POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

Parallel architectures for Processing-in-Memory

Relatori:
Prof. Maurizio Zamboni
Prof. Mariagrazia Graziano
Ph.D. Giovanna Turvani

Candidata:
Milena Andrighetti

Dicembre 2018

Table of contents

List of tables

List of figures

Chapter 1

Introduction

The *Von Neumann* architecture is one of the most widely spread nowadays. It is composed of a CPU, a storage for data and instructions and interconnections.

However, with the fulfillment of Moore's Law, CPUs reached their full potential, but it cannot be said the same about memory performance, that stayed behind, even with the advancement of transistor scaling (figure 1.1). This is called *Von Neumann bottleneck* or *Memory Wall*. As a consequence new approaches and technologies emerged to solve the issue. One of them is *Logic-in-Memory* (LIM).

The *Logic-in-Memory* approach (also often referred to as *Processing-in-Memory* (PIM)) aims to merge the logic and memory elements of a traditional structure, in

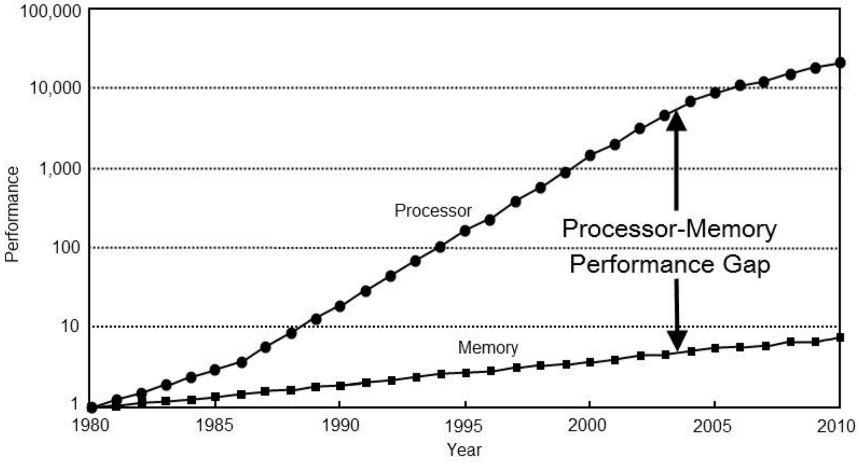


Figure 1.1: CPU vs Memory performances over the years. From [27]

order to reduce the memory bandwidth limitation caused by the fact that common applications are more and more data intensive and the data movement from memory to CPU and back is way too costly.

The PIM approach is suitable for all that kinds of applications that require simple operations, since they would be easier to integrate in the memory array

A lot of researchers already introduced their vision of Logic-in-Memory, acting either from the technological point of view or from the architectural one. Few of them will be reported in the next chapter as state of the art, in order to provide and overview of the already existing proposal and of the promising potential of the Logic-in-Memory paradigm.

Chapter 2

State of the Art

2.1 Magnetic Tunnel Junction

2.1.1 MTJ Basics [1, 2, 3]

Magnetic tunnel junction (MTJ) is a nano-structure composed of three layers. A metal-oxide film is trapped between two ferromagnetic (FM) metals. The structure is represented in figure 2.1.

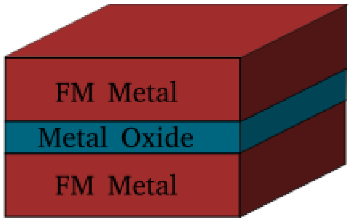


Figure 2.1: Magnetic Tunnel Juction structure

The insulator is thin enough to let the electrons transit from one metal to another, through tunneling, when an appropriate magnetic field is applied on the junction.

The current generated from this effect is proportional to the product of electrodes density of states at Fermi level. This is because in ferromagnetic materials, the ground-state energy bands in the vicinity of the Fermi level are shifted in energy, resulting in separate majority and minority bands for electrons with opposite spins.

Assuming *spin conservation* for the tunnelling electrons, there are two parallel currents: spin-up and spin-down currents.

The magnitude of the tunnelling current depends on the relative orientation of the magnetization of both electrodes. The reason lies in the fact that for aligned magnetization, electrons around the Fermi level are allowed to transit from minority to minority bands and from majority to majority. If the alignment is anti-parallel the transition takes place from majority to minority and vice versa. This results in a higher current for the first case and lower current for the second. In terms of electrical resistance, this corresponds to a low or high resistance, respectively.

This resistance is called Tunnel Magneto Resistance defined as:

$$TMR = \frac{2P_1P_2}{1 - P_1P_2} = \frac{R_{AP} - R_P}{R_P} \tag{2.1}$$

Where P_1 and P_2 are the spin polarization of the two magnetic layers and R_P and R_{AP} are the resistance for parallel and anti-parallel magnetization configuration between the two ferromagnets, respectively.

The digital information is coded by the resistance of the junction: since, according to the relative orientation of the magnetizations, the resistance varies, it is possible to code low resistance as logic '0' and high resistance as logic '1'. It is observed that transitions between parallel and anti-parallel states present an hysteresis trend. The behaviour of MTJ is reported in figure 2.2.

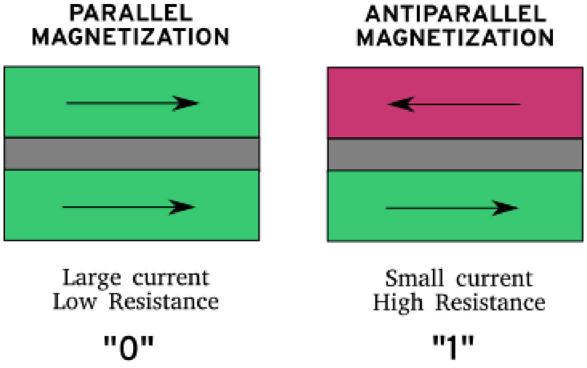


Figure 2.2: Behaviour of Magnetic Tunnel Junction

Magnetic tunnel junctions are widely used in magnetic memories, like *Magnetic Random Access Memory* (MRAM).

The basic MRAM cell is composed of a MTJ connected to a MOS transistor that is used to enable or disable the current flow through the junction.
To make the storage of information programmable, one of the two ferromagnetic layers of the MTJ is set on a fixed magnetization direction and it is called FIXED LAYER. The other one is kept free to rotate its orientation (FREE LAYER) according to external stimuli, like a magnetic field or a current.
Each MTJ cell is controlled by three electrical signals on metal lines, that carry the same names used in a conventional memory structure, i.e. bit line, word line and source line (figure 2.3).

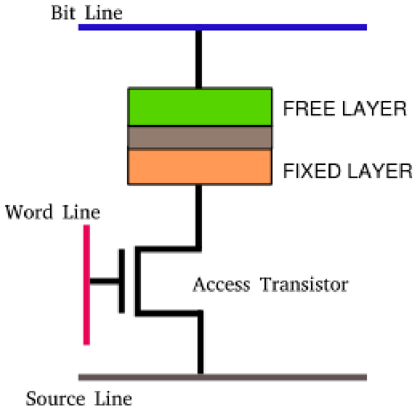


Figure 2.3: MTJ cell structure

Since the information is coded with a variable resistance, reading the cell consists in measuring resistance. On the other hand, writing an MTJ cell can be performed in different ways, resulting in different kinds of MRAMs:

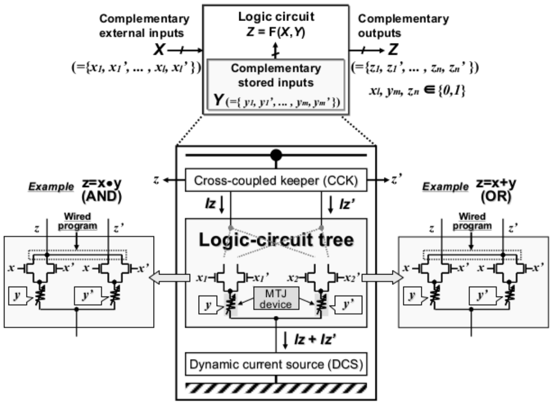
- **FIMS MRAM** (*Field Induced Magnetic Switching*): writing is implemented by the means of a magnetic field generated by current lines close to the junction. If the current densities are strong enough, the magnetic field generated will switch the magnetization of the free layer.

- **TAS MRAM** (*Thermal Assisted Switching*): an additional layer, made of Anti-ferromagnetic material, is used to pin the magnetization of the free layer. A current applied through the stack heats the junction above a critical temperature, freeing the free layer, which can then be easily switched by an applied external field.

- **CIMS MRAM** (*Current Induced Magnetic Switching*): it is based on the Spin Transfer Torque effect. When a spin-polarized current is applied to a ferromagnetic material, a torque on magnetization is exerted. If the current density is high enough this could result in magnetization direction switching. This type of memory is also called STT-MRAM.

Exploiting the MTJ technology (potentially mixed with traditional CMOS transistors) it is possible to implement a large selection of logic circuits, ranging from simple logic gates to Non-volatile LUT or flip-flops, resulting in the generation of more complex systems, such as Magnetic FPGAs.

2.1.2 MTJ-based non-volatile Logic in Memory Circuit, Future Prospects and Issues [4]

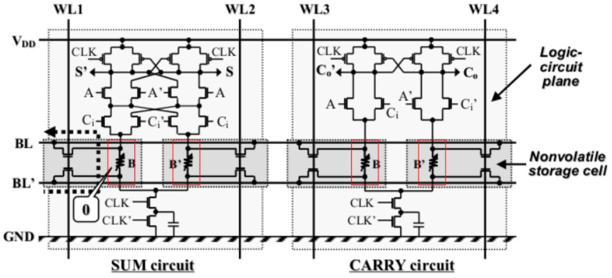

Complementary external inputs
X
(={x1, x1', ... , xl, xl' })
Logic circuit
Z = F(X,Y)
Complementary stored inputs
Y (={ y1, y1', ... , ym, ym' })
Complementary outputs
Z
(={z1, z1', ... , zn, zn' })
xl, ym, zn ∈ {0,1}
Example z=x•y (AND)
Wired program
z
z'
x
x'
y
y'
Cross-coupled keeper (CCK)
Iz
Iz'
Logic-circuit tree
x1
x1'
x2
x2'
MTJ device
Iz + Iz'
Dynamic current source (DCS)
Example z=x+y (OR)

S. Matsunaga et al., 2009

Figure 2.4: Proposed structure of a MTJ-based LiM circuit. From [4]

In figure 2.4 is possible to observe the proposed structure.
The entity is basically composed of three units: the dynamic current source (DCS) that allows to cut off steady current from V_{DD} to GND, which results in low-power dissipation; a logic-circuit tree, where arbitrary logic circuits are realized according to its configuration; a Cross-Coupled-Keeper(CCK) that generates complementary binary outputs congruently to a magnitude comparison result between two current signals (Iz and Iz').

This structure is then applied to a LiM Full Adder, whose architecture is depicted in figure 2.5.



S. Matsunaga et al., 2009

Figure 2.5: Structure of a Logic-in-Memory Full Adder. MTJ elements are highlighted by the red rectangles. From [4]

The stored data is programmed by external signals. Selecting the word lines, is possible to program the complementary inputs B and B'. During this write operation all external inputs and clock signals are switched off.

The dynamic logic, controlled by the complementary clock signals, allows to cut off the steady current flow from the supply voltage to ground, reducing the dynamic power dissipation of the circuit.

Moreover, since the stored data is contained in non-volatile elements (MTJs), it is possible to cut off the supply voltage, maintaining stored data in standby state. This eliminates the static power dissipation.

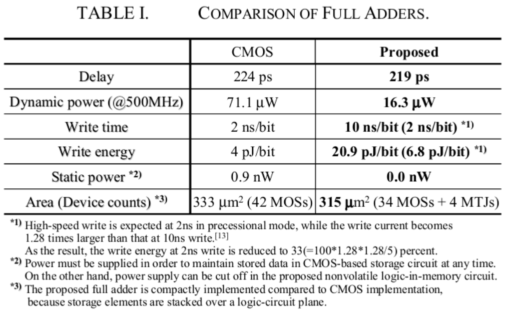


TABLE I. COMPARISON OF FULL ADDERS.

	CMOS	**Proposed**
Delay	224 ps	**219 ps**
Dynamic power (@500MHz)	71.1 μW	**16.3 μW**
Write time	2 ns/bit	**10 ns/bit (2 ns/bit)** *1)
Write energy	4 pJ/bit	**20.9 pJ/bit (6.8 pJ/bit)** *1)
Static power *2)	0.9 nW	**0.0 nW**
Area (Device counts) *3)	333 μm^2 (42 MOSs)	**315** μm^2 (34 MOSs + 4 MTJs)

*1) High-speed write is expected at 2ns in precessional mode, while the write current becomes 1.28 times larger than that at 10ns write.[13]
As the result, the write energy at 2ns write is reduced to 33(=100*1.28*1.28/5) percent.
*2) Power must be supplied in order to maintain stored data in CMOS-based storage circuit at any time.
On the other hand, power supply can be cut off in the proposed nonvolatile logic-in-memory circuit.
*3) The proposed full adder is compactly implemented compared to CMOS implementation, because storage elements are stacked over a logic-circuit plane.

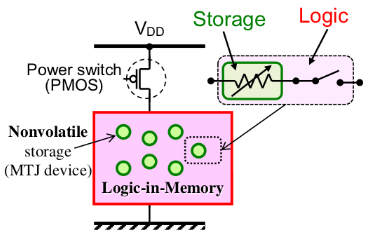
S. Matsunaga et al., 2009

Figure 2.6: Comparison of Full Adders with CMOS-only and MTJ-LiM implementation. This table was extracted from the study under consideration. From [4]

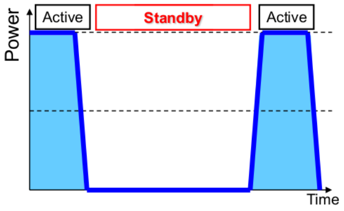
In conclusion, it is demonstrated (figure 2.6) that the proposed structure is characterized by a great reduction in area and power dissipation if compared with the corresponding CMOS-only implementations. As far as delay is concerned an improvement is observed.

2.1.3 Challenge of MTJ-Based Nonvolatile Logic-in-Memory Architecture for Dark-Silicon Logic LSI [5]

It is proposed a non-volatile logic-in-memory (NV-LIM) architecture to solve performance-wall and power-wall problems in the present CMOS-only-based logic-LSI processors(all contents and figures of this section are thus extracted from [5]).
The choice fell on MTJ devices with spin-injection write capability.
MTJ acts either as storage element, either as logic element, because its resistance is two-way programmable in accordance with its stored status.
The non-volatile characteristic of MTJ allows to maintain the data stored even when the power is cut-off. This leads to the possibility to apply power-gating technique to the circuitry, achieving low-power logic LSI. The basic architecture is shown in figure 2.7.



(a)

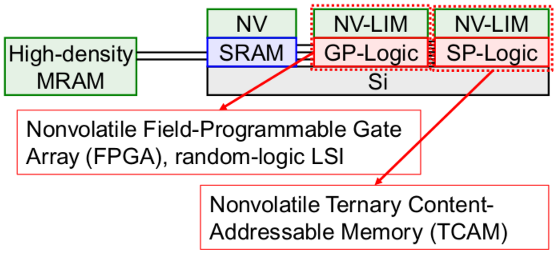


(b)

T.Hanyu et al., 2013

Figure 2.7: MTJ-based NV-LIM architecture: (a) hardware structure with a power-gating capability; (b) power-gating efficiency using the NV-LIM architecture. From [5]

The focus is then moved to VLSI processors. A ”first generation” of NV-VLSI is first described, where high-density MRAM replaces the standard Flash and DRAM memories and also on-chip memories and flip-flops are replaced to NV ones.
To further improve performance of NV-VLSI processors, a ”second-generation NV logic-LSI architecture” is introduced. It consists in merging a part of NV on-chip memory with logic circuit modules (figure 2.8).


High-density
MRAM
NV
SRAM
NV-LIM
GP-Logic
NV-LIM
SP-Logic
Si
Nonvolatile Field-Programmable Gate
Array (FPGA), random-logic LSI
Nonvolatile Ternary Content-
Addressable Memory (TCAM)

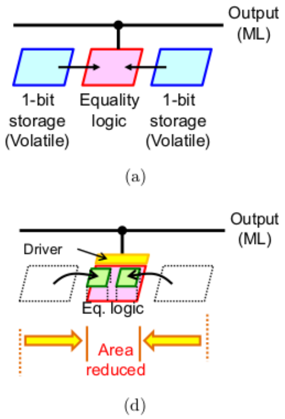
T.Hanyu et al., 2013

Figure 2.8: Second generation NV logic-LSI architecture. From [5]

As next step, NV-LIM components are described in detail.

- **MTJ-based NV TCAM**

  Ternary Content Addressable Memory is a high-speed memory that performs fully parallel search and comparison between stored data and input key. Other than '0' and '1', also the value 'X' (don't care) is included, which makes searching data more flexible. However, conventional CMOS-only implementation requires high cost in terms of complex logic and power dissipation.
  For this reason, a NV MTJ implementation is proposed (figure 2.9).



(a)

(d)

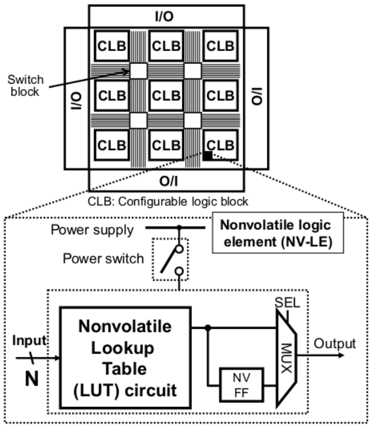
T.Hanyu et al., 2013

Figure 2.9: (a) Conventional TCAM cell structure; (d) Proposed NV-TCAM cell structure. From [5]

  The proposed solution merges storage elements with the logic part, achieving area optimization, since CMOS-based implementation needs 12 MOS transistor while the proposed one takes just 4 MOS and two MTJ devices.
  It is important to notice that MTJ do not affect the total TCAM cell circuit because MTJs are fabricated onto the CMOS plane. The advantage of the compact realization due to NV-LIM architecture can improve the performance of the circuit by inserting a driver (figure 2.9 (d)).
  With this structure it is possible to implement a three-segment based NV TCAM that shows an average activation ratio as low as 2.8%. This implies that about 97% of the TCAM cells can be in standby mode thanks to the fine-grained power gating.

- **MTJ-based NV FPGA**

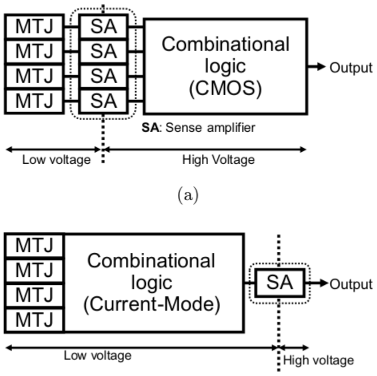
Field programmable gate arrays are widely used in realizing prototyping systems thanks to their changeable configuration according to user needs. However, power consumption and high hardware cost are serious problems in enlarging application fields of FPGAs.
A possible solution to the excessive power consumption is the employment of MTJ devices, obtaining a NV FPGA (figure 2.10) in which each configuration block (CLB) is composed of NV-LUT where configuration data is stored in non-volatile storage element, in the present case, MTJs.
This way, whenever a LUT circuit is in a standby mode, its power supply can be shut down, which completely eliminates the wasted standby power dissipation.


I/O
CLB
Switch block
I/O
I/O
O/I
CLB: Configurable logic block
Power supply
Nonvolatile logic element (NV-LE)
Power switch
SEL
Input
N
Nonvolatile Lookup Table (LUT) circuit
MUX
Output
NV FF

T.Hanyu et al., 2013

Figure 2.10: Overall structure of a NV FPGA. From [5]

However, only the simple replacement of volatile storage elements with NV ones, increases the hardware cost of the circuit. For this reason MTJs are merged with combinational logic in the LUT circuit using the NV-LIM architecture (figure 2.7). Moreover, in this configuration only one sense amplifier is required. This results in a highly compact LUT circuit.



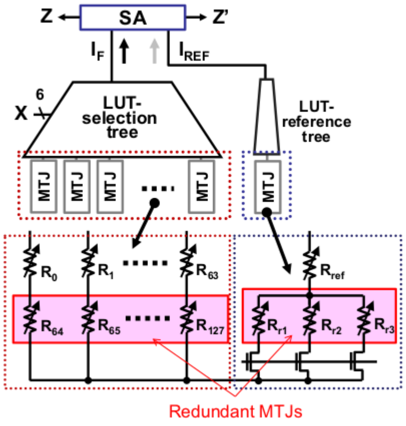
(a)

(b)

T.Hanyu et al., 2013

Figure 2.11: NV LUT circuit: (a) Conventional approach; (b) proposed NV-LIM architecture based approach. From [5]

In FPGAs, LUTs must have four inputs or more. With multiple MTJs, this could be problematic in terms of resistance values. To obtain a stable LUT, it is possible to add additional MTJs to adjust the operating point of the LUT function. In figure 2.12 an example of multi-input NV-LUT circuit is shown.



T.Hanyu et al., 2013

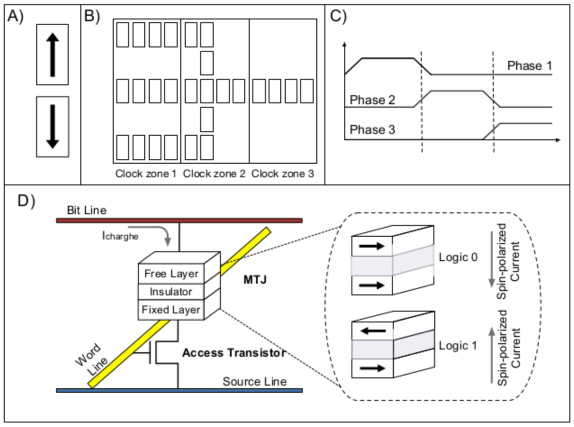
Figure 2.12: Resistance-variation compensation technique using redundant MTJ devices. Twice the number of MTJs are placed in the LUT selection-tree and three additional MTJs are placed in LUT reference-tree. From [5]

The proposed NV-LIM based NV-LUT circuit result more compact with respect to conventional structure even with the increase of the number of inputs.

2.1.4 Design of MRAM-Based Magnetic Logic Circuits [6]

This work presents a new way to implement NML circuits. All the knowledge and figures here exposed are extracted from [6].
The idea is to exploit Magnetic RAM to generate the same logic circuit realised with NML technology.
In Nano Magnet Logic technology, bistable single domain nano-magnets are used to represent digital values.
Since nano-magnets are naturally both logic and storage elements they are the perfect candidate to implement logic-in-memory circuits.
However, even if NML fundamentals were experimentally demonstrated,i.e the clock system and logic gates, a complete NML circuit is not demonstrated yet. MRAM technology, instead, has undergone fast development and evolution and it is currently ready for the commercial stage.

NML basic principle is the transmission of information through magnetodynamic interaction between neighbouring magnets (figure 2.13).



G.Turvani et al., 2017

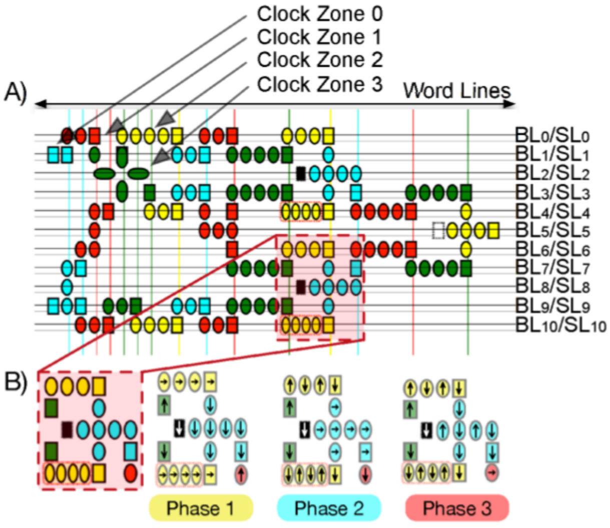
Figure 2.13: NML fundamentals. From [6]

To reproduce the same concept the MRAM (in particular, here STT-RAM is used) is modified:

– Distances among MTJs are reduced so that the free layer can be influenced not only by the current flowing through it, but also by the magnetic field generated by neighbouring MTJs;

– A particular type of MTJ is used: the magnetization of the fixed layer is tilted of 45°. This is done to reduce the influence of the fixed layer on the free one, maintaining at the same time a sufficient difference between the resistance among the two logic states.

Then, to create the same magnetic signal patterns as in the NML circuit, MTJs are removed from specific areas of the MRAM array.
An example of this technology is shown in figure 2.14.

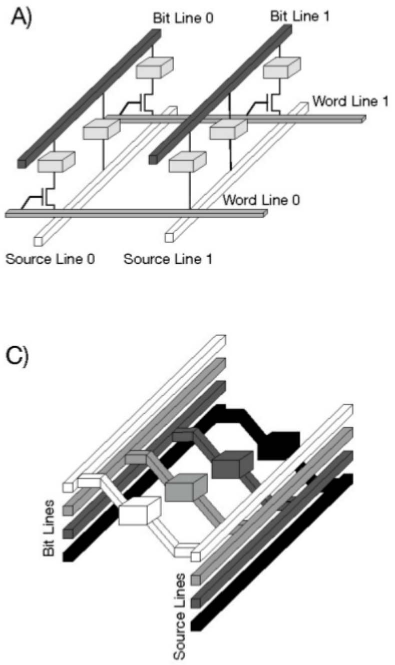


G.Turvani et al., 2017

Figure 2.14: 2-bit XOR circuit implemented with MTJ Bounded layout. (A) Complete circuit layout configuration: Horizontal stripes are Bit-Lines and Source-Lines. Vertical stripes are Word-Lines. (B) Detail and time evolution of a small part of the XOR gate. From [6]

Each MTJ is controlled by word lines (one for each column) and bit and source lines (couple for each row). MTJs are arranged in small groups, identified by different colors. Each color labels a clock zone. The separation of circuits in clock zones is required to control signals propagation and avoid errors.
Since a multiphase clock system is needed in NML circuits to operate properly, the system has to be reproduced using MTJs. However, MTJs placed on a specific row

belong to different clock zones, but they share the same source and bit lines. This fact is in contradiction with the multiphase clocking requirements, because in every clock zone MTJs must be driven independently. The solution of this problem is to exploit the presence of transistors connected in series with every MTJs, that are controlled by the word lines. So, it is possible to enable or disable the current flow through the devices.
However, the constraint imposed by the structure of the MRAM, which implies a rigid placement for the MTJs led to the formulation of a second structure, called *free layout*. The traditional one instead was called *bonded layout*. The new structure requires a change in the technological structure, but comes with no limitations on MTJs placement. This topology has only two control signals and does not require transistors, except for external read and write operations and to generate clock signals. Word lines are not needed anymore (figure 2.15).

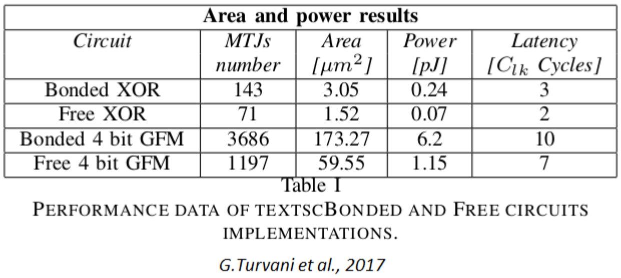


G.Turvani et al., 2017

Figure 2.15: Structure comparison: (A) Bonded Layout; (C) Free Layout From [6]

This way, each clock zone is independently controlled and there are no limitation on MTJs placement, which brings to circuits improvement. The negative aspect is

that this solution requires to modify the fabrication process of the MRAM and its feasibility is not yet demonstrated.
To compare the two different structures, two circuits were implemented: 2-bit XOR and Galois multiplier. Results obtained are illustrated in figure 2.16.



Area and power results				
Circuit	*MTJs number*	*Area* [μm^2]	*Power* [pJ]	*Latency* [C_{lk} Cycles]
Bonded XOR	143	3.05	0.24	3
Free XOR	71	1.52	0.07	2
Bonded 4 bit GFM	3686	173.27	6.2	10
Free 4 bit GFM	1197	59.55	1.15	7

Table I
PERFORMANCE DATA OF TEXTSCBONDED AND FREE CIRCUITS IMPLEMENTATIONS.

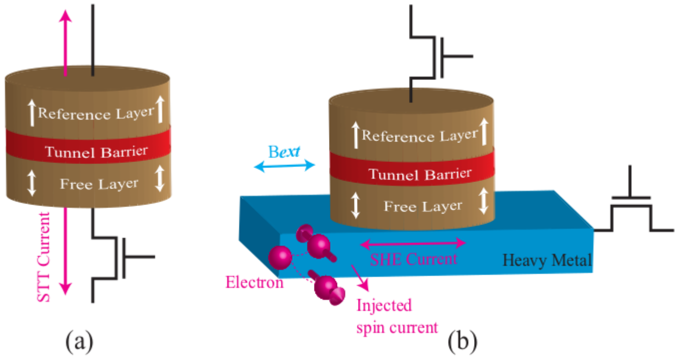
G.Turvani et al., 2017

Figure 2.16: Comparison between the two layout configurations. Table evinced directly from the study under consideration. From [6]

It is possible to observe that free layout presents better results in all the analysed fields but it is the bonded layout that is successful in terms of feasibility, while for the free layout the matter is still to be looked into.
This demonstrates that there is space for improvements and further investigation for this technology.

2.1.5 Reconfigurable Procesing in Memory Architecture Based on Spin Orbit Torque [7]

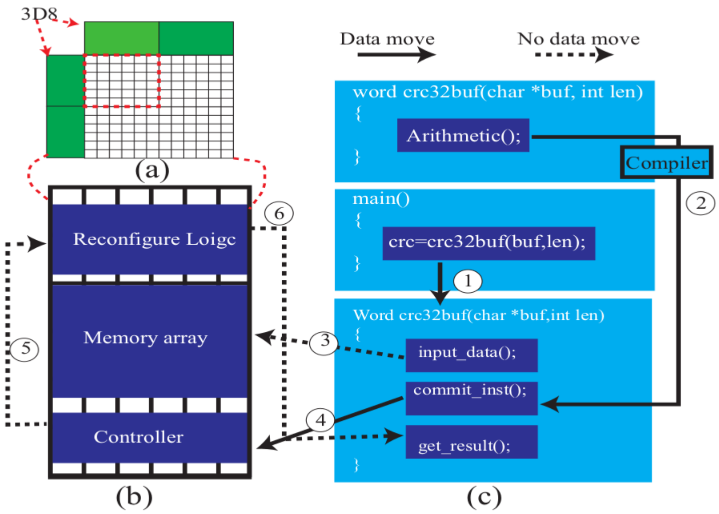
Differently from the STT-RAM, in SOT-RAM the storage element is based on a MTJ above a heavy metal film, resulting in a three terminal device. Write operation is performed letting current flowing through the heavy metal film, then the free layer is able to switch thanks to the spin hall effect. (figure 2.17).



L. Chang et al., 2017

Figure 2.17: (a) STT-MRAM cell (b)SOT-MRAM cell. From [7]

The proposed architecture, called PISOTM (figure 2.18), is composed of SOT-memory, reconfigurable SOT-logic and a controller.
For the SOT-logic, the memory array is partitioned in different logic clusters controlled by several separetd 3 to 8 decoders.



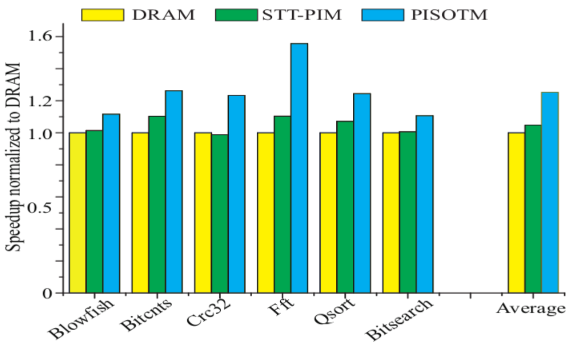
```
word crc32buf(char *buf, int len)
{
    Arithmetic();
}
main()
{
    crc=crc32buf(buf,len);
}
Word crc32buf(char *buf,int len)
{
    input_data();
    commit_inst();
    get_result();
}
```

L. Chang et al., 2017

Figure 2.18: PISOTM Architecture and compile operation. From [7]

The advantage of this structure is that the storage and logic elements are identical, this way any technology conflict is avoided.
Moreover, in SOT-logic there is no data movement, since the required data is already in the memory array.

To evaluate the structure Cadence, modified Multi2Sim and MlBench are used. Results are compared with DRAM and STT-based PIM. The average speedup improvement of PISOTM is about 30% (figure 2.19).



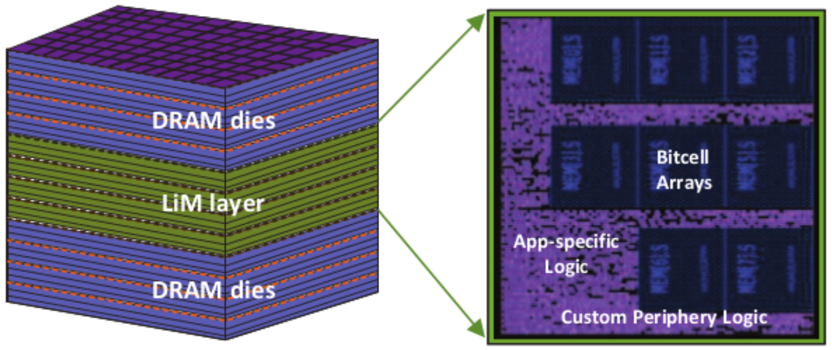
L. Chang et al., 2017

Figure 2.19: Simulation results. From [7]

2.2 3D-Stacking

2.2.1 A 3D-Stacked Logic-in-Memory Accelerator for Application-Specific Data Intensive Computing [8]

Data Intensive Computing applications need high bandwidth to achieve good performance due to their inefficient memory access patterns.
To overcome the memory wall problem 3D-DRAM technology is used. It provides low latency and high bandwidth. The proposed structure is composed of 3D-stacked DRAM and an highly-specialized LiM layer.
The 3D-Lim system is showed in figure 2.20.



(a) 3D LiM Stack: DRAM layers and LiM layers are stacked vertically and communicate with TSVs

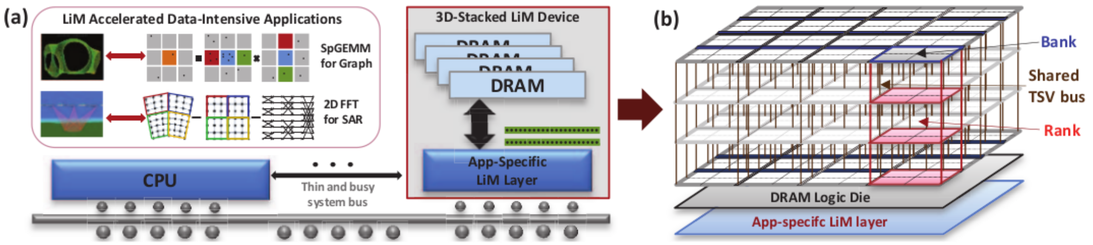
(b) LiM Layer: memory and logic cells are tightly integrated and customized for a particular application

Q.Zhu et al., 2013

Figure 2.20: 3D-Stacked Logic-in-Memory System. From [8]

The overall architecture is depicted in figure 2.21(a). The 3D-LiM device is accessed by the CPU exploiting the standard DRAM interface. The LiM layer is designed to process logic-simple parts of a data-intensive problem in the most efficient way. The data elaborated in the LiM layer are then transferred to the CPU for high-level interpretation. This kind of operations is less memory-bound, this way the traffic on the data bus is greatly reduced.

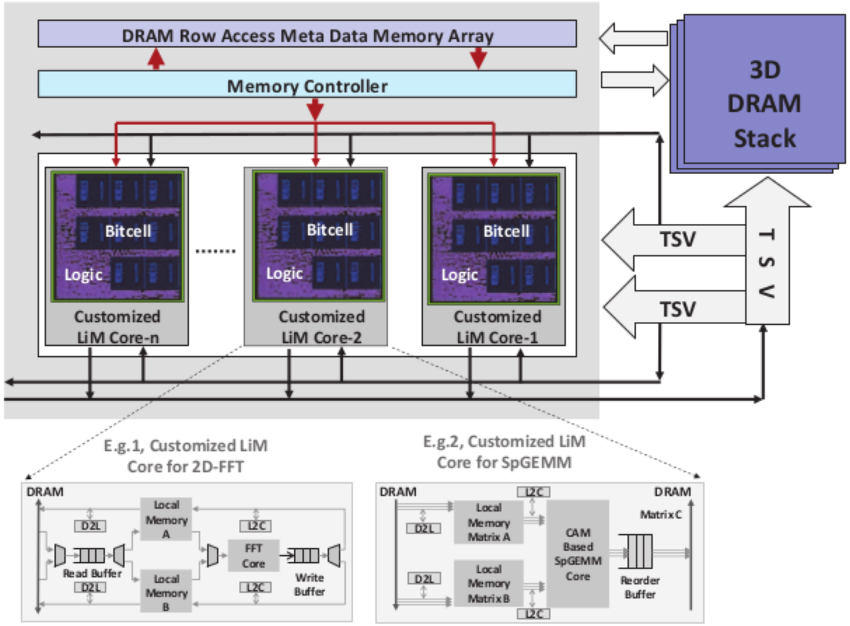
In (b) is represented the structure of the 3D-stacked device. Each DRAM layer is partitioned in banks and a pile of banks form a rank. Vertical connections are formed by dense,short and fast TSV buses, that are capable of transfering a whole DRAM row buffer in a few clock cycles.



Q.Zhu et al., 2013

Figure 2.21: (a) Overall architecture; (b) Fine-grained 3D-stacked DRAM. . From [8]

To understand the structure of the LiM core better, figure 2.22 is provided.

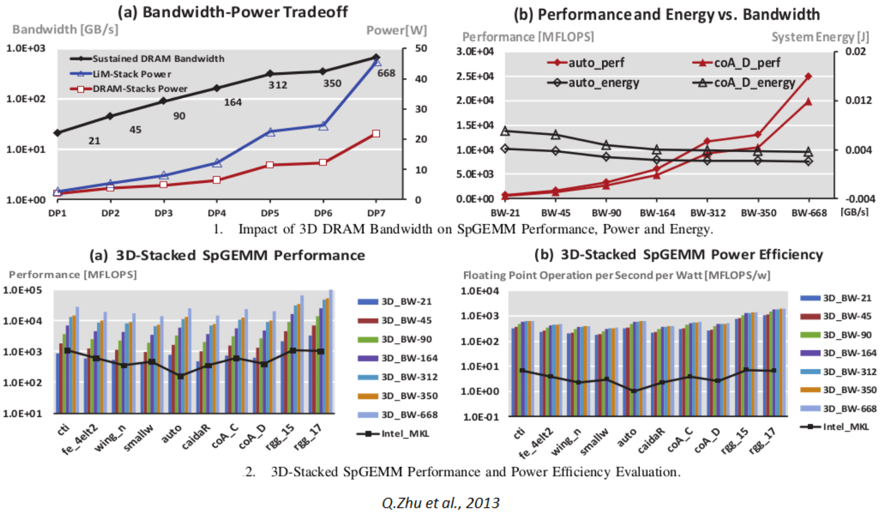


Q.Zhu et al., 2013

Figure 2.22: 3D-Stacked LiM Functional Diagram. . From [8]

The Memory Controller is dedicated for communication among the other core components. The Meta Data Memory Array stores the information which maps the blocks of data to DRAM rows, providing data format, block size and number of non-zero elements. The LiM cores are designed based on a specific application.

This architecture has been tested for both dense and sparse application, 2D-FFT and SpGemm respectively. The 3D-Stacked structure was modelled and simulated with CACTI-3DD and HDL simulations. Results of SpGemm simulations are shown in figure 2.23.



1. Impact of 3D DRAM Bandwidth on SpGEMM Performance, Power and Energy.

2. 3D-Stacked SpGEMM Performance and Power Efficiency Evaluation.

Q.Zhu et al., 2013

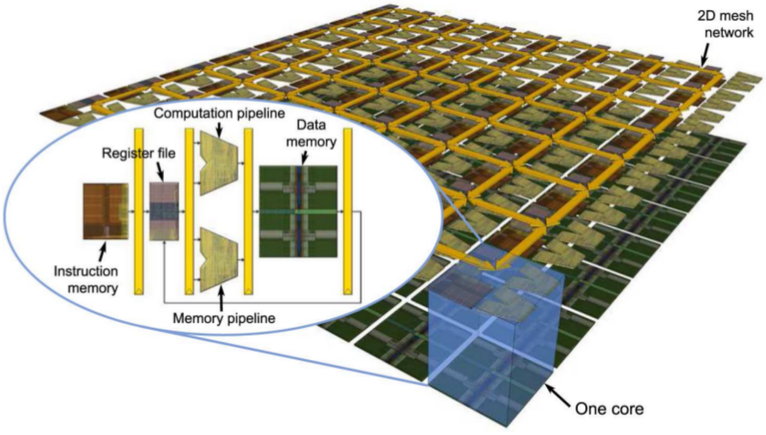
Figure 2.23: Simulation results. . From [8]

Results show that power consumed by DRAM and LiM layer increases with the increasing of the bandwidth. This is because more computational resources are required. In figure 2.23(1.b) SpGemm is simulated with two benchmark matrices and performance keeps increasing on architectures with higher memory bandwidths. Then Intel Math Kernel Library (MKL) Sparse Basic Linear Algebra Subprograms are run on Intel Xeon machines. For comparison, to simulate the same SpGemm, Sniper multicore simulator is used. After that power and performance are evaluated. Results show that the 3D-structure is able to achieve one order of magnitude of performance improvement and more than two of power efficiency compared to Intel MKL.

2.2.2 Design and Analysis of 3D Massively Parallel Processor with Stacked Memory [9]

The proposed architecture is called 3D-MAPS, that stands for Massively Parallel Processor with Stacked Memory. The aim of this work is to fully exploit the high bandwidth provided by the 3D technology. Thanks to the TSV connections or F2F bond pads that are shorter than PCB connections, the stacked structure provides high inter-chip communication bandwidth.

The 3D-MAPS architecture is composed of a core layer made of 64 cores and a memory layer made of 64 4KB SRAM blocks. Each core communicates with its dedicated memory block through F2F bond pads. The architecture is shown in figure 2.24.



Kim et al., 2013

Figure 2.24: 3D-MAPS architecture. From [9]

The memory works at the same frequency of the processors. The processor is general purpose and it is designed to run memory read/write operations every clock cycle in order to fully exploit the memory bandwidth.

Single-Core Architecture The architecture and the ISA of the single core are similar to the MIPS, but they were modified to satisfy system specifications. Since the specification on the area was very strict the architecture and the ISA were simplified by eliminating some of the most expensive components in terms of silicon area, such as floating point units and branch predictors.
Figure 2.25 shows the strcuture of the Single-Core. The pipeline depth is five and the word size is 32bit.

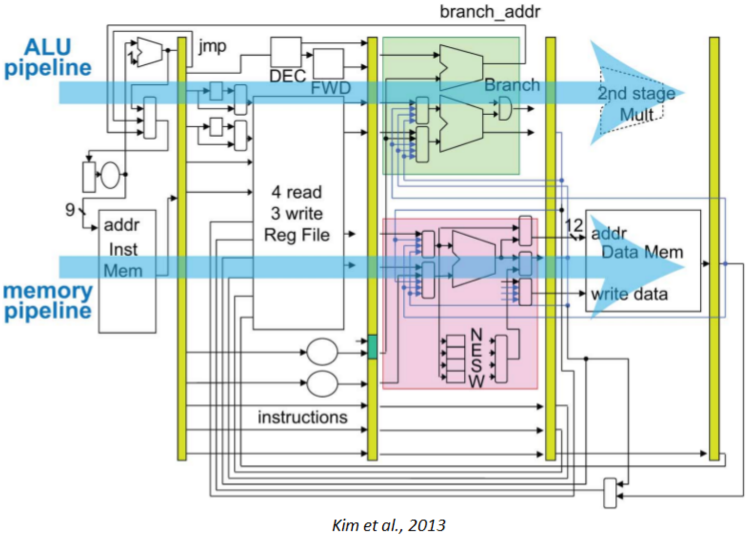


Figure 2.25: Single Core Architecture. From [9]

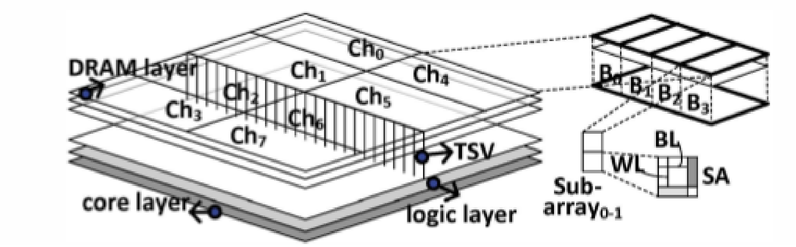
Each core is composed of a general-purpose ALU, a multiplier and 4 core-core communication ports. The communication between cores happens at the third pipeline stage.
To ensure that each core has access to the memory every clock cycle, and thus maximizing the usage of the memory bandwidth, an execution path is reserved for both memory and non-memory operations.

As far as multi-core is concerned, to minimize power consumption a 2D mesh network is used and to synchronize cores an H-tree shaped global barrier instruction is employed.

The 3D-MAPS chip was built with a 130nm process technlogy. The architecture was also simulated using 8 different data intensive benchmarks, such as K-means, String search and matrix multiplication. The simulations were performed using SoC Encounter and Cadence softwares. Results show that the maximum operating frequency is 277Mhz, max peak bandwidth utilization is 63.8GB/s while consuming 4W power.

2.2.3 XNOR-POP: A Processing-in-Memory Architecture for Binary Convolutional Neural Networks in Wide-IO2 DRAMs [10]

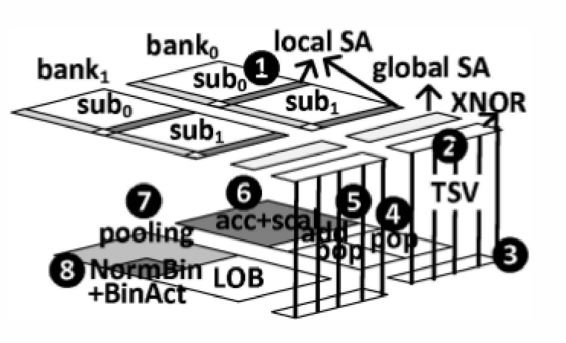
In order to adopt computing intensive CNN a lot of hardware resources and high power budget are required, this is why adopting them in mobile devices is difficult. XNOR-Net, an emerging binary CNN, reduces memory and computational overhead performing XNORs and population count operations instead of floating point multiply accumulate operations.
The proposed architecture aims to support XNOR-Net in mobile devices employing WideIO2 memory. It is a stacked DRAM designed for mobile devices (figure 2.26).



L.Jiang et al., 2017

Figure 2.26: Wide-IO2 DRAM architecture. From [10]

The following figure represent the proposed structure:

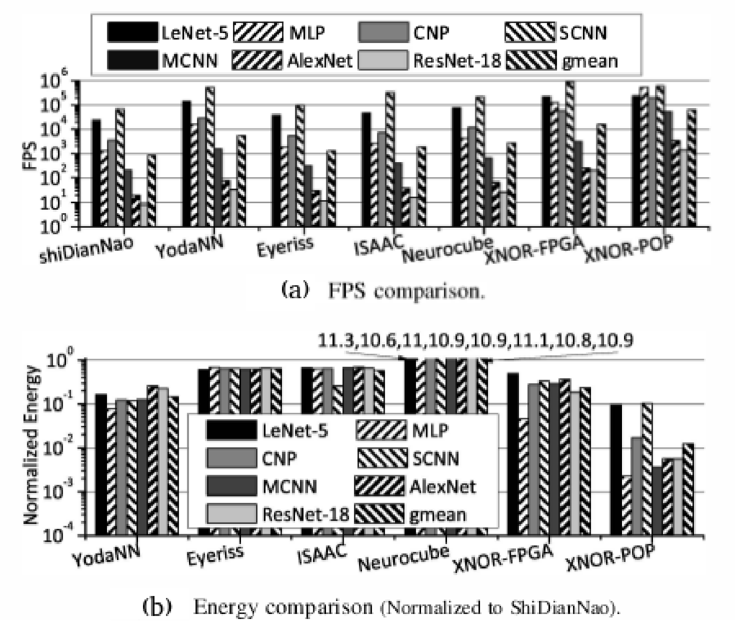


L.Jiang et al, 2017

Figure 2.27: XNOR-POP Flow. From [10]

The XNOR-DRAM performs XNOR operations at row level in each DRAM

bank. Once the result is latched a new operation can start. The result is transferred on the logic die by TSVs. Then the data follow the flow through the population count to the pooling stage till the Layer Output Buffer, which is a 512KB SRAM. The pipeline has to stop when the computation of the convolutional layer is finished or the layer output buffer is full. If this happen, the data in the LOB has to be written back to DRAM dies.
The structure was simulated for different NNs and compared with state-of-the-art accelerators. Results show (figure 2.28) that XNOR-POP on average improves CNN tests by 11 times in performance and 90% of energy consumption.



(a) FPS comparison.

(b) Energy comparison (Normalized to ShiDianNao).

L.Jiang et al., 2017

Figure 2.28: Evaluation Results. From [10]

2.3 ReRAM-Based

2.3.1 ReRAM Basics

Resistive RAM is a non-volatile memory that stores information using a resistive component.

The ReRAM cell is a MIM structure: a metal-oxide layer is sandwiched between two electrodes (figure 2.29).

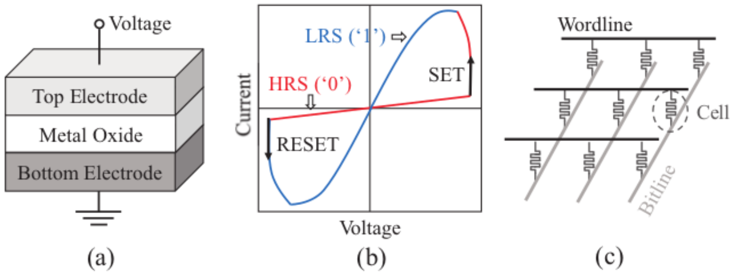


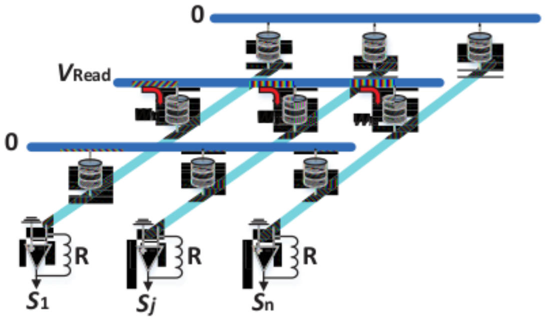
Figure 1. (a) Conceptual view of a ReRAM cell; (b) I-V curve of bipolar switching; (c) schematic view of a crossbar architecture.

P.Chi et al., 2016

Figure 2.29: Basic principles of ReRAM. From [11]

The information is represented by a resistive value: low resistance indicates logic "1"(HRS) whilst high resistance corresponds to logic "0" (LRS).
By applying an external voltage to the cell it is possible to switch between states. The RESET operation consist in switching from LRS to HRS states and requires negative voltage, while SET operation from HRS to LRS and requires positive voltage.[11]

The most common structure of a ReRAM array is the crossbar structure (figure 2.30). This structure allows to implement matrix-vector multiplication and it is often used in neural networks applications.



L.Han et al., 2017

Figure 2.30: ReRAM crossbar structure. From [13]

2.3.2 PRIME: A Novel PIM architecture for Neural Network Computation in ReRAM-based main memory [11]

Artificial Neural Networks are a kind of NN that are implemented through the crossbar structure (figure 2.31).
They implement operations like:

$$b_j = \sigma(\sum_{\forall i} a_i \cdot w_{i,j})$$

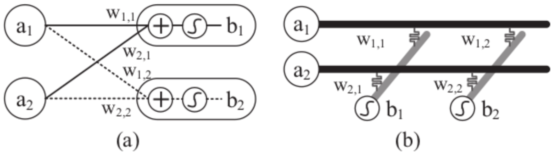


Figure 2. (a) An ANN with one input/output layer; (b) using a ReRAM crossbar array for neural computation.

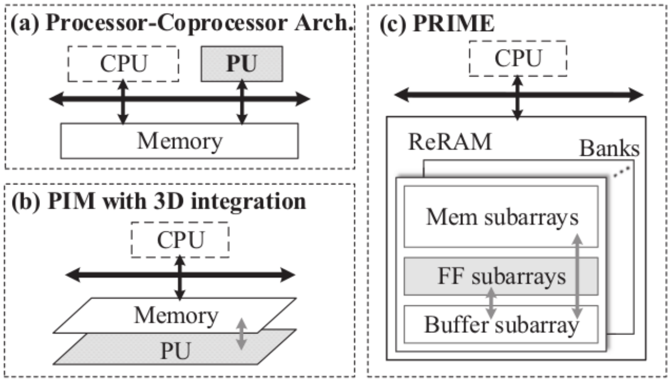
P.Chi et al., 2016

Figure 2.31: Artificial Neural Network structure. From [11]

Since they are similar to matrix-vector operations, ReRAM crossbar structure adapts perfectly to the task.

The proposed architecture, called PRIME, aims to accelerate NN by leveraging ReRAM's computation capability and exploiting processing-in-memory architecture. PRIME acts like a real in-memory architecture, since it does not need supplementary logic components, but it performs the computation directly on the memory array. The add-on components in PRIME are simple modification to the peripheral circuitry of the memory to enable the computation function. This result in a low area overhead.

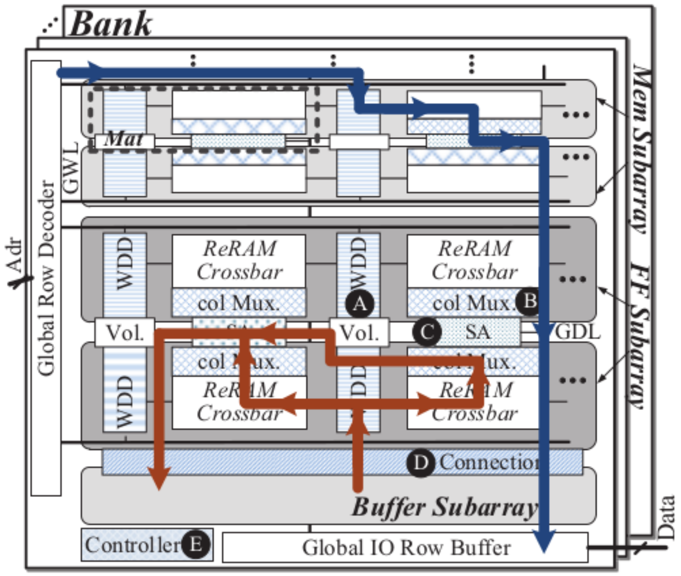
In figure 2.32 a comparison between PRIME and previous approaches is made. It is noticeable that PRIME is the only one to not require an additional processing unit (PU) to operate.



P.Chi et al., 2016

Figure 2.32: Comparison between PRIME and other architectures. From [11]

PRIME performs operations inside the memory banks. To achieve this they are divided into Memory subarrays, which have only storage capability, Full Function subarrays that can work either in computation mode or memory mode and Buffer subarray which serve as data buffers or data storage, if needed.
The overall architecture is shown in figure 2.33.

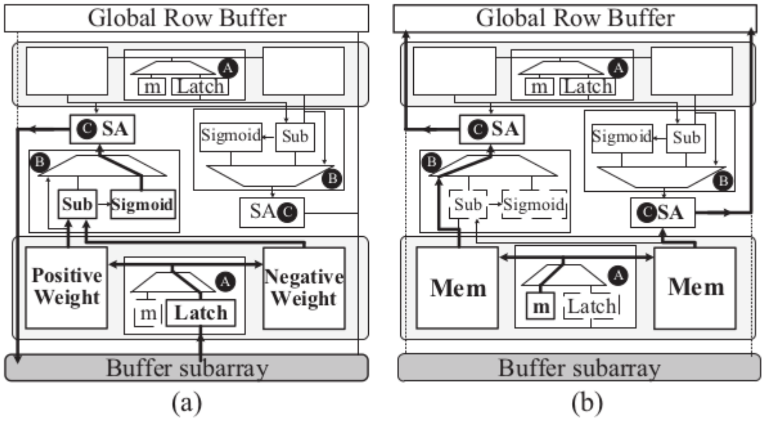


P.Chi et al., 2016

Figure 2.33: PRIME Architecture. From [11]

Buffer subarrays are selected as the closest to the FF subarrays, in order to reduce

latency. The data flow from the Memory Subarrays to the Global Row Buffer and then enter the Buffer Subarray. FF and Buffer subarrays communicate with each other through private ports. This way they do not consume the bandwidth of the memory subarrays and the CPU can still access them and work in parallel.
Moreover, sense amplifiers and write drivers are slightly modified to serve also as ADC and DAC respectively, since these are components required in NN applications.

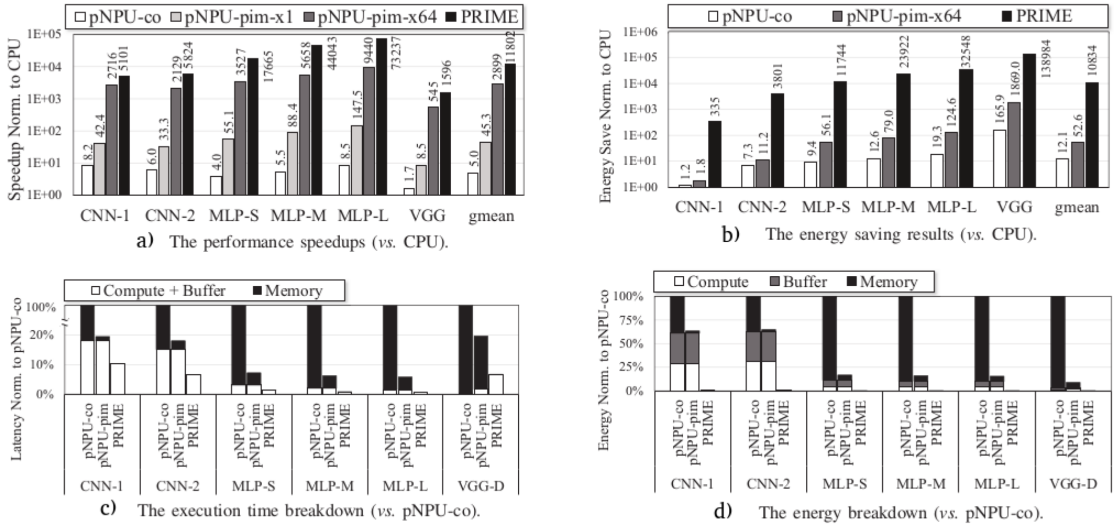


P.Chi et al., 2016

Figure 2.34: Configuration of FF subarrays: (a) Computation mode; (b)Memory mode. From [11]

In figure 2.34 the internal functioning of the FF subarray is depicted.
As previously stated, they can work both in computational and memory mode. The switching between modes is performed by the PRIME Controller that sends control signals to the multiplexers.

The PRIME architecture has been modelled using CACTI-IO, CACTI-3DD and NVSim. Then it was simulated using a trace-based in-house simulator. PRIME was compared to different systems, such as NPU co-processor, CPU-only and NPU PIM processor.
The benchmarks used comprise six NN design (MlBench) for machine learning applications. Results are shown in figure 2.35.



a) The performance speedups (*vs.* CPU).

b) The energy saving results (*vs.* CPU).

c) The execution time breakdown (*vs.* pNPU-co).

d) The energy breakdown (*vs.* pNPU-co).

P.Chi et al, 2016

Figure 2.35: Evaluation results. From [11]

As far as performance is concerned, the advantage of PRIME comes from the fact that the synaptic weights do not need fetching from memory because they are already pre-programmed into the memory cells unlike for instance the NPU. Also from the energy efficiency point of view PRIME shows good results, thanks to the ReRAM structure that is energy efficient for NN applications.

2.3.3 RADAR: A 3D-ReRAM based DNA Alignment Accelerator Architecture

Sequence Alignment is the most fundamental application in bioinformatics. One of the most widely used algorithms is the Basic Local Alignment Search Tool (BLAST). This kind of algorithm involve moving a huge DNA databse from storage to computational components. Such actions are expensive in terms of time and energy. RADAR is an architecture that aims to transfer the computational operations locally in memory to be performed without moving the database.

RADAR uses as memory 3D-ReCAM (3D ReRAM-based CAM) that is very suitable to accelerate BLAST thanks to its low power consumption, high density (it

is capable of storing the whole DNA database) and capability to perform parallel comparisons. Moreover it is a transistor-less structure.
The overall structure is depicted in figure 2.36.

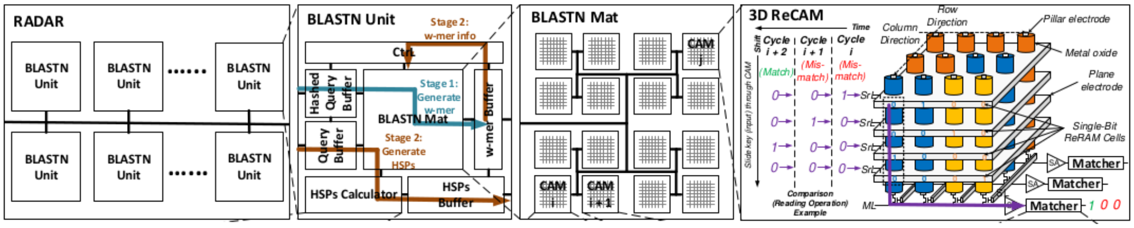


Figure 2.36: RADAR architecture

Each BLASTN Mat is made of multiple 3D reCAMs connected with each other in H-tree.
Comparison operation can be performed in different rows and different CAMs concurrently. This enables both row and CAM level parallelism. Moreover, each BLAST Unit works in parallel independently, providing Unit level parallelism.

In the following few main characteristics of RADAR are listed:

- **Reduction in Data Movement**: instead of moving the whole database into computing components, the query sequence is moved into CAMs. This way the database remains in memory and operations are performed locally. This reduction in data movement leads to speedup and energy saving;
- **Scalability**: 3D ReCAM is characterized by high density, this allows RADAR to store the whole DNA database in a single chip. If the database is too large to fit in a single chip, scaling out is an option. In order to deal with extremely huge databases,it is possible to distribute RADAR;
- There are **no writing operations** in RADAR except for writing the database into ReCAMs just one time;

To evaluate RADAR a C++ simulator was built. 3d ReCAM parameters were extracted from NVSim. RADAR was evaluated for 5 designs, varying CAM parameters (numbers of rows ,columns and so on).
As benchmarks 6 different databases with different sizes were used. The baseline is

NCBI BLASTN running in a server with CPU of Intel Xeon. Results are shown in figure 2.37.

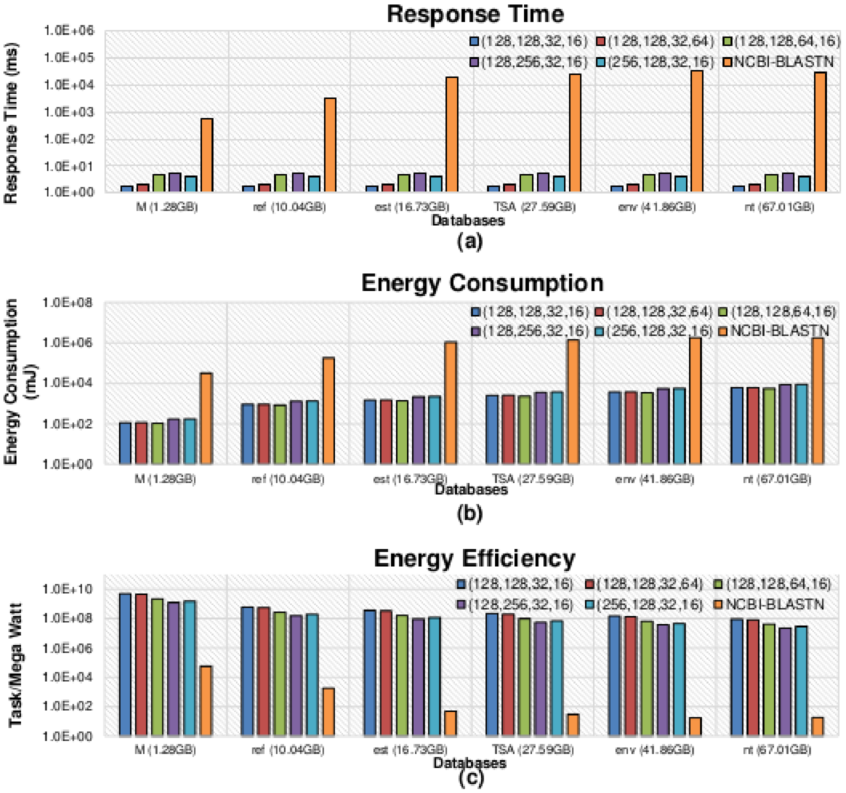


Figure 2.37: Performance results of NCBI BLASTN and RADAR

It is possible to notice that from the response time point of view, the values of the CPU grow dramatically, while RADAR maintains quite stable values even with the increasing of the database size.

As for energy consumption RADAR appear very energy efficient compared with the CPU. This is thanks to the reduction in data movement. However CPU and RADAR move similarly for energy efficiency: it decreases as the database grows. It is due to the correspondent increase in query response time. Still, the CPU efficiency decreases much faster than RADAR.

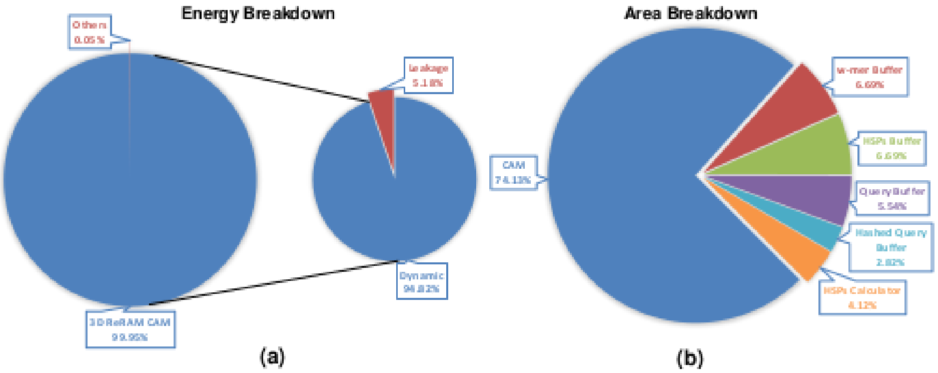
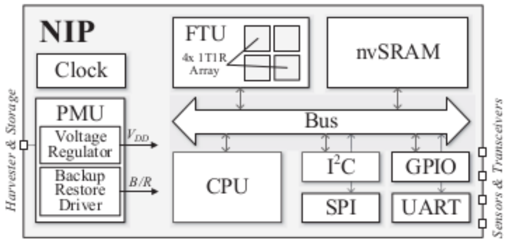


Figure 2.38: Energy and Area breakdown of RADAR

As for Area and Energy breakdown (figure 2.38) , it is noticeable that almost all of area and energy is due to the CAM. This indicates that RADAR is a memory centric accelerator with low leakage and small extra hardware overhead.

2.3.4 A 462GOPs/J RRAM-Based Nonvolatile Intelligent Processor for Energy Harvesting IoE System Featuring NV Logics and PIM [12]

NIP is a nonvolatile intelligent processor capable of both general and neural network computing. It is built in 150nm CMOS process with embedded HfO RRAM. The block diagram is depicted in figure 2.39.



F.Su et al., 2017

Figure 2.39: Top level architecture of NIP. From [12]

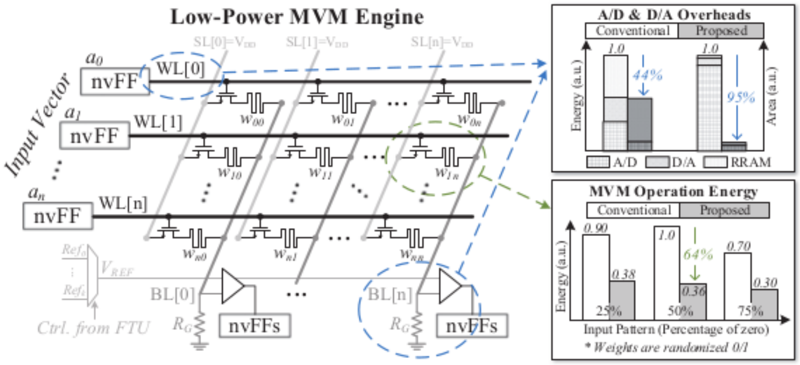
The CPU executes general purpose tasks and manages the communication with off-chip sensors and transceivers; the FCNN Turbo Unit (FTU) deals with FCNN

tasks; nvSRAM is a data memory shared between FTU and CPU.
Power Management Unit (PMU) provides power supply and handles the backup and restore decisions for the whole chip.
However, there are two problems in energy harvesting NIPp with RRAM-based PIMs:

1. CPU and RRAM array interfaces (i.e. DAC and ADC) become the bottleneck of chip area and energy efficiency;

2. To perform matrix-vector multiplication (MVM) operations, all access transistors on the word lines are simultaneously turned on, resulting in a great waste of energy;

For these reasons a low power MVM engine is designed (figure 2.40) with input controlled access transistor and binary interfaces.



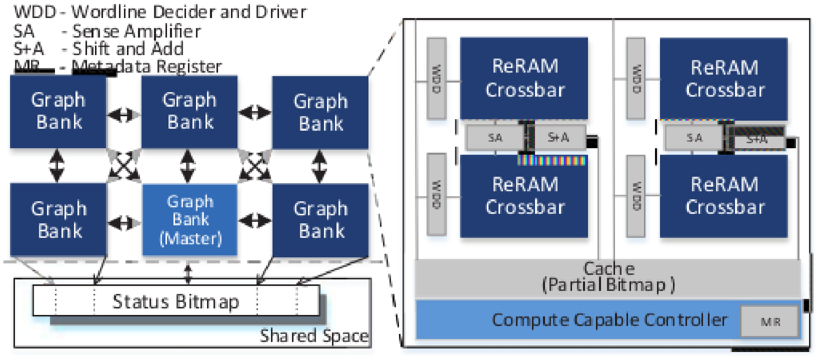
F.Su et al., 2017

Figure 2.40: Proposed Low-power MVM engine and its performance improvements-From [12]

The ADC and DAC overheads are eliminated resulting in energy and area saving. Also, the input-controlled access transistors remain OFF when the row input is zero, resulting in high energy saving. NIP is compared to prior works. It achieves 13x improvement in nergy efficiency over the state of the art.

2.3.5 A Novel ReRAM-based Processing-in-Memory Architecture for Graph Computing [13]

In graph processing application memory bandwidth is the key performance bottleneck.
RPBFS is the proposed ReRAM-based processing-in-memory architecture that implements the Breadth First Search algorithm. This structure is capable of processing the graphs and storing them permanently. RPBFS architecture is represented in figure 2.41.



L.Han et al., 2017

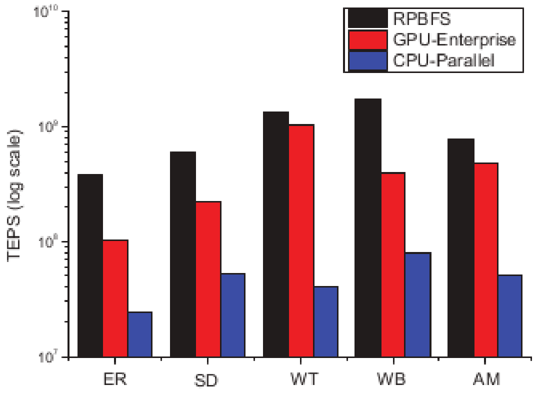
Figure 2.41: RPBFS architecture. From [13]

The ReRAM banks are partitioned into 2 types: graph bank and master bank. Graph banks are used to map the graph and to store its adjiacency list, so for one graph multiple graph banks are involved. The master bank stores corresponding metadata of graph banks.

Each bank is provided with an embedded controller with computing capability to decode instructions and provide control signals. The cache is used to store intermediate data.
Communication between banks is made with a mesh network. Banks share a EDRAM that stores the status bitmap of all vertices in an expansion level.

RPBFS is modelled by modified NVSim. The simulator is modified as trace-based system to evaluate performance with other solutions. RPBFS is compared

with GPU-based solution Enterprise adn state-of-the-art CPU-based parallel implementation.
As benchmarks 5 real world workload are chose, such as Wikipedia Talk network (WT). To evaluate traversal performance traversed edges per second (TEPS) is used. Results reported in figure 2.42 indicate that RPBFS achieve a performance improvement for all benchmarks compared to other techniques.

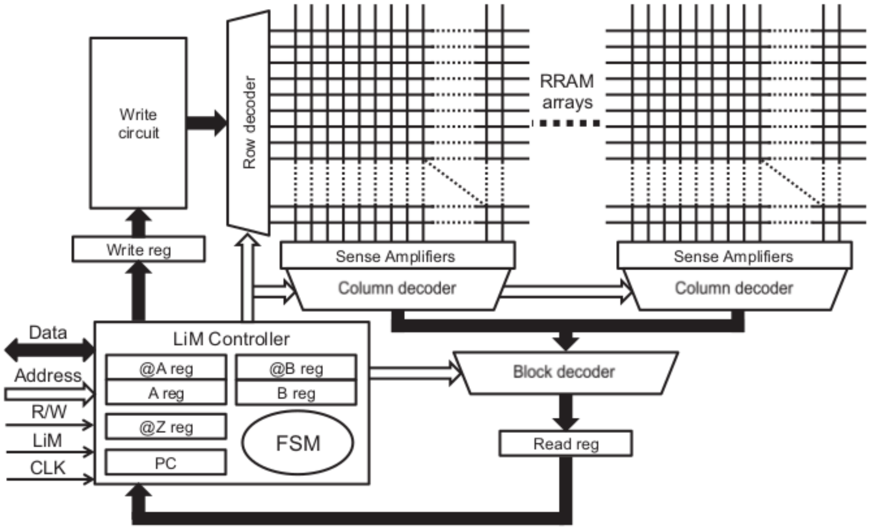


L.Han et al., 2017

Figure 2.42: Performance of RPBFS and direction-optimizing CPU-based and GPU-based solutions. From [13]

2.3.6 The Programmable Logic-in-Memory Computer [14]

The proposed is a fully programmable system composed of a multi-bank ReRAM and a LiM controller (figure 2.43). This architecture is then tested with PRESENT, a primitive used for cryptographic applications.



P. E. Gaillardon et al., 2016

Figure 2.43: PLiM Architecture. From [14]

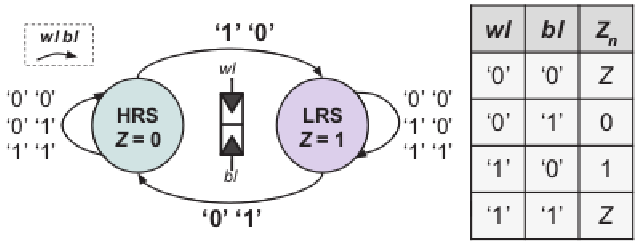
ReRAM memory elements can execute majority voter operations, this allows to perform various operations directly into the memory array. Thanks to the high density of ReRAM, it would be possible to achieve an high level of parallelism but that would imply a complex logic control. Therefore, to simply the controller, only serial operations are taken into account.
A full crypto operation can be implemented in-memory with an energy of 5.88 pJ and a throughput of 120.7 kbps.

The memory is partitioned in multiple banks and it can operate both in memory and computational mode. The LiM controller is composed of a FSM and few registers. When the control signal LIM is low, LIM controller is off and the ReRAM act as standard memory, otherwise computation starts.

2.3.7 ReVAMP: ReRAM based VLIW Architecture for in-Memory comPuting [15]

PLiM architecture allowed only sequential computations, thus under-utilizing the crossbar array. For this reason ReVAMP is proposed.

ReVAMP is a general purpose programmable system that supports VLIW-like instructions and parallel computation.
Logic operations are implemented with the same principle as PLiM: exploiting the ReRAM cell to obtain a majority vore (figure 2.44). With the function Z_n is possible to make any operation.

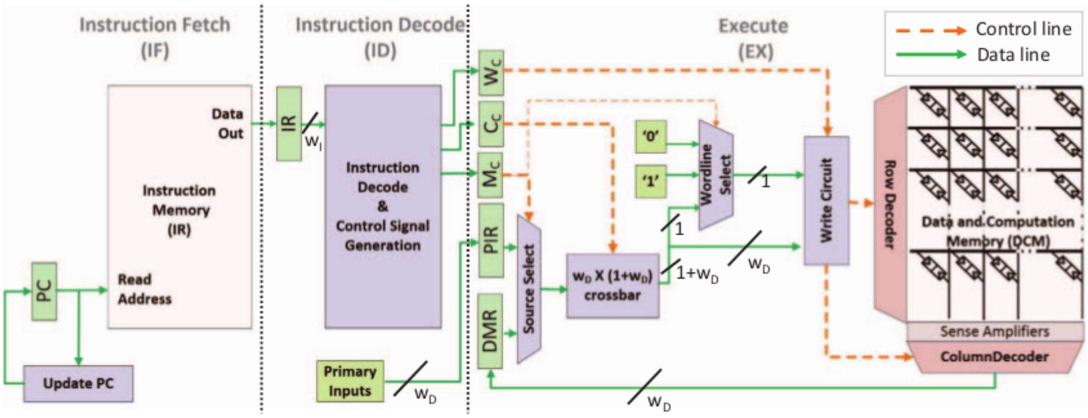


wl	bl	Z_n
'0'	'0'	Z
'0'	'1'	0
'1'	'0'	1
'1'	'1'	Z

D. Bhattacharjee et al., 2017

Figure 2.44: Logic operation using ReRAM devices. $Z_n = M_3(Z, wl, \overline{bl})$. From [15]

ReVAMP architecture (figure 2.45) is composed of two ReRAM crossbar memory: the Instruction Memory accessed by the Program Counter and the Data and Computation Memory (DCM).

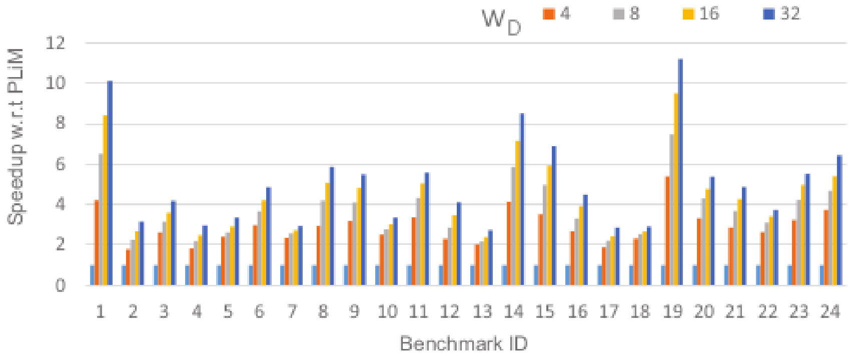


D. Bhattacharjee et al., 2017

Figure 2.45: ReVAMP architecture. From [15]

Since multiple Z_n operations operate in parallel, ReVAMP is a VLIW architecture in nature.

The architecture was test using 24 EPFL benchmarks and compared to PLiM (figure 2.46)for different word lengths. In comparison with the PLiM architecture ReVAMP shows a considerable speed-up improvement.



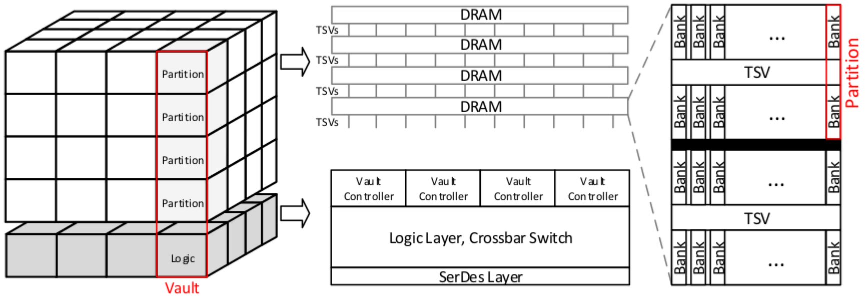
D. Bhattacharjee et al.,2017

Figure 2.46: ReVAMP evaluation results. From [15]

2.4 Processing-in-Memory

2.4.1 Hybrid Memory Cube Basics

Hybrid Memory Cube is a 3D DRAM architecture composed of multiple layer connected with each other via TSVs. The structure is enhanced with a logic die that is responsible of refresh,data routing, DRAM sequencing, error correction and high-speed interconnect to the host [30].



J. Schmidt et al., 2016

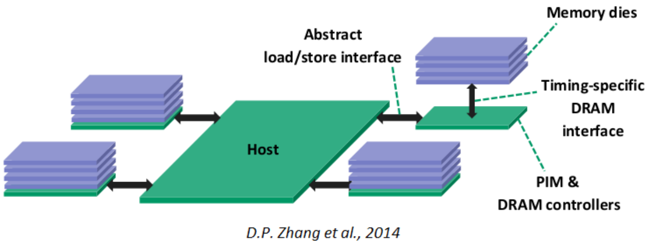
Figure 2.47: HMC Architecture

Each layer of DRAM is partioned; a vertical stack of partitions is called vault [31]. Each vertically stacked memory module operates in parallel to achieve up to 320GB/s bandwidth [32].

2.4.2 TOP-PIM: Throughput-Oriented Programmable Processing in Memory [16]

3D-stacking technology provides high bandwidth and it can be exploited to move memory-intesive computations closer to memory.
However, due to thermal constraints, stacking memory directly on top of a high-performance processor is not advisable.

TOP-PIM presents an alternative. The system organization is conceived as an host processor interconnected with multiple 3D stacked memories enhanced with an in-memory processor incorporated on the logic die of each stack (figure 2.48).



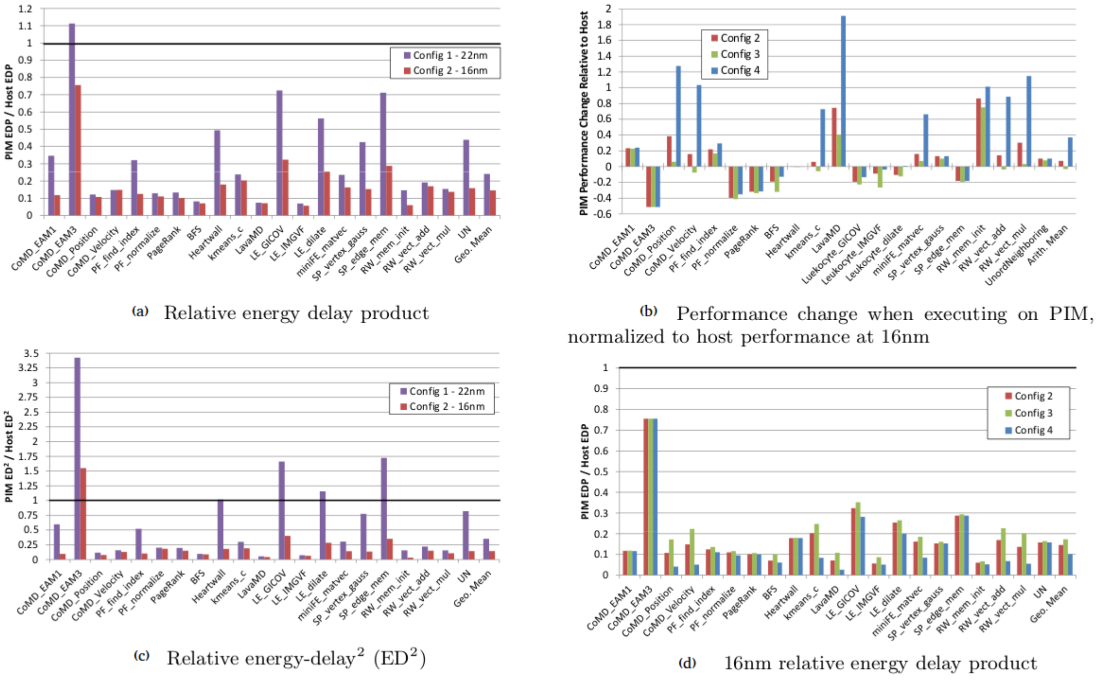
D.P. Zhang et al., 2014

Figure 2.48: System with in-memory processors. From [16]

Both in-memory processors and the host in this organization are accelerated processing units (APU), each of which consists of GPU and CPU cores on the same silicon die. Since the host processor does not suffer from any thermal constraint it can support computing-intensive high performance operations.

The system was modelled with an in-house ML-based model for performance and power estimation. As GPU AMD Radeon was chosen.

The benchmark used for evaluation are graph processing algorithms such as Breadth First Search or high performance compute such as MiniFE. Results are reported in figure 2.49.



(a) Relative energy delay product

(b) Performance change when executing on PIM, normalized to host performance at 16nm

(c) Relative energy-delay^2 (ED^2)

(d) 16nm relative energy delay product

D.P. Zhang et al., 2014

Figure 2.49: Evaluation results. From [16]

Simulations show that PIM can provide both performance and energy benefits for a various range of applications.

2.4.3 The Architecture of the DIVA Processing-in-Memory Chip [17]

The DIVA (Data IntensiVe Architecture) system is composed of a set of PIM chips serving as smart-memory co-processor to a standard microprocessor (figure 2.50).

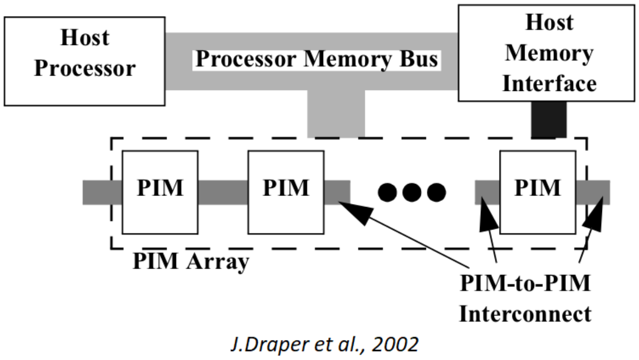


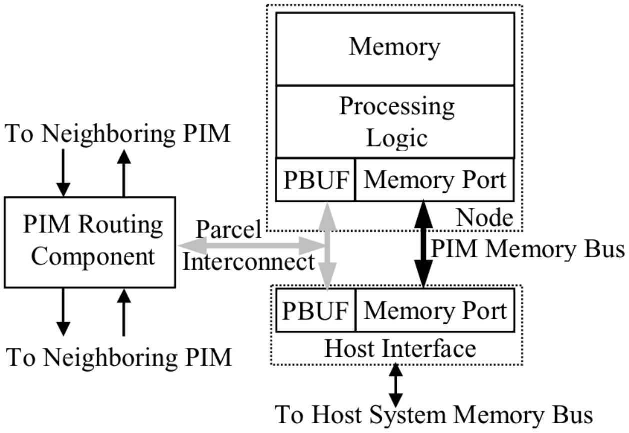
Figure 2.50: The DIVA system architecture. From [17]

DIVA aims to improve performance in bandwidth limited applications such as sparse-matrix and multimedia. DIVA accelerates these applications by executing computation directly in memory.

PIM chips communicate with each other through a separate interconnection in order not to interfere with host-memory traffic. Data, parcels and application code contain virtual addresses. DIVA memory is partitioned based on usage in order to avoid the overhead of page tables at each node to translate the addresses.
The partition is the following:

- **Dumb Memory**: allocated in a host's application virtual space and not reached by PIM chips processing;
- **Local Memory**: used only by PIM node routines;
- **Global Memory**: visible to the PIM nodes and the host;

A PIM chip is general purpose and it is composed of multiple PIM nodes (figure 2.51).

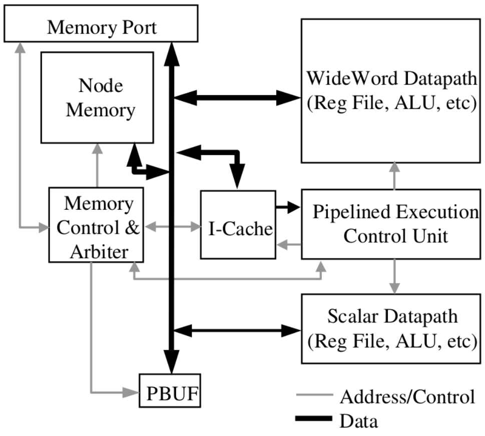


J.Draper et al., 2002

Figure 2.51: DIVA PIM chip architecture. From [17]

PIM Routing Component has the duty to route parcels off and on the chip. PIM chips share the host interface.

Figure 2.52 shows the internal organization of a PIM node.


Memory Port
Node Memory
WideWord Datapath (Reg File, ALU, etc)
Memory Control & Arbiter
I-Cache
Pipelined Execution Control Unit
Scalar Datapath (Reg File, ALU, etc)
PBUF
Address/Control
Data

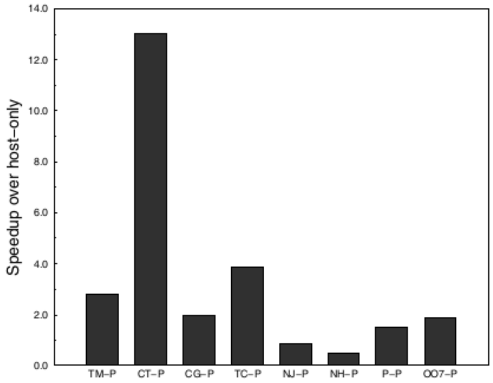
J.Draper et al., 2002

Figure 2.52: DIVA PIM node organization. From [17]

PIM node supports single-issue, in order execution. There is a single instruction control unit that coordinates two different datapaths: *Wide Word Datapath* executes fine-grained parallel executions and a *Scalar Datapath*.

DIVA structure has been tested with various types of benchmarks, ranging from image processing to database. To perform evaluation DSIM was developed. It is an

event-driven simulator based on the RSIM framework.
Results (figure 2.53) has been obtained from a simulation considering the system composed of only one PIM chip and they were compared to the ones obtained with a system provided only with the host processor (based on the MIPS R10000).



J.Draper et al., 2002

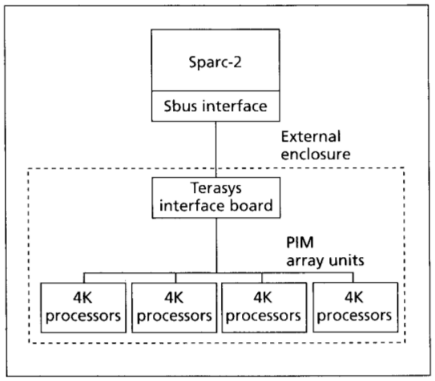
Figure 2.53: DIVA evaluation results. From [17]

Only with one PIM chip it is possible to achieve x3.3 speed-up improvement. This is thanks to the reduction of the stall time and to parallelism.

2.4.4 Processing in Memory: The Terasys Massively Parallel PIM Array [18]

SIMD processors can provide high performances for massively parallel problems, in which threads execute the same operation repeatedly. However, the computational load distribution is not homogeneous and this result in a waste of energy and decrease in performance.

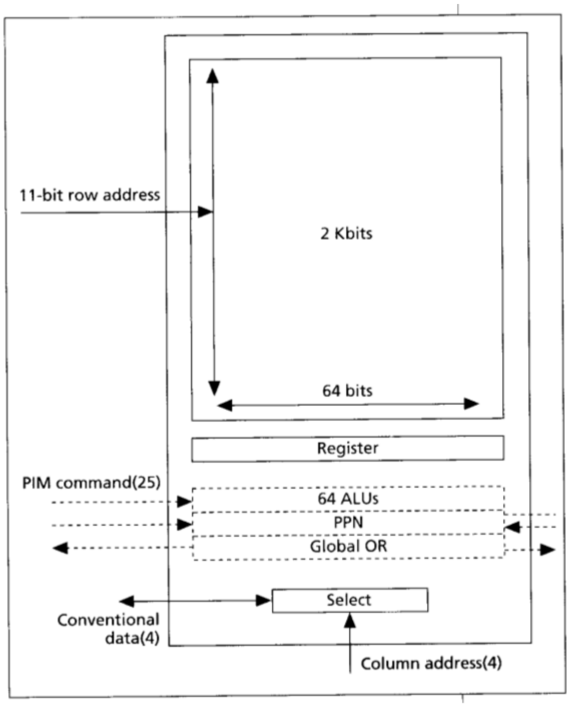
Terasys aims to embed the SIMD so close to host processor architecture that the SIMD array can be seen both as conventional memory and as a processor array. Structure of Terasys is depicted in figure 2.54.



M. Gokhale et al., 1995

Figure 2.54: Terasys workstation. From [18]

The Sparc processor executes conventional sequential operations. Instructions on data parallel operands are conveyed to Terasys.
Terasys interface can support up to 8 PIM array units, each of which contains 4096 processors, giving a total of 32768 processors. The PIM chip is depicted in figure 2.55. It is composed of 64 single-bit processors, 2k x 64 bits of SRAM and error detection and control circuitry. To operate in computation mode Terasys interface board sends determined control signals.

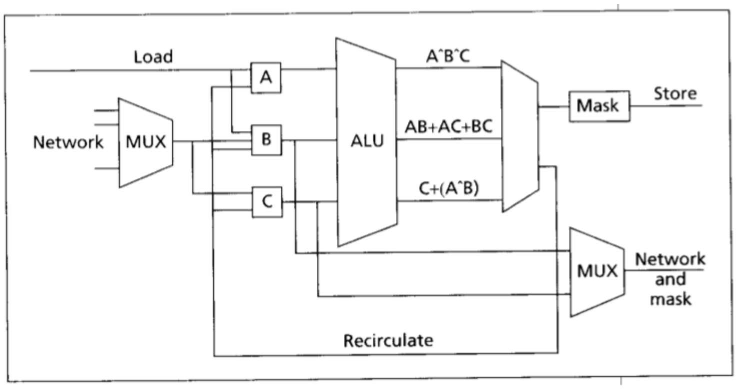


M. Gokhale et al., 1995

Figure 2.55: A Processor-in-memory chip. From [18]

Figure 2.56 shows the PIM processor. It is divided into 2 parts. The lower half performs masking and routing operations, while the upper half executes computations on data.

At each clock cycle, the pipelined ALU can storage data in memory or load data from data, but these operations can not be performed at the same time.



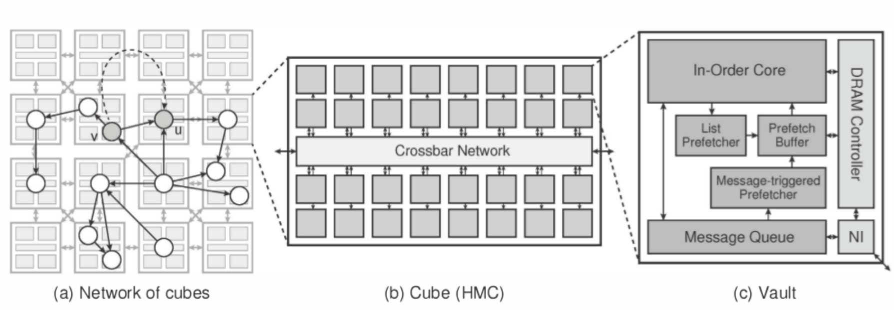
M. Gokhale et al., 1995

Figure 2.56: A Processor-in-memory processor. From [18]

Terasys was tested for 20 different applications, such as DNA sequence match or image processing. Results show that Terasys achieves $3.2 \cdot 10^{11}$ peak bit operations per second.

2.4.5 A Scalable Processing-in-Memory Accelerator for Parallel Graph Processing [19]

The proposed PIM architecture, called Tesseract, aims to be the solution to achieve memory-capacity-proportional performance in large-scale graph-processing.
Tesseract does not depend on a particular memory organization, but for analysis purposes HMC with eight 8 Gb DRAM layers was chosen as baseline (figure 2.57).



(a) Network of cubes (b) Cube (HMC) (c) Vault

J. Ahn et al., 2015

Figure 2.57: Tesseract architecture. From [19]

In order to execute computations in-memory a single-issue in-order core is placed

in the logic die of each vault.
In the proposed system the host processor has its own memory and Tesseract serves as accelerator. Moreover, it does not support virtual memory, in order to avoid the overhead derived from the address translation.

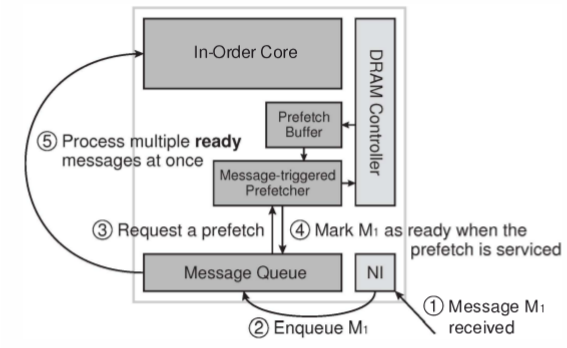
The host processor has access to the entire memory of Tesseract and it is the host's duty to distribute the input graph inside the vaults.

On the other hand, each core has access only to its own local DRAM partition. Therefore, message passing mechanism is implemented in order to connect cores with each other.

Since a single-issue in-order core is not able to exploit all the available bandwidth, 2 types of hardware prefetchers were designed:

- *List Prefetcher*: for constant-stride sequential accesses, so it is used a stride prefetcher based on a reference prediction table;
- *Message-triggered prefetching*: for random access patterns

Since graph processing often requires a large amount of random accesses, Message-triggered prefetching(figure 2.58) is included. It exploits message communication adding as additional field the memory address to be pre-fetched.

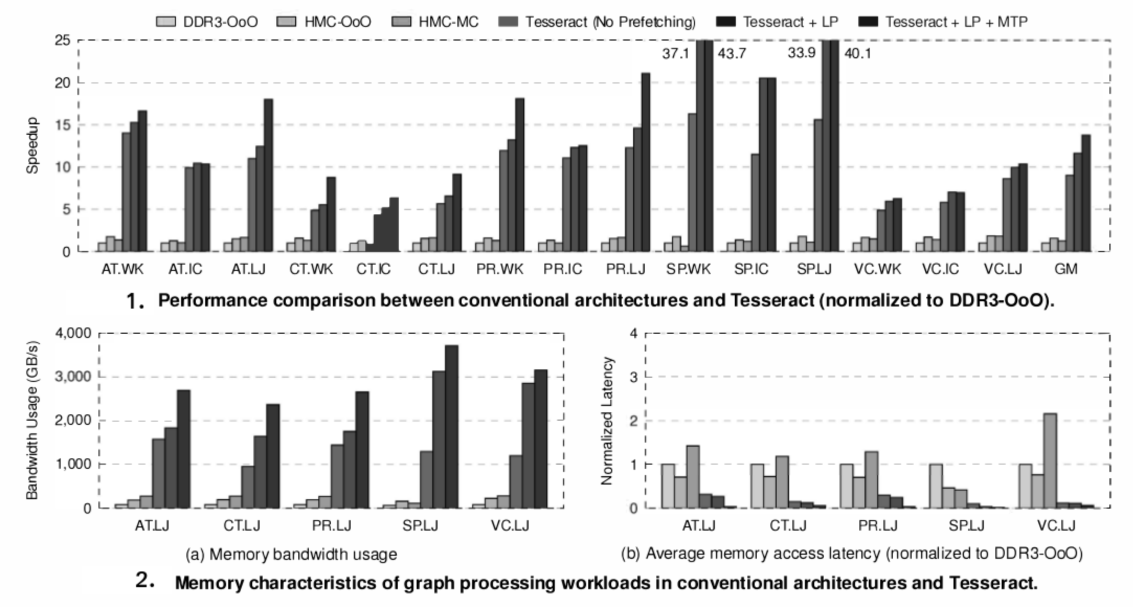

In-Order Core
DRAM Controller
Prefetch Buffer
⑤ Process multiple **ready** messages at once
Message-triggered Prefetcher
③ Request a prefetch
④ Mark M_1 as ready when the prefetch is serviced
Message Queue
NI
① Message M_1 received
② Enqueue M_1

J. Ahn et al., 2015

Figure 2.58: Message-triggered prefetching mechanism. From [19]

In order to evaluate Tesseract an in-house cycle-accurate simulator was developed. As benchmarks 5 graph algorithms were picked, such as Average Teenage

Follower (AT) or Page Rank (PR). Then, 3 input graphs were used.
Results show (figure 2.59) that Tesseract achieves better performances than DDR3 even without prefetching.



1. Performance comparison between conventional architectures and Tesseract (normalized to DDR3-OoO).

2. Memory characteristics of graph processing workloads in conventional architectures and Tesseract.

J. Ahn et al., 2015

Figure 2.59: Evaluation results. From [19]

The reason why Tesseract gives better performances compared with other structures is that they are not able to exploit the wide bandwidth. In fact, as tables show, Tesseract achieves a bandwidth usage of the order of TB/s.

2.4.6 Prometheus: Processing-in-Memory Heterogeneous Architecture Design From a Multi-layer Network Theoretic Strategy

In order to deal with the great amount of data found in nowadays application, Prometheus aims to provide a solution to the memory bottleneck.

The goal is to propose an approach to partition data across different vaults in HMC-based systems, in order to exploit high intra-vault memory bandwidth and at the same time reduce energy consumption and improving performance.

The Prometheus Framework (figure 2.60)takes into account the interactions between computation and communications. It follows three steps:

1. An input C/C++ application is modeled as a two-layered graph, where nodes denote LLVM IR (low level virtual machine intermediate representation) instructions and edges represent the data and control dependencies among LLVM instructions. Moreover, the weight associated with the edge represent the amount of time required for that specific operation;

2. The two-layered network is highly partitioned in order to minimize the energy consumption required for data movement and accesses;

3. Community-to-vault mapping strategy is applied

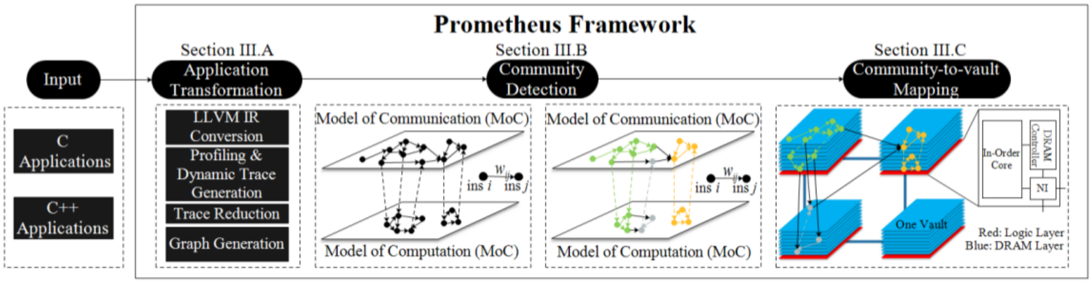
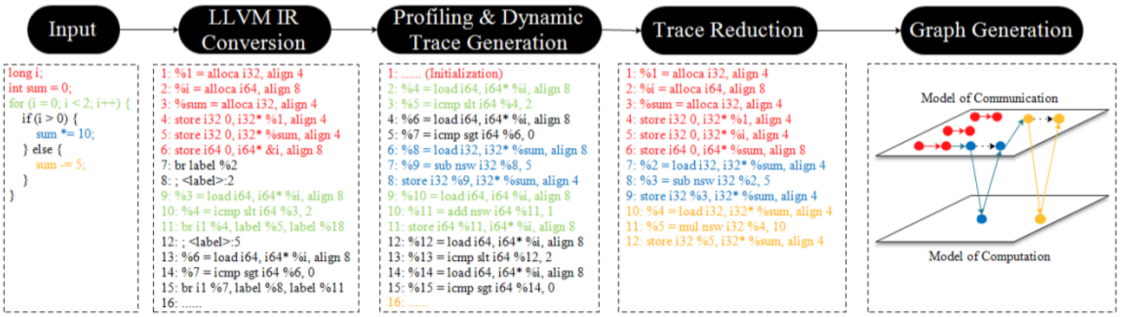


Figure 2.60: Overview of the Prometheus framework

In the following the three phases are analysed in detail:

1. **Application Transformation**: The C/C++ application in input is turned into a two-layered graph through different steps. In the computation layer nodes represent computations, the second layer models communication. Edges of the graph correspond to data and control dependencies.



```
long i;
int sum = 0;
for (i = 0; i < 2; i++) {
    if (i > 0) {
        sum *= 10;
    } else {
        sum -= 5;
    }
}
```

```
1: %1 = alloca i32, align 4
2: %i = alloca i64, align 8
3: %sum = alloca i32, align 4
4: store i32 0, i32* %1, align 4
5: store i32 0, i32* %sum, align 4
6: store i64 0, i64* &i, align 8
7: br label %2
8: ; <label>:2
9: %3 = load i64, i64* %i, align 8
10: %4 = icmp slt i64 %3, 2
11: br i1 %4, label %5, label %18
12: ; <label>:5
13: %6 = load i64, i64* %i, align 8
14: %7 = icmp sgt i64 %6, 0
15: br i1 %7, label %8, label %11
16: ......
```

```
1: ...... (Initialization)
2: %4 = load i64, i64* %i, align 8
3: %5 = icmp slt i64 %4, 2
4: %6 = load i64, i64* %i, align 8
5: %7 = icmp sgt i64 %6, 0
6: %8 = load i32, i32* %sum, align 8
7: %9 = sub nsw i32 %8, 5
8: store i32 %9, i32* %sum, align 4
9: %10 = load i64, i64 %i, align 8
10: %11 = add nsw i64 %11, 1
11: store i64 %11, i64* %i, align 8
12: %12 = load i64, i64* %i, align 8
13: %13 = icmp slt i64 %12, 2
14: %14 = load i64, i64* %i, align 8
15: %15 = icmp sgt i64 %14, 0
16: ......
```

```
1: %1 = alloca i32, align 4
2: %i = alloca i64, align 8
3: %sum = alloca i32, align 4
4: store i32 0, i32* %1, align 4
5: store i32 0, i32* %i, align 4
6: store i64 0, i64* %sum, align 8
7: %2 = load i32, i32* %sum, align 4
8: %3 = sub nsw i32 %2, 5
9: store i32 %3, i32* %sum, align 4
10: %4 = load i32, i32* %sum, align 4
11: %5 = mul nsw i32 %4, 10
12: store i32 %5, i32* %sum, align 4
```

Figure 2.61: Application Transformation

2. **Community Detection**: is a technique to partition in clusters vertices which have higher probability of connecting with each other with respect to other vertices in different groups.

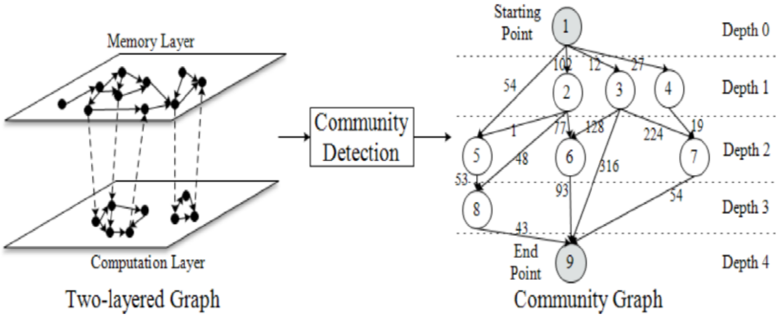
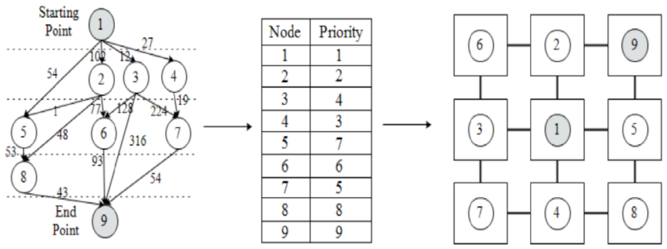


Figure 2.62: Community Detection

3. **Community-to-vault mapping**:the graph from previous step is re-mapped. Nodes are ordered according to priority (the lower the depth, the higher the priority). If more communities have the same depth they are sorted by communication cost (represented by the edges in the graph). Then, communities are mapped onto NoC in a way that the ones with higher priority are placed in a better position.



Node	Priority
1	1
2	2
3	4
4	3
5	7
6	6
7	5
8	8
9	9

Figure 2.63: Community-to-Vault Mapping

Prometheus has been tested and compared with an HMC- and a DDR3-based systems. Results (figure 2.64) shows that Prometheus bring improvement both in performance and energy efficiency compared to other systems.

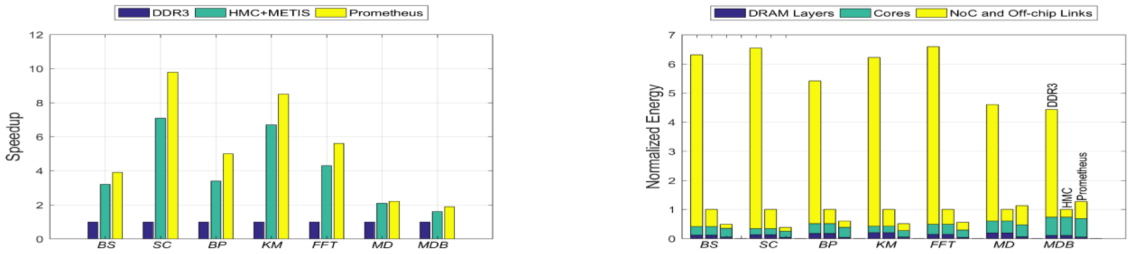
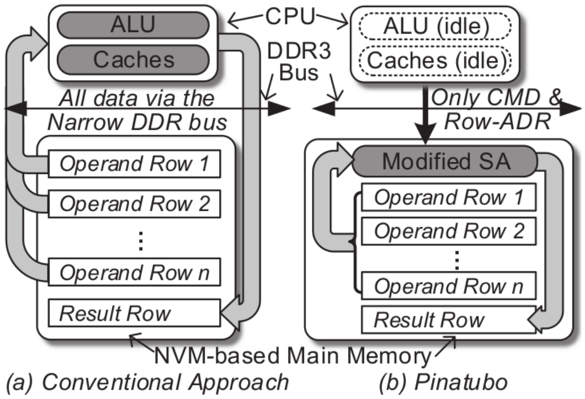


Figure 2.64: Speedup and energy consumption comparison between DDR3, HMC and Prometheus

2.4.7 PINATUBO: A Processing-in-Memory Architecture for Bulk Bitwise Operations in Emerging Non-volatile Memories [20]

The aim of this project is to present a valid alternative for the DRAM-based PIM architectures that are dependant on 3D integration. Instead of using DRAM, Pinatubo exploits the emerging non-volatile memories, such as STT-RAM or ReRAMs. Pinatubo does not depend on a specific technology, as long as the technology is based on resistive cell. Moreover it does not rely on 3D integration. Pinatubo is designed to accelerate Bulk Bitwise Operations.

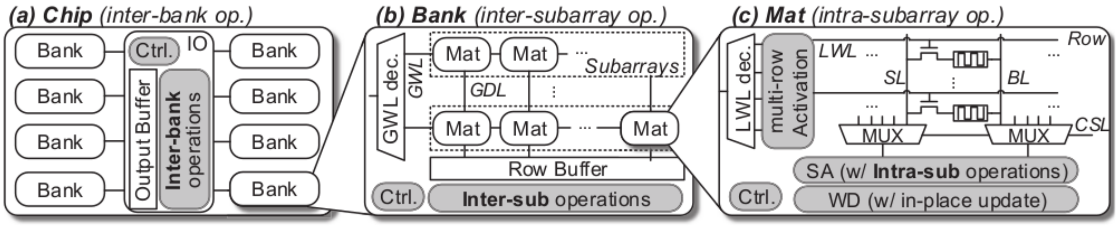


S. Li et al., 2016

Figure 2.65: Comparison between Pinatubo and conventional approach. From [20]

Pinatubo eliminates data movement for computation, since computation is performed in the memory itself. Moreover it provides high bandwidth and parallelism since it allows multi-row operations.

As presented in figure 2.65, the conventional approach consists in fetching data from memory, moving them to CPU and then write back. Pinatubo, instead, communicates with CPU only for control commands and row addresses.



S. Li et al., 2016

Figure 2.66: Pinatubo Architecture. From [20]

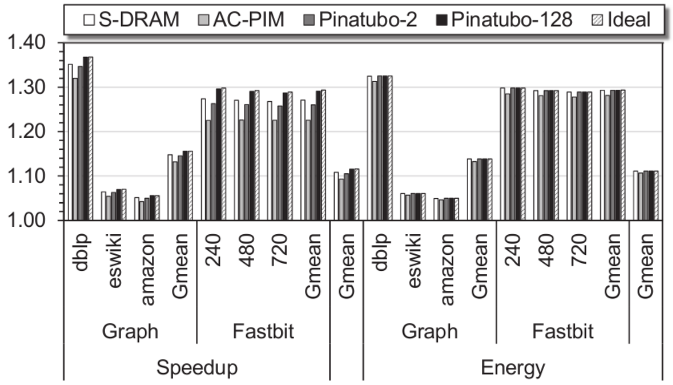
Figure 2.66 shows the 3 different levels of bitwise operations supported by Pinatubo.

- (c) **Intra-subarray ops**: Pinatubo performs intra-subarray operations if operands are all in one subarray. Multi-row activation is used. The operation is perfomed by the sense amplified, in which the reference values has been shifted in order to perform the logic operation;

- (b) **Inter-subarray ops**: If operands are all on the same bank. Operations are performed by additional logic components. Final results is sampled by the global row buffer;

- **(a) Intra-bank ops**: if operands are on different banks but on the same chip. They are executed by add-on logic in the I/O buffer.

Pinatubo has been simulated using HSPICE, NVSim and CACTI-3DD and compared with SIMD, S-DRAM and AC-PIM systems. Pinatubo 2 and 128 indicate the number of row operations. Bechmarks used are vector, graph processing and database.



S. Li et al., 2016

Figure 2.67: Speed-up and energy saving normalized to SIMD baseline. From [20]

The label "ideal" indicates result with zero latency and energy spent on bitwise operations. Results (figure 2.67) show that Pinatubo almost achieves the ideal value for acceleration. Compared with other systems, Pinatubo can improve database application energy saving and speed-up of about 1.29 times, graph processing speedup (x1.15) and energy saving (x1.14).

2.4.8 Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology [21]

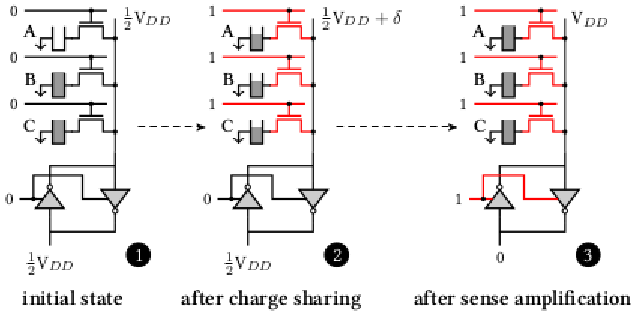
Many applications require bulk bitwise operations, ranging from databases to encryption algorithm and so on. However, this kind of operations require a large amount of data transitions that results in high bandwidth, latency and energy consumption. In order to compensate the limitation on the throughput of bulk bitwise operations due to limited memory bandwidth, Ambit is introduced. This architecture exploits DRAM technology in order to execute bulk bitwise operations completely inside the memory, using the entirety of its bandwidth.

Ambit can be thought split in two parts:

- **Ambit-AND-OR**: it is possible to perform AND or OR operations exploiting a three-row activation with rows that share the same sense amplifiers;
- **Ambit-NOT**: since each sense amplifier has two inverters, with a modest change it is possible to perform bitwise NOT.

With the capability to execute AND, OR and NOT operations, Ambit can execute any bitwise operation. All of that is obtained entirely using DRAM technology.

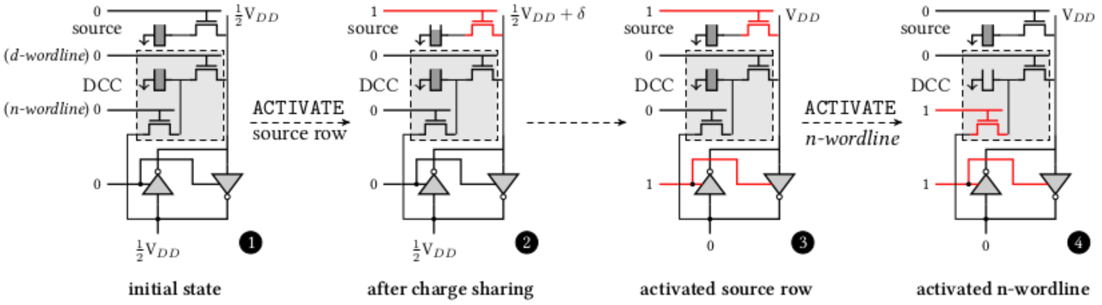
Ambit-AND-OR Knowing that in a subarray, each SA is shared by many DRAM cells on the same bitline and that the final state of the bitline after sense amplification depends primarily on the voltage deviation on the bitline after the charge sharing phase, it is possible to observe that simultaneously activating three cell results in a bitwise majority function.
The final state of the bitline is$AB + BC + CA = C(A + B) + \overline{C}(AB)$.
This means that by controlling the value of cell C, it is possible to exploit Triple Row Activation (figure 2.68) to execute a bitwise AND or bitwise OR of cells A and B.



V.Seshadri et al., 2017

Figure 2.68: Triple Row Activation. From [21]

Ambit-NOT Ambit-NOT exploits the fact that at the end of the sense amplification process the voltage level of the $\overline{bitline}$ represents the negated logical value of the cell (figure 2.69).

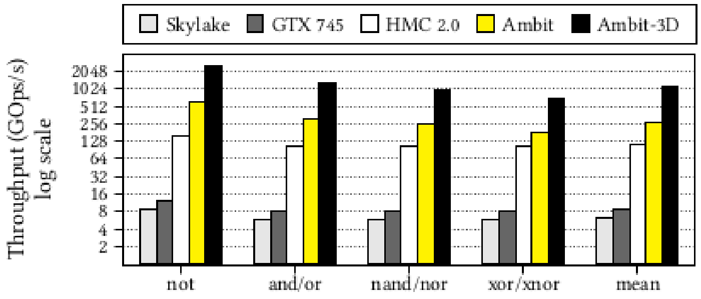


V.Seshadri et al., 2017

Figure 2.69: Bitwise NOT using a dual-contect cell. From [21]

Ambit interface is exactly the same of conventional DRAM, so it is possible to connect it directly on the system memory bus and handle it using memory controller. Since the CPU can access Ambit directly, there is no need to transfer data between CPU memory and accelerator.

Ambit was modelled using SPICE and gem5. The architecture was compared with Intel Skylake CPU, NVIDIA GTX and HMC 2.0. Ambit is tested both in normal and 3D stacked version. Throughput results (figure 2.70) show that Ambit outperforms other systems.

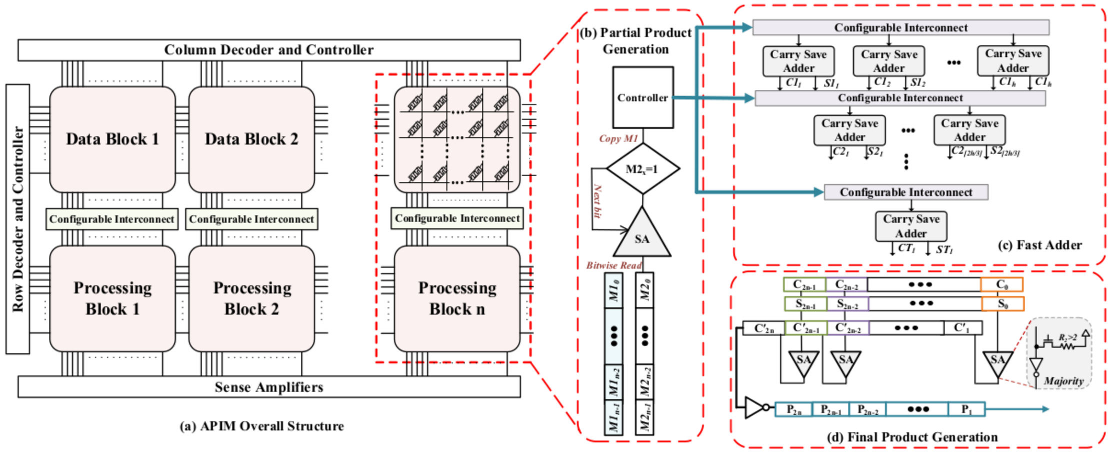


V.Seshadri et al., 2017

Figure 2.70: Throughput of bulk bitwise operations. From [21]

2.4.9 Ultra-Efficient Processing In-Memory for Data Intensive Applications [22]

APIM is an Approximate Processing-in-Memory architecture that exploits the statical nature of data in some applications, such as Internet of Things or multimedia applications, to obtain better performance in spite of some accuracy.
The proposed architecture makes use of emerging non-volatile memory, in particular ReRAM, in a blocked cross-bar structure, which introduces flexibility in executing operations and facilitates shift operations in memory.
Moreover, APIM can dynamically configure the precision of computation in order to tune the level of accuracy during runtime.


Column Decoder and Controller
Row Decoder and Controller
Data Block 1
Data Block 2
Configurable Interconnect
Configurable Interconnect
Configurable Interconnect
Processing Block 1
Processing Block 2
Processing Block n
Sense Amplifiers
(a) APIM Overall Structure
(b) Partial Product Generation
Controller
Copy M1
M2$_x$=1
Next bit
SA
Bitwise Read
$M1_0$
$M1_{n-2}$
$M1_{n-1}$
$M2_0$
$M2_{n-2}$
$M2_{n-1}$
Configurable Interconnect
Carry Save Adder
CI_1 SI_1
CI_2 SI_2
CI_h CI_h
Configurable Interconnect
$C2_1$ $S2_1$
$C2_{[2h/3]}$ $S2_{[2h/3]}$
Configurable Interconnect
CT_1 ST_1
(c) Fast Adder
C_{2n-1} C_{2n-2} C_0
S_{2n-1} S_{2n-2} S_0
C'_{2n} C'_{2n-1} C'_{2n-2} C'_1
SA
$R_t>2$
Majority
P_{2n} P_{2n-1} P_{2n-2} P_1
(d) Final Product Generation

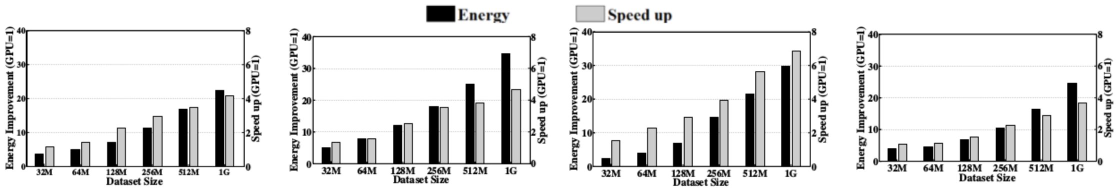
M.Imani et al., 2017

Figure 2.71: Overall structure of APIM. From [22]

As figure 2.71 shows, APIM is partitioned in blocks linked with each other through configurable interconnections.
There are two kinds of block but since they are structurally the same, they are interchangeable.
Data blocks serve as data storage, while processing blocks perform computation. Interconnections support shifting operation, this way latency is reduced. Fast addiotin is performed by carry save adders.

As far as approximation is concerned there are two approaches. One is approximating at the beggining of operation, masking some of the LSBs. This approach is faster and less consuming but the error propagates through the entire process, possibly resulting in very high inaccuracy.
The second approach consists in approximating at the final stage of execution. It is more accurate but also slower.

APIM was compared with state-of-the-art AMD Radeon GPU. Simulation were carried out using multi2sim, Cadence Virtuoso. Six general OpenCL were run for comparison, such as FFT. Results are reported in figure 2.72. APIM was run in exact mode.



(a) Sobel (b) Robert (c) FFT (d) DwtHaar1D

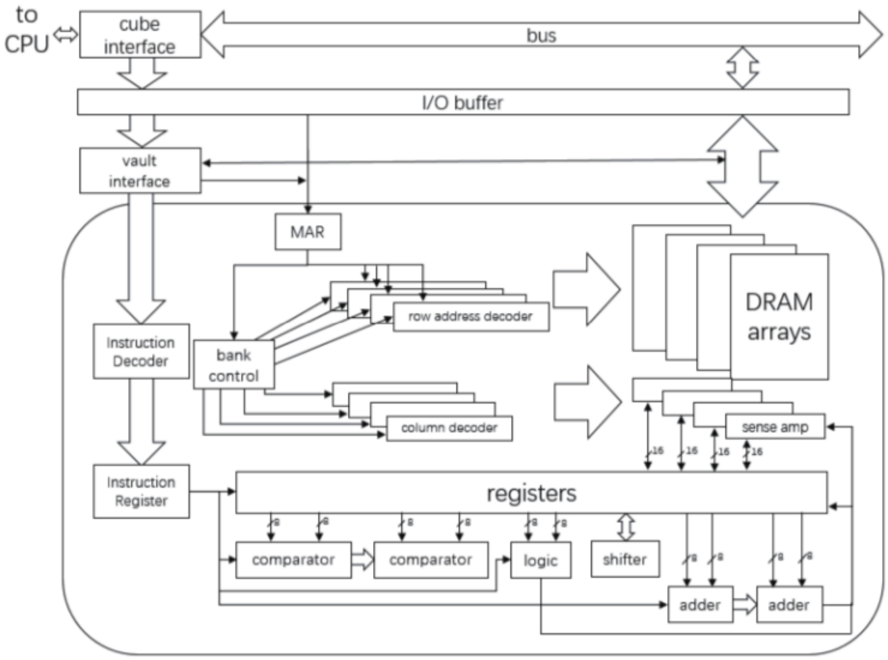
M.Imani et al., 2017

Figure 2.72: Energy consumption and speedup of exact APIM normalized to GPU vs different dataset sizes. From [22]

Energy and speed-up results of APIM outperforms the one obtained for GPU.

2.4.10 ApproxPIM: Exploiting Realistic 3D-stacked DRAM for Energy-Efficient Processing-in-memory [23]

The purpose of ApproxPIM is to investigate the potential and feasibility of diploying PiM in realistic HMC products without adding any computation logics or cores. ApproxPIM is proposed to enable PIM in HMC for domain-specific computing. The overall architecture is shown in figure 2.73.

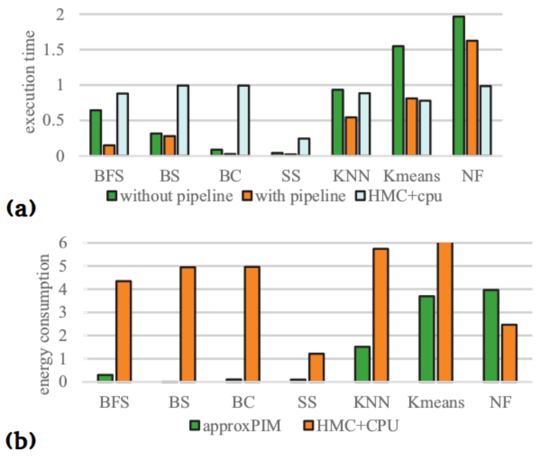


Y. Tang et al., 2017

Figure 2.73: Overview of ApproxPIM. From [23]

ApproxPIM is based on HMC, which is composed of vaults. Each vault has its own controller and it is completely independent from each other. The communication between HMC and the host processor is based on a parcel transmission protocol. At the first stage of executing an application ApproxPIM configures its lane into quarter-width links, in order to reduce energy cost. When execution is finished, ApproxPIM notify the host which restore the full-duplex connection.

To evaluate ApproxPIM CACTI-3DD, Multi2Sim and McPAT were used. The architecture was tested with different workloads such as graph processing, sorting algorithms and machine learning applications. ApproxPIM supports boh sequential and parallel execitions. Results (figure 2.74, 2.75) were then compared with a CPU+HMC system.

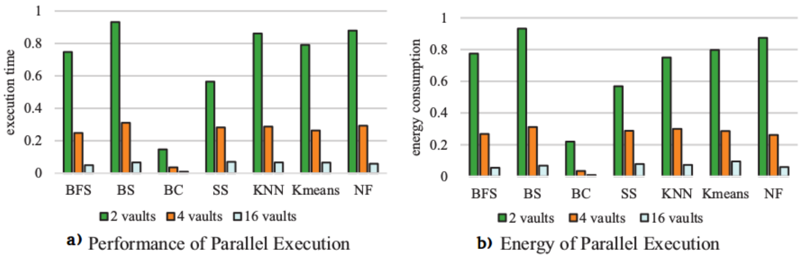


Y. Tang et al., 2017

Figure 2.74: Sequential Execution results. (a) Performance in terms of total execution time; (b) Energy consumption. From [23]

For sequential execution ApproxPIM achieves 21% speedup improvement with respect to the baseline (Host processor +2D DRAM), 8% speedup improvement with respect to HMC+CPU system and 68% less energy consumption with respect to HMC+CPU.

HMC provides inter-vault parallelism. Figure 2.75 shows performance of ApproxPIM for different numbers of vaults. It is noticeable that performance is improved when it is possible parallelize the benchmark and they further improve, the more parellism is added.



a) Performance of Parallel Execution

b) Energy of Parallel Execution

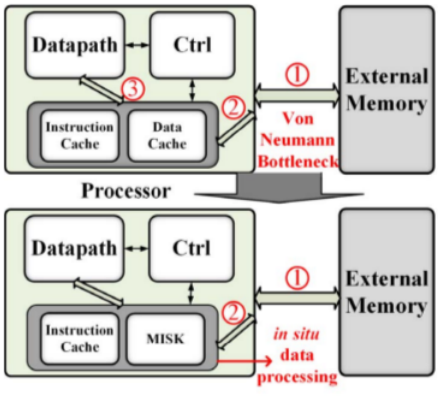
Y. Tang et al., 2017

Figure 2.75: Parallel Execution results. (a) Performance in terms of total execution time; (b) Energy consumption. From [23]

2.4.11 Interleaved Logic-in-Memory Architecture for Energy-Efficient Fine-Grained Data Processing [24]

This architecture aims to present a solution for the Von Neumann bottleneck, that is the disparity in speed between memories and processors.

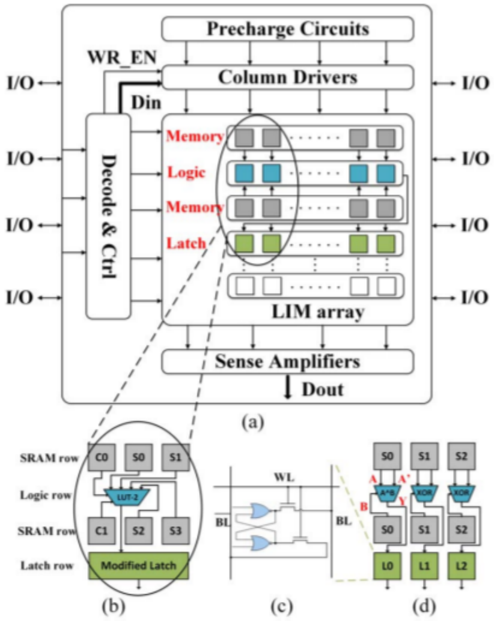
Nearly 75% of energy consumption is due to data movement, so reducing this would imply a great improvement in performance.
This architecture, called MISK, tries to implement a single, monolithic logic-in-memory structure, rather than keeping CPU and memory phisically separated.



K. Yang et al., 2017

Figure 2.76: MISK and conventional approach. From [24]

Rather than transferring data into processing elements to elaborate, with MISK (figure 2.76) data only need to be moved into the cache. This results in a reduction of data transfer overhead and enables massively parallel execution.



K. Yang et al., 2017

Figure 2.77: (b) LUT-based LIM unit; (c) Modified RS-latch; (d)XOR-based LIM unit. From [24]

CMOS logic for data processing is integrated into a 6T SRAM which is in a memory-logic-memory-latch configuration (figure 2.77).

Two classes of LIM units are introduced: XOR-based unit and LUT-based unit. XOR is a commonly used operation for cryptographic application and LUT offers FPGA-like flexibility.

MISK targets a wide application space, ranging from image processing to fine-grained computation.
To evaluate MISK architecture, 9 different applications were mapped into OpenRISC 1200 CPU with and without MISK integrated into the data cache. Simulations were performed using Or1ksim and Synopsys Design Compiler.

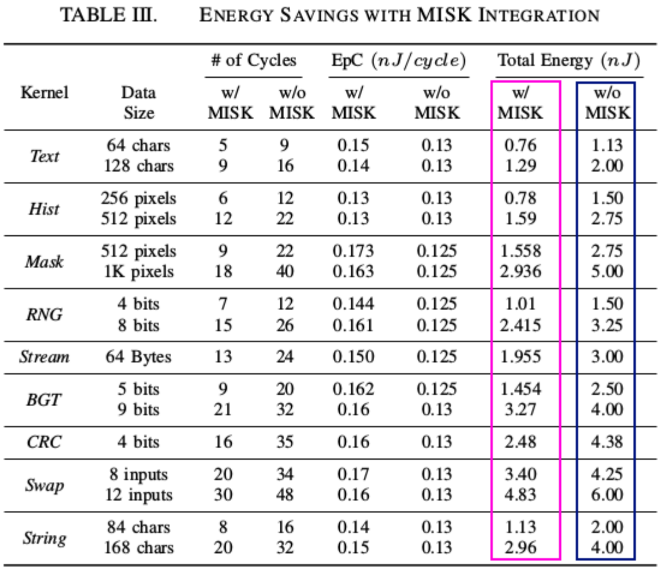


TABLE III. ENERGY SAVINGS WITH MISK INTEGRATION

Kernel	Data Size	# of Cycles		EpC $(nJ/cycle)$		Total Energy (nJ)	
		w/ MISK	w/o MISK	w/ MISK	w/o MISK	w/ MISK	w/o MISK
Text	64 chars 128 chars	5 9	9 16	0.15 0.14	0.13 0.13	0.76 1.29	1.13 2.00
Hist	256 pixels 512 pixels	6 12	12 22	0.13 0.13	0.13 0.13	0.78 1.59	1.50 2.75
Mask	512 pixels 1K pixels	9 18	22 40	0.173 0.163	0.125 0.125	1.558 2.936	2.75 5.00
RNG	4 bits 8 bits	7 15	12 26	0.144 0.161	0.125 0.125	1.01 2.415	1.50 3.25
Stream	64 Bytes	13	24	0.150	0.125	1.955	3.00
BGT	5 bits 9 bits	9 21	20 32	0.162 0.16	0.125 0.13	1.454 3.27	2.50 4.00
CRC	4 bits	16	35	0.16	0.13	2.48	4.38
Swap	8 inputs 12 inputs	20 30	34 48	0.17 0.16	0.13 0.13	3.40 4.83	4.25 6.00
String	84 chars 168 chars	8 20	16 32	0.14 0.15	0.13 0.13	1.13 2.96	2.00 4.00

K. Yang et al., 2017

Figure 2.78: Evaluation results. From [24]

Results (figure 2.78) show that MISK-integrated CPU achieves impressive improvement in terms of execution time; even if the energy-per-cycle value is grater than the simple CPU, MISK-integrated CPU requires less cycles, resulting in energy savings.

2.4.12 Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing [25]

Gilgamesh is a system based on shared and distributed memory. It is composed of multiple MIND chips linked with each other through a Global Interconnection (figure 2.79).

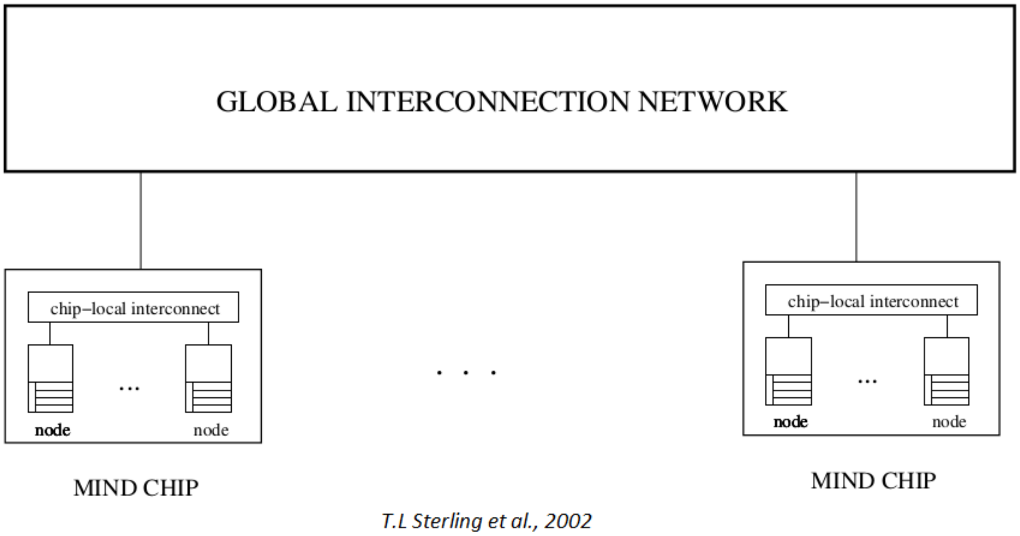


Figure 2.79: Gilgamesh Architecture. From [25]

Communication between chips is implemented with a parcel protocol. MIND chips (figure 2.80) are formed by multiple DRAM banks, processing logic, I/O interfaces and inter-chip communication channels. It is a general purpose structure.

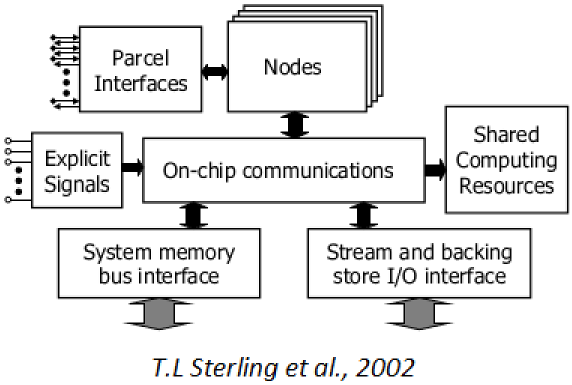
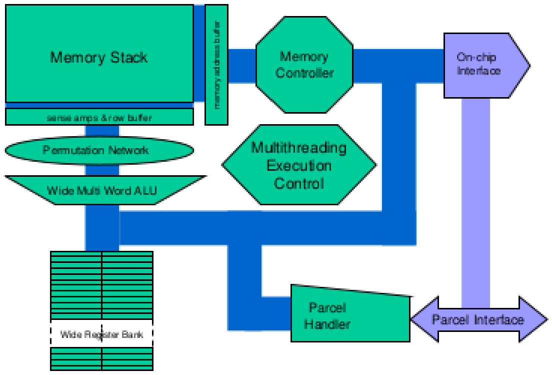

Parcel Interfaces
Nodes
Explicit Signals
On-chip communications
Shared Computing Resources
System memory bus interface
Stream and backing store I/O interface
T.L Sterling et al., 2002

Figure 2.80: MIND Chip Architecture. From [25]

MIND nodes (figure 2.81) provide computational capability to the chip, integrating memory with logic and control blocks.
Since a wide memory bandwidth is available, provided by DRAM row-wide access and the large amount of memory banks in a single chip, each memory bank is integrated with a wide ALU inside the row buffer, in order to process row-wide data.



T.L Sterling et al, 2002

Figure 2.81: MIND Node Architecture. From [25]

The system memory bus interface gives the means to interconnect MIND chips to conventional workstation and server motherboard memory buses.

A Gilgamesh prototype has been made using a specifically designed board, containing 4 high-density FPGAs and 8MB SRAM to represent two nodes and their interconnection. The prototype board operates at one tenth of speed of what an actual chip would be capable of but its performance is at least a thousand times greater than a gate level cycle-by-cycle software simulator.

2.4.13 Design and Evaluation of a Processing-in-Memory Architecture for the Smart Memory Cube [26]

Smart Memory Cube is a PIM architecture that enhances the capabilities of the logic base die in HMC.
The structure has been analyzed with an in-house simulation environment called SMCSim based on gem5. The simulator is capable of modeling a SMC device linked to a SoC Host. The host used is a Cortex A15.

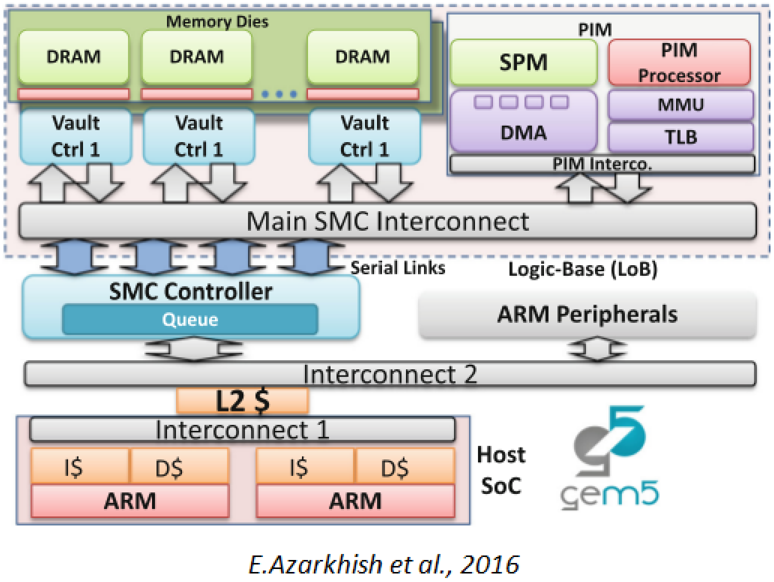
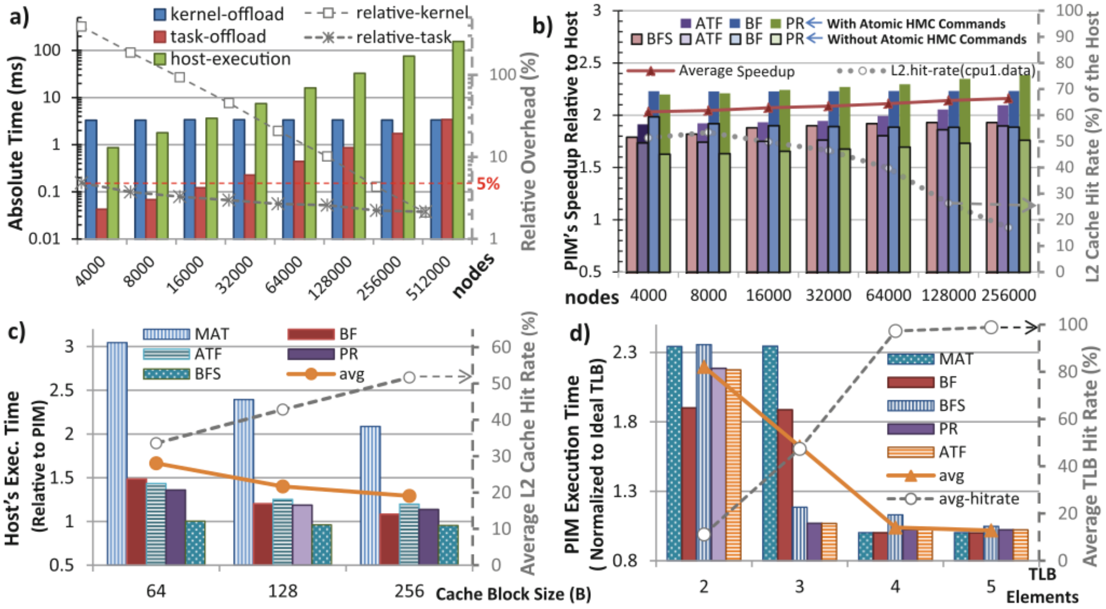


Figure 2.82: SMCSim Environment. From [26]

As figure 2.82 shows, PIM component is connected to the logic base through local interconnection. It is composed of a Scratchpad memory (SPMs), DMA engine, Translation Look Aside Buffer and Memory Management Unit.

TLB is used so that PIM can access user-space virtual memory directly. PIM has been enhanced with a DMA engine capable of bulk data transfer between the DRAM vaults and its SPMs.

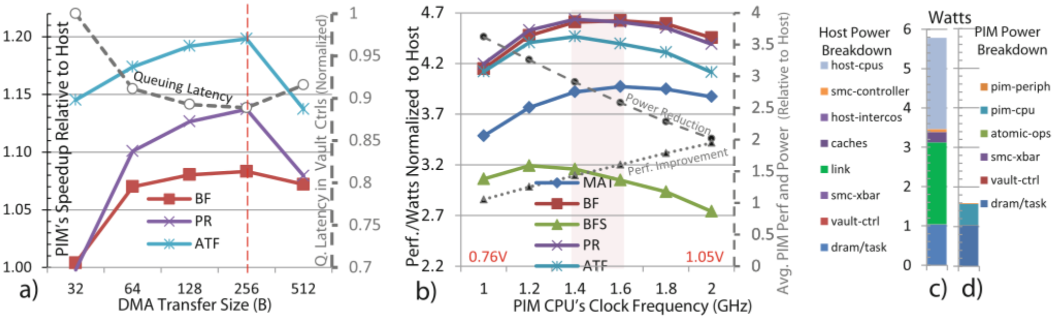
SMC has been simulated for data intensive application such as graph processing applied to social network applications such as Average Teenage Follower (ATF), Breadth First Search and Page Rank.



E.Azarkhish et al., 2016

Figure 2.83: Performance results. From [26]

SMC can reach up to 2x performance improvement in comparison with the host SoC and about 1.5x against a similar host-side accelerator (figure 2.83). Moreover, by scaling down frequency and voltage it is possible to reduce energy by about 55% and 70% with respect to the accelerator and the host respectively (figure 2.84).



E.Azarkhish et al., 2016

Figure 2.84: Energy efficiency results. From [26]

Chapter 3

The CLIMA Architecture

The architecture under develompment was named CLIMA, which stands for *Configurable Logic-in-Memory Array*. The main aim was then to achieve a configurable structure capable of performing operations directly inside the memory array.
For the development of the architecture, a *bottom-up* approach was adopted (figure 3.1).

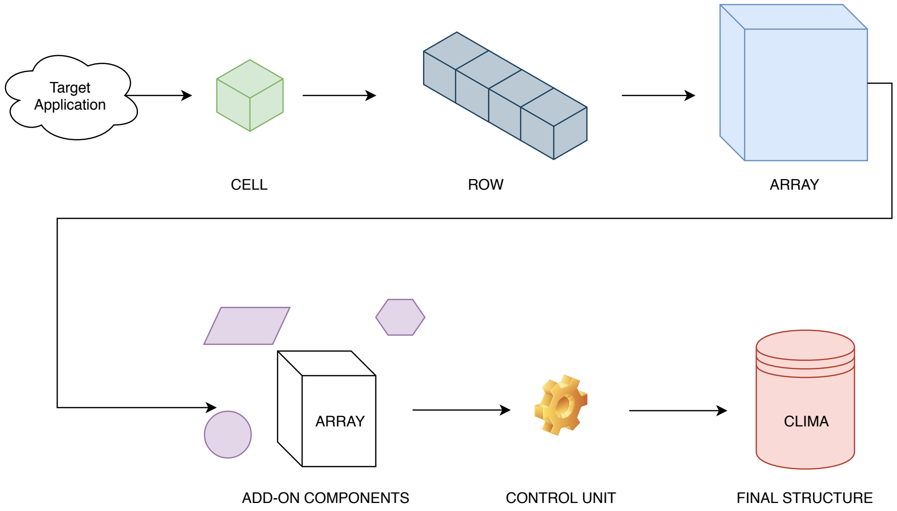


Figure 3.1: Bottom-up flow followed to develop CLIMA

Once a target application and thus an algorithm was selected, the first step was to design the LiM cell that was capable of performing the required actions. After that, a series of cells can compose a row and in the same way multiple rows would form the memory array. The next step was to define all the remaining components used to guarantee the correct functioning of the array. Once the datapath was fully designed, the control unit was implemented. The union of datapath and control unit brings to light the final CLIMA structure.

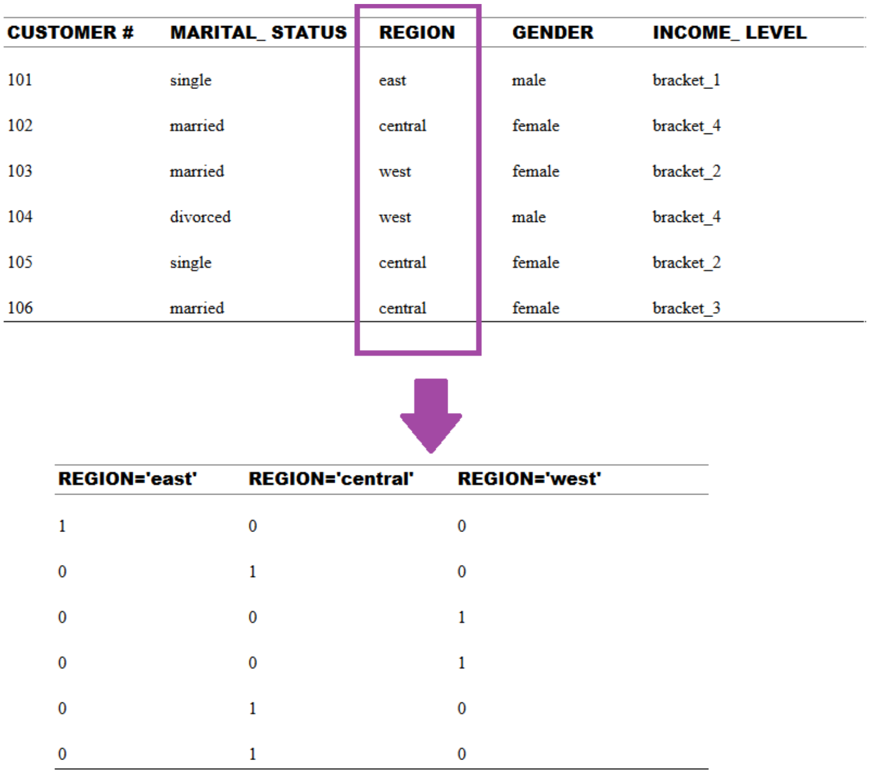
3.1 The application: Bitmap Indexing

Sometimes, when trying to implement something new, having a blank page and no boundaries can do more harm than good. However, thanks to the wide spectrum of possibilities gathered with the state of the art, a few possible applications and algorithms compatible with the Logic-in-Memory mentality were selected. Out of them, Bitmap indexes were chosen as a starting point.

Bitmap indexes are widely used in database management systems. Typically, a column in a table is composed of different key-values. Bitmap indexing transforms that column in as many bitmap indexes as each distinct key-value of that column. A bitmap is an array of bits in which the *i-th* bit of the bitmap is set to 1 if the value of the column in the *i-th* row has the same value corresponding to the one represented by the bitmap index. Any other position of the bitmap is set to 0. [33] An example is shown in figure 3.2.

Bitmap indexing is indicated for database with a low *degree of cardinality*, that is a table in which the number of distinct key-value is smaller with respect to the number of rows. It also provides reduced response time and it is most effective with queries in the ”where” clause [28]. This is because with bitmap indexing resolving queries results in performing simple **bitwise** logic operations, as shown in figure 3.3.

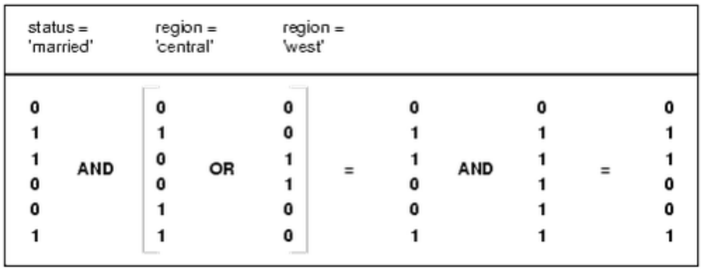
In figure 3.3, to answer the ”how many” query a counter that counts the hits in the resulting bitmap would be needed.



CUSTOMER #	MARITAL_ STATUS	REGION	GENDER	INCOME_ LEVEL
101	single	east	male	bracket_1
102	married	central	female	bracket_4
103	married	west	female	bracket_2
104	divorced	west	male	bracket_4
105	single	central	female	bracket_2
106	married	central	female	bracket_3

REGION='east'	REGION='central'	REGION='west'
1	0	0
0	1	0
0	0	1
0	0	1
0	1	0
0	1	0

Figure 3.2: An example of bitmap index. From [28]

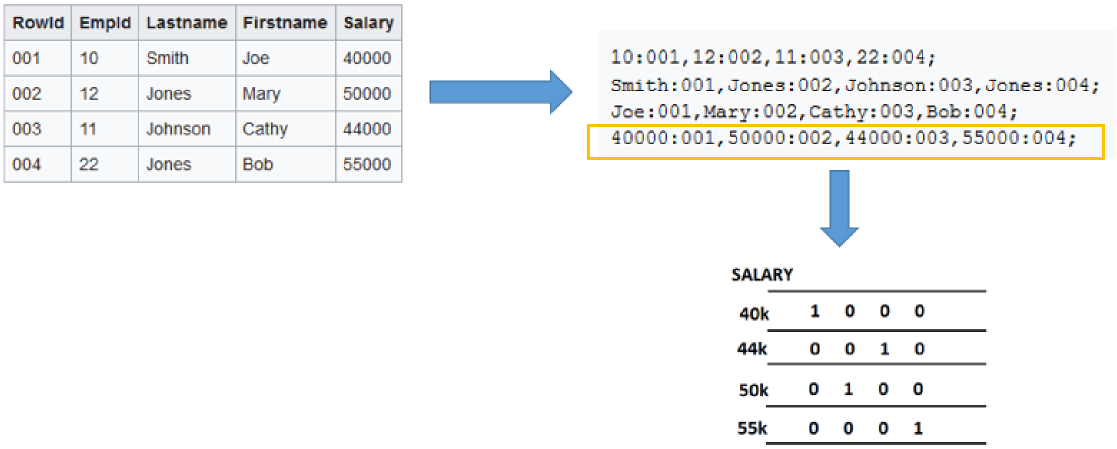


status = 'married'		region = 'central'		region = 'west'						
0		0		0		0		0		0
1		1		0		1		1		1
1	AND	0	OR	1	=	1	AND	1	=	1
0		0		1		0		1		0
0		1		0		0		1		0
1		1		0		1		1		1

Figure 3.3: An example of query using bitmap indexes. "How many married customers live in the west or central region?".From [28]

3.1.1 Data Organization

Usually, data of a table are stored in columns, in such a way that each column represents a particular attribute of the data set and one row contains the entire profile of a particular entry. However, for a Logic-in-Memory approach, this organization would imply operations between columns and so, accessing multiple rows and discarding unwanted data. Such an approach would have been too costly. For this reason, a Column-Oriented organization was preferred [29]. The entire table is transposed so that each column now lies as a row. With this method accessing one row will allow to have access to the whole index (figure 3.4).



RowId	EmpId	Lastname	Firstname	Salary
001	10	Smith	Joe	40000
002	12	Jones	Mary	50000
003	11	Johnson	Cathy	44000
004	22	Jones	Bob	55000

```
10:001,12:002,11:003,22:004;
Smith:001,Jones:002,Johnson:003,Jones:004;
Joe:001,Mary:002,Cathy:003,Bob:004;
40000:001,50000:002,44000:003,55000:004;
```

SALARY				
40k	1	0	0	0
44k	0	0	1	0
50k	0	1	0	0
55k	0	0	0	1

Source: https://en.wikipedia.org/wiki/Column-oriented_DBMS

Figure 3.4: Column-oriented bitmap memory organization. From [29]

3.1.2 The Algorithm

As said in the previous section, a query can be fragmented into many bitwise logic operations.

For *bitwise*, it is intended the processing of a bit vector with another making interact the bits belonging to the same position in the array. For instance, if an AND operation is to be performed between two operands (e.g 8-bit each), the steps to follow are:

1. Read Operand1;
2. Read Operand2;
3. Perform AND between each bit of the operands, respecting their position;

4. Logic result is computed;

This example is depicted in figure 3.5.

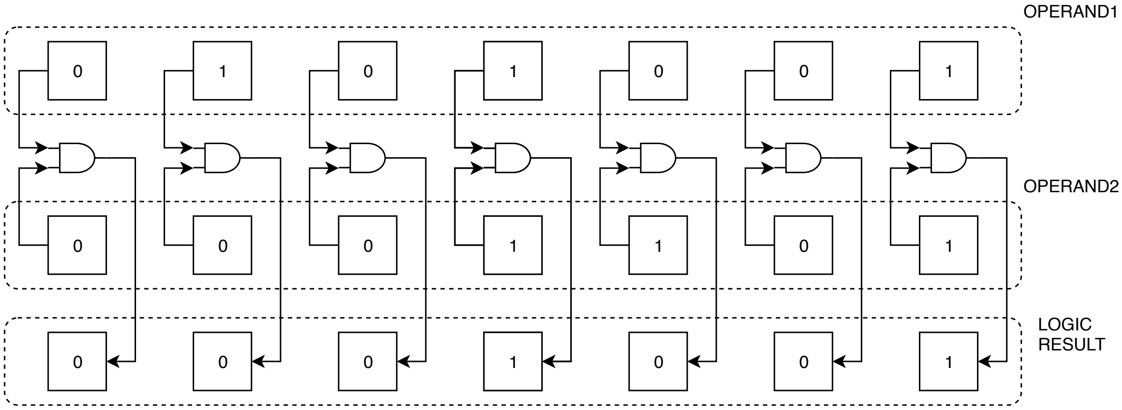


Figure 3.5: Bitwise AND between two operands

Types of Query

In this implementation will be considered two different types of query:

- Simple: a query composed only of an operation between two operands.
  E.g: $A \cdot B$;

- Composed: a query composed of two operations.
  E.g: $(A \cdot B) + C$;

3.2 CLIMA Datapath

3.2.1 The Cell

The first step consisted in defining the LiM cell. According to the selected application, what was needed were bitwise logical operations between rows inside the array. So, the characteristics to implement were the following:

- standard memory features (*read,write*);

- configurable logic operations(*inter-row bitwise operations*);

For this reason the cell was divided into two different elements: memory and logic. An internal view of the cell is shown in figure 3.6.

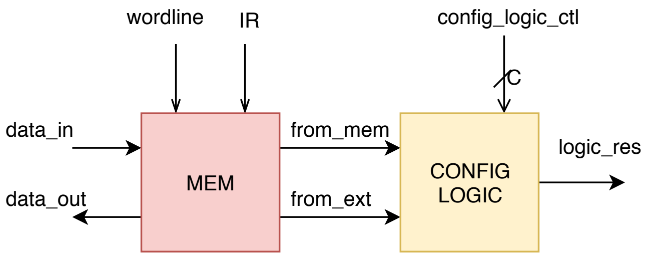


Figure 3.6: Internal view of the LiM cell

The memory element is in charge of storing information and connecting external data with the logic, this is made to avoid a second input in the cell. A permanent connection between the memorized data and the logic is present. This way, when performing an operation, the data stored in the cell does not need to be read. The IR control signal, alongside the traditional wordline, is used to manage the data flow inside the cell (figure 3.7).

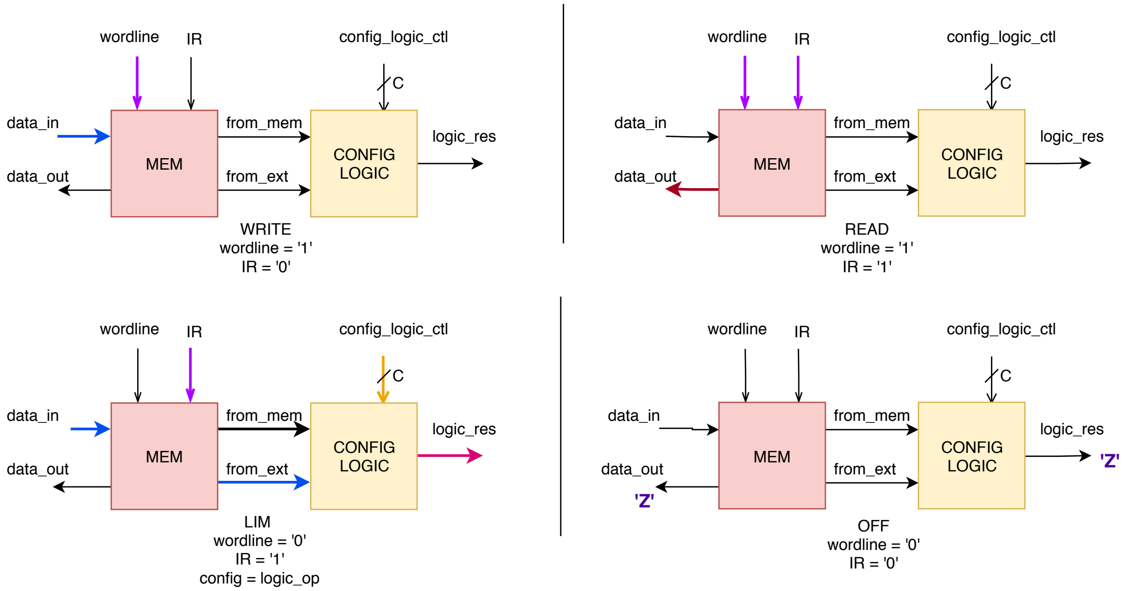


Figure 3.7: Data flow inside the cell, according to different input. Highlighted arrows indicate active signals

In figure 3.7 are depicted all the operation modes of the LiM cell.

- WRITE: the wordline is active and the input data is saved inside the cell;
- READ: wordline and IR are set to 1 and the data stationing in the cell is sent as output;
- LIM: only IR is active. An external data enters the cell up the logic block so that the cell is able to execute the wanted operation, according to the logic configuration control signals that are sent in input;
- OFF: both wordline and IR are set to 0. This means that the entire cell is switched off and its outputs impose high impedance , in order not to disturb other cells.

The logic element has the external data and the stored one as input. According to the different number and the kind of operation to perform, a C-bit configuration control vector is sent as input to the logic. The result of the logic operation is then sent in output. For this application, AND,OR and XOR operations were implemented, with all the possible combinations of inputs (e.g A, not(A)),for a total of 12 possible operations and a 4-bit control vector. The logic block was implemented as a structural block, since the behavioral counterpart had too much area overhead. An internal view of the configurable logic block is represented in figure 3.8.

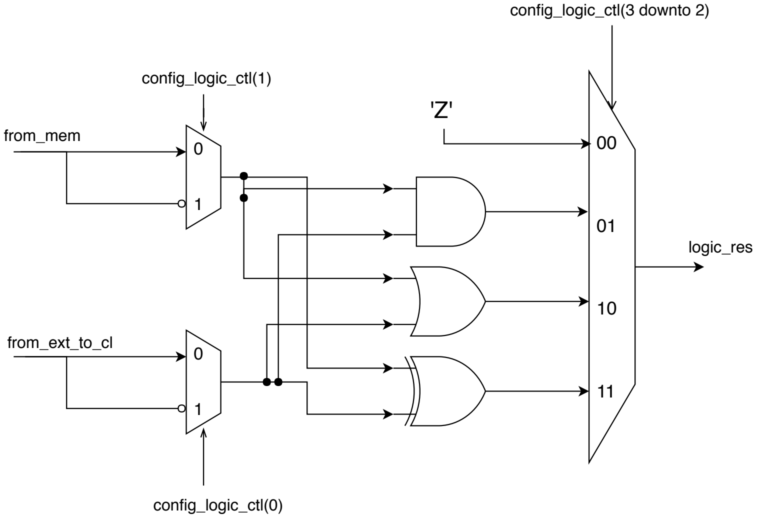


Figure 3.8: Internal view of Configurable Logic

As shown in figure 3.8, the logic block perform the operation represent by the configuration signal. The all-zero signal is used to keep the block switched off sending as output high impedance.
The look of the cell as a whole is shown in figure 3.9.

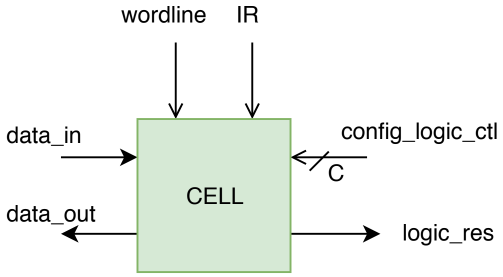


Figure 3.9: Top view of the LiM cell

Now, the interaction between cells will be explained.
Computing is performed between two operands, so, since each cell is capable of logic computation, a decision about which one actually performs it is needed.
Considering two cells belonging to different rows, but that have the same position in the row, as bitwise operations demand, one of them has to be read, so that the data inside it will travel to the logic block of the second cell to execute the operation (figure 3.10).

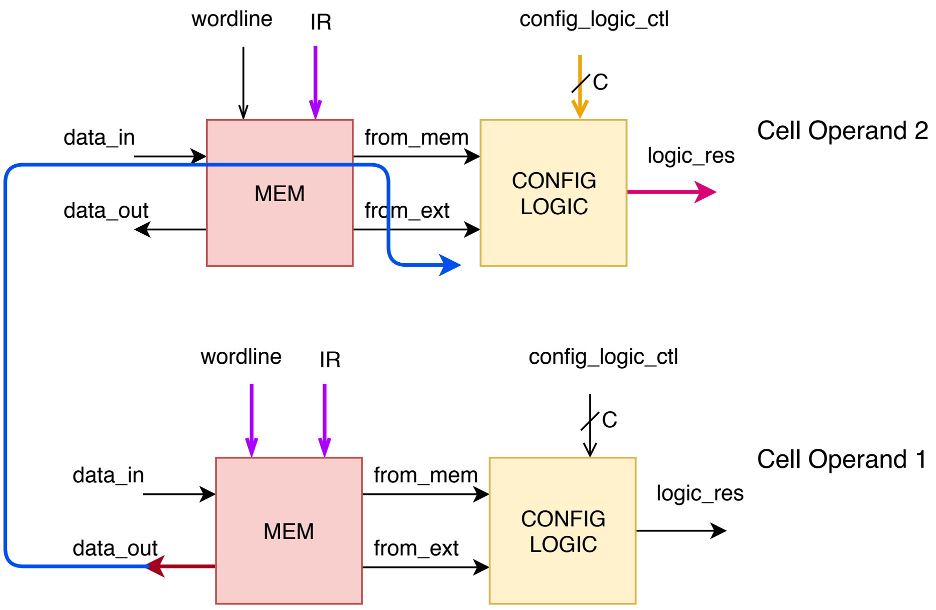


Figure 3.10: Interaction between cells during LiM operation

This means that the input of every cell has to be common to the input of every other cell, same for the outputs and a connection between input and output is necessary. Thinking to a standard memory, this principle is implemented by the bitlines shared between cells.

As a consequence, a way to avoid conflicts along the lines was needed. Thus, the cell control signals `wordline` and `IR` are used not only to read, write or to connect input to logic but also to set the outputs as high impedence, so that if the cell is not used it will not cause interference to the others.

Ghost Cell

As said before, the logic result is sent as one of the output signals of the cell. But as for where it goes it was not yet specified. There were many options to consider, such as overwriting the content of the cell who performed the operation to save the result, but it would have meant the deletion of the database. Another choice was to add a second element of memory in the cell, resulting in a doubling of the total area.

A third way was chosen. An additional row was added, composed of a cell equipped only with the memory element in order to store the freshly computed logic output.

Obviously, the same interface as the LiM cell has been kept to guarantee congruence with the rest of the architecture. These cells were given the name of *ghost cells*, to underline the purpose to store temporary data (figure 3.11).

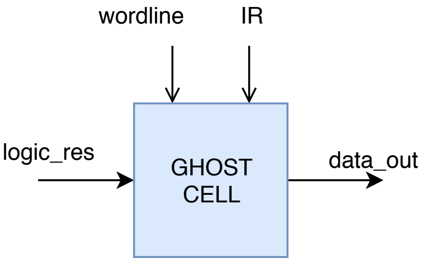


Figure 3.11: View of a Ghost Cell

The output of the Ghost Cell is common to the LiM cells. This is done to enable the possibility to read the content of the Ghost Cell whenever needed.

The acronym IR stands for Interal/external Read/write. It refers to one of the first drafts of the LiM cell, before the introduction of the ghost one, when the case of overwriting the content of the cell was implemented, making that the fifth functioning mode of the cell, coded with three bits: two of IR and the wordline. Without it, the possible actions are restricted to four, coded in a two bit control signal. For this reason IR switched from a two-bit to a one-bit signal, but its label stayed unchanged.

3.2.2 The LiM array

Once the two types of cell were defined, it was possible to implement the whole LiM array.

A series of ghost-cells were put in line to form the ghost row, while the LiM-cells were put together to compose the rest of the array.

Each row is controlled by the same wordline and IR control signal, the same for the ghost one. But the logic configuration bits are sent only to LiM-rows.

The input line is common for every LiM-row, while the output line is also common with the ghost row giving the user the possibility to read its content or re-using it for further computation.

In a standard memory, an organization based on rows divided into words is a commons solution, since moving big amounts of data all together would be too costly. So, each row was divided into 2^W words, where W is the bit size of the word address. To avoid conflicts between words belonging to the same row, an enable word bit vector signal had to be introduced for each row to activate one word at a time. The interaction between words is the same as explained previously for the cells. Between two operands, one is set to be read and the second to "absorb" the first operand and perform execution. Then, the result is immediately saved in the ghost word having the same word-address as the second operand. This was made to simplify the control of the array, since having to choose where to sent the result would have implied the need to manage a third address. To make connection between input and output possible when needed, a multiplexer was inserted as input to choose between an external data, as it would be required in a normal memory functioning, and the output of one of the other LiM-words necessary for LiM operations. The selection bit of the multiplexer was then called LIM, and it is set to 0 if the array is working in `standard memory mode`, to 1 if it is in `LIM mode`. The final array composition is shown in figure 3.12.

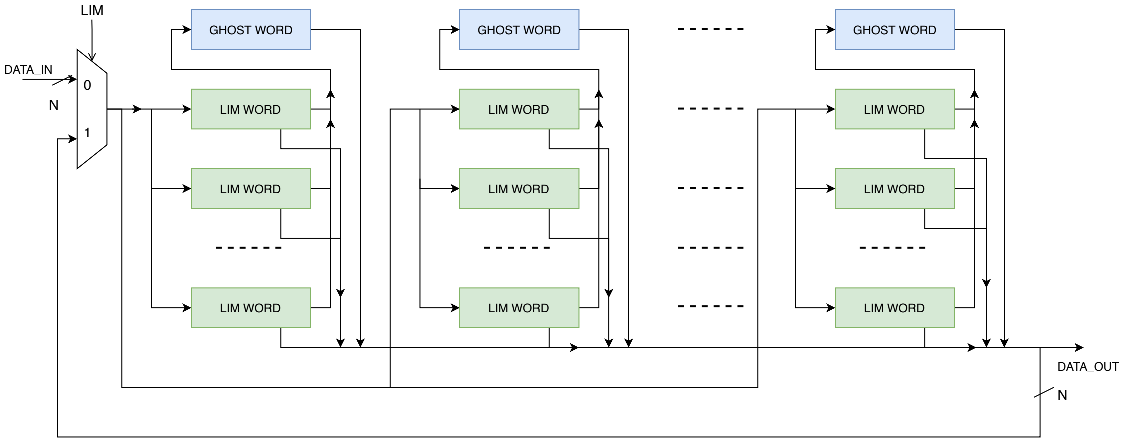


Figure 3.12: View of the LiM array. Word control signals are omitted for clarity. N represent data size

Decoders

The presence of so many control signal imposed the use of custom-made decorders, to simplify the control at a higher level. Two decoders were implemented:

- Configurable Logic Decoder;
- IR Decoder;

Configurable Logic Decoder Having so many bit of configuration for each row, connecting all of them from the outside would be too expensive. This is why it was implemented a decoder capable of managing all the configuration bits of an array. As input, an enable signal, the logic operation coded on C-bit (considering the implemented logic, C will be 4-bit large) and the address are sent. The decoder set the configuration bits in correspondence to the input address and all the rest are set to 0, that is the default configuration that indicates that the logic is switched off. The enable signal is used to activate or not the decoder (figure 3.13).

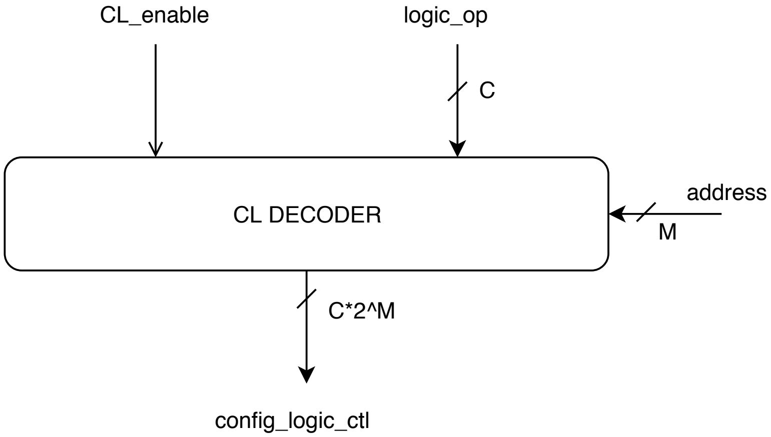


Figure 3.13: View of the Configurable Logic Decoder

IR Decoder As figure 3.14 shows, IR Decoder has many inputs. They will be explained briefly:

- `Address 1` is composed of word address and row address of the first operand of the operation, that, as said before, is the one to be read, travelling to the second;

- Address 2 is composed of word address and row address of the second operand of the operation. The row segment of this address is also sent to the CL decoder;

- mem is set to 0 when the structure is working in standard memory mode. In this mode, only address 1 is decoded, to activate the corresponding word to be read or written. If it is set to 1, the structure enters LIM mode and outputs enable_word, wordlines, IRs are set as the operation demands;

- IR1,IR2 are the IR signals to insert according to the input addresses, the working principle is the same as the CL decoder;

- en_wl is used during LIM mode to read the content of the ghost word, sending it to another word in the array for further computation;

- enable is used to enable the decoder. If disabled, it keeps the whole array to a switched off status.

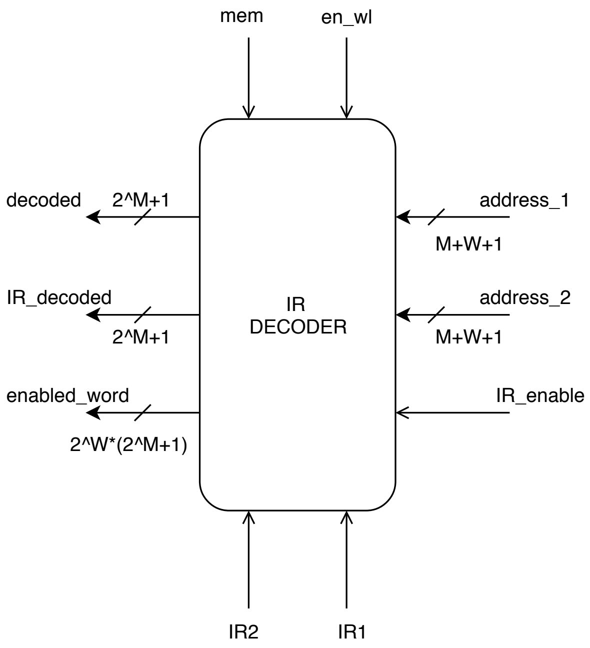


Figure 3.14: View of the IR decoder

Array size of the addresses and as a consequence of the input vector is `M+W+1` because it includes word address as well as row addresses, that in one more then the wanted size due to the presence of the ghost row.

Word and Row Selectors

During the early stages of the synthesis phase the array as it was had a pretty long output delay. The reason turned out to be the large amount of load capacitance oppressing the output line, due to the fact that each single word of the array was connected to it. To lessen the weight on the line, a word selector was inserted in each row, exploiting the enabled_word output of the IR decoder to sent the desired word in output and a row selector was inserted for each array to select between the pre-selected words, exploiting the wordlines signal from the IR decoder. This way the control unit did not need to be modified and the delay on the line was deeply reduced. In figure 3.15, it is possible to observe what would be the load on the output line without this component.

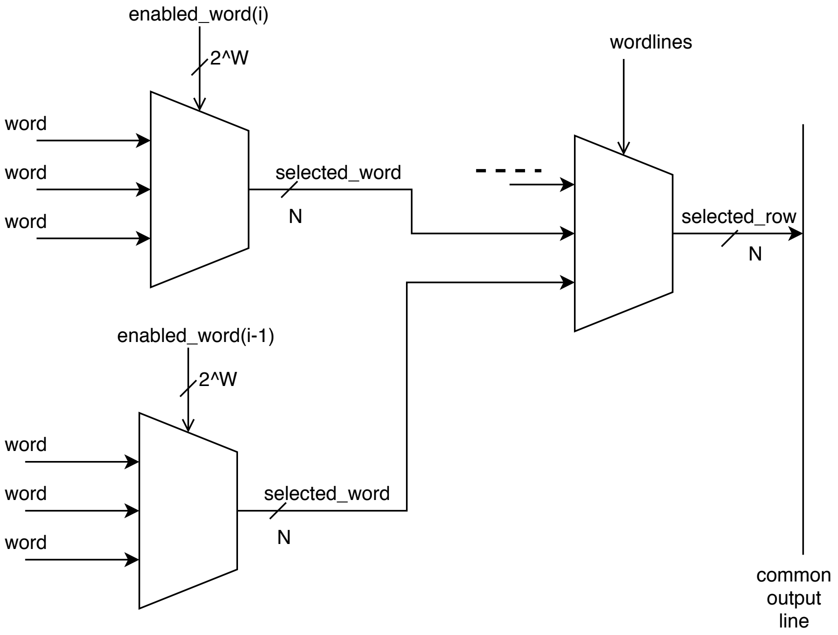


Figure 3.15: View of two word selector connected to the output line

3.2.3 CLIM Bank

The structure depicted so far is shown in figure 3.16.
Since in this structure input and output are connected together during the LIM operation, inside this structure can be performed only one operation at a time.
So, it was decided to divide the whole memory array into such units, from now called **CLIM Bank**. A bank is the smallest unit of parallelism inside the whole array.

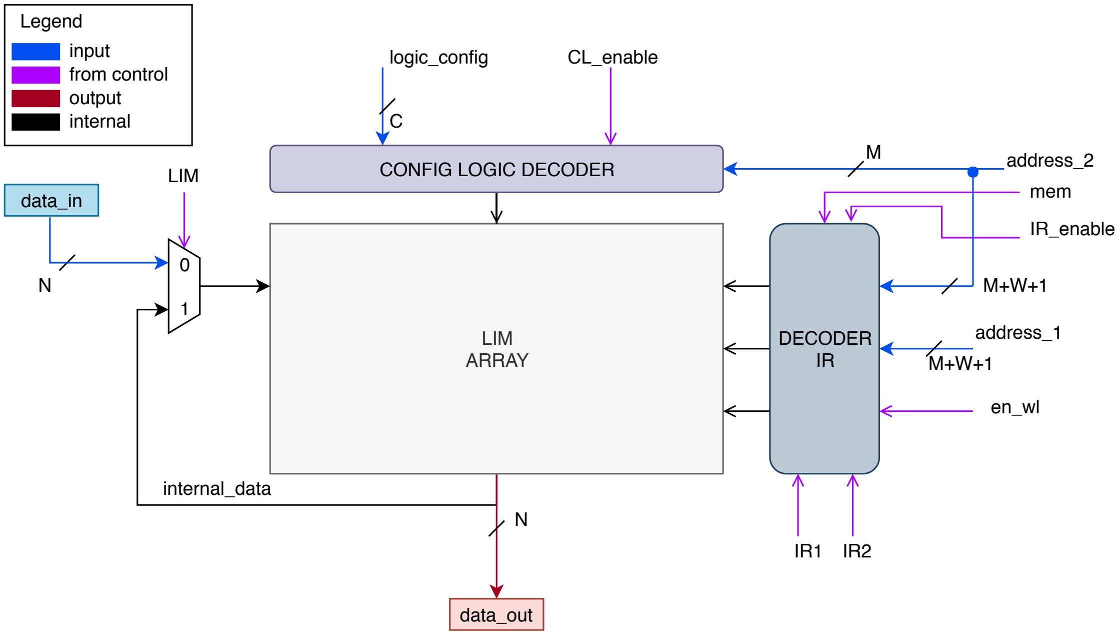


Figure 3.16: Internal view of a CLIM Bank

The complete structure can be composed of as many banks as it is thought to be appropriate. The idea was to exploit the bank ability to isolate itself to implement a degree of parallelism that could be at maximum equal to the number of banks, allowing more of them to work together or independently in parallel with other banks.

Bidirectional Breaker

One of the most tricky issue about the development of CLIMA was to manage interconnections. This was already observed during the implementation of the LiM rows themselves.

As far as multiple banks are concerned the same principle of interconnection between rows in a bank still stands: the input line is common to each bank and the same happens for the output line. This condition imply that if two banks are intended to work with each other, they will occupy the data bus, preventing other banks to do the same. For this reason, a method to isolate the banks with each other, in order to guarantee the maximum possible parallelism, was introduced. A Bidirectional Breaker is instantiated for each clim bank. The component is shown in figure 3.17.

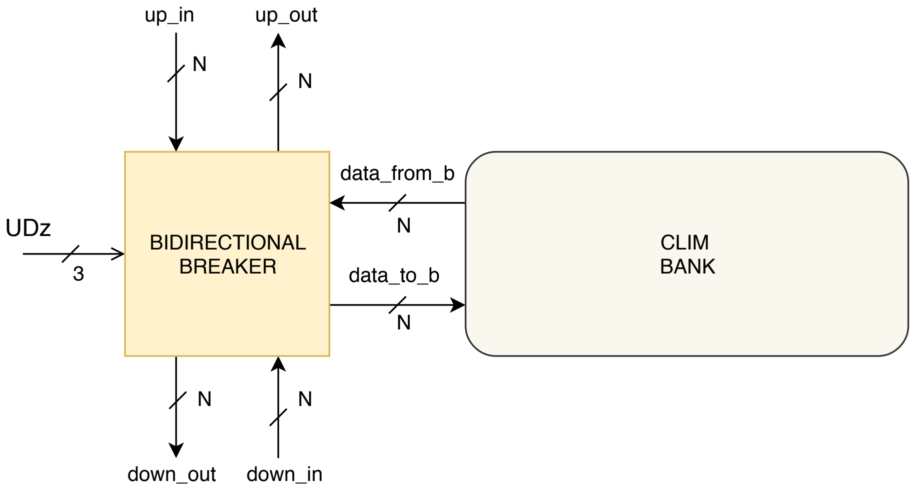


Figure 3.17: View of the Bidirectional Breaker

According to the 3-bit control signal, the Bidirectional Breaker is in charge of diverting the data that passes through it to the desired direction, allowing data to travel everywhere in the array. If switched off, the breaker imposes high impedance to all of its outputs, isolating the bank or entire regions of the whole array (figure 3.18).

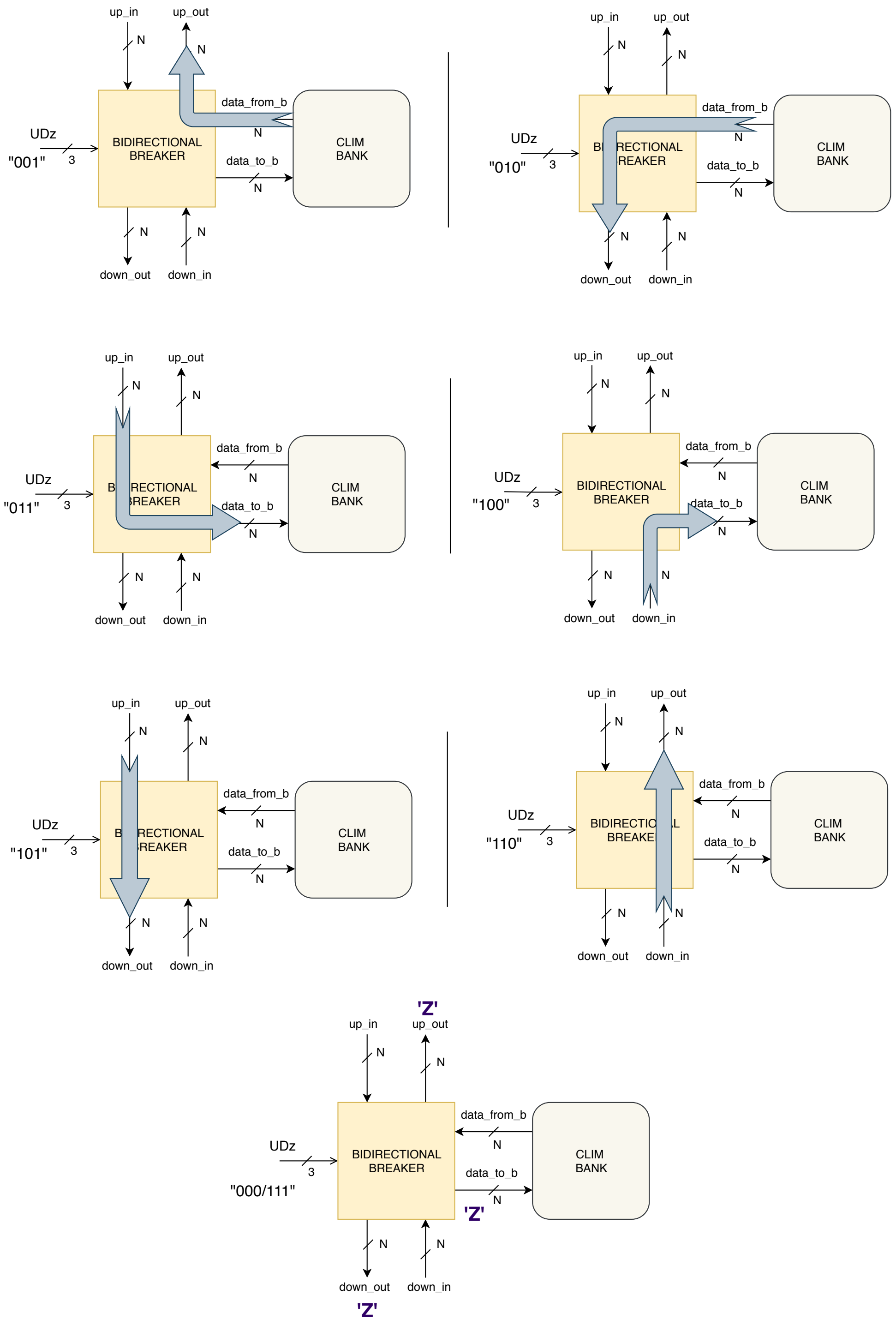


Figure 3.18: Data flow inside the Bidirectional Breaker according to different control inputs

Looking at figure 3.18, it is possible to observe that the breaker opens only one output at time, leaving the other two switched off. It can be ”active” sending the data of its own bank to the rest the array or receiving data for it, or it can act as a ”passive” component, letting data pass through it. Thanks to the fact that even if the breaker lets data pass , this does not interfere with its bank, it is possible to implement parallel operations, with banks that work with each other, while the isolated ones can work by themselves.

3.2.4 CLIM Bank Array

The duo Bank-Breaker is then used to compose the final structure of the LiM array. Multiple copies of the duo are connected together to form the array (figure 3.19). Since each bank represent the smallest unit of parallelism and the entire implementation is parametric, a lot of combinations can be made, according to the performance to achieve. It is possible to choose a configuration with lots of banks with few rows, or few banks with more rows and so on.

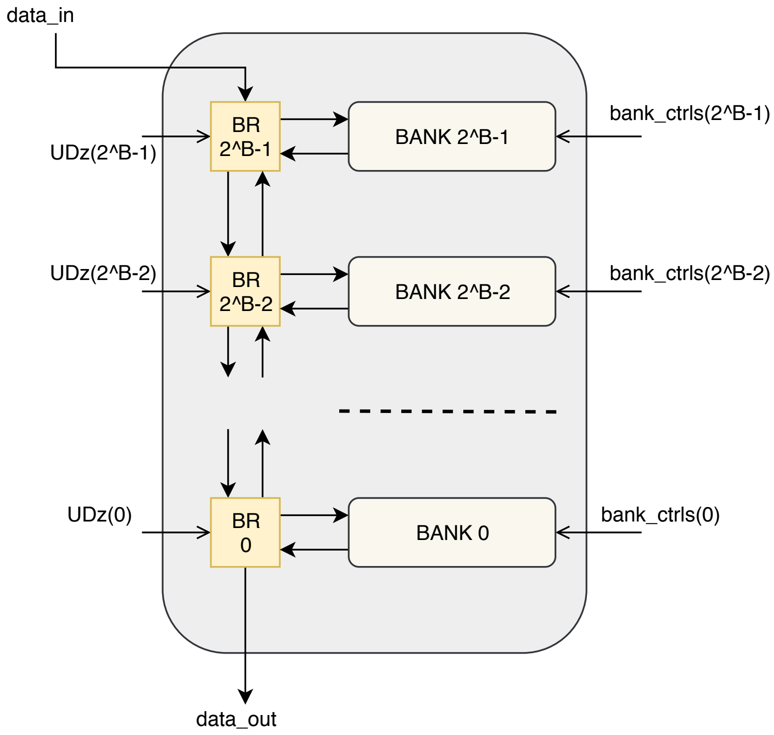


Figure 3.19: Array of banks and breakers

3.2.5 Instruction Memory

To allow the structure to work at the maximum of its capabilities, that is performing as many parallel operations as the number of banks, it was necessary to provide the architecture with the components to manage such eventuality.
The Instruction Memory is a register file containing as many registers as the number of banks of the array, with dimensions equal to the sum of twice the size of a complete address (bank, row and word) and the size of the logic configuration(identified in figure 3.20 with letter O) . Therefore, a so called "instruction" is a segment of a query, which contains the addresses of the two operand and the desired operations. The rest of the structure of a complete query will be explained later.
A single load signal was used, to simplify the control unit.

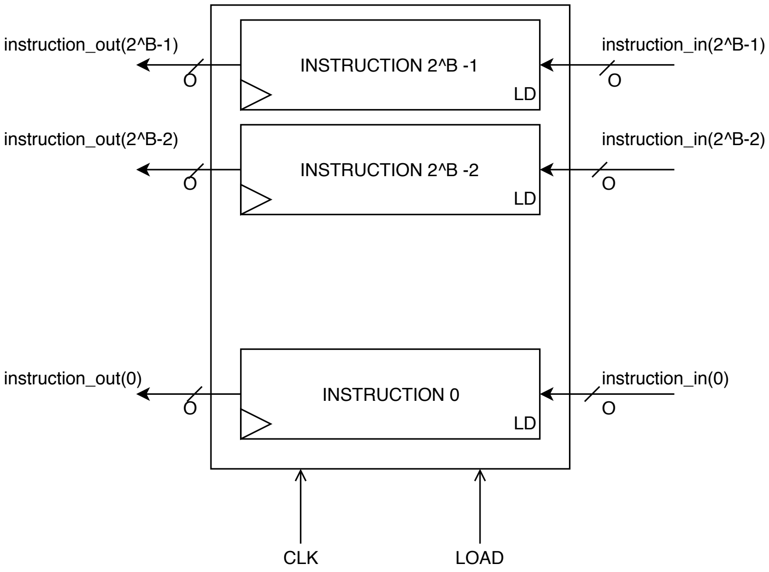


Figure 3.20: View of the Instruction Memory

3.2.6 Operation Dispatcher

Due to the versatility of the architecture, the operands of the operation to be performed could be found everywhere in the array. Thus, the needed addresses (row+word) had to be sent to the appropriate bank. A logic block, the Operation Dispatcher served this purpose.

As said previously, CLIMA had to be equipped according to its major potential, but this does not mean that it is the only way it can perform. Therefore, the instruction memory was needed as big as possible, but in most cases, it will not be updated fully. This is why an `enable_operation` control signal was needed. According to how many new parallel operation are loaded and thus enabled, the dispatcher is in charge of reordering the addresses, sending them to the right bank and ignoring the disabled ones. Also, two arrays `bank_1` and `bank_2` are sent to the control unit to manage the operations. They contain the addresses of the banks that are used in operation order (figure 3.21). For instance, if during the operation with position 0 bank 2 and 3 are used, the arrays will contain 2 and 3 in position 0, respectively.

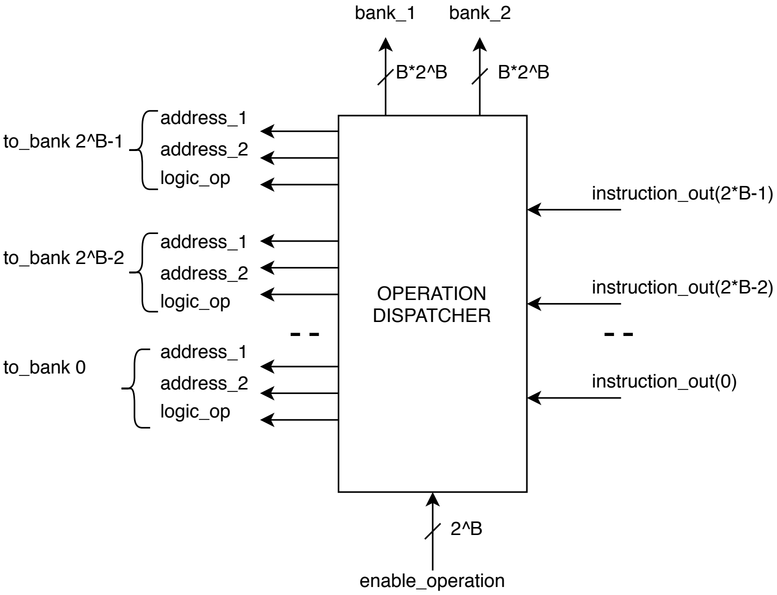


Figure 3.21: View of the Operation Dispatcher

3.2.7 Addresses Register File

To ensure synchronization, the well-ordered addresses, before being directly connected to their banks, are sent to a register file, where they are loaded and then sent to the array. Each banks own its triplette of registers with two addresses and one logic configuration. Clock and load signals are common for all the internal registers (figure 3.22).

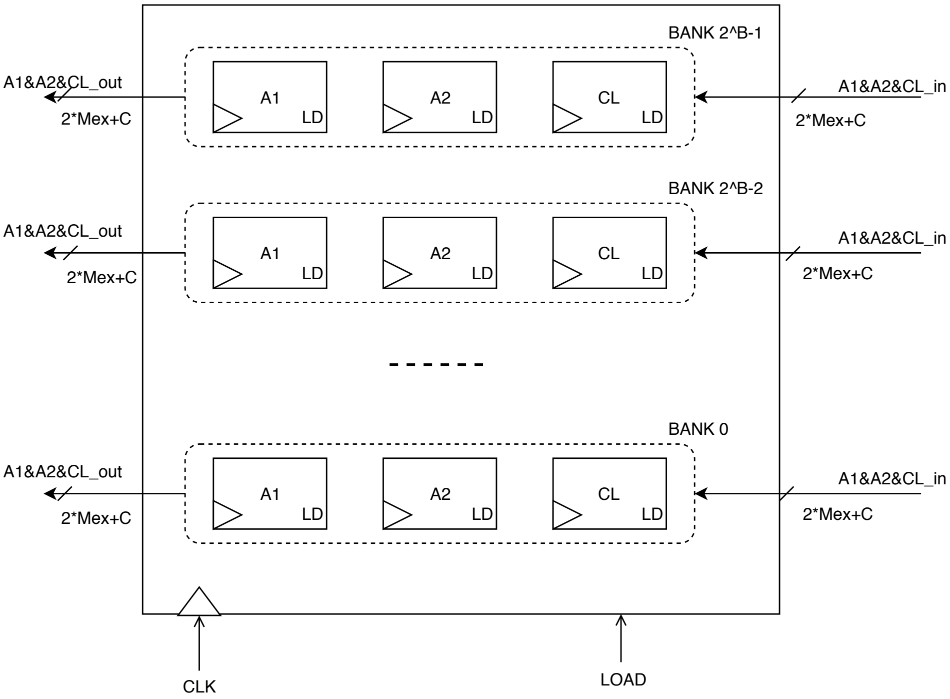


Figure 3.22: View of the Addresses Register File. Mex indicates the extended address composed of word and row, considering the plus one of the ghost row

3.2.8 Ones Counter

Even if at the beginning the idea was to incorporate every single aspect of computation in the memory array, it was soon realized that inserting a ones counter in the array would have been too costly. Moreover, a simple counter in which the input data would have been analyzed bit-by-bit, incrementing a counter for each '1' found, was too slow. For this reason a tree-structured counter was implemented.

The data array is firstly divided into D segments, each composed of $\frac{N}{D}$-bits, then each segment is processed at the same time to count how many ones it contains and eventually all segments are added together following a tree structure. To make implementation easier, the adders that compose the tree structure are all of the same dimension, computed to avoid overflow, set as $\frac{D}{2} + \frac{N}{D}$.

The counter is also sensitive to the negative edge of the clock, giving the component at least half a clock to perform the counting before the output register, that is sensitive to the positive edge, loads the result. A reset signal is present to reset the counter and it is set to 1 when the counter is needed. The top view and internal structure of a 8 bit counter with 4 segments is shown in figure 3.23 and 3.24.

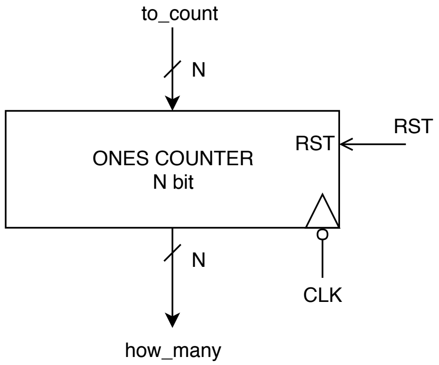


Figure 3.23: Top view of the Ones Counter

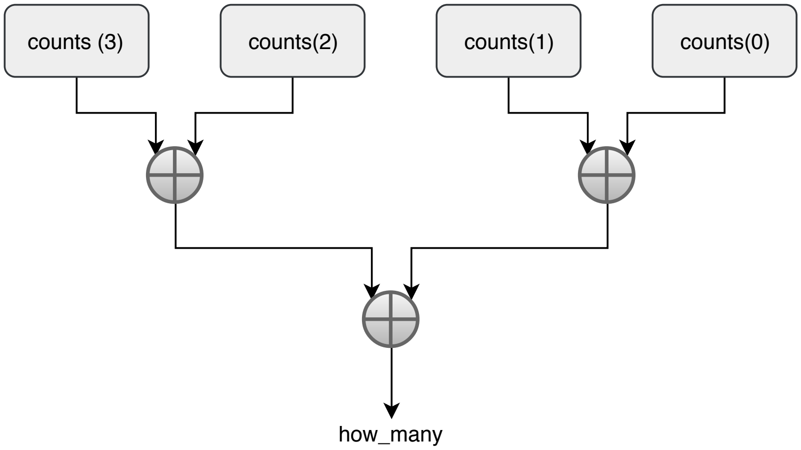


Figure 3.24: Internal view of an 8-bit ones counter divided into 4 segments

3.2.9 The complete CLIMA Datapath

The complete structure of the CLIMA Datapath is shown in figure 3.25. Input (Data In) and output (Result of Query) registers were added. A multiplexer is interposed between the array and the input register, giving the user the possibilty also to save in the array the result of a previous query, maybe instantiating an empty bank to save results, in order to avoid deleting part of the database. Another multiplexer at the output chooses between the ”who” answer of the query, represented by the output of the array, or the ”how many” answer represented by the output of the counter.

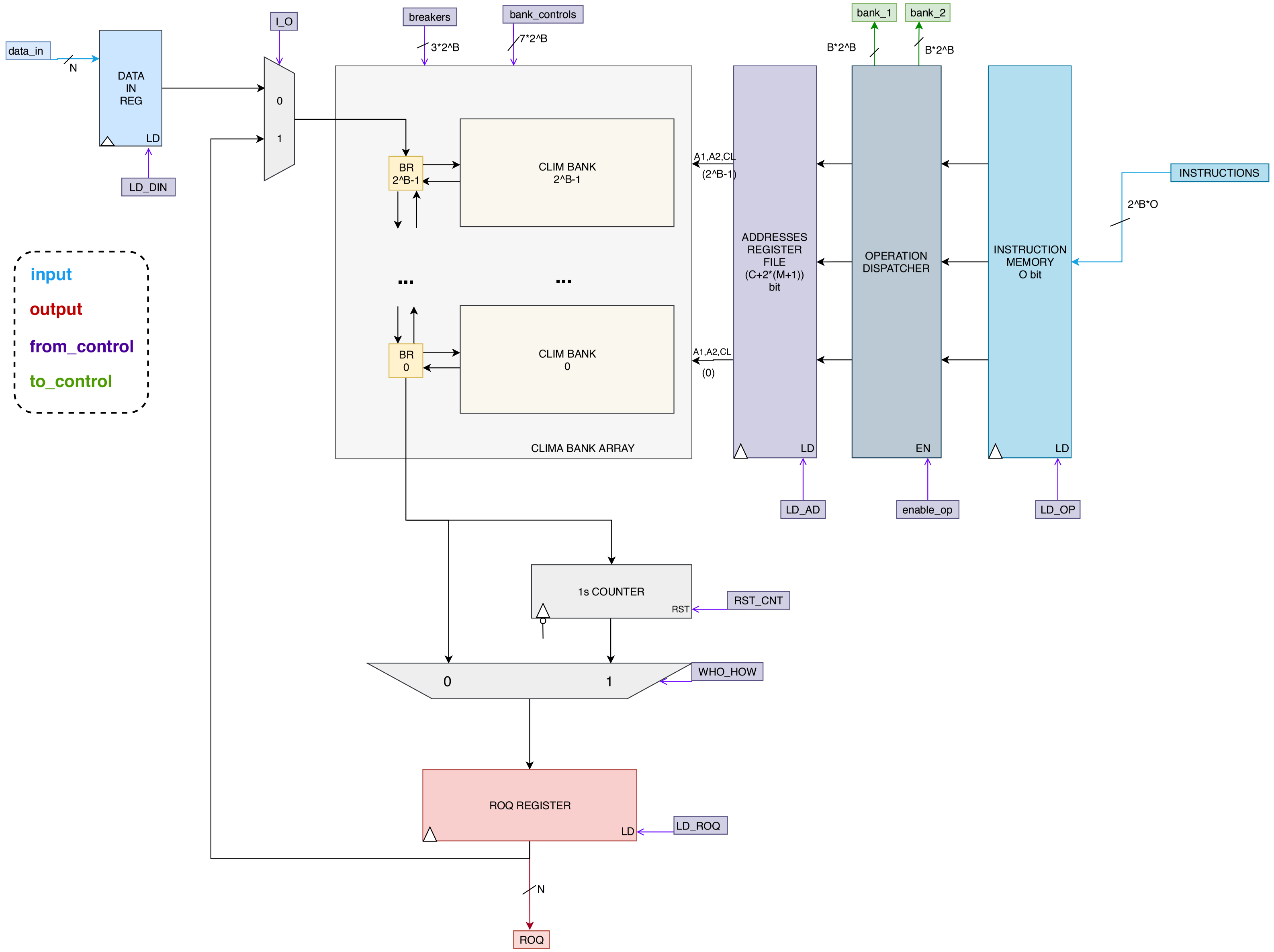


Figure 3.25: View of the CLIMA Datapath

3.3 CLIMA Control Unit

The final step of the implementation of CLIMA was the design of the Control Unit. The management of all the possible operating modes of CLIMA and as a consequence the management of the data flowing everywhere in the array thorugh the breakers, made it difficult to implement the control as a microprogrammed unit. This is why the Control Unit was implemented as a Finite State Machine.
8 different operation modes were defined, coded on a 3-bit vector called OP_MODE:

- WRITE(000): an external data is written in the location pointed by the input address;
- READ(001): the data located in the input address is sent as output
- SAVE(010): the result of a query loaded in the output register is saved in the location pointed by the input address;
- LIM single same Bank(011): a LIM operation is performed between two operands belonging to the same bank. The result is immediately sent in output;
- LIM single different Bank(100) : a LIM operation is performed between two operands belonging to different banks. The result is immediately sent in output;
- LIM Multiple Banks(101): multiple LIM operations are performed in parallel, between operand belonging either to the same banks or to different ones;
- LIM composed single(110): a complex LIM operation is performed in two steps. Then the result is sent as output
- LIM multi composed(111): Multiple complex LIM operation are performed in parallel.

Each operation mode is the starting point of a query, which is composed as shown in figure 3.26.

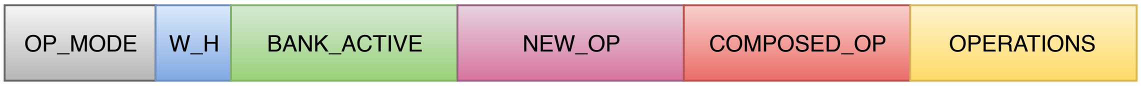


Figure 3.26: Composition of a complete query

- OP_MODE: 3-bit vector to decide the operation mode of the query;

- W_H: bit set to 0 is the query to answer is ”who”, set to 1 is the query asks ”how many”;

- bank_active: mask composed of as many bits as the number of banks in the array. It is used to simplify the control unit and it indicates which banks are active during the execution of the query. This is done to avoid interference caused by unused banks;

- new_op: mask composed of as many bits as the number of banks in the array. It is sent to the dispatcher which will perform assignation of the operations that are actually being executed;

- composed_op: mask with as many bits as the number of banks in the array. During a simple operation is set to 0, whilst during a composed operation it is in part complementary with new_op (depending on the total number of operations, the rest is set to 0 for both masks) and it is used in the second part of the execution, substituing new_op as enable for the dispatcher;

- operations: is a sequence of bit vectors (separated by a space), each of which is as long as the dimension of an instruction (twice a complete address plus the logic operation, as explained in section 3.2.5) and it is sent as input to the datapath, in particular to the Instruction Memory;

The external view of the Control Unit is shown in figure 3.27.

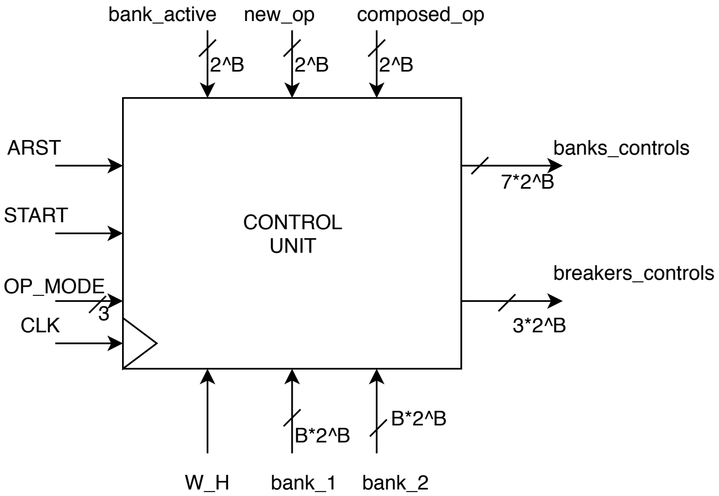


Figure 3.27: Top view of the Control Unit

In the following the inputs not yet mentioned will be explained:

- `ARST`: is the asynchronous reset. It resets the machine to an idle state;
- `START`: if is set to 0 the machine remains in the idle state, otherwise it starts working. The Control Unit is designed in such a way that if START remains set to 1 the machine continues working sampling another query, skipping the idle state;
- `bank_1,bank_2`: are bank address vectors coming from the datapath. They are ordered according to the operation sequence and are used to both manage the banks control signals and the breakers control signals;

The FSM chart of all operation modes are reported in the following figures.

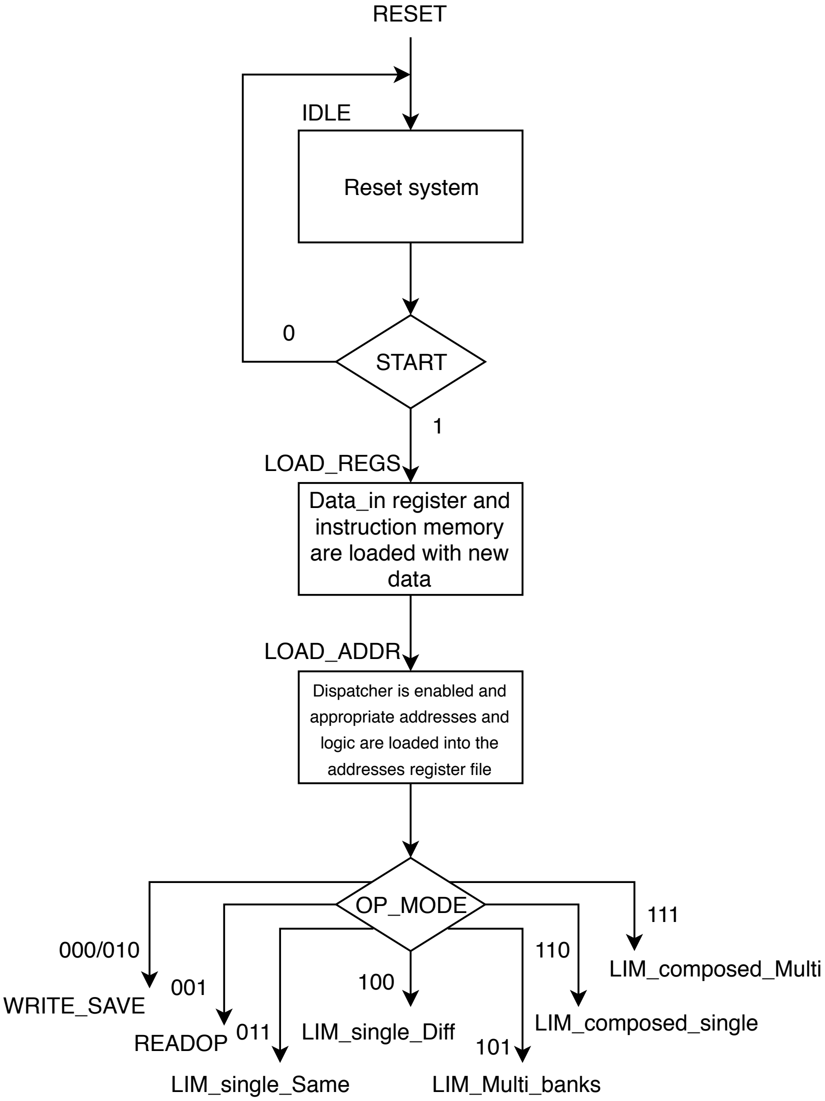


Figure 3.28: Preliminary stages of a CLIMA operation

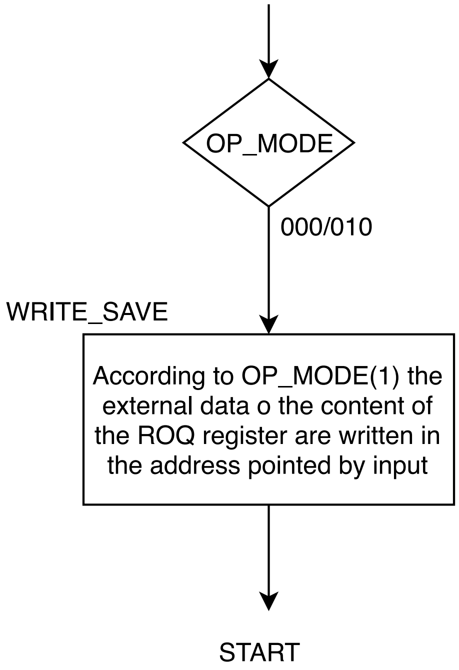

OP_MODE
000/010
WRITE_SAVE
According to OP_MODE(1) the external data o the content of the ROQ register are written in the address pointed by input
START

Figure 3.29: Flow of a WRITE/SAVE operation

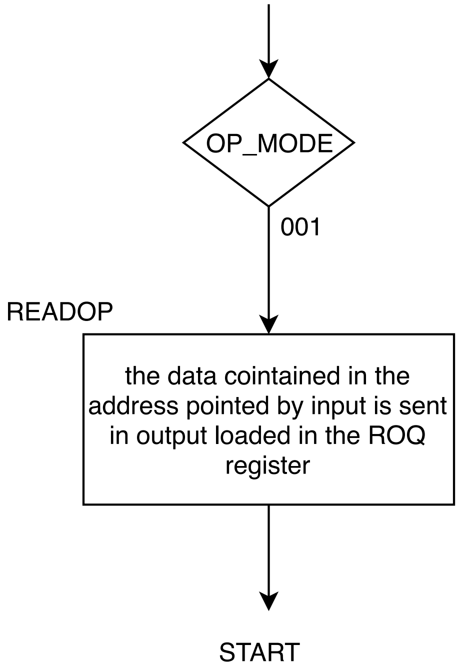

OP_MODE
001
READOP
the data cointained in the address pointed by input is sent in output loaded in the ROQ register
START

Figure 3.30: Flow of a READ operation

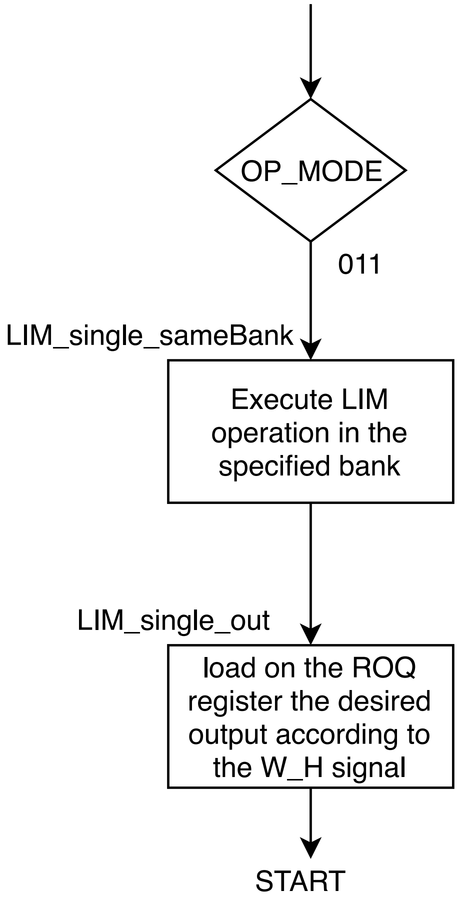

OP_MODE
011
LIM_single_sameBank
Execute LIM operation in the specified bank
LIM_single_out
load on the ROQ register the desired output according to the W_H signal
START

Figure 3.31: Flow of a single LIM operation inside a bank

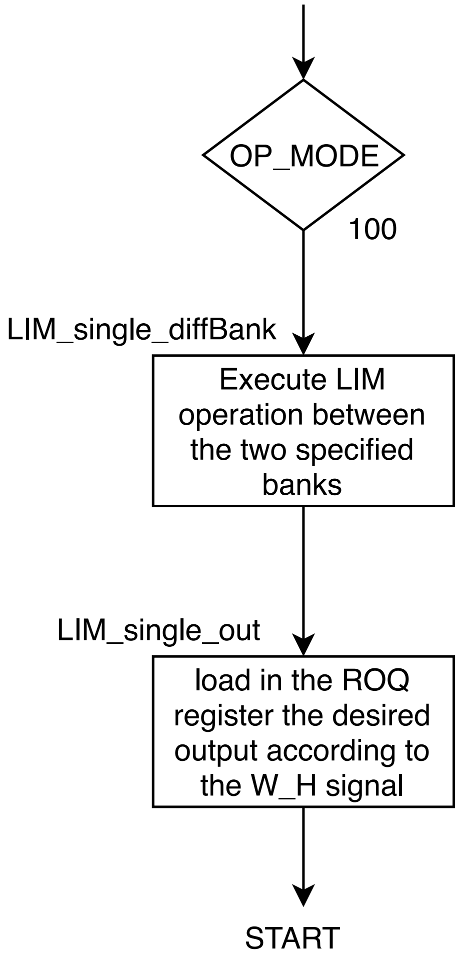

OP_MODE
100
LIM_single_diffBank
Execute LIM operation between the two specified banks
LIM_single_out
load in the ROQ register the desired output according to the W_H signal
START

Figure 3.32: Flow of a single LIM operation between two different banks

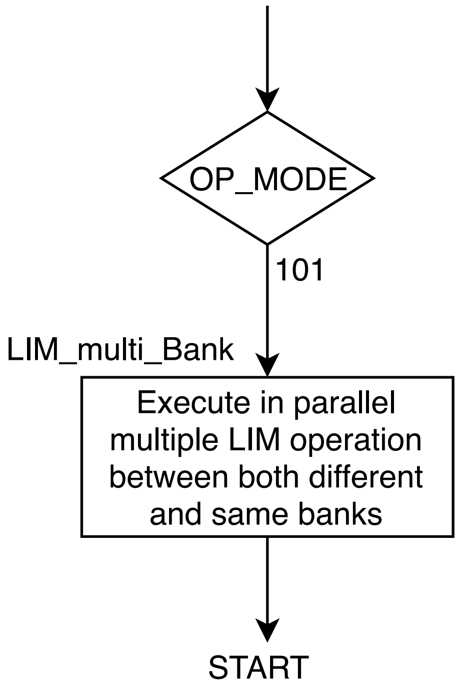

OP_MODE
101
LIM_multi_Bank
Execute in parallel multiple LIM operation between both different and same banks
START

Figure 3.33: Flow of a parallel LIM operation

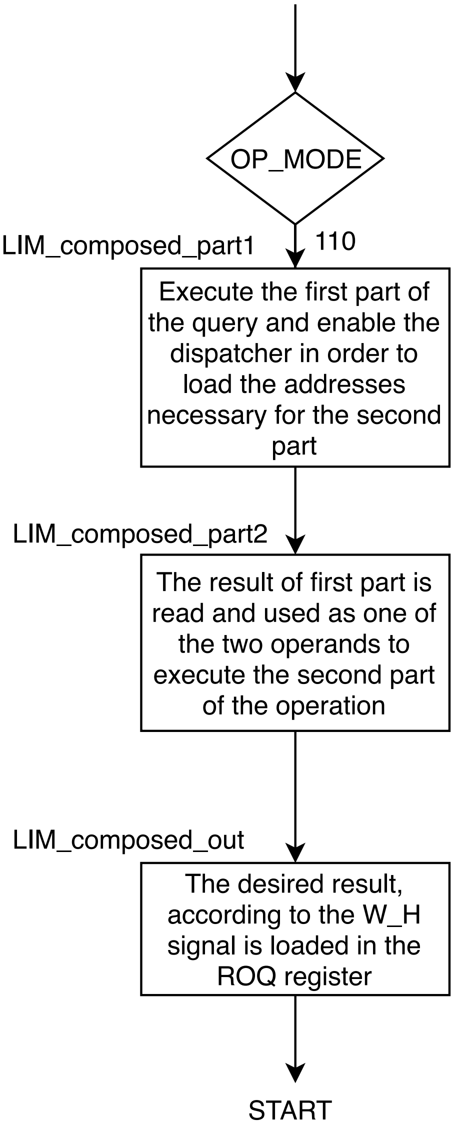

OP_MODE
LIM_composed_part1
110
Execute the first part of the query and enable the dispatcher in order to load the addresses necessary for the second part
LIM_composed_part2
The result of first part is read and used as one of the two operands to execute the second part of the operation
LIM_composed_out
The desired result, according to the W_H signal is loaded in the ROQ register
START

Figure 3.34: Flow of a single composed LIM operation

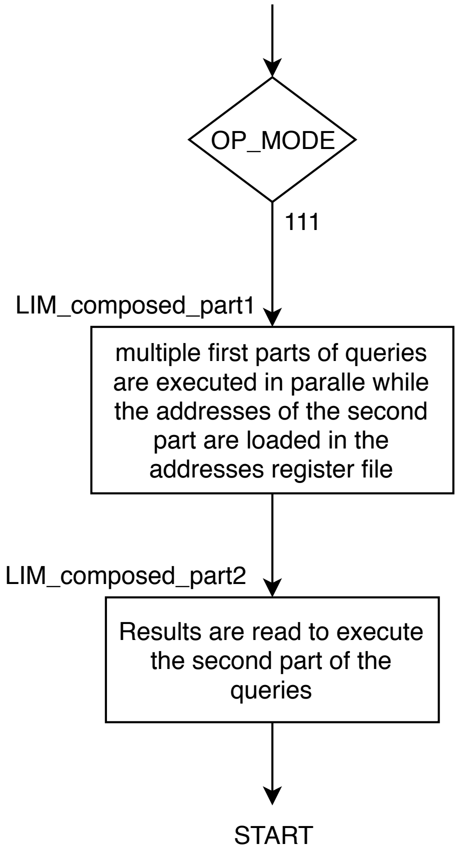

OP_MODE
111
LIM_composed_part1
multiple first parts of queries are executed in paralle while the addresses of the second part are loaded in the addresses register file
LIM_composed_part2
Results are read to execute the second part of the queries
START

Figure 3.35: Flow of multiple composed LIM operations

3.4 The complete CLIMA structure

The relantioship between Datapath and Control Unit is depicted in figure 3.36, while a top view of the structure as a whole is shown in figure 3.37.

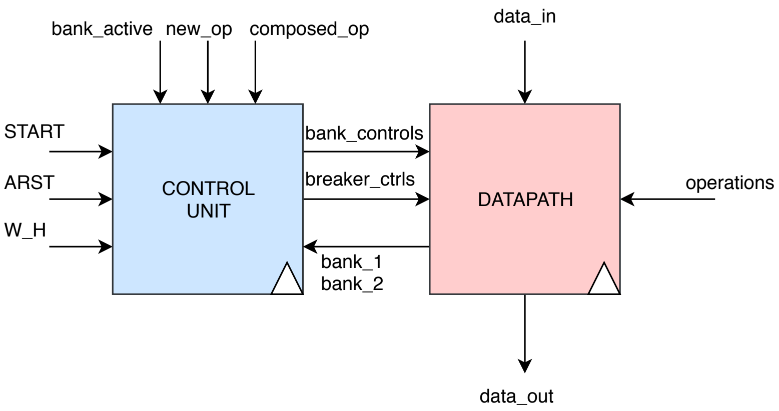


Figure 3.36: Internal view of CLIMA. Signal parallelism is omitted for clarity

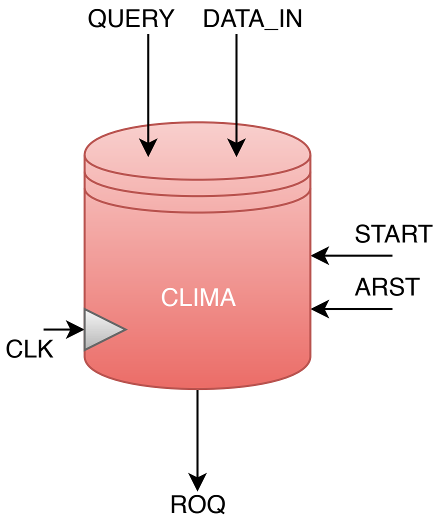


Figure 3.37: Top view of CLIMA

As it is possible to observe, CLIMA has as inputs external data, the desired query other than traditional control signals while the only output is Result Of Query.

Chapter 4

Test and Simulation

After implementation was completed, the natural following step was to ensure that CLIMA worked properly. All of CLIMA internal structures have been kept parametric to give the possibility to implement the architecture composed of how many banks, rows and words needed according to the target database.
For this study it was decided to implement a 256x256 bit architecture, divided into 16 banks, with 16 rows each, with 16 words each, for a total of 4096 words. Data size is set as 16 bit.
The flow implemented for this stage of the project is show in figure 4.1.

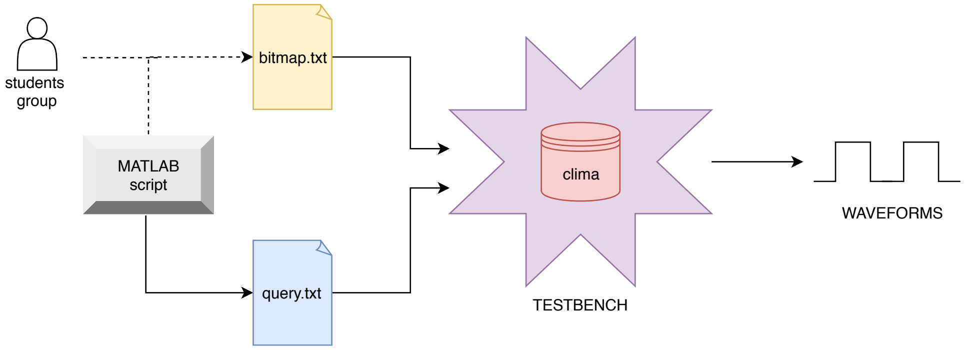


Figure 4.1: Flow of the Test phase

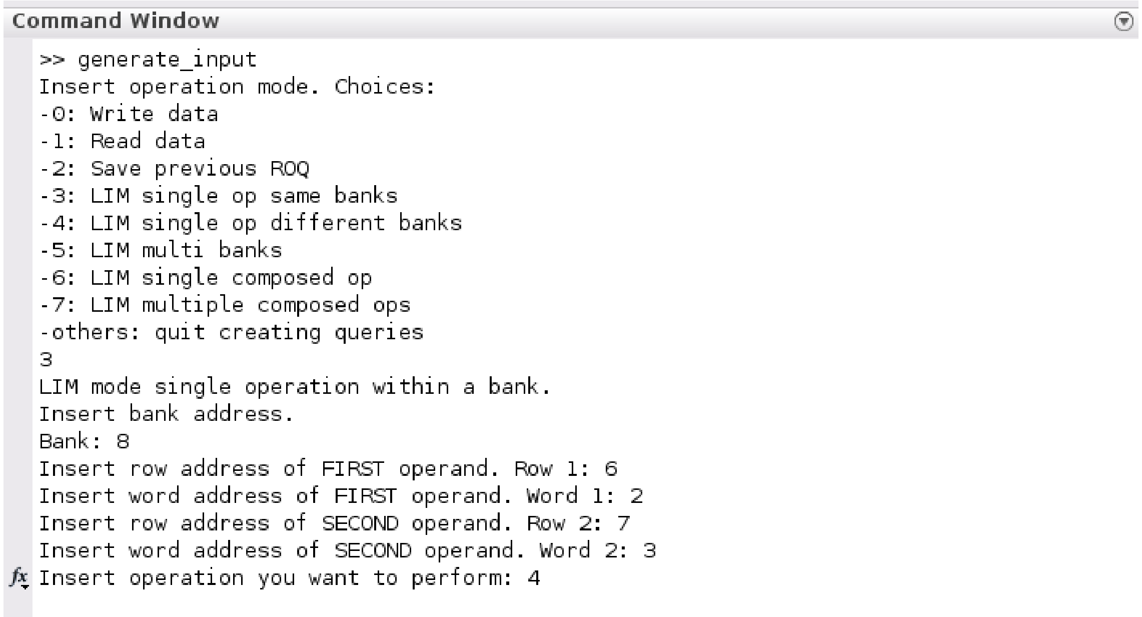
From a MATLAB script (or from an external source in the case of the bitmap) were extracted both the bitmap and the queries to execute. The files were then set as input for the VHDL Testbench and finally it was run a simulation. The VHDL Testbench is structured in two loops:

1. Fill the memory extracting data from the bitmap;

2. For each cycle of the loop, read one query and execute it. The loop stops at the end of the file

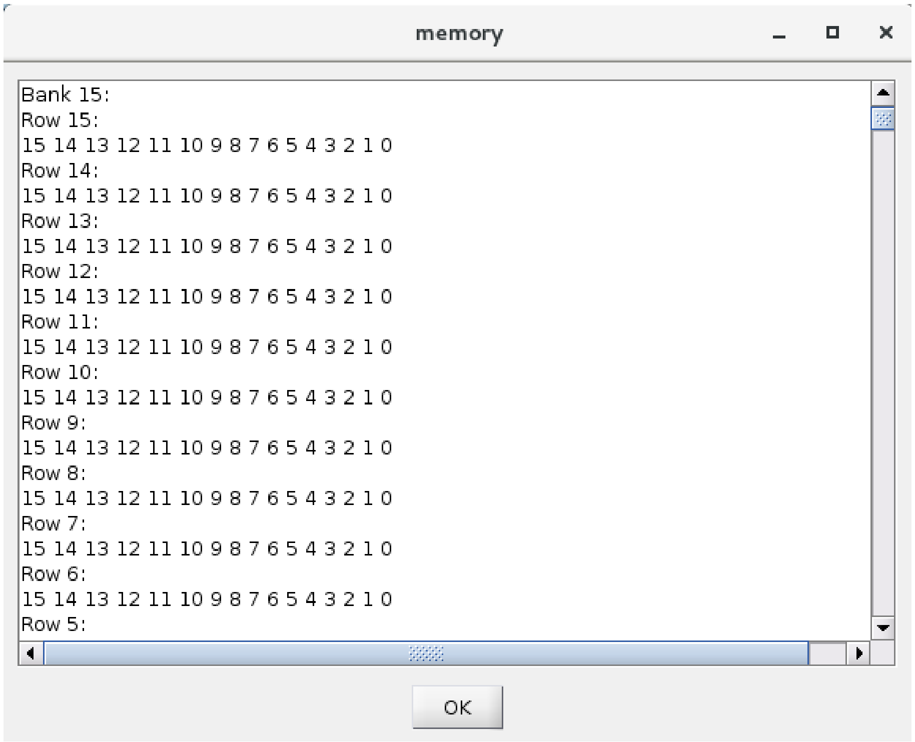
4.1 Generate Input MATLAB script

To make the generation of queries easier, a very simple MATLAB script was written and used. When started, the script enters a loop that terminates only when the user decides not to create any more queries and a file `query.txt` is generated as output. The completion of the query is assisted by two pop-up windows: one shows the internal composition of the memory (figure 4.3) and the other shows the available logic operations and their correspondent code(figure 4.4).
An example of the user interface is shown in figure 4.2.



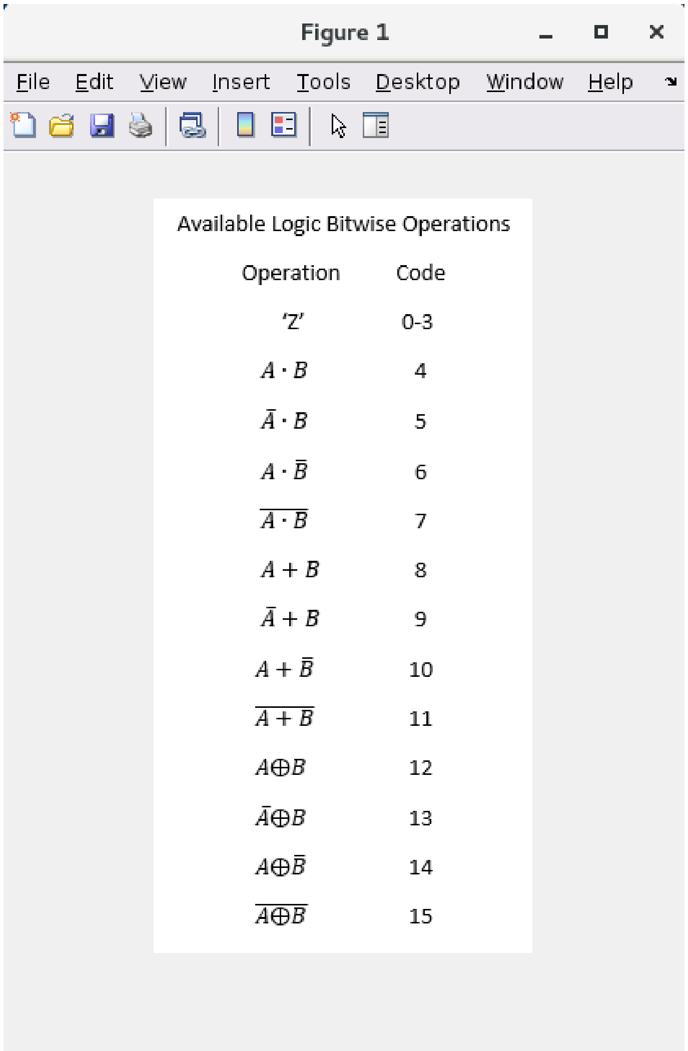
```
>> generate_input
Insert operation mode. Choices:
-0: Write data
-1: Read data
-2: Save previous ROQ
-3: LIM single op same banks
-4: LIM single op different banks
-5: LIM multi banks
-6: LIM single composed op
-7: LIM multiple composed ops
-others: quit creating queries
3
LIM mode single operation within a bank.
Insert bank address.
Bank: 8
Insert row address of FIRST operand. Row 1: 6
Insert word address of FIRST operand. Word 1: 2
Insert row address of SECOND operand. Row 2: 7
Insert word address of SECOND operand. Word 2: 3
fx Insert operation you want to perform: 4
```

Figure 4.2: Screenshot of the generate input Matlab script interface



```
Bank 15:
Row 15:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 14:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 13:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 12:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 11:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 10:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 9:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 8:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 7:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 6:
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
Row 5:
```

Figure 4.3: Description of the internal organization of the array



Available Logic Bitwise Operations

Operation	Code
'Z'	0-3
$A \cdot B$	4
$\bar{A} \cdot B$	5
$A \cdot \bar{B}$	6
$\overline{A \cdot B}$	7
$A + B$	8
$\bar{A} + B$	9
$A + \bar{B}$	10
$\overline{A + B}$	11
$A \oplus B$	12
$\bar{A} \oplus B$	13
$A \oplus \bar{B}$	14
$\overline{A \oplus B}$	15

Figure 4.4: Description of available logic operation

The script also generates a text file called `memory.txt` that can be used during testing. It can be used as filler of the array, but each words corresponds to the ID of the word in which it is saved (e.g 1,2,...4095,4096), so it is not a real bitmap, but it was still used because simple to obtain, in order to check if architecture worked as planned.

Later in the test phase a real bitmap was provided by a student group who implemented a Python compiler capable of manipulating an integer table to obtain the bitmap. Since both of the text files were used to fill the array, from now on it will be referred as `bitmap.txt`.

4.2 Operation Mode Testbench

In this section will be reported a simulation for each operation mode of CLIMA. For clarity, only the operations string of the query will be reported in the following format:

```
OP_MODE WHO/HOW_MANY BxRxWx ByRyWy LOGIC_OP
```

Where x,y corresponds to the addresses of the first and second operand, respectively.WHO/HOW_MANY and LOGIC_OP will be omitted for read and write cases. The radix of the data input and output is set as ”unsigned” to make the results easier to understand.

4.2.1 Write/Save

Query:

WRITE **B**14**R**7**W**1

Expected behaviour

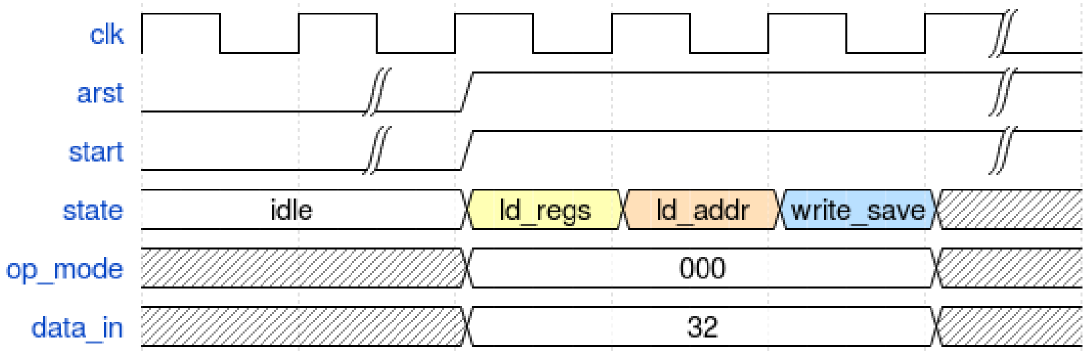


Figure 4.5: Expected waveform of a write operation

Actual behaviour

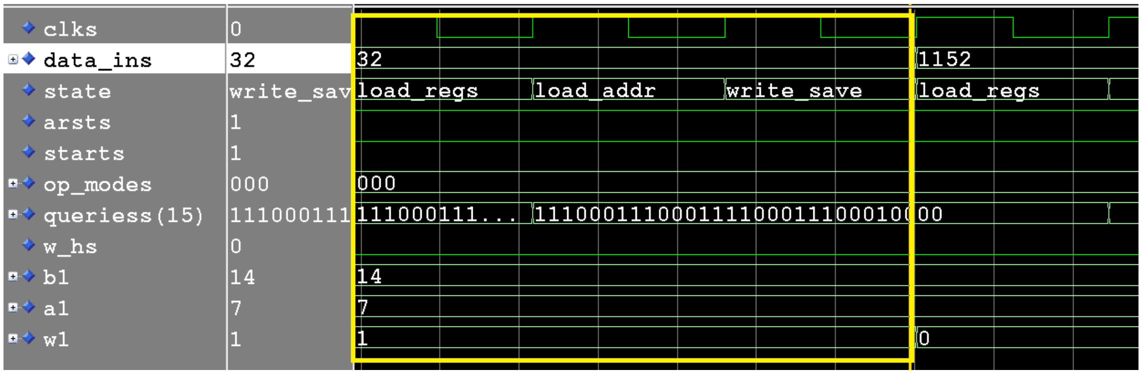


Figure 4.6: Waveform of a Write operation

4.2.2 Read

Query:

READ **B**14**R**7**W**1

Expected behaviour

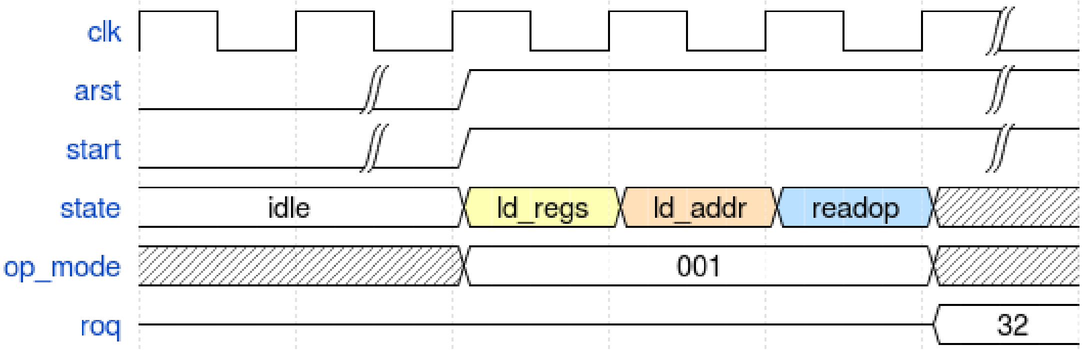


Figure 4.7: Expected waveform of a read operation

Actual behaviour

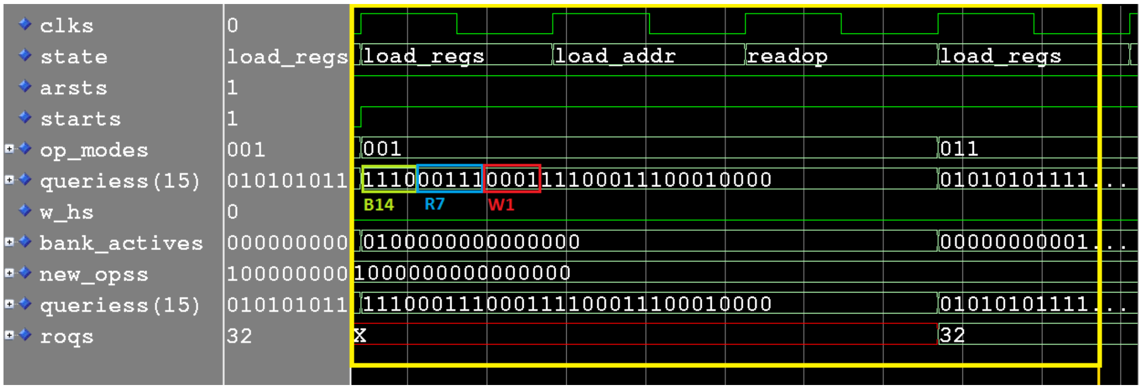


Figure 4.8: Waveform of a Read operation

4.2.3 LIM single same bank

Query:

LIMsingleSame WHO **B**5**R**11**W**13 **B**5**R**8**W**2 AND

Expected behaviour

Operation:

$$72 \text{ AND } 4 = 0$$

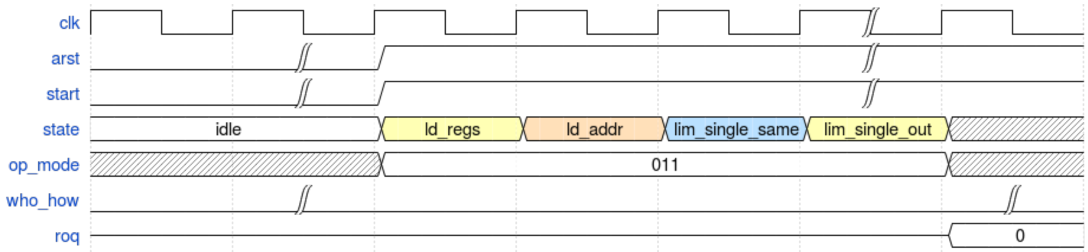


Figure 4.9: Expected waveform of a LIM single same bank operation

Actual behaviour

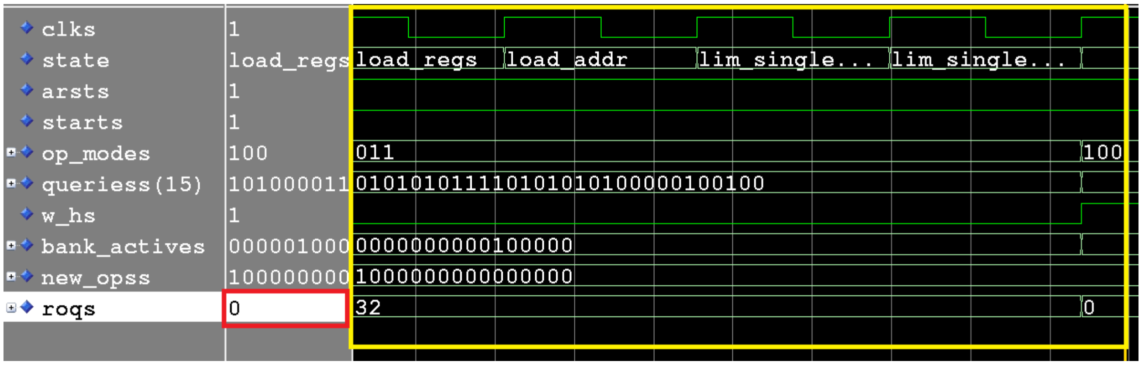


Figure 4.10: Waveform of a LIM single same bank operation

4.2.4 LIM single different banks

Query: LIMdiffSame HOW_MANY **B**10**R**3**W**3 **B**1**R**0**W**0 XOR

Expected behaviour

Operation:

$$0 \text{ XOR } 8 = 8$$
$$\text{number of ones} = 1$$

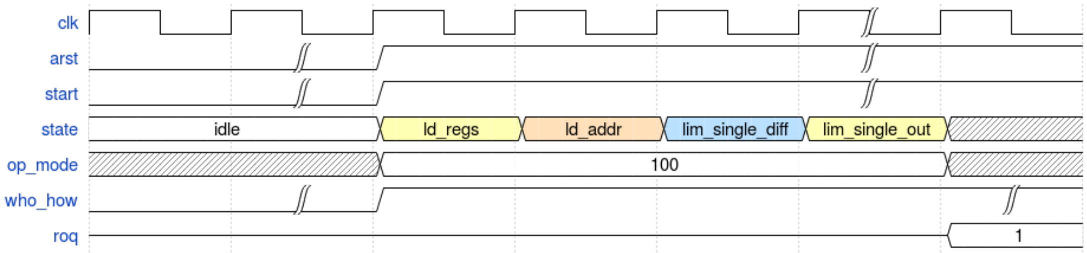


Figure 4.11: Expected waveform of a LIM single different banks operation

Actual behaviour

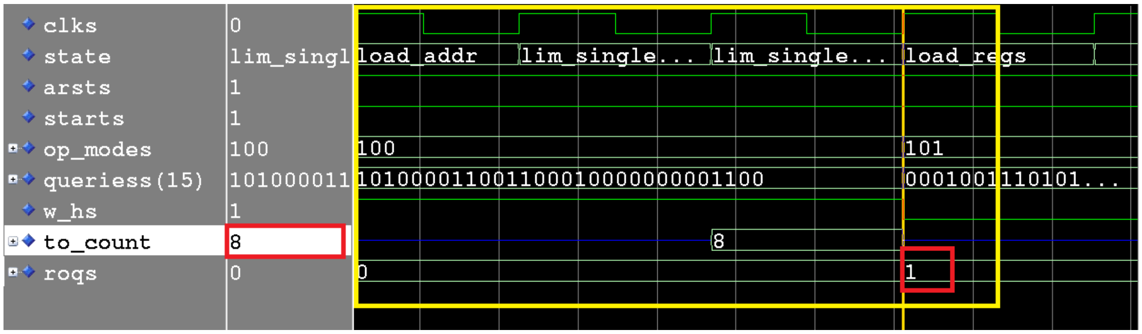


Figure 4.12: Waveform of a LIM single different banks operation

4.2.5 LIM multiple operations

Query:

$$\texttt{LIMmultiple WHO }\mathbf{B}1\mathbf{R}7\mathbf{W}5\ \mathbf{B}3\mathbf{R}0\mathbf{W}10\ \texttt{OR}$$
$$\mathbf{B}8\mathbf{R}8\mathbf{W}\ \mathbf{B}8\mathbf{R}1\mathbf{W}9\ \overline{AND}$$
$$\mathbf{B}7\mathbf{R}7\mathbf{W}7\ \mathbf{B}11\mathbf{R}11\mathbf{W}11\ \overline{OR}$$
$$\mathbf{B}14\mathbf{R}0\mathbf{W}0\ \mathbf{B}14\mathbf{R}10\mathbf{W}5\ \texttt{AND}$$
$$\mathbf{B}13\mathbf{R}2\mathbf{W}2\ \mathbf{B}15\mathbf{R}9\mathbf{W}6\ \texttt{OR}$$

Expected behaviour

Operations:

$$8192 \text{ OR } 2048 = 10240$$
$$\overline{5120} \text{ AND } \overline{0} = 60415$$
$$\overline{0} \text{ OR } \overline{0} = 65535$$
$$264 \text{ AND } 4224 = 0$$
$$0 \text{ OR } 256 = 256$$

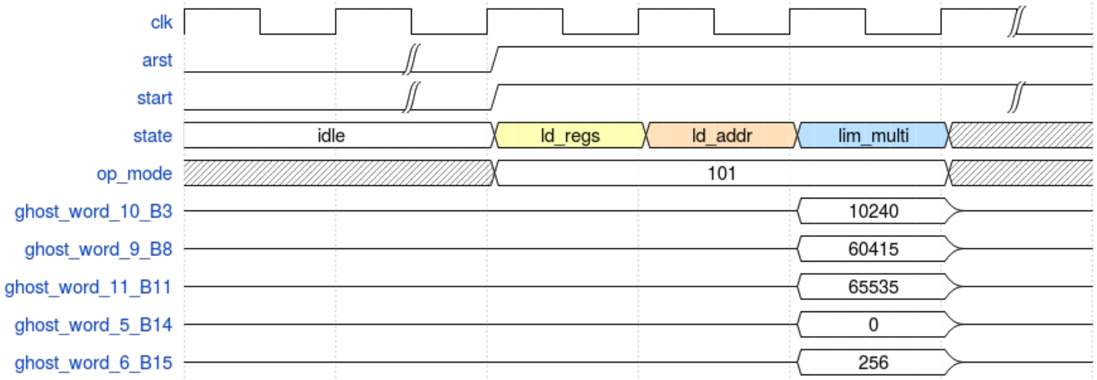


Figure 4.13: Expected waveform of a LIM multiple operations

Actual behaviour

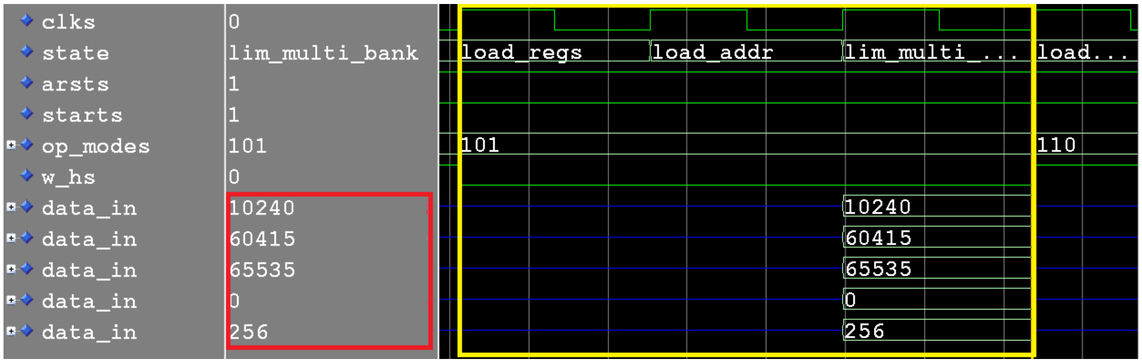


Figure 4.14: Waveform of a LIM multiple banks operation

4.2.6 LIM single composed

Query:

LIMsingleComposed HOW_MANY B2R2W2 B1R1W1 XOR
B1R16W1 B7R11W0 OR

Expected behaviour

Operations:

$$18432 \text{ XOR } 264 = 18696$$

$$18696 \text{ OR } 0 = 18696$$
$$\text{number of ones} = 4$$

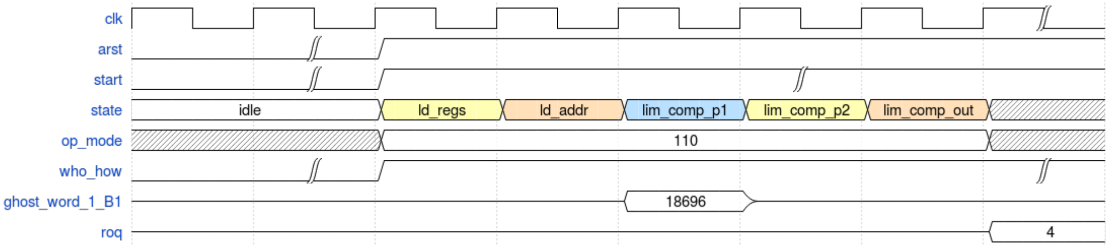


Figure 4.15: Expected waveform of a LIM single composed operation

Actual behaviour

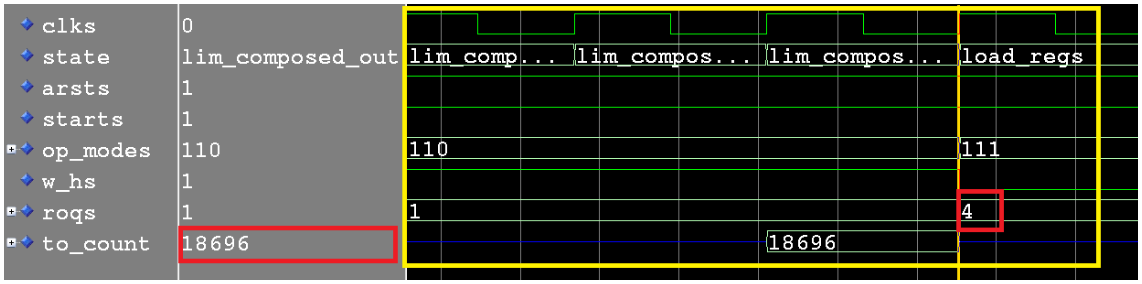


Figure 4.16: Waveform of a LIM single composed operation

4.2.7 LIM multiple composed

Query:

`LIMmultiComposed WHO` **B**15**R**15**W**15 **B**15**R**0**W**0 `AND`
B15**R**16**W**0 **B**10**R**2**W**2 `OR`
B4**R**4**W**4 **B**3**R**3**W**3 `XOR`
B3**R**16**W**3 **B**1**R**10**W**11 `OR`

Expected behaviour

Operations:

$$32768 \text{ AND } 16384 = 0$$
$$0 \text{ OR } 36865 = 36865$$
$$—$$
$$1280 \text{ XOR } 0 = 1280$$
$$1280 \text{ OR } 0 = 1280$$

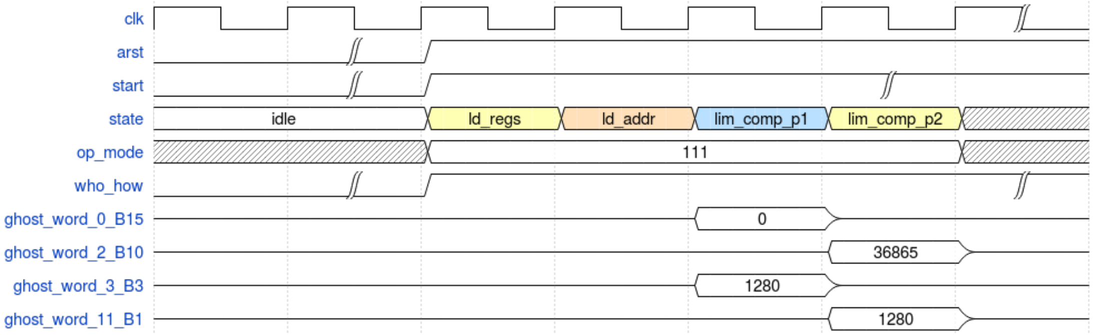


Figure 4.17: Expected waveform of a LIM multiple composed operation

Actual behaviour

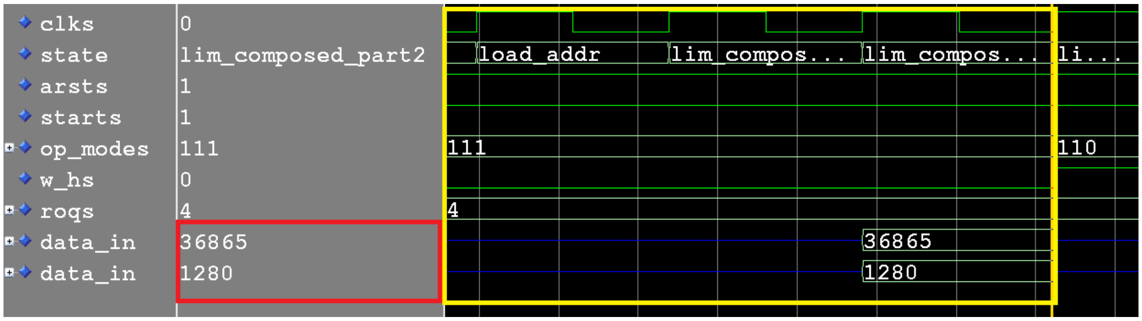


Figure 4.18: Waveform of a LIM multiple composed operation

Chapter 5

Synthesis Results

After ensuring that CLIMA worked properly, the process of synthesis begun. The flow of the synthesis phase is shown in figure 5.1.

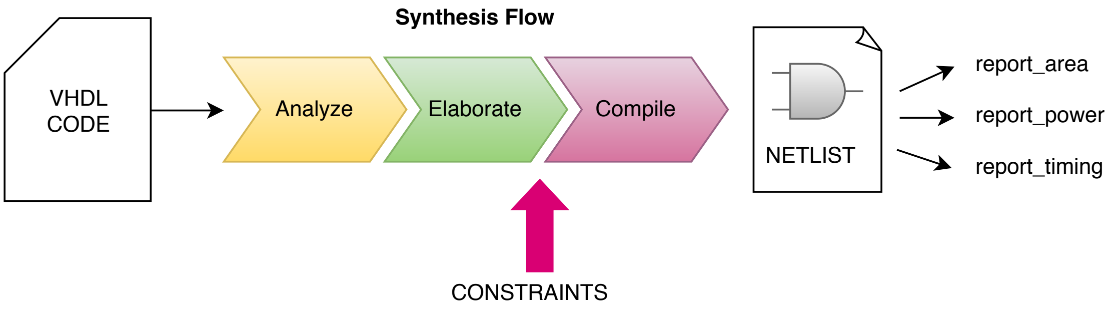


Figure 5.1: Flow of the synthesis process

Synthesis was performed using Synopsys Design Compiler with a 45 nm CMOS technology.

It was necessary to proceed by baby steps to avoid errors. For this reason, instead of synthesizing the whole structure in one step, as for the implementation phase a bottom-up approach was adopted.

At first, Cell and Ghost Cell were analyzed and compiled. When both of their best delay was found, it was decided to impose a `don't touch` constraint on them, to ensure that the fundamental element of the entire architecture would not undergo modification due to optimization during synthesis.

5.1 Cell

As the fundamental element of the whole structure, the Cell was the first to be analyzed and optimized.
The obtained results are reported in tables 5.1, 5.2.

	memory	logic	LiM Cell
Non-Combinational Area [μm^2]	9.31	2.12	11.43
Combinational Area [μm^2]	5.32	15.43	20.75
Total Area [μm^2]			32.18
Delay [ns]			0.45

Table 5.1: Area and Time Results of the LiM Cell

	Internal Power	Switching Power	Tot.Dynamic Power	Leakage Power
	[μW]	[μW]	[μW]	[nW]
LiM Cell	5.93	4.53	10.46	700.85

Table 5.2: Power consumption of the LiM Cell

The resulting schematics of both mem and logic components are shown in figures 5.2 and 5.3. As it is noticeable both logic and memory elements are coherent with the implementation.

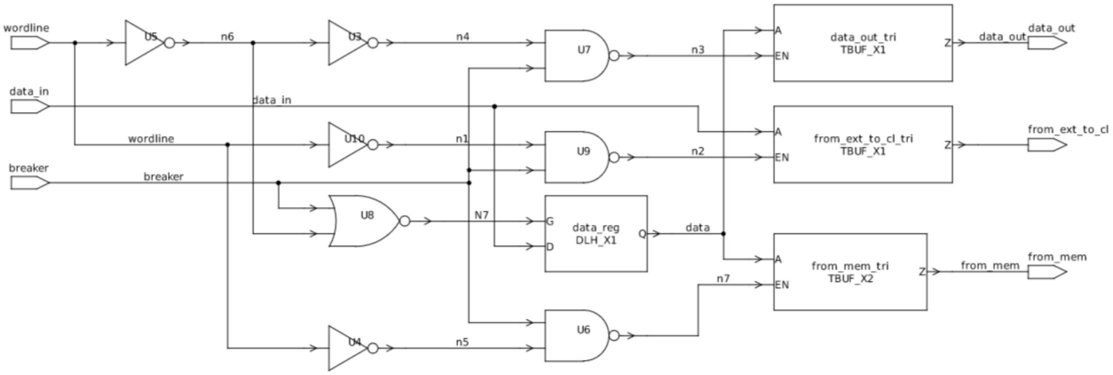


Figure 5.2: Schematic of the memory element of the LiM Cell

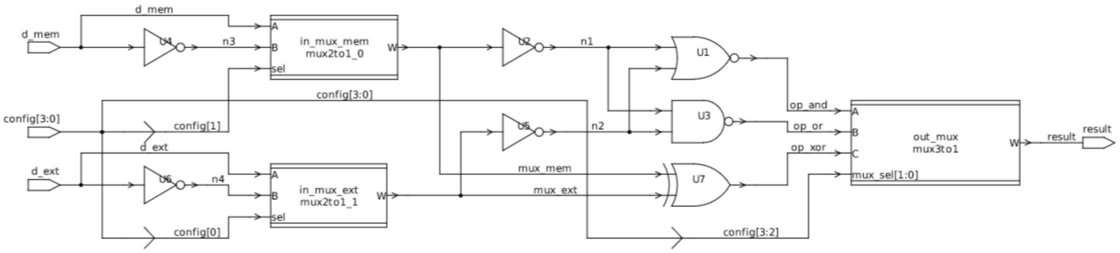


Figure 5.3: Schematic of the logic element of the LiM Cell

5.2 Ghost Cell

The ghost cell is the second basic component of CLIMA. The results of the synthesis are reported in tables 5.3 and 5.4.

	Ghost Cell
Non-Combinational Area [μm^2]	2.13
Combinational Area [μm^2]	5.05
Total Area [μm^2]	7.18
Delay [ns]	0.16

Table 5.3: Area and Time Results of the Ghost Cell

	Internal Power [μW]	Switching Power [μW]	Tot.Dynamic Power [μW]	Leakage Power [nW]
Ghost Cell	0.755	0.475	1.23	142.61

Table 5.4: Power consumption of the Ghost Cell

The schematic obtained with the synthesis is depicted in figure 5.4.

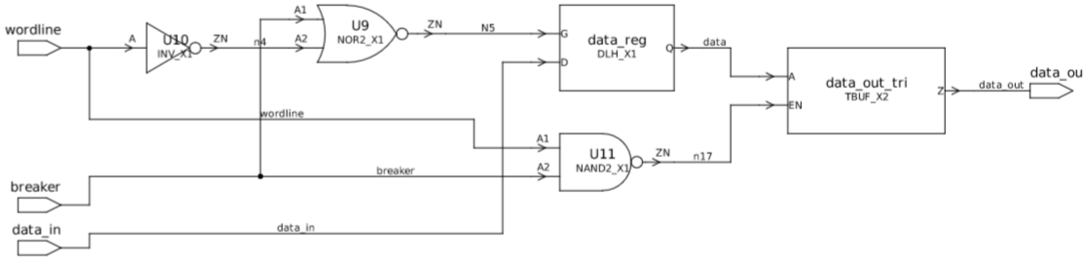


Figure 5.4: Schematic of the Ghost Cell

5.3 CLIMA

As next step, the entire structure was synthesized. As mentioned at the beginning of this section, a `don't touch` constraint was posed on Cell and Ghost Cell to avoid modification during synthesis.
CLIMA was synthesized of dimension 256x256 bit divided into 16 banks, with 16 rows each, with 16 words each, with a data size of 16 bit, the same as the simulation phase.

5.3.1 Parametric Ones Counter

In the first drafts of CLIMA, the Ones counter was implemented as a behavioural block with a loop that counted the ones present in the vector. However, such a counter had a delay of 3 ns. To speed it up, the counter was re-implemented as explained in section 3.2.8.
To make the structure as flexible as possible, the counter was implemented with the parametric value `D`, that indicates in how many segments the input vector is divided. Figure 5.5 show the relation between D and the total delay of the counter.

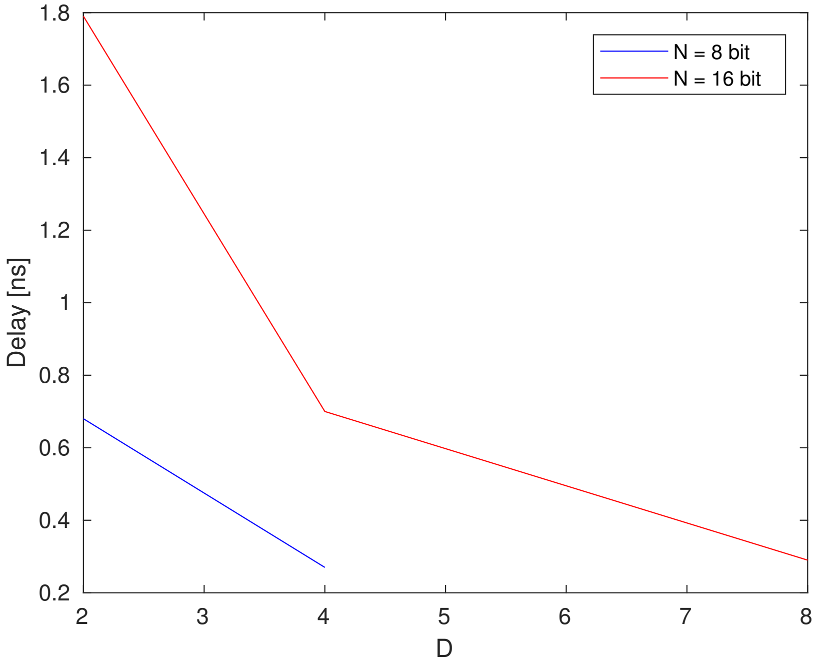


Figure 5.5: Relation between delay and number of segments in the counter

It is noticeable that for a data size of 16 bits, for $D = 8$ the counter shows the best delay. For this reason CLIMA was synthesized with a fixed value of $D = 8$.

5.3.2 CLIMA results

Synthesis results of the CLIMA architecture are reported in tables 5.5 and 5.6.

	CLIMA
Non-Combinational Area $[mm^2]$	1.554
Combinational Area $[mm^2]$	0.785
Total Area $[mm^2]$	2.33
Delay [ns]	12.14
f_{CLK} [MHz]	82.4

Table 5.5: Area and Time Results of Clima

	Internal Power $[\mu W]$	Switching Power $[\mu W]$	Leakage Power [nW]	Total Power $[\mu W]$
CLIMA	582.55	448.04	47.9e6	48.9e3

Table 5.6: Power consumption of CLIMA

For comparison purposes CLIMA was also synthesized with a 28nm CMOS library. In order to compare CLIMA with the non-LiM ASIC architecture specifically implemented for the Bitmap Index algorithm provided by the group of students mentioned previously, since their implementation can perform only AND, OR and NOT operations, the logic block of CLIMA was modified to comply with theirs and then synthesized with 45 nm. This version will be referred to as CLIMAmod. All the results are reported in tables 5.7 and 5.8.

	CLIMA 45nm	CLIMA 28nm	CLIMAmod 45nm
Non-Combinational Area $[mm^2]$	0.785	0.105	0.768
Combinational Area $[mm^2]$	1.554	0.947	0.835
Total Area $[mm^2]$	2.33	1.052	1.603
Delay [ns]	12.14	3.46	12.14
f_{CLK} [MHz]	82.4	289.02	82.4

Table 5.7: Area and Time Results of different versions of CLIMA

As far as throughput is concerned, when CLIMA is working in multiple parallel mode, one clock cycle is enough to execute operations. So, the resulting throughput for a multiple parallel mode simple operation is:

$$throughput_{simple} = f_{clk} \cdot N_{ops}$$

With N_{ops} corresponding to the number of operations performed in parallel.
As for the composed operations, since it takes two clock cycles to complete them, the resulting throughput is:

$$throughput_{composed} = \frac{f_{clk}}{2} \cdot N_{ops}$$

	Internal Power [μW]	Switching Power [μW]	Leakage Power [nW]	Total Power [μW]
CLIMA 45 nm	582.55	448.04	47.9e6	48.9e3
CLIMA 28 nm	1.23e3	361	10.86e6	12.45e3
CLIMAmod 45nm	578.4	446.1	28.4e6	29.4e3

Table 5.8: Power consumption of different versions of CLIMA

As expected CLIMA 28nm is the best out of the three versions, since it was synthesized with a newer technology.
Observing synthesis results, it is possible to notice that CLIMA and CLIMAmod share the delay, but as far as area is concerned , CLIMAmod is smaller. This is due to the fact that the logic block of the LiM cell has a different configuration and a smaller amount of internal block, since it perform a less wide spectrum of operations. This is also the reason why CLIMAmod consumes less power.

Array organization

Since all the sub-structures of CLIMA were implemented as parametric, it was interesting to analyze how synthesis results would change varying the distribution of the array. For this reason, another LiM array was synthesized, again with a dimension of 256x256 bit but distributed in 128 banks with 2 rows each, having 16 words each. Data size is kept unchanged. Results are reported in table 5.9.

	Array B16R16W16	Array B128R2W16
Total Area [mm^2]	2.292	2.575
Delay [ns]	3.16	16.48

Table 5.9: Area and delay results of the two instances of LiM array

Comparing the two instances, it is possible to say that the configuration with 128 banks is bigger and way slower. The overhead in area is due to the fact that, even if the overall number of rows is the same, more banks imply more ghost rows and more breakers. The bigger amount of breakers is also the cause of the longer delay. However, it must be said that such a distribution of the array could perform 128 parallel operations instead of 16, resulting in a much higher throughput. So, it is possible to implement any configuration of CLIMA according to the desired performances and area-delay trade-off.

5.3.3 Optimization

Analyzing synthesis results of CLIMA it was noted that an optimization could be carried out.
The delay of CLIMA is dependent on the fact that the ones counter is sensitive to the negative edge of the clock. This imply that the critical path results in half of the overall clock. This choice was made since if the counter was sensitive to the positive edge, it did not have time to sample the correct data and compute result. In order to try and optimize the clock, a register with the load always set to 1 was inserted right after the output of the array, then the counter was modified to be sensitive to the positive edge of the clock. Adding a delay represented by the new register, now the counter would have the time to count the correct input. However, it was also necessary to modify lightly the control unit, in particular adding more states to obtain the output. The modified structure was then again synthesized in all the previous versions. Results are reported in tables 5.10 and 5.11.

	CLIMA 45nm	CLIMA 28nm	CLIMAmod 45nm
Non-Combinational Area [mm^2]	0.785	0.105	0.768
Combinational Area [mm^2]	1.542	0.953	0.837
Total Area [mm^2]	2.33	1.058	1.606
Delay [ns]	6.52	1.74	6.52
f_{CLK} [MHz]	153.4	574.7	153.4

Table 5.10: Area and Time Results of different versions of CLIMA after optimization

	Internal Power [μW]	Switching Power [μW]	Leakage Power [nW]	Total Power [μW]
CLIMA 45 nm	1.09e3	797.1	47.8e6	49.7e3
CLIMA 28 nm	2.43e3	0.689	10.954e6	14.07e3
CLIMAmod 45nm	1.07e3	781.5	28.4e6	30.3e3

Table 5.11: Power consumption of different versions of CLIMA after optimization

It is possible to notice that the one value that changed considerably after optimization was the delay. Now the clock frequency of CLIMA is almost twice the previous one and the same could be said for the resulting throughput.

5.3.4 Comparison

CLIMAmod was implemented to be compared with the architecture provided by the student group. They implemented a non-LiM ASIC architecture composed of traditional memory, an ALU (which could perform only AND, OR and NOT operations, as explained previously) and the Control Unit, aimed to the Bitmap index algorithm. They synthesized it and provided the results. To make the comparison coherent, CLIMAmod was necessary. Results of both architectures are reported in table 5.12.

	CLIMAmod 45nm	non-LiM
Total Area [mm^2]	1.606	0.226
Delay [ns]	6.52	2.2
f_{CLK} [MHz]	153.4	454.5
Total Power [mW]	30.3	15.2

Table 5.12: Synthesis results of CLIMAmod 45nm and the standard architecture provided by the student group

To compare the two structures the number of accesses to memory would probably be worth mentioning. To perform an operation between two operands in a standard architecture three accesses would be needed: two to read the operands and one to write the result back in the array. As for the LiM architecture, to execute an operation, two accesses are enough: one to transport one operand to the other and one to save the result in the correspondent ghost word. Moreover, the fact that these data movements are performed inside the memory array, instead that going back and forth from the memory, must be taken into account. Also, CLIMA is capable of performing multiple operations in parallel, while a standard structure is able to execute one at a time. To implement parallel operation it would be necessary to instance as many units as needed, resulting in a multiplied architecture.

Furthermore, it should be noted that the values of the memory of the non-LiM architecture were collected from the model of a real memory, while CLIMA could not be synthesized as a real one. This resulted in the memory elements being synthesized as latches, which are far more expensive than a traditional memory cell. As far as power consumption of CLIMA is concerned it has to be considered that data movement inside a Bank would probably be less expensive than in the case of operations between banks, especially if distant ones. This shows that with CLIMA there is a lot of room for study, improvements and optimization.

Chapter 6

Conclusion and future work

The project carried out with this thesis aimed to explore the *Logic-in-Memory* paradigm, starting from a target application and then implementing a parallel architecture.

CLIMA is a configurable parallel architecture which bases itself on the application of *Bitmap indexing*, an algorithm used in database systems.

CLIMA is implemented as a modular and parametric structure, to be as versatile as possible. It is capable of performing as many parallel operations as the number of its banks. It is also pretty easy to modify if some improvements or reconfiguration are needed. In fact, even if it started from *bitmap indexing*, CLIMA is suitable for any kind of bitwise applications.

The architecture was tested and synthesized and it showed promising results compared with a standard architecture, taking into account its high degree of parallelism and the fact that CLIMA could not be synthesized to be modeled as a real memory.

6.1 Future Work

CLIMA is still far from being a perfect structure and there is a lot of room for improvement. With more study and analysis it could be possible to further optimize CLIMA.

Few examples will be reported:

- **3D structure**: multiple layers of LiM array could be put together to obtain a more efficient architecture;

- **Flushing outputs**: aimed to improve parallelism, while multiple banks are busy working independently, the ones that completed their job send the results in output one by one and resume work when free.

- **Avoid ghost overwriting**: one of the bad sides of multiple parallel operations is that the result cannot be sent right in output. This is why it is saved in the ghost word. However, if the result is not read soon after, there is a possibility that it will be overwritten by a subsequent operation. A possible improvement could be finding a solution to avoid this eventuality.
- **General purpose**: CLIMA was implemented with the idea of bitwise operations in mind. But it should not be too difficult to further modify the structure to make CLIMA suitable for all kinds of applications.

To conclude, this work comes to an end with the hope that CLIMA could one day be a starting point for new ideas that will bring the concept of Logic-in-Memory even further.

Bibliography

[1] S. Ikeda, J. Hayakawa, Y. M. Lee, F. Matsukura, Y. Ohno, T. Hanyu, and H. Ohno. Magnetic tunnel junctions for spintronic memories and beyond. *IEEE Transactions on Electron Devices*, 54(5):991–1002, May 2007.

[2] G. Prenat, M. El Baraji, Wei Guo, R. Sousa, L. Buda-Prejbeanu, B. Dieny, V. Javerliac, J. P. NoziERES, Weisheng Zhao, and E. Belhaire. Cmos/magnetic hybrid architectures. In *2007 14th IEEE International Conference on Electronics, Circuits and Systems*, pages 190–193, Dec 2007.

[3] W. Zhao, E. Belhaire, B. Dieny, G. Prenat, and C. Chappert. Tas-mram based non-volatile fpga logic circuit. In *2007 International Conference on Field-Programmable Technology*, pages 153–160, Dec 2007.

[4] S. Matsunaga, J. Hayakawa, S. Ikeda, K. Miura, T. Endoh, H. Ohno, and T. Hanyu. Mtj-based nonvolatile logic-in-memory circuit, future prospects and issues. In *2009 Design, Automation Test in Europe Conference Exhibition*, pages 433–435, April 2009.

[5] TAKAHIRO HANYU. Challenge of mtj-based nonvolatile logic-in-memory architecture for dark-silicon logic lsi. *SPIN*, 03(04):1340014, 2013.

[6] G. Turvani, M. Bollo, M. Vacca, F. Cairo, M. Zamboni, and M. Graziano. Design of mram-based magnetic logic circuits. *IEEE Transactions on Nanotechnology*, 16(5):851–859, Sept 2017.

[7] L. Chang, Z. Wang, Y. Zhang, and W. Zhao. Reconfigurable processing in memory architecture based on spin orbit torque. In *2017 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, pages 95–96, July 2017.

[8] Q. Zhu, B. Akin, H. E. Sumbul, F. Sadi, J. C. Hoe, L. Pileggi, and F. Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive

computing. In *2013 IEEE International 3D Systems Integration Conference (3DIC)*, pages 1–7, Oct 2013.

[9] D. H. Kim, K. Athikulwongse, M. B. Healy, M. M. Hossain, M. Jung, I. Khorosh, G. Kumar, Y. J. Lee, D. L. Lewis, T. W. Lin, C. Liu, S. Panth, M. Pathak, M. Ren, G. Shen, T. Song, D. H. Woo, X. Zhao, J. Kim, H. Choi, G. H. Loh, H. H. S. Lee, and S. K. Lim. Design and analysis of 3d-maps (3d massively parallel processor with stacked memory). *IEEE Transactions on Computers*, 64(1):112–125, Jan 2015.

[10] L. Jiang, M. Kim, W. Wen, and D. Wang. Xnor-pop: A processing-in-memory architecture for binary convolutional neural networks in wide-io2 drams. In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 1–6, July 2017.

[11] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie. Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 27–39, June 2016.

[12] F. Su, W. H. Chen, L. Xia, C. P. Lo, T. Tang, Z. Wang, K. H. Hsu, M. Cheng, J. Y. Li, Y. Xie, Y. Wang, M. F. Chang, H. Yang, and Y. Liu. A 462gops/j rram-based nonvolatile intelligent processor for energy harvesting ioe system featuring nonvolatile logics and processing-in-memory. In *2017 Symposium on VLSI Technology*, pages T260–T261, June 2017.

[13] L. Han, Z. Shen, Z. Shao, H. H. Huang, and T. Li. A novel reram-based processing-in-memory architecture for graph computing. In *2017 IEEE 6th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, Aug 2017.

[14] P. E. Gaillardon, L. AmarÃº, A. Siemon, E. Linn, R. Waser, A. Chattopadhyay, and G. De Micheli. The programmable logic-in-memory (plim) computer. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 427–432, March 2016.

[15] D. Bhattacharjee, R. Devadoss, and A. Chattopadhyay. Revamp: Reram based vliw architecture for in-memory computing. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pages 782–787, March 2017.

[16] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L.

Greathouse, Lifan Xu, and Michael Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 85–98, New York, NY, USA, 2014. ACM.

[17] Jeff Draper, Jacqueline Chame, Mary Hall, Craig Steele, Tim Barrett, Jeff LaCoss, John Granacki, Jaewook Shin, Chun Chen, Chang Woo Kang, Ihn Kim, and Gokhan Daglikoca. The architecture of the diva processing-in-memory chip. In *Proceedings of the 16th International Conference on Supercomputing*, ICS '02, pages 14–25, New York, NY, USA, 2002. ACM.

[18] M. Gokhale, B. Holmes, and K. Iobst. Processing in memory: the terasys massively parallel pim array. *Computer*, 28(4):23–31, Apr 1995.

[19] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, June 2015.

[20] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie. Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.

[21] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. Ambit: In-memory accelerator for bulk bitwise operations using commodity dram technology. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-50 '17, pages 273–287, New York, NY, USA, 2017. ACM.

[22] Mohsen Imani, Saransh Gupta, and Tajana Rosing. Ultra-efficient processing in-memory for data intensive applications. In *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, pages 6:1–6:6, New York, NY, USA, 2017. ACM.

[23] Y. Tang, Y. Wang, H. Li, and X. Li. Approxpim: Exploiting realistic 3d-stacked dram for energy-efficient processing in-memory. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 396–401, Jan 2017.

[24] K. Yang, R. Karam, and S. 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)*, pages 409–412, Aug 2017.

[25] T. L. Sterling and H. P. Zima. Gilgamesh: A multithreaded processor-in-memory architecture for petaflops computing. In *Supercomputing, ACM/IEEE 2002 Conference*, pages 48–48, Nov 2002.

[26] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. Design and evaluation of a processing-in-memory architecture for the smart memory cube. In *Proceedings of the 29th International Conference on Architecture of Computing Systems – ARCS 2016 - Volume 9637*, pages 19–31, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[27] https://www.extremetech.com/computing/261792-what-is-speculative-execution.

[28] https://docs.oracle.com/cd/A84870_01/doc/server.816/a76994/indexes.htm.

[29] https://en.wikipedia.org/wiki/Column-oriented_DBMS.

[30] J. Jeddeloh and B. Keeth. Hybrid memory cube new dram architecture increases density and performance. In *2012 Symposium on VLSI Technology (VLSIT)*, pages 87–88, June 2012.

[31] Juri Schmidt, Holger Fröning, and Ulrich Brüning. Exploring time and energy for complex accesses to a hybrid memory cube. In *Proceedings of the Second International Symposium on Memory Systems*, MEMSYS '16, pages 142–150, New York, NY, USA, 2016. ACM.

[32] Y. Yu and N. K. Jha. Energy-efficient monolithic 3d on-chip memory architectures. *IEEE Transactions on Nanotechnology*, PP(99):1–1, 2017.

[33] Elizabeth O'Neil, Patrick O'Neil, and Kesheng Wu. Bitmap index design choices and their performance implications. In *Proceedings of the 11th International Database Engineering and Applications Symposium*, IDEAS '07, pages 72–84, Washington, DC, USA, 2007. IEEE Computer Society.