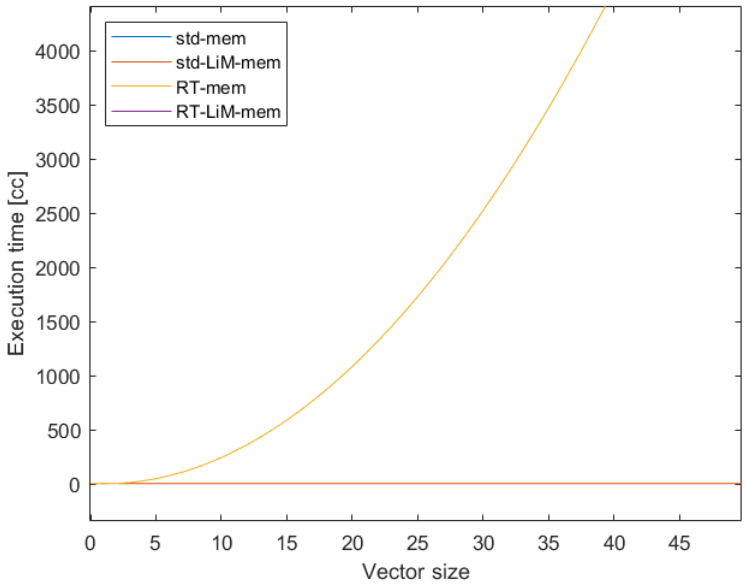

Figure 4.6. Execution time estimation for *xnor_net.c* with different vector size *N*

Simulation logs were taken with N=4 and adopting sign_function(1) for weight variables, there is a good improvement with the LiM configuration. The final XNOR operation is performed with a single logic store instruction repeated for the number of channels while without LiM instructions the line of code is repeated as many times the for-loop requires.

Listing 4.18. Extract of simulation log of *xnor_net.c* - standard configuration with new compiler

```
Time       Cycles PC    Instr    Mnemonic
2736ns     270 000002dc 00082403 lw              x8, 0(x16)          x8=00000000 x16:00030800  PA:00030800
2746ns     271 000002e0 0006a803 lw              x16, 0(x13)         x16=00000000 x13:00020000  PA:00020000
2766ns     273 000002e4 00181813 slli            x16, x16, 0x1       x16=00000000 x16:00000000
2776ns     274 000002e8 00886833 or              x16, x16, x8        x16=00000000 x16:00000000  x8:00000000
2786ns     275 000002ec 0106a023 sw              x16, 0(x13)         x16:00000000 x13:00020000  PA:00020000
2796ns     276 000002f0 00039c63 bne             x7, x0, 24           x7:00000000
2806ns     277 000002f4 00359813 slli            x16, x11, 0x3       x16=00000000 x11:00000000
2816ns     278 000002f8 010f0833 add             x16, x30, x16       x16=00003000 x30:00003000 x16:00000000
2826ns     279 000002fc 00082803 lw              x16, 0(x16)         x16=00000001 x16:00003000  PA:00003000
2836ns     280 00000300 00161613 slli            x12, x12, 0x1       x12=00000000 x12:00000000
2846ns     281 00000304 00c86633 or              x12, x16, x12       x12=00000001 x16:00000001 x12:00000000
2856ns     282 00000308 00670733 add             x14, x14, x6        x14=00000001 x14:00000000  x6:00000001
2866ns     283 0000030c 00271713 slli            x14, x14, 0x2       x14=00000004 x14:00000001

3416ns     338 00000330 00fe07b3 add             x15, x28, x15       x15=00000000 x28:00000000 x15:00000000
3426ns     339 00000334 00279793 slli            x15, x15, 0x2       x15=00000000 x15:00000000
3436ns     340 00000338 00fe87b3 add             x15, x29, x15       x15=00020000 x29:00020000 x15:00000000
3446ns     341 0000033c 0007a703 lw              x14, 0(x15)         x14=00000000 x15:00020000  PA:00020000
3466ns     343 00000340 00e64733 xor             x14, x12, x14       x14=0000000f x12:0000000f x14:00000000
3476ns     344 00000344 00e7a023 sw              x14, 0(x15)         x14:0000000f x15:00020000  PA:00020000
3486ns     345 00000348 00030793 addi            x15, x6, 0          x15=00000001  x6:00000001
3496ns     346 0000034c 02530063 beq             x6, x5, 32           x6:00000001  x5:00000003
```

Listing 4.19. Extract of simulation log of *xnor_net_lim.c* - LiM configuration with new compiler

```
Time       Cycles PC    Instr    Mnemonic
2736ns     270 000002dc 00082403 lw              x8, 0(x16)          x8=00000000 x16:00030800  PA:00030800
2746ns     271 000002e0 0006a803 lw              x16, 0(x13)         x16=00000000 x13:00020000  PA:00020000
2766ns     273 000002e4 00181813 slli            x16, x16, 0x1       x16=00000000 x16:00000000
2776ns     274 000002e8 00886833 or              x16, x16, x8        x16=00000000 x16:00000000  x8:00000000
2786ns     275 000002ec 0106a023 sw              x16, 0(x13)         x16:00000000 x13:00020000  PA:00020000
2796ns     276 000002f0 00031c63 bne             x6, x0, 24           x6:00000000
2806ns     277 000002f4 00361813 slli            x16, x12, 0x3       x16=00000000 x12:00000000
2816ns     278 000002f8 010e8833 add             x16, x29, x16       x16=00003000 x29:00003000 x16:00000000
2826ns     279 000002fc 00082803 lw              x16, 0(x16)         x16=00000001 x16:00003000  PA:00003000
2836ns     280 00000300 00171713 slli            x14, x14, 0x1       x14=00000000 x14:00000000
2846ns     281 00000304 00e86733 or              x14, x16, x14       x14=00000001 x16:00000001 x14:00000000
2856ns     282 00000308 01c787b3 add             x15, x15, x28       x15=00000001 x15:00000000 x28:00000001
2866ns     283 0000030c 00279793 slli            x15, x15, 0x2       x15=00000004 x15:00000001

9026ns     899 00000358 f5c592e3 bne             x11, x28, -188      x11:00000003 x28:00000003
9036ns     900 0000035c 000207b7 lui             x15, 0x20000        x15=00020000
9046ns     901 00000360 ffc78693 addi            x13, x15, -4        x13=0001fffc x15:00020000
9056ns     902 00000364 00a00613 addi            x12, x0, 10         x12=0000000a
9066ns     903 00000368 0006963b sw_active_xor   Nx12 0(x13)         x13:0001fffc x12:0000000a  PA:0001fffc
9076ns     904 0000036c 00e7a023 sw              x14, 0(x15)         x14:0000000f x15:00020000  PA:00020000
9086ns     905 00000370 00000793 addi            x15, x0, 0          x15=00000000
9096ns     906 00000374 000687bb sw_active_none  Nx15 0(x13)         x13:0001fffc x15:00000000  PA:0001fffc
9106ns     907 00000378 00c12403 lw              x8, 12(x2)           x8=00000000  x2:000060a0  PA:000060ac
9116ns     908 0000037c 00000513 addi            x10, x0, 0          x10=00000000
```

4.4 Simulation Results Analysis

Once defined the equations to estimate Execution Time, it is possible to compare real simulation results to understand the improvement brought by the LiM paradigm and the adoption of an innovative technology as Racetrack. Each program was run for all the three different Racetrack implementations, Execution time results will be shown in the following, this comparison is extremely useful to understand the performance improvement from the initial serial design to the final one.

4.4.1 Original version

In the following an analysis of the real execution times for the initial Racetrack design is given. In this first implementation, the Racetrack's clock frequency is the same as the system clock frequency, thus multiples clock cycles are required for different memory accesses. In particular, considering the implemented FSM algorithm, a standard access or a XOR/XNOR access takes 3 clock cycles to be performed and all the other access types take 6 clock cycles. Here, the system clock is still at $100MHz$ as in the original work presented in [5].
Table 4.1 summarizes the results of custom programs in terms of total Execution Time derived from Synopsys VCS simulations.

Table 4.1. Custom programs simulation results comparison

Program	Std-mem [ns]	Std-LiM-mem [ns]	RT NO LiM [ns]	RT LiM [ns]
bitwise.c	4220	3300 (-21,8%)	6430	5840 (-9,18%)
bitwise_inv.c	4390	3300 (-24,83%)	6600	5840 (-11,52%)

Considering at first the adoption of a standard memory technology, improvements are around $\simeq -21\%$ for *bitwise.c*, exploiting the range operations of the LiM architecture it is possible to have a quite interesting result in terms of performance. The improvement is even higher in *bitwise_inv.c* and it reaches nearly $\simeq -25\%$, inverting operations require more internal instructions with respect to normal one, this additional latency is avoided using the LiM structure.
For the Racetrack case, there is still some improvements adopting a LiM architecture ($\simeq -9\%$ and $\simeq -12\%$ respectively) but it is less than the other memory type. One of the reasons is the additional latency for built-in operations, like NAND/NOR and AND/OR which are performed with additional cycles with respect to XOR/XNOR ones. In addition, when adopting a Racetrack memory, it is not possible to perform parallel operations, in fact they are serialized.
In any case, these programs are tailored on the LiM architecture, thus the importance of these results is marginal, in Table 4.2 Standard programs results are reported, these can give a better idea of the improvements in real case scenarios.

Table 4.2. Custom programs simulation results comparison

Program	Std-mem [ns]	Std-LiM-mem [ns]	RT-mem [ns]	RT-LiM-mem [ns]
bitmap_search.c	4590	5520 (+20,26%)	8720	10140 (+29,67%)
aes128_addroundkey.c	5600	4350 (-22,32%)	8720	7070 (-18,92%)
xnor_net.c	10720	10160 (-5,22%)	15160	14530 (-4,16%)

Result for *bitmap_search.c* reported in the original word [5] showed a marginal improvement in terms of execution time which is limited to -2% (estimated in cc).
The overall Execution time for *bitmap_search.c* program in this Thesis has a completely different trend, in fact the adoption of a LiM architecture degrades the performance. Execution time increases of $\simeq +22\%$ adopting a standard technology memory but performance decreases dramatically using the Racetrack memory ($\simeq +30\%$), these results are of course unacceptable.
The reason of such bad results could reside in the code design, remember that in [5], LiM programs were manually modified adding required assembly instructions, this of course increases the software efficiency because it is possible to have a register-level optimization. Adopting in-line assembly pieces of codes is much easier but at the same time the code optimization is in charge of the Compiler. Probably in this case, the Compiler is not able to optimize the code. Taking into account the original assembly trace reported in [5], between one cycle iteration and the next one, only few instructions are implemented (i.e. addi for address update used in the lw_mask, logic operation, sw result and a bne for testing if the loop is ended). In the current implementation, between one cycle iteration and the next one, many other instructions are executed, this is an example on how the Compiler is not able to optimize the code. Another reason could be the code itself, standard code design is more restricting with respect to directly introduce modifications in .hex file, so it is possible that the designed LiM code is not much efficient.
Not considering software-related issues, this worse result of the Racetrack implementation is not unexpected because this program exploits only OR computations. This memory carries out this operation with the built-in NOR operation which requires more clock cycles with respect to a standard access. This was partially anticipated by the Execution time estimations, Figure 4.3

shows clearly how built-in NOR operation have a huge impact on performance.
Results for *aes128_addroundkey.c* program are very interesting. For the standard technology memory case, adopting a LiM configuration can achieve a remarkable improvement ($\simeq -22\%$), also in the case of a Racetrack memory it is possible to have a quite reasonable result ($\simeq -19\%$) which is very close to the previous one and even higher than Custom programs results. In this program only LiM XOR operations are exploited, as explained previously, in the Racetrack this computation is carried out with a latency equal to a standard memory access, this is the reason of this result. Remember that a NAND or NOR logic operation would have required more clock cycles.
For program *xnor_net.c* Execution time reduction is quite small, it reaches $\simeq -5\%$ in the standard technology memory case and $\simeq -4\%$ in the Racetrack case. These results have been obtained with a small number of elements (N=4) due to simulation time reasons, possibly improvement could be even higher with a higher value of N. Also in this case the program uses XOR operations, this is executed with the same latency as standard accesses in the Racetrack memory, this is the reason of the similar improvement.
It is clearly visible that different memories could lead to different results and satisfy different needs. In a standard technology memory configuration, the adoption of the LiM paradigm could achieve, for specific programs, interesting improvements in terms of Execution time. Of course area and power are necessarily higher because additional logic is required to support LiM operations.
Simulation results show how the adoption of a Racetrack memory leads to a deterioration of the Execution time, Table 4.3 shows the increase with the adoption of not of LiM functionalities with respect to the standard technology memory case.

Table 4.3. Execution time degradation with Racetrack memory

Program	RT-mem vs Std-mem	RT-LiM-mem vs Std-LiM-mem
bitwise.c	+52,37%	+76,97%
bitwise_inv.c	+50,34%	+76,97%
bitmap_search.c	+70,37%	+83,70%
aes128_addroundkey.c	+55,71%	+62,53%
xnor_net.c	+41,42%	+43,01%

Data show how Execution time increases on average in the range of $+40 - 80\%$. This analysis has not taken into account Area and Power, in [7] it is highlighted that Racetrack technology has a smaller cell size ($\leq 2\ F^2$) than classic SRAM ($\leq 100 - 200\ F^2$) and DRAM cells ($\leq 4 - 8\ F^2$). Furthermore, Read energy, Write energy and Leakage Power is lower in all the three cases with respect to SRAM and DRAM.
In addition, a lot of operations are carried out by means of magnetic interactions (i.e. data shifting and NAND/NOR computations) thus it is reasonable to expect a lower power consumption.

4.4.2 Parallel version

Results presented in the previous Section present a huge performance degradation when adopting Racetrack technology. In the following an analysis of Racetrack implementing parallel LiM store access feature is presented. As explained in the previous Chapter, these modifications allows to overcome the bottleneck generated by the serialization of parallel operations. This memory version exploits the same waveforms as in the previous one, it is expected to see quite remarkable improvement in terms of Execution time.
Table 4.4 reports the exact Execution times derived from VCS simulations for custom programs.

Table 4.4. Custom programs simulation results comparison

Program	Std-mem [ns]	Std-LiM-mem [ns]	RT NO LiM [ns]	RT LiM [ns]
bitwise.c	4220	3300 (-21,8%)	6430	5240 (-18,51%)
bitwise_inv.c	4390	3300 (-24,83%)	6600	5240 (-20,61%)

It is clearly visible how Execution time for the Racetrack LiM case is reduced and percentage improvements are similar to Std-LiM-mem ones. Execution time is still higher due to the multiple access cycles required for memory accesses.
Table 4.5 shows the result for standard programs.

Table 4.5. Custom programs simulation results comparison

Program	Std-mem [ns]	Std-LiM-mem [ns]	RT-mem [ns]	RT-LiM-mem [ns]
bitmap_search.c	4590	5520 (+20,26%)	7820	10140 (+29,67%)
aes128_addroundkey.c	5600	4350 (-22,32%)	8720	7070 (-18,92%)
xnor_net.c	10720	10160 (-5,22%)	15160	14260 (-5,94%)

Execution times reduces only for *xnor_net.c* program because it is the only one involving range operations. In this case percentage improvement is even higher than in the standard memory case. In *bitmap_search.c* program, Racetrack memory shows a worse behaviour due to the high number of OR operations involved in the programs.
Table 4.6 reports the comparison between adopting or not LiM functionalities for both memory types.

Table 4.6. Execution time degradation with Racetrack memory

Program	RT-mem vs Std-mem	RT-LiM-mem vs Std-LiM-mem
bitwise.c	+52,37%	+58,79%
bitwise_inv.c	+50,34%	+58,79%
bitmap_search.c	+70,37%	+83,70%
aes128_addroundkey.c	+55,71%	+62,53%
xnor_net.c	+41,42%	+40,35%

Results suggest that programs which use heavily range operations get a better improvement in terms of Execution time, in fact *bitwise.c* and *bitwise_inv.c* programs, considering the Racetrack case, go from +50/52% to +58,79% in the LiM case, a remarkable improvement. Considering *xnor_net.c* program, the improvement is marginal because the the number of elements used for the simulation is small, an higher improvement is expected using a higher value of N.

4.4.3 Core compliant version

This subsection analyzes results adopting a system clock frequency equal to $25MHz$ and a Racetrack working frequency equal to $100MHz$. It is expected to observe an important improvement in terms of Execution time because now memory accesses are performed in one or at most in two clock cycles.

Table 4.7 reports the exact Execution times derived from VCS simulations for custom programs.

Table 4.7. Custom programs simulation results comparison

Program	Std-mem [ns]	Std-LiM-mem [ns]	RT NO LiM [ns]	RT LiM [ns]
bitwise.c	16880	13200 (-21,8%)	16880	13400 (-20,62%)
bitwise_inv.c	17560	13200 (-24,83%)	17560	13400 (-23,69%)

Now for both programs performances are similar to the original ones, Execution time for both programs is a little bit higher for the Racetrack memory case due to built-in NAND/NOR operations performed in two clock cycles. In absolute terms, Execution time in all the cases increases due to the longer system clock period.
In Table 4.8 all the results for standard programs are shown.

Table 4.8. Standard programs simulation results comparison

Program	Std-mem [ns]	Std-LiM-mem [ns]	RT-mem [ns]	RT-LiM-mem [ns]
bitmap_search.c	18360	22080 (+20,26%)	18360	23760 (+29,41%)
aes128_addroundkey.c	22400	17400 (-22,32%)	22400	17400 (-22,32%)
xnor_net.c	42880	40640 (-5,22%)	42880	40640 (-5,22%)

Also in this case performances are basically the same for all the analyzed cases. Considerations for *bitmap_search.c* program are the same as in the previous Subsection.
Table 4.9 shows the comparison between adopting or not LiM functionalities.

Table 4.9. Execution time degradation with Racetrack memory

Program	RT-mem vs Std-mem	RT-LiM-mem vs Std-LiM-mem
bitwise.c	+0%	+1,52%
bitwise_inv.c	+0%	+1,52%
bitmap_search.c	+0%	+7,61%
aes128_addroundkey.c	+0%	+0%
xnor_net.c	+0%	+0%

Thanks to the new clock system frequency and the modifications applied to the Racetrack memory, in most of the programs the overhead due to the Racetrack memory is almost negligible (below 8% for in the worst case). Racetrack memory suffers when multiple NAND/NOR (AND/OR) operations are involved in program execution, in this case the latency is double with respect to a standard access or another logic operation, so it is important to take into account also the types of LiM operations implemented in the software.

4.5 Racetrack organization analysis

In the following a theoretical analysis on the possible Racetrack internal organizations will be given. It has been taken into account many different solutions with respect to the ones shown during the design followed in this Thesis.
The reference structure is the one used in [5], here a $2^{10}B$ memory was synthesized, thanks to the particular structure, it was possible to perform a store operation on 256 words, this result is taken as top level reference for all the other configurations. On the other side, the bottom level reference is represented by the first serial Racetrack design in which parallel store operations were

performed serially.
Different design solutions were explored and they focused on different design aspects:

- MU parameters (Type 1): In this category, the MU design parameters were modified to achieve a larger logic store operation parallelism. Here the number of bits per Racetrack is ranged from 1 to 32, but ports are arranged in a way in which the port alignment is performed always in a single clock cycle. In fact, for some configurations, the total number of ports remain stable as the number of bits per Racetrack increases. For example, the configuration $1 \times 8\_1024\_2048 - 32$ corresponds to a MU with 8 bits per Racetrack, 1024 Racetracks, 2048 ports and store parallelism equal to 32 words. The number of ports is double because in this way it is possible to perform a port alignment in the second half of the Racetrack with the same latency.
  These configurations are quite different from the other two, here a single MU is used and the concept of *Block* is different from the one explained previously because it can be considered as the whole MU;

- Active ports (Type 2): Here, a single 32-bit word is accessed within a *Block*, the difference is the possibility to access multiple words in parallel activating the same word-line in all the available *Blocks*. This solution is the one adopted in the final design of this Thesis and it requires a different word organization. The store parallelism depends on the available *Blocks*, larger the memory, larger is the store parallelism. In all these configurations it was assumed to have a *Block* structure as the one implemented in this Thesis, thus it contains 32 words. Fort this category it was analyzed also the case in which a larger memory size is used;

- *Block* parallelism (Type 3): The idea is to exploit the already available ports within each *Block*. Instead of accessing a single word within a *Block*, it is possible to access multiple words inside the same *Block*, this feature is combined to the possibility to access multiple *Blocks* in parallel. For example, a memory composed by 8 *Blocks* in which it is possible to access 2 words per *Block*, would result in a logic store parallelism equal to 16 words.
  Also in this case a different word organization is required. Also here it was assumed to have a *Block* structure as the one implemented in this Thesis;

All the configurations are represented with a code with the following format:
$Mtype - Type - N \times Nb\_Nr\_Np - N_{LSP}$.

- Mtype = Memory type
- Type = Test type
- N = Number of *Blocks*
- Nb = Number of bits per Racetrack
- Nr = Number of Racetracks
- Np = Number of ports per Racetrack
- N_{LSP} = Logic store parallelism expressed in terms of accessed words

In the following a brief description of all the three different design solutions is given. For all the analysis it was considered to adopt the same FSM algorithm described in the previous section and the configuration with the Core working at $25MHz$ and the Racetrack working at $100MHz$, thus accesses require one or at most two clock cycles.
In the following multiple analysis are shown, parameter N is ranged from 5 to 256 and it refers to the number of words accessed in parallel during a parallel store logic. Execution Time (in terms of

clock cycles) is used as comparison parameter between the different configurations, this metric was estimated with the same methodology adopted in previous analysis, taking into account the different logic store parallelism.
Configurations with a lower N_{LSP} are expected to have a higher Execution time because less words can be processed in parallel. In all the cases the first plot is taken with N equal to the one used in real tests, then it was increased starting from 32 and arriving to 256 (operation on the whole memory). Using N equal to the real parameter helps to have an idea of the actual configuration, while increasing N is helpful to understand the performances of each different configuration.
In Table 4.10 a summary of all the analyzed configurations is given.

Table 4.10. Configuration summary

N.	Mype	Type	N	N_b	N_r	N_p	N_{LSP}
1	std						
2	std-lim						
3	rt-std		8	5	3	2	1
4	rt-lim	1	1	1	8192	8192	256
5	rt-lim	1	1	2	4096	4096	128
5	rt-lim	1	1	4	2048	2048	64
6	rt-lim	1	1	8	1024	2048	32
7	rt-lim	1	1	16	512	2048	16
8	rt-lim	1	1	32	256	2048	8
9	rt-lim	2	8	32	4	8	1
10	rt-lim	2	8	32	4	8	8
11	rt-lim	2	16	32	4	8	16
12	rt-lim	2	32	32	4	8	32
13	rt-lim	2	64	32	4	8	64
14	rt-lim	2	128	32	4	8	128
15	rt-lim	2	256	32	4	8	256
16	rt-lim	3	8	32	4	8	8
17	rt-lim	3	8	32	4	8	16
18	rt-lim	3	8	32	4	8	32
19	rt-lim	3	8	32	4	8	64

4.5.1 Bitwise & inverted_bitwise

The following Figures show the Execution time, expressed in terms of clock cycles, changing the parameter N. Results show that for N equal to 5, as in the original program, performances are almost the same for all the cases. The LiM architecture presented in [5] has the best performance because all the operations are performed with the same latency. The worst result for the LiM approach is the one with the serial approach (like in the first Racetrack design), this is reasonable because parallel accesses are serialized.
On the right y axis the memory size is shown, for configurations of Type 2 the size increases due to the higher number of *Blocks*. Thus, for a limited N, in general lower than 8 (minimum logic SW parallelism, with the exception of the serial configuration), Execution times are more or less the same as the reference LiM case, with only a small overhead brought by the well known operations which are performed directly in memory.
As parameter N increases, performances will depend deeply from parameter N_{LSP}. The behaviour is the same in all the plots, configurations with higher N_{LSP} show a lower Execution time because they can perform accesses on a higher range of words, while other configurations require multiple accesses to complete the range operations.

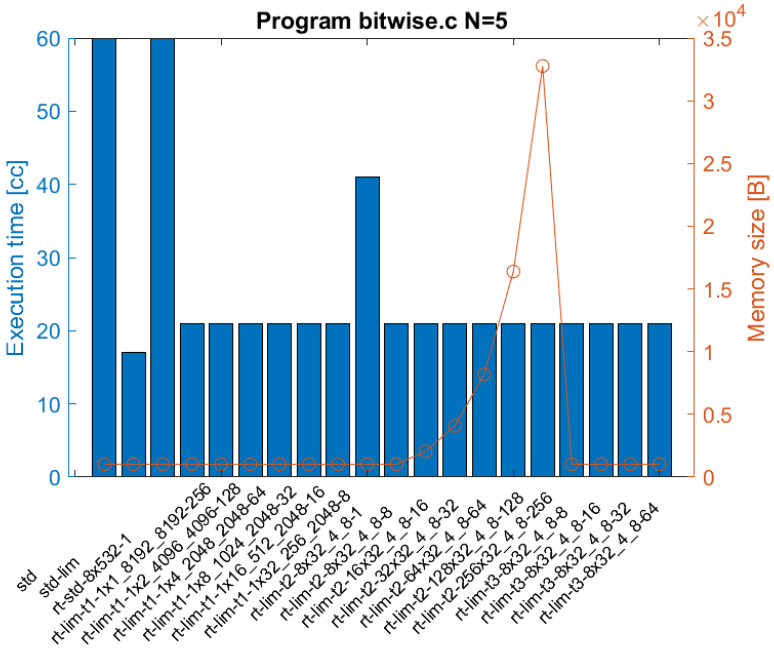

Figure 4.7. Theoretic analysis N=5

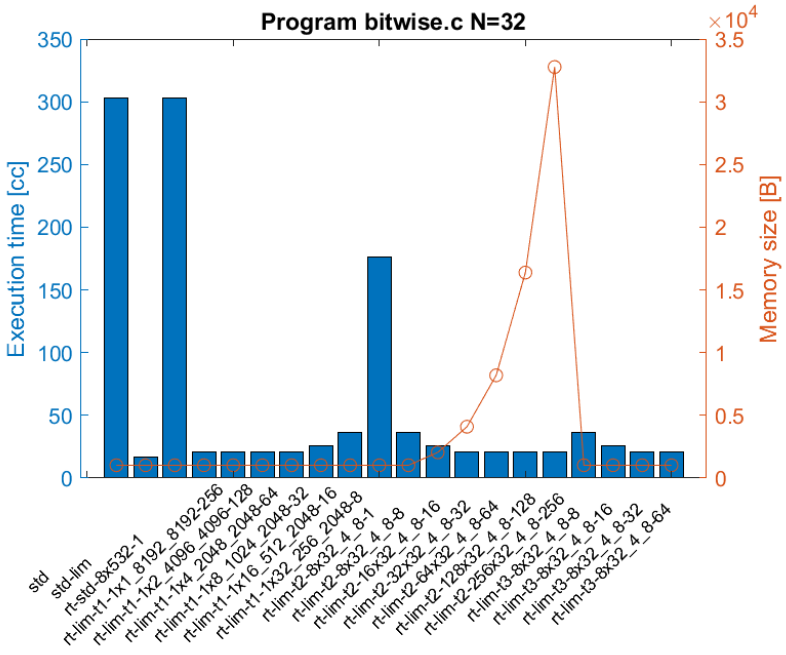

Figure 4.8. Theoretic analysis N=32

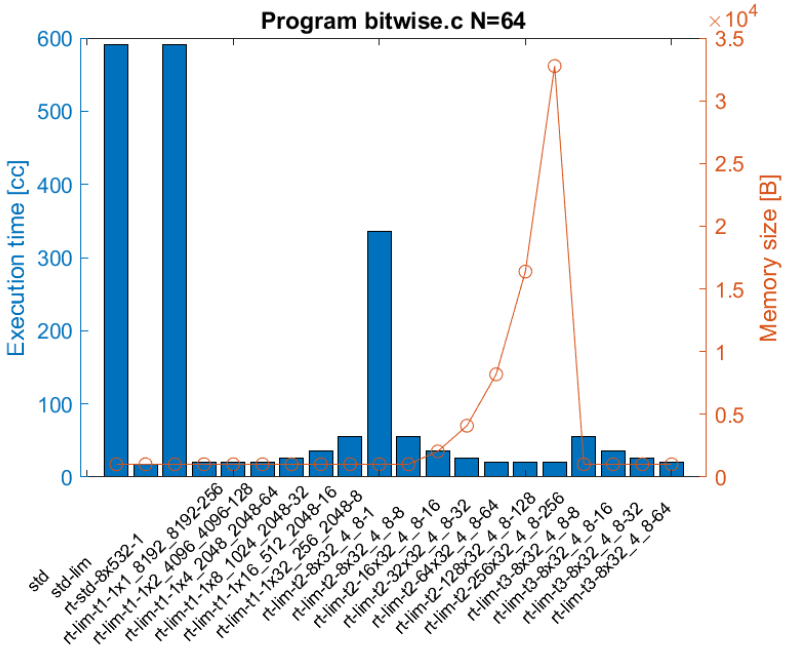

Figure 4.9. Theoretic analysis N=64

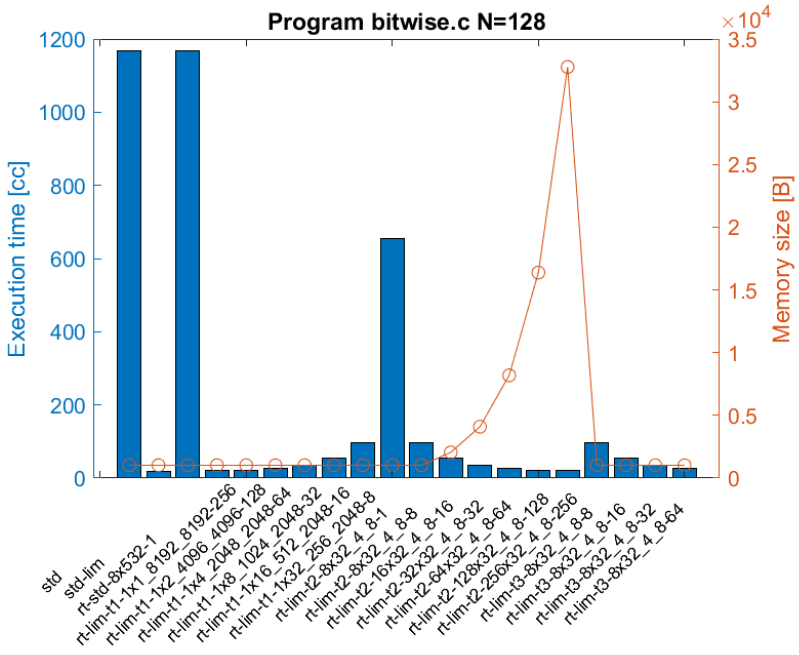

Figure 4.10. Theoretic analysis N=128

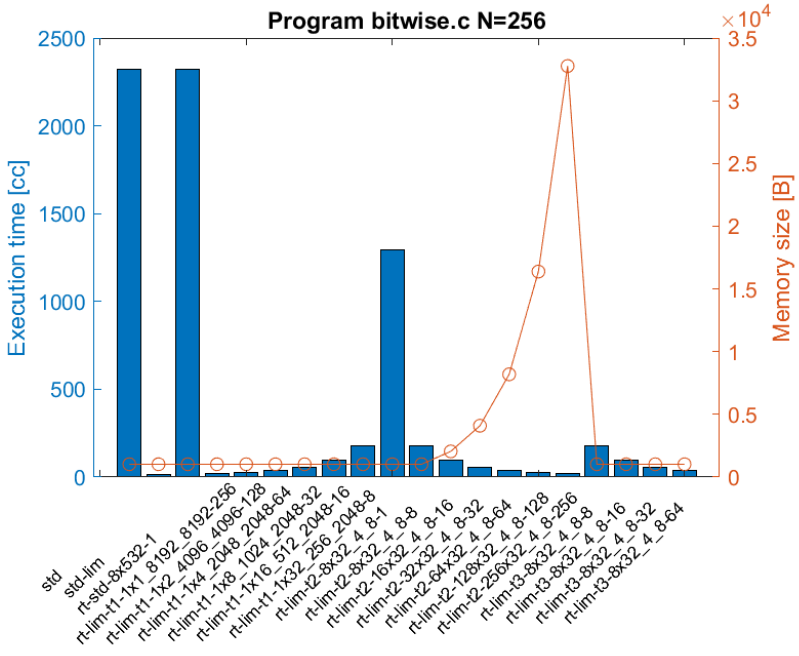

Figure 4.11. Theoretic analysis N=256

4.5.2 Bitmap algorithm

As analyzed in the previous Section, Execution time for this type of algorithm has a linear behaviour. It adopts for-cycles that unfortunately could not be executed with a single range logic store operation. As clear from the histograms, Execution time increases with the number of processed elements N (remember that N is the number of 32-bits vectors). Regardless the internal organization, Racetrack memory with LiM functionalities has a worse performance with respect to all the remaining types, notice that also the Racetrack adopted without LiM functionalities has a better behaviour. The best performance is reached with the original LiM architecture because in that case all the logic operations are performed with a single clock cycle. Racetrack memory with LiM functionalities is slower because the algorithm adopts heavily lw-OR operations, as known this operation is performed internally with an additional latency of one clock cycle.

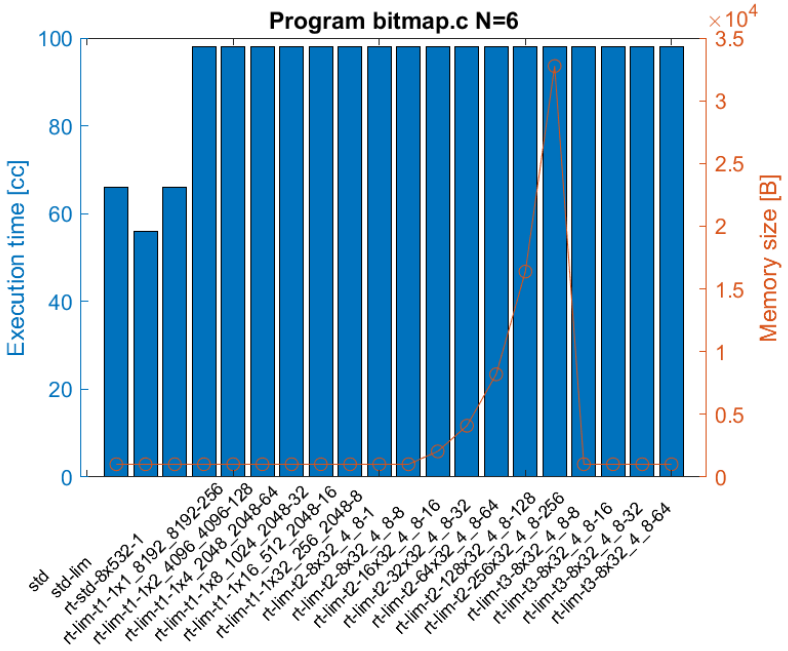

Figure 4.12. Theoretic analysis N=6

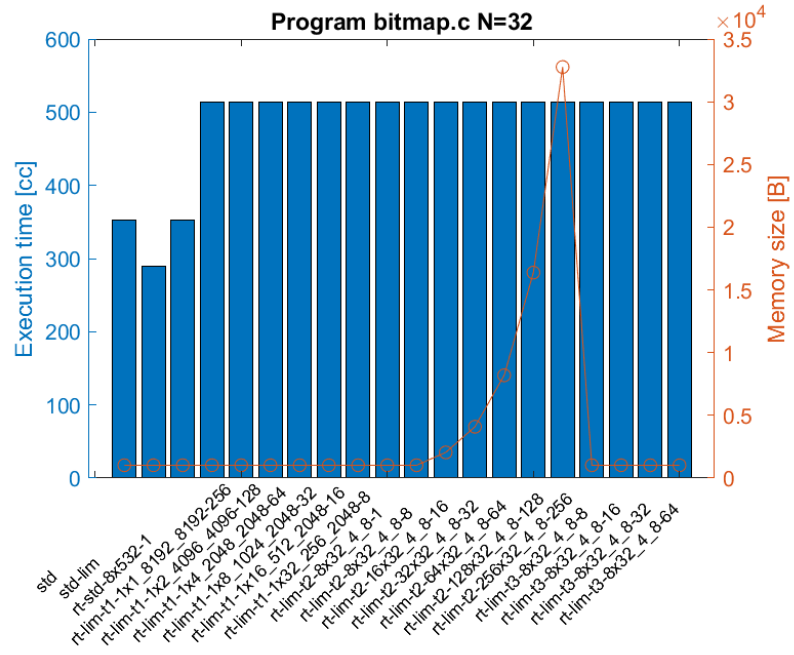

Figure 4.13. Theoretic analysis N=32

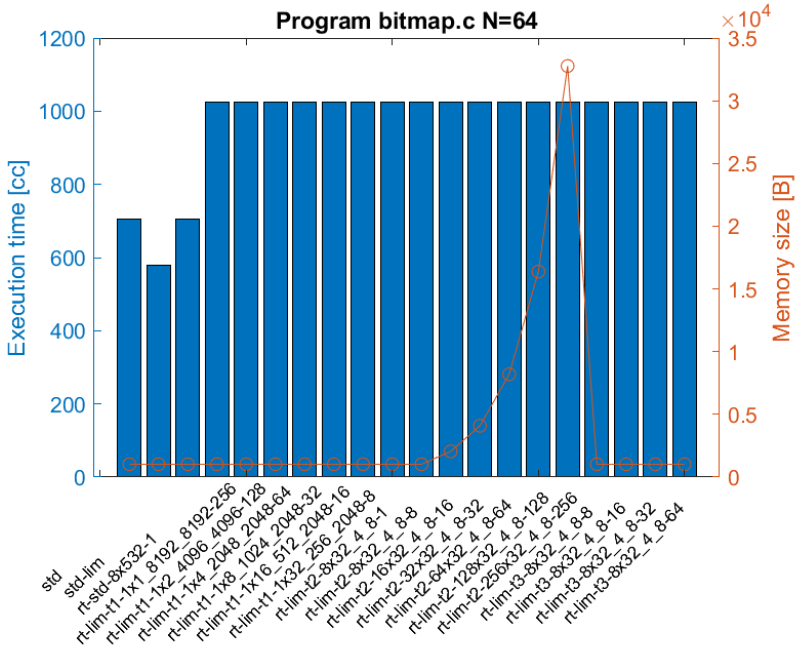

Figure 4.14. Theoretic analysis N=64

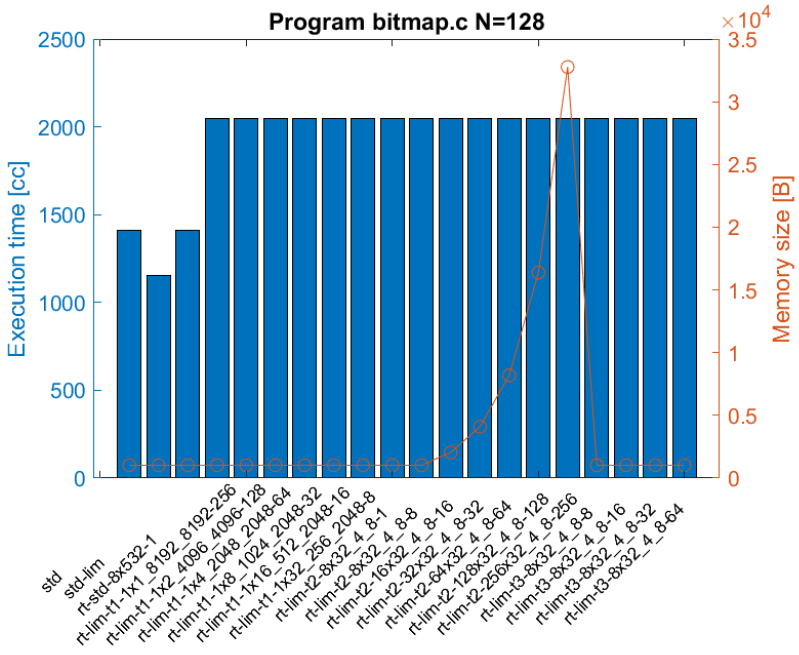

Figure 4.15. Theoretic analysis N=128

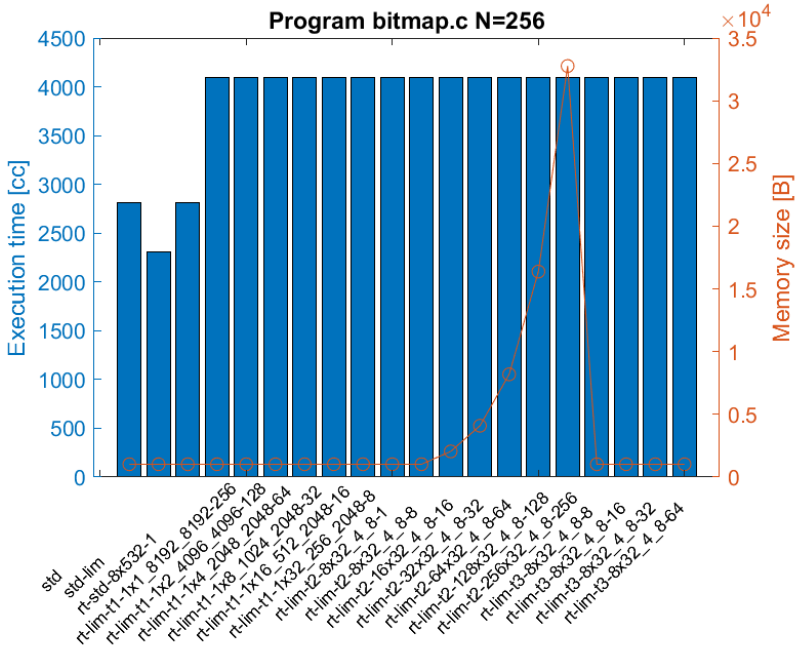

Figure 4.16. Theoretic analysis N=256

4.5.3 AES_128 Addroundkey algorithm

For this algorithm, chunks of 16 data are processed, thus there is no need to show results for different values of N. As in the previous algorithm logic range sw operations cannot be performed directly, because i-th, j-th elements should be loaded in memory and then adopted as mask vector. Fortunately, all these operations exploit logic sw/lw XOR operations, which have the same latency as a standard access.For this reason all the different Racetrack configurations have the same Execution time as the original LiM reference architecture.

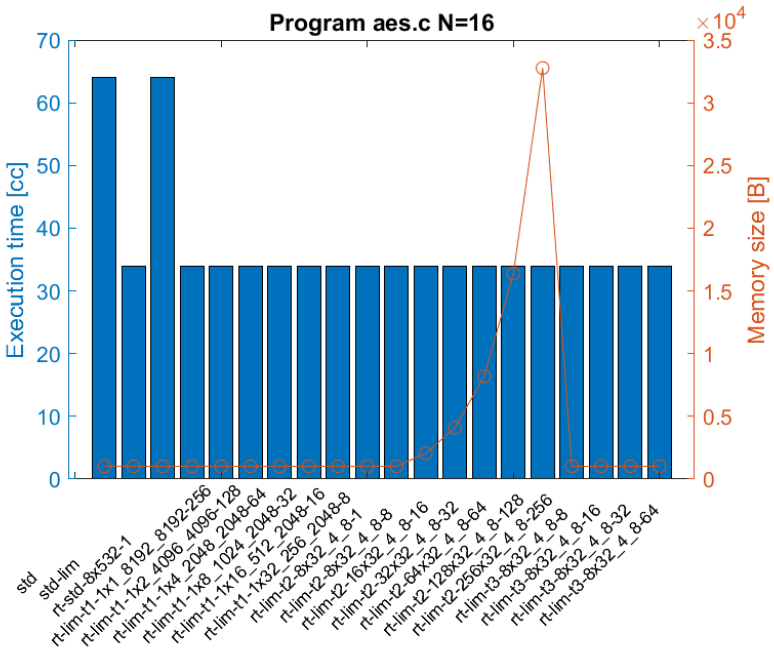

Figure 4.17. Theoretic analysis N=16

4.5.4 Xnor net algorithm

This algorithm exploits the logic range sw feature, results are very clear, configurations with a higher N_{LSP} show better performances because they are able to process multiple words in the same clock cycle. Here N represents the number of column or rows in a NxN matrix, as this parameter increases, the benefits brought by the LiM are more evident.

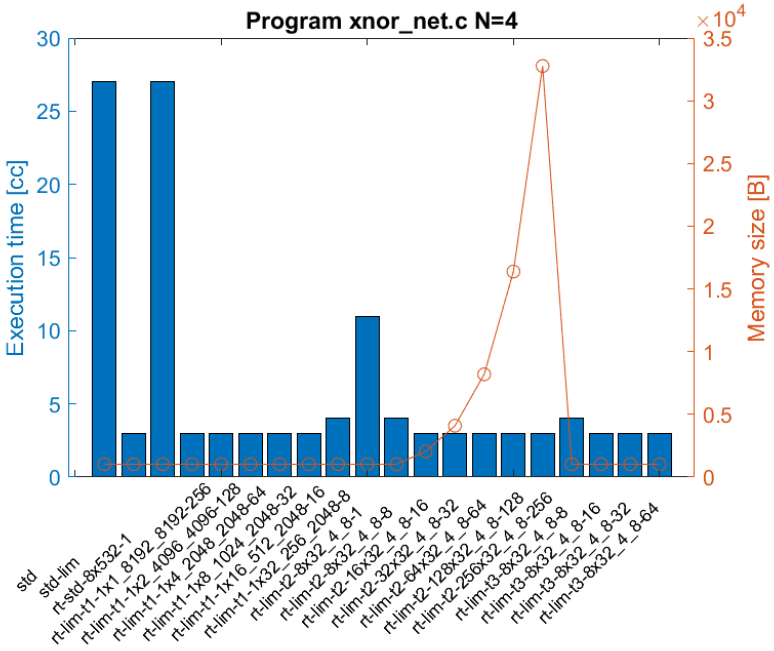

Figure 4.18. Theoretic analysis N=32

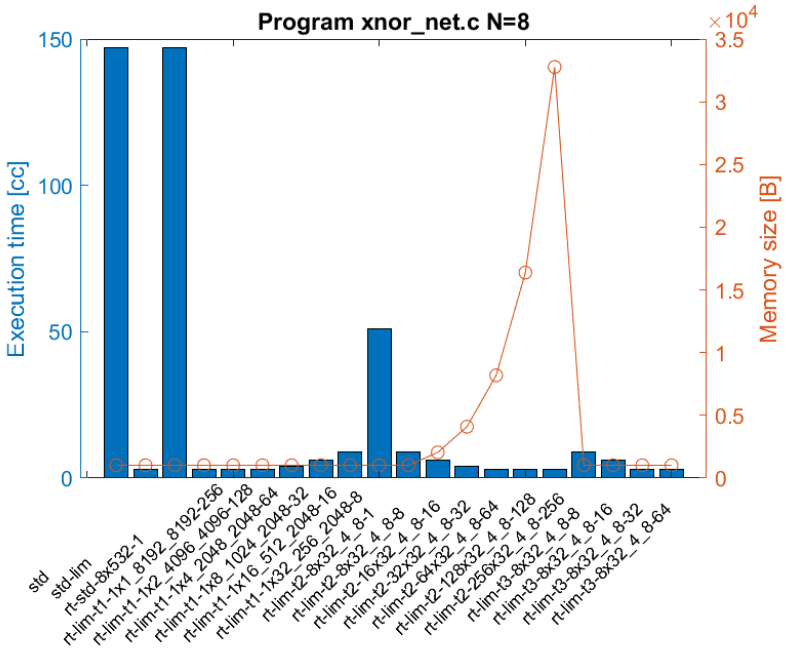

Figure 4.19. Theoretic analysis N=64

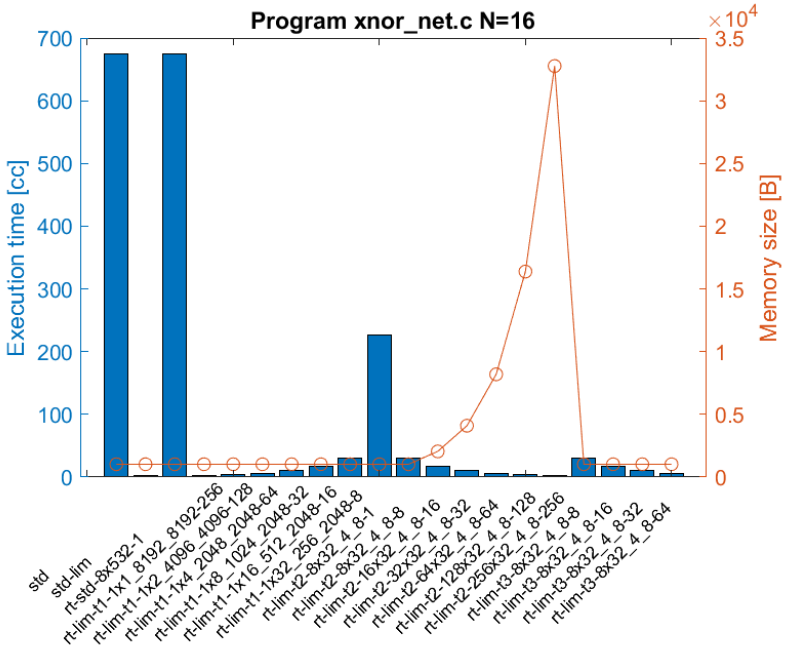

Figure 4.20. Theoretic analysis N=128

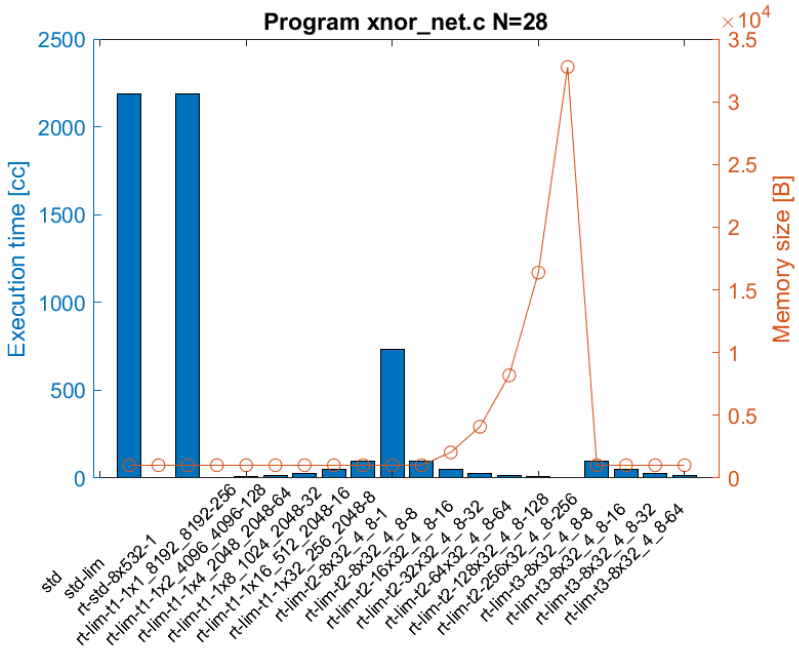

Figure 4.21. Theoretic analysis N=256

Chapter 5

Conclusion and future works

The aim of this Thesis was to implement an open and configurable Logic-in-Memory framework capable to support different types of memory implemented with standard an novel technologies. The Thesis work followed two tasks, the first was to expand the already available LiM structure integrated in a Microprocessor context, the second one was to apply the concept of Logic-in-Memory to a novel technology as Racetrack.
In the first part of the Thesis, the architecture was improved to execute a larger amount of LiM functions, furthermore also the RISC-V GNU GCC compiler was modified to support the definition of new LiM instructions. This makes the architecture expandable and configurable for future improvements and decouples the framework from the hardware necessary to implement the LiM functionalities.
The second part of the Thesis focused on the design of a LiM architecture based on Racetrack technology exploiting the concept of pNML logic. Results are comparable to the original LiM system implemented with standard technology, even if the different access latency of bit-wise LiM operations should be taken into account during the algorithm selection and implementation to speed-up the execution.

Future works should focus on the study of new architectures based on the Racetrack and LiM concepts. This technology allows a high degree of flexibility and this aspect should be investigated to find the best internal organization.
The expansion the available LiM instruction set should be another point to be taken into account. This would require further work on the RISC-V compiler in order to support new and more complex LiM operations.
Furthermore, another focus should be the analysis and comparison of new and more complex algorithms, this would give the possibility to understand the effectiveness of the application of the Racetrack technology in real-case scenarios.

Bibliography

[1] *256Mb ST-DDR3 Spin-transfer Torque MRAM*. EMD3D256M08BS1/16BS1. Revision 1.3. Everspin Technologies Inc. 2018. URL: https://www.everspin.com/family/emd3d256m.

[2] S Agatonovic-Kustrin and Rosemary Beresford. «Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research». In: *Journal of pharmaceutical and biomedical analysis* 22.5 (2000), pp. 717–727.

[3] Benny Akesson. *An introduction to SDRAM and memory controllers*. [2ONAvailable at https://www.es.ele.tue.nl/premadona/files/akesson01.pdf. [Online; accessed 2022-03-15].

[4] Kaya Can Akyel et al. «DRC2: Dynamically Reconfigurable Computing Circuit based on memory architecture». In: *2016 IEEE International Conference on Rebooting Computing (ICRC)*. 2016, pp. 1–8. DOI: 10.1109/ICRC.2016.7738698.

[5] Ieva Antonia. «Speed-up of RISC-V core using Logic-In-Memory operations». Master's Thesis. Politecnico di Torino, 2019-20.

[6] Ieva Antonia. «Studio di gate LiM programmabili sviluppati su tecnologia Racetrack memory». Master's Thesis. Politecnico di Torino, 2019-20.

[7] Robin Bläsing et al. «Magnetic Racetrack Memory: From Physics to the Cusp of Applications Within a Decade». In: *Proceedings of the IEEE* 108.8 (2020), pp. 1303–1321. DOI: 10.1109/JPROC.2020.2975719.

[8] Hao Cai et al. «A survey of in-spin transfer torque MRAM computing». In: *Science China Information Sciences* 64 (June 2021). DOI: 10.1007/s11432-021-3220-0.

[9] Valeria Cardellini. *La gerarchia di memorie*. [Online; accessed 2022-06-01]. URL: http://www.ce.uniroma2.it/courses/aac07/lucidi/Cache%5C%5F2%5C%5F4pp.pdf.

[10] Andrea Coluccio, Marco Vacca, and Giovanna Turvani. «Logic-in-Memory Computation: Is It Worth It? A Binary Neural Network Case Study». In: *Journal of Low Power Electronics and Applications* 10.1 (2020). ISSN: 2079-9268. DOI: 10.3390/jlpea10010007. URL: https://www.mdpi.com/2079-9268/10/1/7.

[11] Andrea Coluccio et al. «Hybrid-SIMD: a Modular and Reconfigurable approach to Beyond von Neumann Computing». In: *IEEE Transactions on Computers* (2021), pp. 1–1. DOI: 10.1109/TC.2021.3127354.

[12] Jim Cook. *Introduction to Flash Memory (T1A)*. [Online; accessed 2022-06-01]. URL: https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2008/20080813%5C%5FT1A%5C%5FCooke.pdf%22.

[13] OPENHW Group. *CVA6*. URL: https://github.com/openhwgroup/cva6.

[14] openRISC Group. *OR1200*. URL: https://github.com/openrisc/or1200.

[15] Roland Höller et al. «Open-Source RISC-V Processor IP Cores for FPGAs – Overview and Evaluation». In: *2019 8th Mediterranean Conference on Embedded Computing (MECO)*. 2019, pp. 1–6. DOI: 10.1109/MECO.2019.8760205.

[16] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, pp. 1–4. ISBN: 0123797519.

[17] Philip Koopman. *Multi-Level Strategies*. [Online; accessed 2022-04-04]. URL: https://users.ece.cmu.edu/~koopman/ece548/handouts/10levels.pdf.

[18] Donghyuk Lee et al. «Tiered-latency DRAM: A low latency and low cost DRAM architecture». In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 2013, pp. 615–626. DOI: 10.1109/HPCA.2013.6522354.

[19] Prerna Mahajan and Abhishek Sachdeva. «A study of encryption algorithms AES, DES and RSA for security». In: *Global Journal of Computer Science and Technology* (2013).

[20] *NAND Flash Controller*. Reference Design RD1055. rd1055_01.2. Lattice Semiconductor Corporation. 2010. URL: https://www.latticesemi.com/-/media/LatticeSemi/Documents/ReferenceDesigns/NR/NANDFlashControllerDesign-Documentation.ashx?document%5C%5Fid=34185.

[21] *Northwest Logic Offers MRAM Controller IP compatible with Everspin's ST-MRAM*. https://www.everspin.com/sites/default/files/pressdocs/Northwest_Logic_and_Everspin_PR.pdf. [Online; accessed 2022-03-20].

[22] David A. Patterson and John L. Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2020, p. 474.

[23] Andrei Pavlov and Manoj Sachdev. *CMOS SRAM circuit design and parametric test in nano-scaled technologies: process-aware SRAM design and test*. Vol. 40. Springer Science & Business Media, 2008, p. 14.

[24] Mohammad Rastegari et al. «Xnor-net: Imagenet classification using binary convolutional neural networks». In: *European conference on computer vision*. Springer. 2016, pp. 525–542.

[25] Fabrizio Riente et al. «Parallel Computation in the Racetrack Memory». In: *IEEE Transactions on Emerging Topics in Computing* 10.2 (2022), pp. 1216–1221. DOI: 10.1109/TETC.2021.3078061.

[26] Hadi R. Sandid. *Adding Custom Instructions to the RISC-V GNU-GCC toolchain*. [Online; accessed 2022-06-02]. URL: https://hsandid.github.io/posts/risc-v-custom-instruction/.

[27] Giulia Santoro. «Exploring New Computing Paradigms for Data-Intensive Applications». PhD thesis. PhD thesis, Politecnico di Torino, 2019.

[28] Jennifer Tran. *Synthesizable DDR SDRAM Controller*. XAPP200. v2.4. XILINX. 2002. URL: http://www.cisl.columbia.edu/courses/spring-2004/ee4340/restricted%5C%5Fhandouts/xapp200.pdf.

[29] Rangharajan Venkatesan et al. «Cache Design with Domain Wall Memory». In: *IEEE Transactions on Computers* 65.4 (2016), pp. 1010–1024. DOI: 10.1109/TC.2015.2506581.

[30] Jingcheng Wang et al. «A 28-nm Compute SRAM With Bit-Serial Logic/Arithmetic Operations for Programmable In-Memory Vector Computing». In: *IEEE Journal of Solid-State Circuits* 55.1 (2020), pp. 76–86. DOI: 10.1109/JSSC.2019.2939682.

[31] Wikipedia contributors. *Cache inclusion policy*. [Online; accessed 2022-04-05]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Plagiarism%5C&oldid=5139350.

[32] Ming-Chuan Wu and A.P. Buchmann. «Encoded bitmap indexing for data warehouses». In: *Proceedings 14th International Conference on Data Engineering*. 1998, pp. 220–230. DOI: 10.1109/ICDE.1998.655780.

[33] Kai Yang, Robert Karam, and Swarup Bhunia. «Interleaved logic-in-memory architecture for energy-efficient fine-grained data processing». In: *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2017, pp. 409–412. DOI: 10.1109/MWSCAS.2017.8052947.

[34] Hongbin Zhang et al. «Performance analysis on structure of racetrack memory». In: *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. 2018, pp. 367–374. DOI: 10.1109/ASPDAC.2018.8297351.

[35] Xinmiao Zhang and K.K. Parhi. «High-speed VLSI architectures for the AES algorithm». In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 12.9 (2004), pp. 957–967. DOI: 10.1109/TVLSI.2004.832943.