POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

# Exploration and Modeling of Logic in Memory Architectures


**Supervisors**
Prof.ssa Mariagrazia Graziano
Prof. Marco Vacca
Dott. Ing. Alessio Naclerio

**Candidate**
Francesco Marino
ID: 296840

Academic year 2023 – 2024

# Summary

All modern CPU architectures are based on the **Von Neumann model**, which consists of two entities connected by a bus: the **processing unit** and the **memory unit**. Although processing units have improved their performance over the years, memory capacity has not. This has led to a significant performance gap between the two, known as the **Memory Wall**. Memories are very slow to access in comparison with the processor, which leads to bottlenecks. As a result, the CPU must reduce its performance to mitigate these bottlenecks.

The Memory Wall has been discussed for several years, and many solutions have been proposed. The **Logic in Memory (LiM)** paradigm is considered a viable solution. There are several types of Logic in Memory, based on different approaches to move the computation inside (or near) the memory itself, such as Computation-near-Memory (CnM), Computation-with-Memory (CwM), Computation-in-Memory (CiM), and the "pure" Logic in Memory (LiM).

The **LiM paradigm** aims to reduce the number of load and store operations by performing part of the computation (mainly simple calculations) directly inside the memory. This leads to a reduction in power consumption and data-fetching latency.

Moreover, LiM-based architectures can perform **parallel computation** due to the presence of **multiple memory banks**, each with its own computational capabilities, independent from the others. This results in even better performance.

Logic in Memory architectures can be implemented using standard technologies, such as SRAM or DRAM. However, the best performance is achieved by utilizing emerging technologies, such as resistive random-access memories (RRAMs), phase change memories (PCMs), ferroelectric field-effect transistors (FeFETs), hybrid CMOS, and Spintronic Devices based on the Magnetic Tunnel Junction (MTJ), such as Spin-Orbit Torque (SOT) Magnetic RAMs

(MRAMs) and Spin-transfer Torque RAMs (STT-RAMs). These technologies offer advantages in terms of **power consumption**, **area**, and **performance**.

The literature presents multiple LiM implementations across different levels of abstraction, from LiM Cell characterization to LiM Array Design. The approaches found in the literature vary in certain aspects, such as supported algorithms, memory cell characterization, operating modes, and data mapping. However, many commonalities exist across these methods. What is truly lacking is a framework that consolidates these shared aspects while also enabling the application of algorithms to assess the benefits of using LiM banks. However, there is no **high-level description** for these designs.

The goal of this thesis is to develop a **high-level architectural model** for LiM Banks that can establish a standard and be compatible with common interfaces, such as the Open Bus Interface (OBI). The model's key feature is its versatility, as it can be tailored to emulate more specific LiM Bank architectures.

After formulating the key concept, the scientific literature was consulted to gather information on technologies to employ, mathematical operations to implement, overall architecture structure, and supported algorithms.

Before designing the architecture, it was essential to choose the supported **mathematical operations** and determine how to **translate** them into instructions, taking into account constraints like the number of **operands** and the **bit-width** for processing. This selection was made with the goal of aligning with existing literature, ensuring that the developed model encompasses and supports all the key features of the analyzed implementations. This led to the development of a custom **Instruction Set** with a layout similar to the **RISC-V ISA**, which has been used as a standard in previous works and is both simple and effective. The Resulting Instruction Word can be represented as:

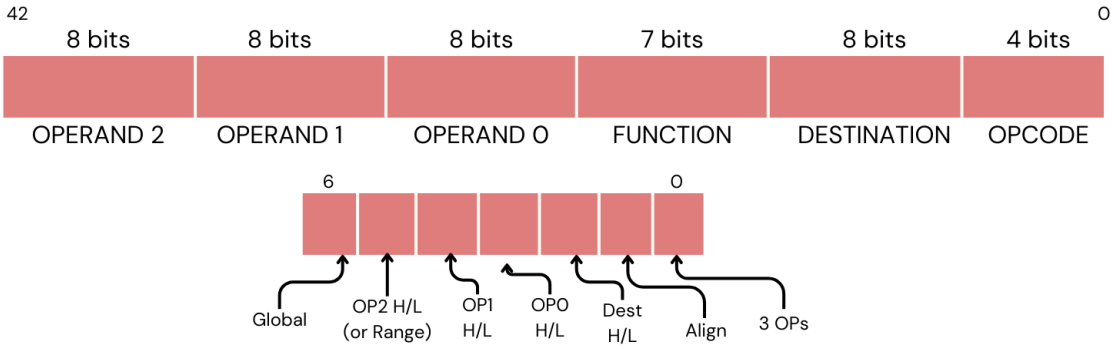


Figure 1. LiM Instruction Word

The model operates in two main modes: **Memory** (for LOAD and STORE operations) and **LiM** (for bitwise and mathematical operations). The Model is divided in three main levels:

1. Level 1 $\longrightarrow$ **Top** Level:

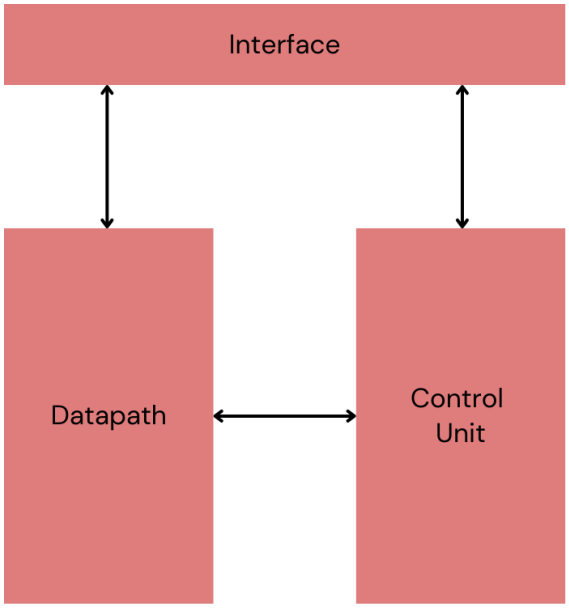


Figure 2. LiM Level 1 Schematic

2. Level 2 $\longrightarrow$ **Datapath and Control Unit** Level:

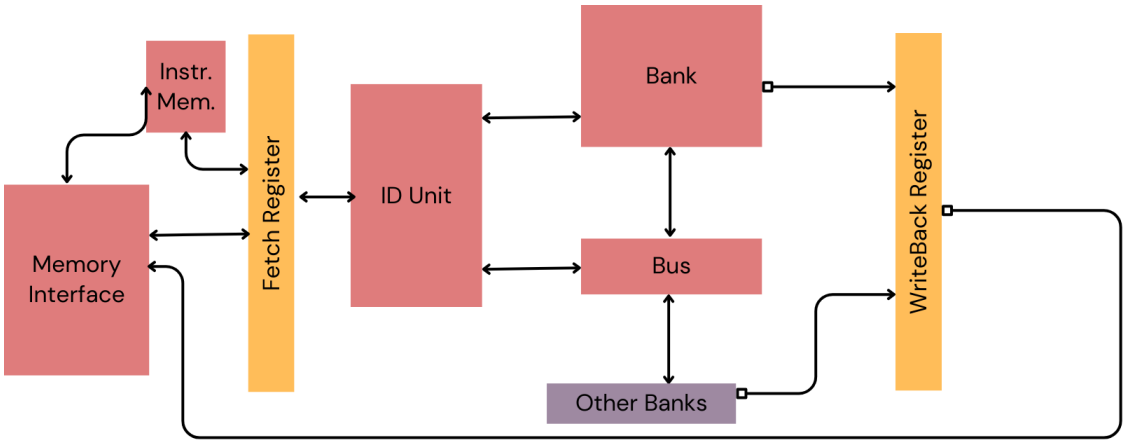


Figure 3. LiM Level 2 Schematic

3. Level 3 $\longrightarrow$ **Bank** Level.

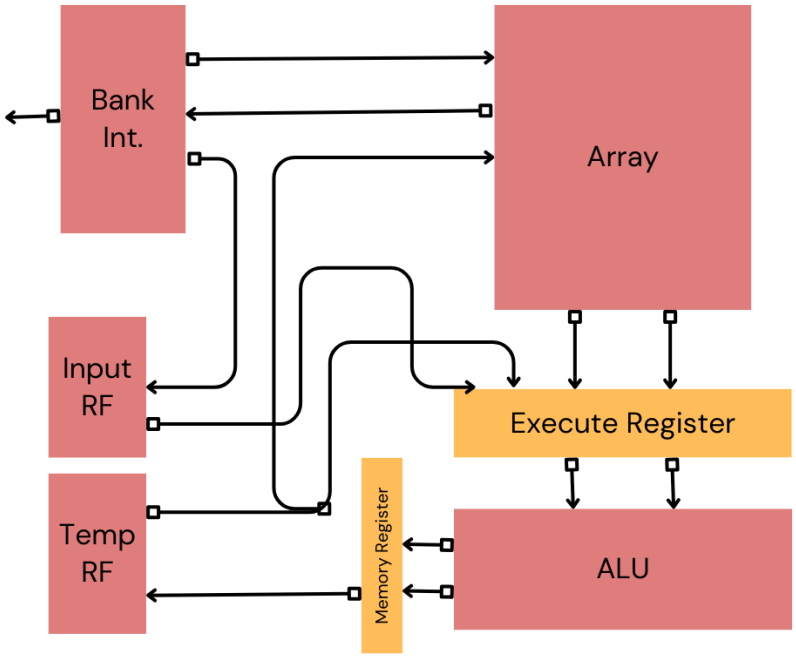


Figure 4. LiM Level 3 Schematic

At start-up, all instructions are stored in an **Instruction Memory**. Similar to a standard CPU architecture, in LiM mode, the current instruction address is stored in the Program Counter (PC) and is then forwarded to the **Instruction Decoding Unit** (ID Unit).

The ID Unit decodes the instruction and activates the Memory Banks to perform LiM and memory operations. Each bank features its own **Arithmetic and Logic Unit** (ALU), **Register Files** (RFs), and **Memory Array**, allowing them to operate in parallel, thus improving performance. All banks are connected via a common **bus**.

The model was tested using a traditional **testbench** and later embedded in the **X-Heep** microcontroller to verify the correctness of the operations in a real environment. The resulting Schematic of the Overall Architecture compatible with X-Heep can be observed as:

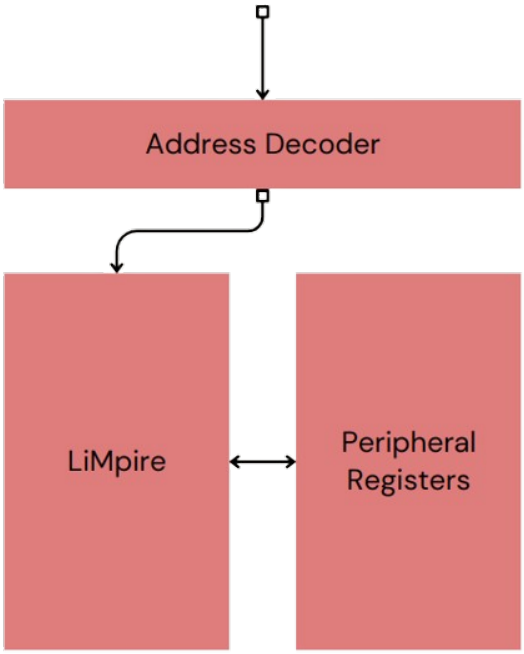


Figure 5. LiM X-Heep Implementation Schematic

The use of X-Heep allowed for the development of more complex benchmarks based on existing **cryptographic algorithms**, enabling performance comparisons between the CPU and the accelerator. The benchmarks showed a variable **speedup** of up to **10x**, depending on the algorithm and the operations performed.

The architecture was synthesized using the **SAED 14nm library** to collect comprehensive data on the dimensions of the blocks that make up the architecture. However, to achieve better performance, it is crucial to design a custom library based on **emerging technologies**, which would enable **area**

and **timing** optimizations.

# Contents

# List of Tables

# List of Figures

# Introduction

The work carried out in this thesis focuses on **Logic in Memory (LiM)**. The **memory wall** is a frequently debated issue in CPU architectures, as it limits the performance gains achieved through advancements in CPU technology. While **CPUs** continue to get faster over time, **memory capacity** has not progressed at the same rate. One solution is to modify the **CPU architecture** to support additional features that can slightly reduce the gap, such as out-of-order execution [1]. Another approach is to modify the **memory** by adding computational blocks inside or close to it, reducing both **power consumption** and **data-fetching latency**. This is known as the **Logic in Memory paradigm**.

This thesis is divided into two main parts: **literature analysis** and **architectural design**. The reason for this division lies in the close correlation among the phases: no design can be developed without first investigating the theory behind it.

The first part is titled **"Logic in Memory: A Physical and Architectural Overview"** and can be further divided into two blocks:

1. The first block investigates the most promising **emerging technologies** that can be used for manufacturing LiM Memory Arrays, such as resistive random-access memories (**RRAMs**), phase change memories (**PCMs**), ferroelectric field-effect transistors (**FeFETs**), hybrid CMOS, and Spintronic Devices based on the Magnetic Tunnel Junction (**MTJ**), such as Spin-Orbit Torque (**SOT**) Magnetic RAMs (**MRAMs**) and Spin-transfer Torque RAMs (**STT-RAMs**). These technologies offer advantages in terms of **power consumption**, **area**, and **performance**.

2. The second block is an analysis and comparison of the **architectural implementations** developed by researchers worldwide. This section highlights the most relevant features of each architecture.

The second part, titled **"The birth of LiMpire"**, describes and tests a

**High-Level Architectural Model** for LiM. The key concepts of the design are **versatility** and **customization**. Thus, there may be potential optimizations to improve the architecture's performance when executing specific **algorithms** or **benchmarks**.

The **Design Phase** was divided into the following steps:

- Identify and select all the **Boolean** and **Arithmetic Operations** to implement;
- Design an **Instruction Set** and a **Compiler**;
- Design the **datapath** of the architecture;
- Describe the **hardware** in SystemVerilog;
- Test the hardware using **TestBenches**;
- Install the LiM Architecture in the **X-Heep Microcontroller** and test it;
- Compare the performance of executing various **benchmarks** based on **cryptographic algorithms** when run on both the **CPU** and the **memory**;
- **Synthesize** the LiM.

This Block primarily focuses on the design, testing, and synthesis of the architecture.

Finally, the **Conclusions** and **Future Works** sections will follow.

# Part I

# Logic in Memory: a Phisical and Architectural overview

# Chapter 1

# Motivation and Background

## 1.1 An Introduction to the Logic-in-Memory Paradigm

Modern **CPU architectures** adopt the **Von Neumann** architectural model, which consists of:

- A **Processing Unit**, to perform all the calculations required by every instruction;
- A **Memory Unit**, to store all the data from the CPU (and peripherals) and load it to the CPU.

These units are connected through buses. Although **microprocessors** are significantly increasing their performance by becoming much faster, **memory systems** are not keeping pace in terms of capacity: there is currently no high-capacity memory that can match the performance of modern processors. This leads to a slowdown of the **CPU**, since it needs to adapt its speed to avoid causing any **bottleneck** during executing a LOAD or a STORE. Modifying at least one of the two units is the only way to tackle this issue.

The **Microprocessor** can be modified by implementing **pipelining** or **out-of-order execution**, which will consistently impact **energy consumption** [1].

Another solution is to modify the **memory** itself by implementing **computational units** near or inside it that can handle the simplest calculations

(mostly bitwise operations) and reduce the number of loads and stores to execute. This concept can be summarized as **"Logic in Memory" (LiM)**.

The purpose of this thesis is to provide a **High-Level Architectural Model** of **Logic in Memory** that can establish a standard and be compatible with the most common interfaces, such as the Open Bus Interface (OBI).

## 1.2 An Overview of LiM Implementations

The **Logic in Memory paradigm** allows for a very high degree of freedom when it comes to implementation; therefore, we can classify four main types of **LiM architectures** [1]:

1. **Computation-near-memory (CnM)**: We have two distinct logic units for computation and storage in this approach.[1] [2] The performance improvement is given by the very short interconnections, since the two units are close.[1] [2] This is possible due to **3D integration technology**.[1] [2] An example is **WIDE-IO2**, a 3D stacked DRAM memory that has a logic layer located at the bottom of the stack [1] [2].

2. **Computation-with-memory (CwM)**: This model is composed of a Memory Array, Look-Up Tables (LUT), and Content Address Memories (CAM).[1] The **LUT** is responsible for providing the result given a set of inputs, while the **CAM** is responsible for storing the results.[1] Inputs are provided to the LUT, which accesses the CAM, retrieving the address.[1] Once the address is obtained, it reads the stored results.[1] The **Memory Array** stores only precomputed results [1].

3. **Computation-in-Memory (CiM)**: In this category, the **Memory Array** is standard and unaltered.[1] Data computation is performed by modified peripheral circuits supporting bitwise operations.[1] Specifically, **Sense Amplifiers (SAs)** are modified to implement bitwise operations, and some specific **Decoders** are implemented to perform boolean bitwise operations among a set of addresses.[1]
   There is a sub-version of CiMs, called **Configurable Logic-in-Memory Architecture (CLiMA)**, that includes an in-memory computing unit.[1] **CiMs** can be manufactured using emerging resistive technologies, such as **memristors**, Magnetic Tunnel Junctions **(MTJs)**, and Phase Change Memories **(PCMs)**.[1] They offer high scalability and efficiency, as well as high density and low power consumption [1].

4. **Logic-in-Memory (LiM)**: This is the purest form, as the **Memory Array** is modified to add logic blocks in each memory cell.[1] This solution guarantees more flexibility and allows for the execution of more complex algorithms with reasonable performance [1].
   This is the chosen approach for this thesis' **Design Project** due to the previously described advantages.

# Chapter 2

# Overview of Technologies for Logic in Memory

The pathway of the first part of this thesis moves from analyzing the **technologies** that can be employed for Logic in Memory manufacturing to an overview of implementations for both **Memory Array** Architectures and **Top-Level** Architectures. This chapter will primarily focus on **FeFETs** and **Spin-Tronic** technologies.

## 2.1 FerroelectricFET Devices Overview

Transistors with integrated **ferroelectrics** offer unique possibilities for low-power and dense CiM applications, as they can function as standard transistors while retaining their logical state even without a power supply, offering high reliability in data retention.[3] **Ferroelectric FETs (FeFETs)** are similar to **MOSFETs**, except for the presence of a layer of ferroelectric (**FE**) **oxide** deposited in the gate stack.[3][4] Device non-volatility arises from hysteresis due to the coupling between FE and CMOS capacitances ($C_{FE}$ and $C_{CMOS}$).[4]
The **information** stored in a FeFET corresponds to one of two logic levels: "0" or "1".[3][4] The logic "0" is associated with high $V_{th}$, while logic "1" is associated with low $V_{th}$.[3][4] The **inherent gain** of FeFETs ($I_{on}/I_{off}$) is on the order of $10^6$, providing better state distinguishability, low leakage currents, and scalability to large arrays.[5][4] The elevated **ON/OFF ratio** enables energy-efficient voltage-domain operations with a full swing.[5]
These devices present a three-terminal structure (as shown in Figure 2.1)

that enables separate write and read paths.[3][4] The **write** process depends on switching the FE polarization with an appropriate $V_{gs}$, while **reading** is performed by sensing the drain-source current path.[5]

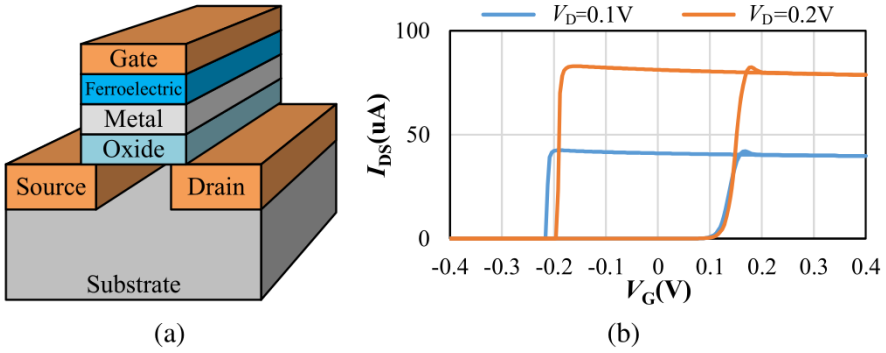


Figure 2.1. (a) FeFET structure. (b) Simulated hysteresis curve

### 2.1.1 FeFET Write Operation

During a write operation, the FeFET **gate** behaves as a **capacitive load**, eliminating the need for any drain-source current ($I_{ds}$), thus resulting in low writing energy as no DC power is consumed.[6] By applying an **external voltage** to the gate, the device can be set to the corresponding polarization state, which exhibits a different resistance state that can be read out by applying an appropriate $V_G$.[6]

## 2.2 Magnetic Tunnel Junction Devices Overview

**Magnetic Tunnel Junction** (MTJ) transistors are the core of MRAM bit cells, consisting of two ferromagnetic layers—called the "**pinned layer**" (orange) and the "**free layer**" (green)—separated by an oxide layer known as the tunneling barrier (blue) (Figure 2.2).[7] There are two stable magnetization orientations for the ferromagnetic layers: **parallel** (P) and **antiparallel** (AP).[7] As a result, MTJ devices exhibit two different resistance states due to the TMR effect: low resistance ($R_P$) and high resistance ($R_{AP}$).[7]

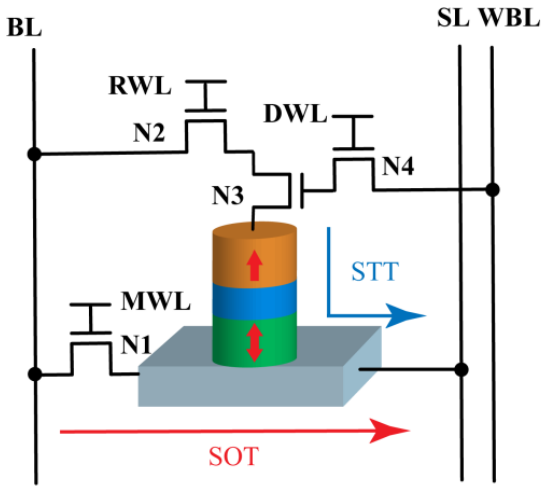


Figure 2.2. Hybrid memory cell schematic

### 2.2.1 STT-MRAM Write Mechanism

The write mechanism of MRAMs is **spin-transfer torque (STT)**.[7] In STT-MRAMs, data is written by a **bidirectional current** passing through a magnetic tunnel junction (MTJ).[7] The current switches the **magnetization** of the free layer, rewriting the stored data.[7] However, STT has an inherent incubation delay and requires a high critical current, which limits the write speed and results in poor endurance with high energy consumption.[7] To eliminate incubation delays and enable high speed, the **spin-orbit torque** (SOT) mechanism can be employed, which requires low-energy switching.[7][8] The **free layer** of the MTJ in SOT-MRAMs is connected to a **heavy metal strip**.[7] An in-plane current flowing through this metal generates a **Spin-Orbit Torque** (SOT), switching the magnetization direction of the free layer.[8][7] This process is mathematically described by the **Landau-Lifshitz-Gilbert** (LLG) equations:[8]

$$\frac{\partial \vec{m}}{\partial t} = -\gamma \mu_0 \vec{m} \times \vec{H}_{\text{eff}} + \alpha \vec{m} \times \frac{\partial \vec{m}}{\partial t} + \vec{\tau}_{DL} + \vec{\tau}_{FL}$$

$$\vec{\tau}_{DL} = -\lambda_{DL} J_{SOT} \xi \vec{m} \times (\vec{m} \times \vec{\sigma})$$

$$\vec{\tau}_{FL} = -\lambda_{FL} J_{SOT} \xi \vec{m} \times \vec{\sigma}$$

The equations describe how the **unit magnetization vector** ($\vec{m}$) of the free layer interacts with the **effective magnetic field** ($\vec{H}_{eff}$), along with

additional torque terms representing the damping-like ($\lambda_{DL}$) and field-like ($\lambda_{FL}$) components of the SOT.[8] These torques are influenced by the SOT current density ($J_{SOT}$) and a material-dependent parameter ($\xi$), which modulates the spin polarization.[8] $\gamma$ is the gyromagnetic ratio, $\mu_0$ is the vacuum permeability, and $\alpha$ is the Gilbert damping constant.[8] Additionally, conventional SOT-MRAMs suffer from **source degeneration**, where the driving ability of the access transistors is asymmetric, limiting the effectiveness of bidirectional current driving.[8]

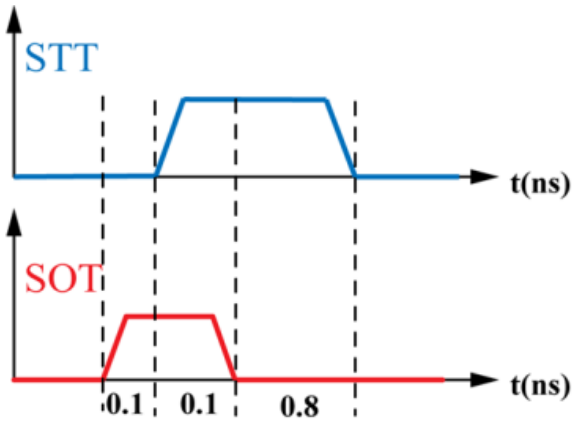


Figure 2.3. TST writing mechanism, showing two current pulses with time overlap

### 2.2.2 Field-Free Switching Mechanism in p-MTJ Devices

However, an **external magnetic field** (or special manufacturing processes) is required to achieve bipolar switching in perpendicular magnetic anisotropy MTJ (p-MTJ) devices.[7][8] The **toggle spin torques** (TST) mechanism combines both **STT** and **SOT** effects, enabling efficient, field-free switching in common **p-MTJ** devices.[7][8] The TST writing Mechanism is shown in Figure 2.3
Alternatively, a new **field-free** (FF) **SOT** mechanism has been developed, enabling **ultra-fast switching** of MTJs without the need for an external magnetic field, thus reducing energy overhead.[8] By optimizing the ratio between **damping-like torque** ($\lambda_{DL}$) and **field-like torque** ($\lambda_{FL}$), the MTJ can switch its magnetization direction in sub-nanosecond timescales.[8] This mechanism introduces **unipolar switching**, where the magnetization of

the free layer switches reliably in one direction, regardless of current polarity, eliminating the bidirectional current problem.[8]

# Chapter 3

# Overview on Architectural Implementations for Logic in Memory

Literature describes multiple **approaches** for architectural models and implementations for Logic in Memory, each one with different characteristics depending on:

1. **Technology** to employ for manifacturing;
2. Memory Array **Layout**;
3. **Operations** and **Algorithm** to execute;
4. Data **location** management;
5. Working **Modes**.

This thesis reports about the most significant implementations of Memory Array Architectures and Overall Architectures.

## 3.1 Architecture I: FePIM

There are roughly two categories of modern PIM (Processing-in-Memory) **implementations**:[2]

1. The first category takes advantage of **3D integration**.[2] Processing units are placed on the bottom layer of 3D stacking.[2]

2. The second category employs the emerging **non-volatile memory devices**.[2] The **peripheral circuitry** is modified in order to perform bitwise logic operations without the 3D integration.[2] This category can be manifactured by using resistive random-access-memories (**RRAMs**), **spin**-transfer torque random access memories (**STT-RAMs**) and Ferroelectric field-effect transistors (**FeFETs**).[2]

FePIM is based on the second approach, and it employs **FeFET** technology.[2] The proposed **FePIM** architecture can work in two modes: **Memory Mode**, to perform **LOAD** and **STORE** operations, and **PIM Operation Mode**, to perform mathematical operations. [2]
To schedule PIM operations, the memory features a **PIM Global controller** that fetches one PIM command for each clock cycle, decodes it, and forwards it to the appropriate bank(s) for computation.[2] Once the **Bank Controller** receives commands from the **Global Controller**, it generates all the control signals and the addresses to schedule the PIM operation inside the bank.[2] The **Bank Controller** is a Finite State Machine (FSM).[2] It supports: **AND**, **NAND**, **OR**, **NOR**, **XOR**, **XNOR**, and **NOT** operations.[2] Each bank is composed of:

- **FeFET memory array** for storing data, the core of the whole architecture;
- **Sense amplifiers** (SAs);
- **PIM Logics**;
- **Bank Controller**;
- Two **Forwarding Rows** (FwRows);
- Other **peripheral circuitry**.

The Architecture can be observed in Figure 3.3:

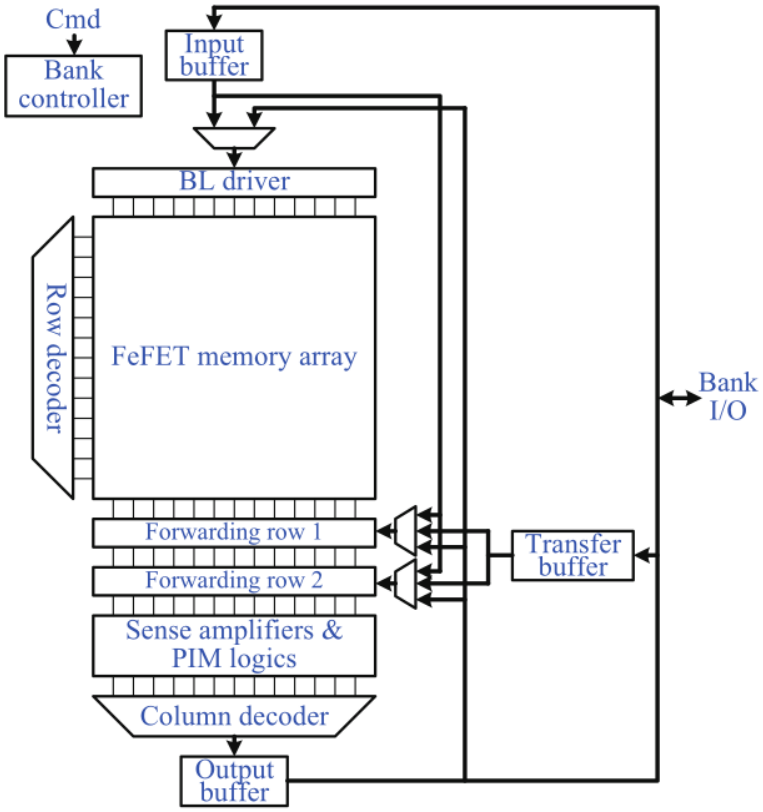


Figure 3.1. FePIM architectural Architecture

**PIM operations** can be performed between two rows (thus, two addresses) and between one immediate value and a row; the result is computed by the **Sense Amplifiers** and **PIM logics**.[2]
To explain how the **Forwarding Rows** work, let's make an example: we have to perform two operations: $a = b\ op1\ c$ and $d = a\ op2\ e$. In the first clock cycle, the two operands $b$ and $c$ are read, and therefore $a$ is computed.[2]
In the second clock cycle, the result of the first operation ($a$) is written back.[2] In order to compute $d$, $a$ has to be read along with $e$, thus bringing **SRAW** operations on the same row.[2] To do so, the architecture has to be **stalled** for one clock cycle, resulting in a slowdown in performance.[2] To avoid this issue, the architecture features **Forwarding Rows**.[2] Thus, $a$ is written to the forwarding row, which is stored in the output buffer.[2] In the meantime, $a$ can also be stored in memory, and $d$ can be computed by fetching $a$ from the forwarding row and $e$ from the memory row.[2] Briefly, Forwarding Rows are effectively two **additional memory rows**.

## 3.2 Architecture II: FeFET-CiM

Another architectural implementation for Logic in Memory is **FeFET-CiM**, featuring **2T+1 FeFET** memory cells.[4] This implementation supports the same operations as FePIM along with the addition of two words in memory.[4][2] The **Memory Array Architecture** includes the following components:

- **Column Decoder**;
- **Sense Amplifiers**, which are responsible for executing the operation;
- **Wordline Drivers**;
- Two **Row Decoders** (one for each operand);
- **Bitline Driver**;
- **Memory Array**.

The whole **Architecture** can be observed in Figure 3.3

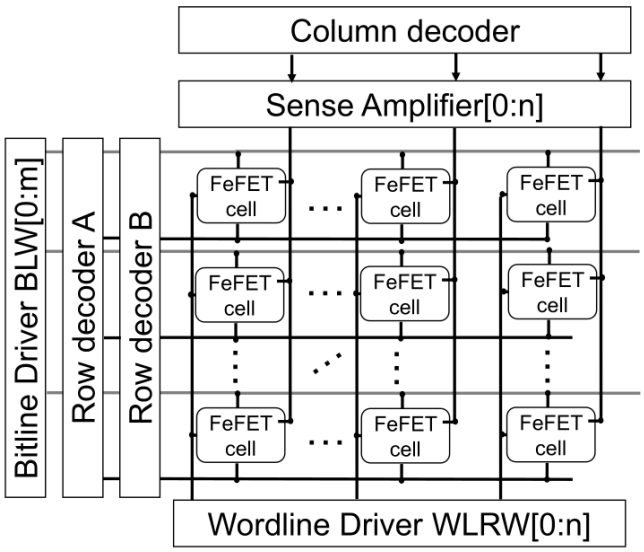


Figure 3.2. FePIM Architecture

The architecture works in two modes: **Memory Mode (MM)** and **Compute Mode (CM)**.[4] The **Column Decoder**, **Bitline**, and **Wordline**

**drivers** function the same way regardless of the mode, while the **Sense Amplifiers** are modified to efficiently accomplish CiM operations, and they can be either voltage-based and/or current-based.[4] **LOAD** (in MM), **OR**, and **NOR** (in CM) are performed entirely through a voltage-based sense scheme, while the other CiM operations (**AND**, **NAND**, **XOR**, **XNOR**, **NOT**, and **ADD**) are performed by exploiting a mixed voltage and current sense scheme.[4] Clearly, in order to perform a two-operand operation, it is necessary to activate two rows at the same time.[4]

## 3.3 Architecture III: FeMIC

**FeMIC** is another LiM architectural implementation based on **FeFETs**. Like the other implementations, it supports **Multi-Operands In-Memory Computing**.[9]
This approach allows for the integration of **processing units** inside the memory to locally perform part of the computation.[9] The FeMIC architecture supports:

- All the **2-operands** operations;
- **Multi-Operands** operations.

To support the latter type, the architecture features an elaborate **3T cell design** and a **forwarding row** (FwRow) mechanism.[6][9] The architecture presents the following blocks (Figure (3.3)) (as it can be observed in Figure 3.3):

- **Input and Output Buffers**;
- **Forwarding Row**;
- **SAs** and **CiM logics**;
- A **Sense Line** (SL) Driver;
- An **Address Decoder**;
- **Memory Array** made with a **3T cell design** (one **FeFET** and two access transistors).

FeMIC can work in three different modes: **Memory** Mode, **Two-operands Compute** Mode, and **Multiple Operands Compute** Mode.[6][9] The

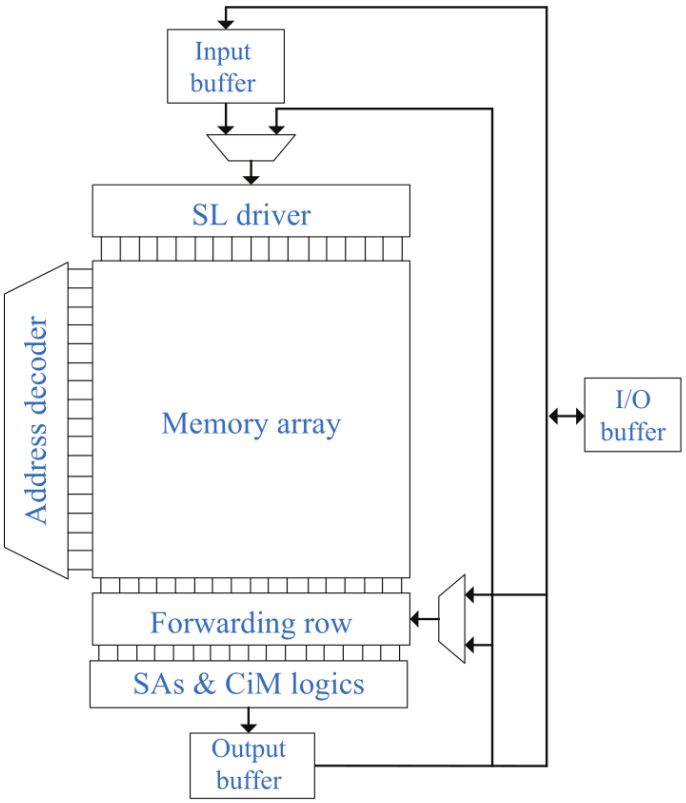


Figure 3.3. FeMIC Bank Architecture

working principle of two-operand CiM operations consists of activating the corresponding **memory rows** so the **SAs** (which have been accordingly modified to support data computation) can compute and output the results.[9] Then, results are written back to the **memory array** through the **SL driver**.[9] For subsequent CiM operations, they are written in the **FwRow**.[9] For multi-operand operations, the FwRow is used to store **temporary results**.[6][9]

## 3.4 Architecture IV: FeCrypto

**FeCrypto** was developed to support multiple cryptographic functions, and it is based on FeFET technology.[6] This architecture presents a custom **Instruction Set Architecture** (ISA). An ISA is an abstraction of the hardware-software interface that specifies what fundamental operations hardware supports.[6]

FeCrypto is based on a **multi-operand CIM** architecture, which supports all the **2-operands** operations along with the multi-operand due to an elaborate **3T cell design** and a **forwarding row** (FwRow) mechanism.[6][9] The **memory array** block is exactly the same as FeMIC (Figure (3.3)).
This architecture has been developed to support **Advanced Encryption Standard** (AES) algorithms and several hash functions, such as **MD5**, **SHA-1** (described in Section 9.1.5), and **SM3**.[6][9] FeCrypto supports seven types of operations: **AND**, **NAND**, **OR**, **NOR**, **XOR**, **XNOR**, **ADD**, **MUL**, **NOT**, and **ROL/ROR**.[6] It is worth noting that it includes both **MUL and ROL/ROR** operations since cryptographic algorithms need both cyclic and logical shifts.[6] It supports three types of instructions: **Memory Access**, **Shift**, and **CiM**.[6] FeCrypto **Overall Architecture** consists of:

- **Instruction Fetcher**;
- **Instruction Decoder**;
- **FeMICs** (in-memory computing units), which include the memory array;
- **S-Boxes**;
- **Shifters**;
- **Output Register**.

The **Architecture** can be observed in Figure 3.4:

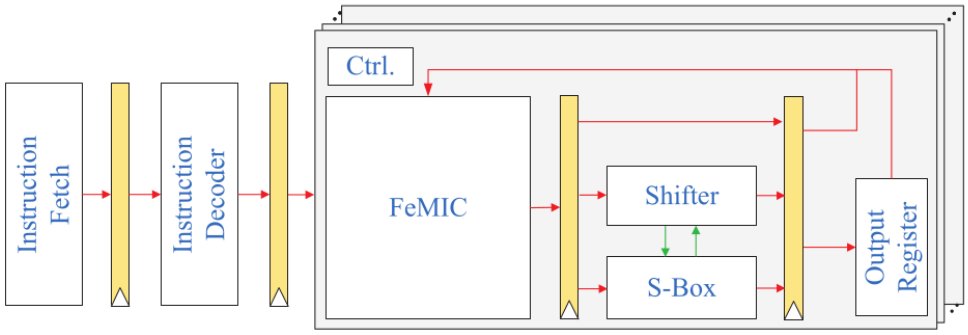


Figure 3.4. FeCrypto Architecture

The architecture presents five pipeline stages: **fetching**, **decoding**, **CiM operations**, **shifting/replacing**, and **writing back**.[6] As a standard pipeline, in **CiM mode**, the instruction is fetched, decoded, and sent to FeMICs, which are in charge of executing the corresponding operation.[6] Then, data is forwarded to the following pipeline stages, even though they might not need to perform shifts or S-Box replacement operations.[6] Indeed, if the data doesn't need to be further elaborated, these blocks will only forward it to the next stage to keep the architecture synchronized.[6] Finally, output **results** are written to registers or memory.[6]

## 3.5 Architecture V: BLiM

**BLiM** architecture is based on **FeFets**.[5] This architecture adopts single-level per FeFET cell (**SLC**) to enable a high noise margin for data storage, sensing, and driving.[5] The architecture is composed of the following blocks (as shown in Figure 3.5):

- **Memory Array**;
- **Row** and **Column Decoders**;
- **Controller**;
- **Sensing Interface**;
- Optional **Peripherals**.

This architecture supports two types of BLiM computing: **Type-I operations** and **Type-II operations**.[5] The former processes all the inputs in parallel without the need for external logic gates, while the latter supports multiple inputs or more complex logic.[5]
The **Memory Array** is composed of single-level FeFET cells (SLC) to enable a high noise margin for data storage, sensing, and driving.[5] Therefore, **2 transistors per cell** (2T/C) and 3T/C designs are used.[5] This choice enhances the computing capability by adding more functionalities and increasing reliability.[5] These cells present a **write line (WL)** for writing, a **read line (RL)** for reading, and an additional horizontal line for computing.[5]

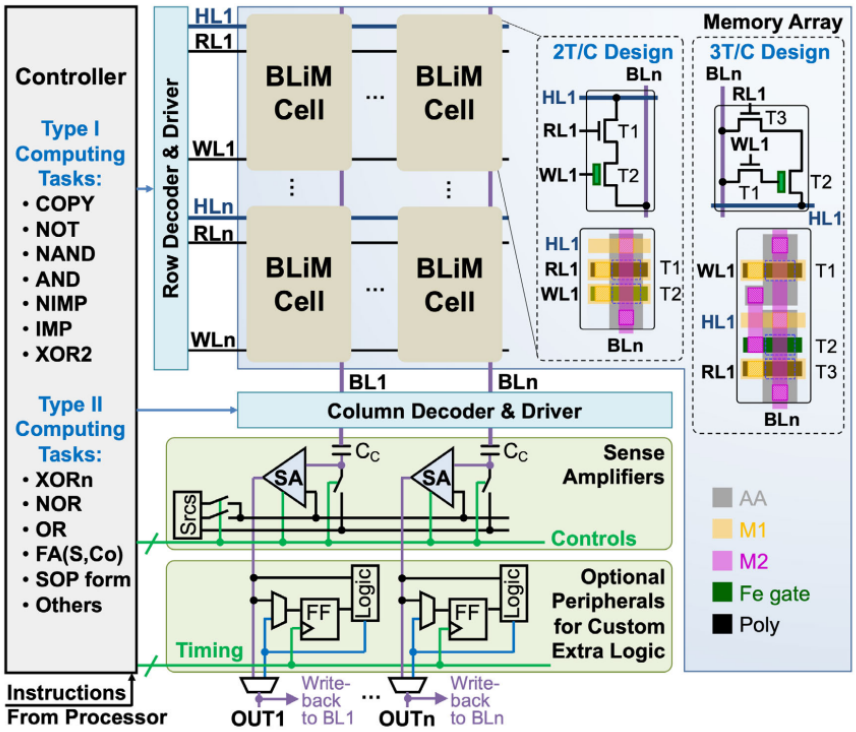


Figure 3.5. BLiM Architecture

## 3.6 Architecture VI: FeMAT

The **FeMAT** array introduces an innovative design that integrates multiple functions into a FeFET-based memory array.[3] It is composed of **3T-based memory cells**, each containing one FeFET and two access transistors.[3] The architecture can operate in four modes: **memory**, **computational memory**, **BCNN acceleration**, and **TCAM**.[3]
The array is composed of (as shown in Figure 3.6):

- **FeFET-Based Memory Cells**: Each cell is composed of **3T cells** (one FeFET, an access transistor for writing (ATW), and an access transistor for reading (ATR));
- **Memory**, **PIM**, **TCAM**, and **BCNN SAs**;
- **Address decoders**;
- **BL**, **DL**, **WL**, and **RSL drivers**.

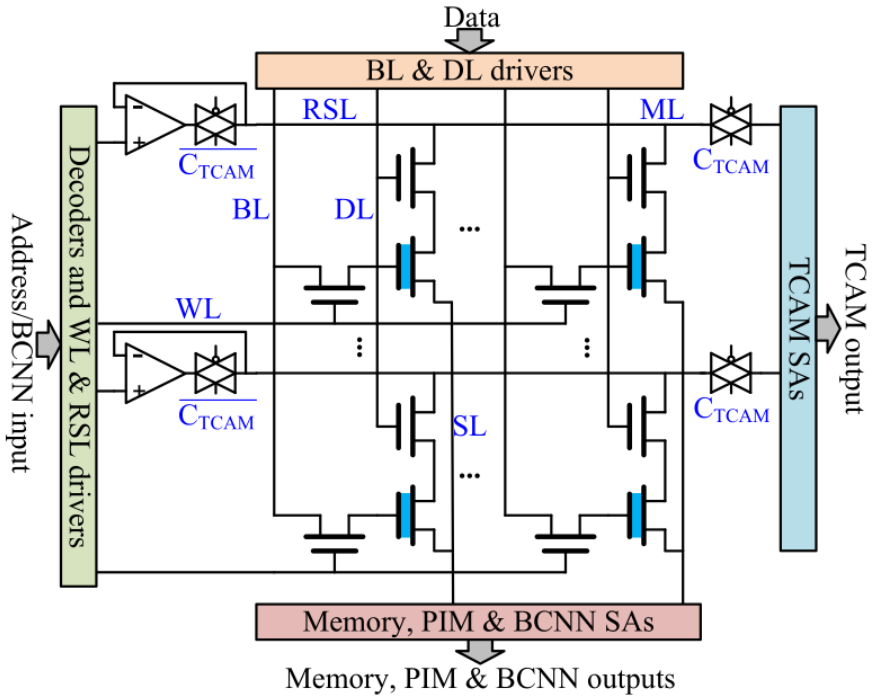


Figure 3.6. FeMAT Cell Architecture

The working modes of the array are:

- **Memory and Computational Memory** Modes: These modes are described together since they share the same data path and **SAs**.[3] The **CTCAM** signal controls the connection of TCAM SAs and voltage followers.[3] In this mode, FeMAT supports row-wise **read** and **write** operations.[3] It can also perform bit-wise **operations** and **additions** among two rows in computational memory mode;[3]
- **TCAM Mode**: In this mode, the TCAM SAs are connected to match lines (MLs), and all FeFETs maintain their states for comparison with the input word.[3] FeMAT functions as a TCAM by **comparing** the search data with all stored data in parallel, allowing for high-speed content-based data retrieval.[3]
- **BCNN Acceleration Mode**: The array is configured to perform **XNOR** operations followed by **accumulations** to execute binary convolutions.[3] Indeed, the array behaves as a BCNN accelerator by performing XNOR operations and accumulations, which are equivalent to convolutional operations with binarized inputs and weights.[3]

## 3.7 Architecture VII: reFeMAT

Another considerable implementation for Logic in Memory is **reFeMAT**, whose memory array architecture is a modified version of FeMAT, based on FeFET technology.[10][3] reFeMAT supports five functions: **PIM logic operations**, **BCNN and CNN accelerations**, **TCAM**, and **nonvolatile memory**.[10] These functions can be performed depending on the following modes:

- **PIM Logic Operations**. In this mode, reFeMAT can perform processing-in-memory (**PIM**) logic operations, thus executing logic functions directly within the memory to reduce data movement and latency;[10]
- **BCNN Acceleration**. This mode enables the architecture to accelerate binary convolutional neural networks (**BCNNs**), which are neural networks that operate on binary data, while optimizing the efficiency of neural network computations;[10]
- **CNN Acceleration**. In this mode, reFeMAT can accelerate convolutional neural networks (**CNNs**), which are crucial for tasks like image recognition and are more complex and accurate than BCNNs;[10]
- **TCAM**. **reFeMAT** can operate as a ternary content-addressable memory (**TCAM**), allowing rapid searches for matching data patterns, useful in applications requiring high-speed data retrieval and comparison;[10]
- **Nonvolatile Memory**. Finally, reFeMAT can function as a conventional nonvolatile memory, storing data persistently without the need for a constant power supply, ensuring data retention even when powered off.[10]

The major component of the memory array is a **crossbar**, composed of FeFETs basic cells.[10] The peripheral circuits include:

- **Read and Write Address Decoder**;
- **BL Driver**;
- **DL Driver**;
- **WL Driver**;
- **RSL Driver**;

- **Sense Amplifiers (SAs)** for PIM Logic, BCNN acceleration, memory, and TCAM modes.

The Full **Cell Architecture** can be observed in the following picture:

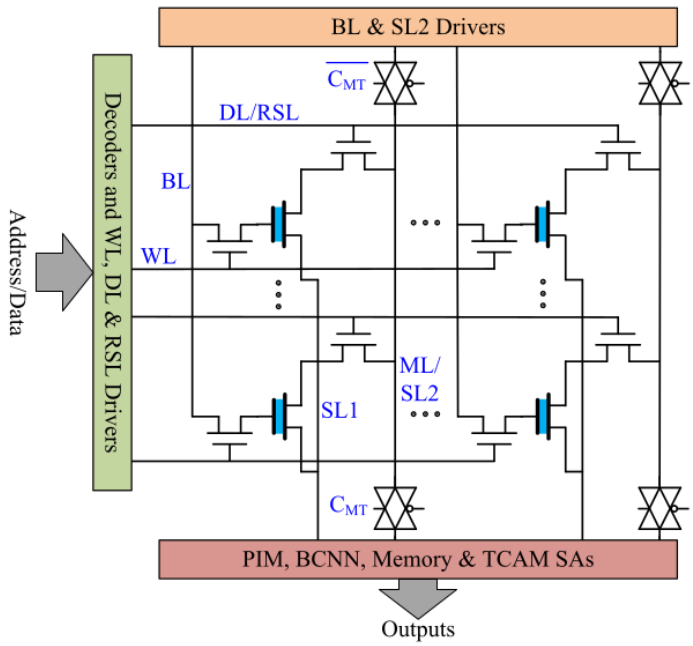


Figure 3.7. reFeMAT Cell Architecture

For **mode switching**, some transmission gates are employed, along with multiplexers and other transmission gates inside the peripheral circuits.[10]

The **top-level architecture** (as shown in Fig. 3.8) resembles an **FPGA**, where the configuration process is executed offline, ensuring that the circuit will operate in one mode at startup.[10] The main blocks of this architecture are: **Processing Elements** (PEs), **Connection Blocks** (CBs), and **Switching Blocks** (SBs).[10] Each PE is composed of a FeFET array and the following peripheral circuits: **analog-to-digital converters** (ADCs), **input registers** (IRs), **output registers** (ORs), **shift and add units**, and **digital units**.[10] PEs are interconnected through SBs and CBs.[10] It's worth noting that SAs operate in the same way in every mode.[10]

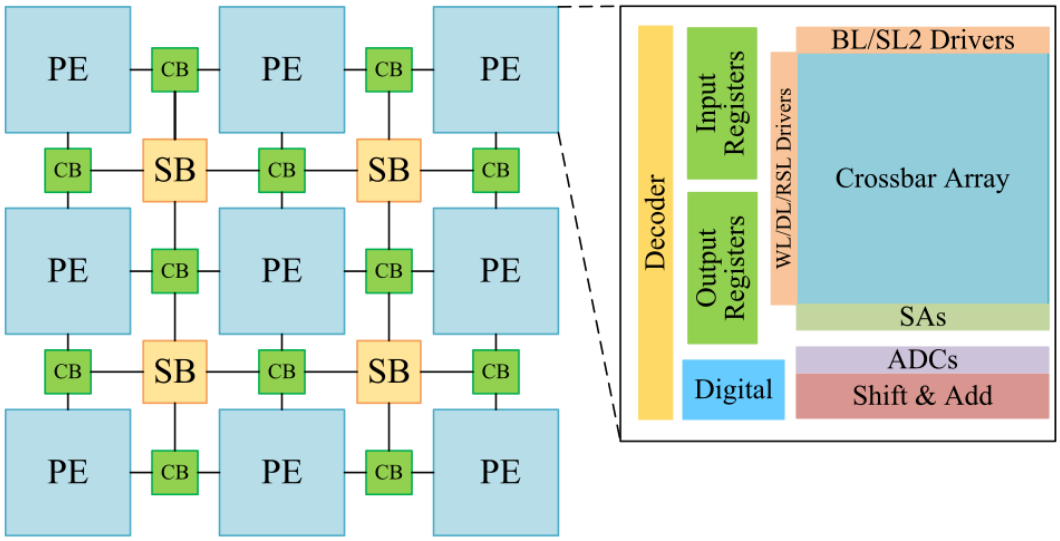


Figure 3.8. reFeMAT Architecture

## 3.8 Architecture VIII: STT-CiM

**STT-CiM** (STT-MRAM based compute-in-memory) is a design for in-memory computing that uses standard **STT-MRAM** arrays.[11] The key concept behind this implementation is to enable multiple wordlines simultaneously in an STT-MRAM array, leading to multiple bit-cells connected to each bitline.[11] This architecture can perform the following operations:

- **OR and NOR**;
- **AND and NAND**;
- **XOR and XNOR**;
- **ADD** by means of XOR, AND, and OR operations.

To implement these operations inside the array, the **ISA** has been expanded to include some custom instructions, such as **CiMXOR**, **CiMAND**, **CiMADD**, etc.[11] When performing a **LOAD**, the requested address is sent to the memory, which provides the data located at that address.[11] When performing a **CiM Operation**, two locations need to be read simultaneously using registers.[11] Traditional system buses can only transmit one address at a time, but we need to send two.[11] To achieve this, we utilize the **write-data**

**channel** of the system bus, which remains idle during CiM operations.[11] Along with the addresses, the CPU sends the operation (**CiMType**) to the memory through the **Control Bus**, which has been modified by adding 3 bits.[11] The **Architecture** presents the following Layout:

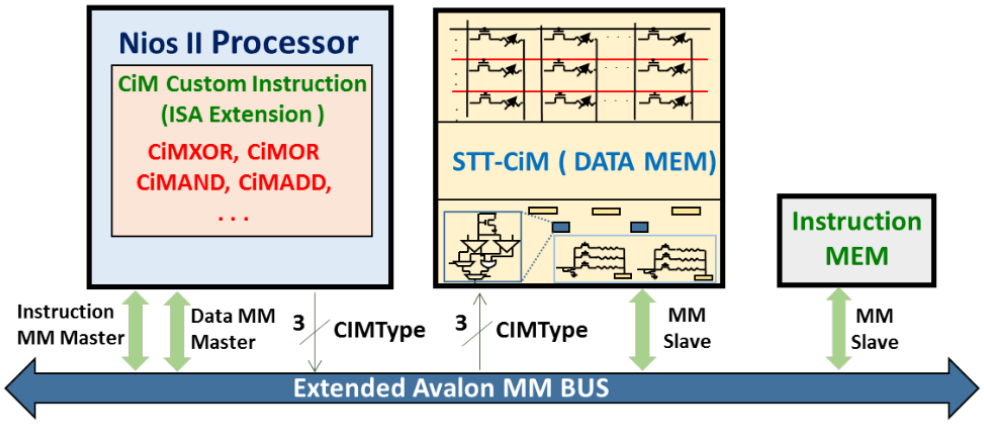


Figure 3.9. STT-CiM Architecture

**CiM Operations** can be executed only if the following constraints are satisfied:

1. Operands are stored in the **same bank**;
2. Operands are mapped to **different rows**;
3. Operands are stored in the **same set of columns**.

Let's consider a **multi-bank memory**, where each bank contains an array characterized by rows and columns.[11] We can distinguish three types of operations (as shown in Figure 3.10):

1. **Type I**. This is a standard two-operands operation.[11] To ensure that they are aligned in the same column, the array alignment technique is exploited.[11] An extension of this technique, called **"row-interleaved placement"**, is applicable to larger data structures not fully residing in the same memory bank by ensuring proper mapping.[11]
2. **Type II**. This is a one-to-many operation, where one operand must be operated on several elements of an array.[11] The **spare row technique**

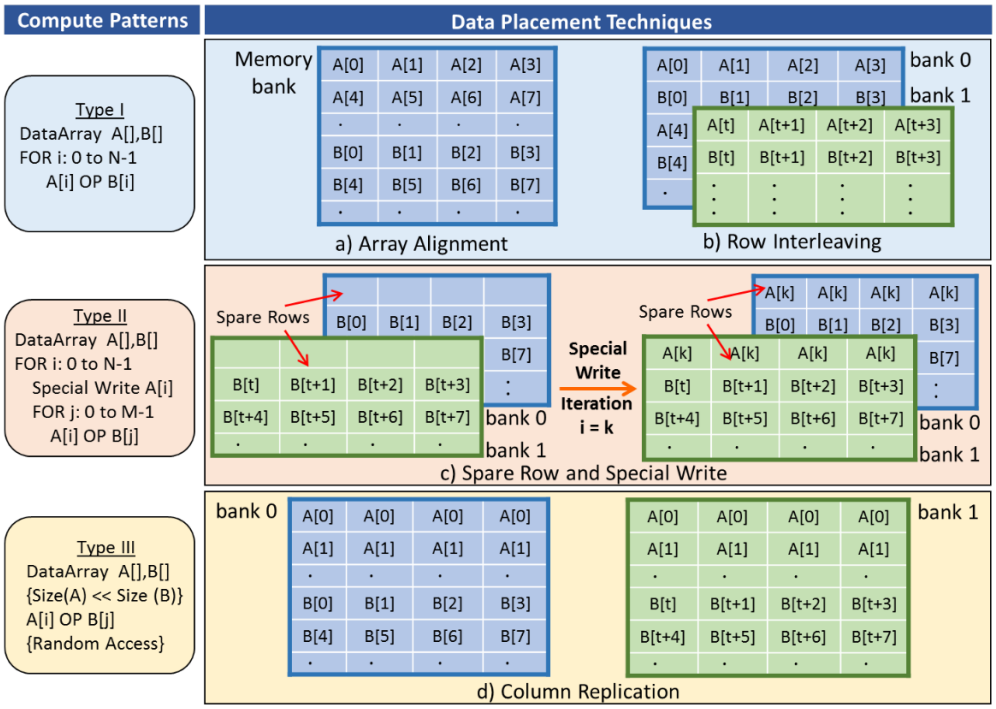


Figure 3.10. STT-CiM Data Placement Techniques

is used for data alignment, which involves reserving one spare row in each memory bank to store copies of an element.[11] The spare row is filled with an element of the first operand, and then the operation is treated the same way as Type I.[11]

3. **Type III**. In this case, operations are performed on an element drawn from a small array and an element from a much larger array.[11] The elements are selected arbitrarily, for example, for executing search algorithms.[11] To enable CiM operations, **column replication** is used, where a single element of an array is replicated across columns to fill the entire row.[11] This ensures that each replicated element of the first operand is aligned with every element of the other operand.[11]

## 3.9 Architecture IX: GraphS

?? **GraphS** is a SOT-MRAM graph processing accelerator built with MTJ technology.[12] The main memory chip is divided into **multiple banks**, connected with each other via an **I/O buffer**.[12] Each bank is composed of

multiple memory matrices (**mats**).[12] Each mat consists of multiple computational memory sub-arrays connected to a **Global Row Decoder** (GRD) and a shared **Global Row Buffer** (GRB).[12] The **GraphS Controller** (Ctrl) is responsible for configuring the data array to execute data-parallel inter- and intra-sub-array computations.[12] Between every two sub-arrays, there is a **Local Row Buffer** (LRB) and a **Digital Processing Unit** (DPU) to process the data.[12]

The array features the following components (as shown in Fig. 3.11):

- **Write Driver** (WD);
- **Memory Row Decoder** (MRD);
- **Memory Column Decoder** (MCD);
- **Reconfigurable SAs**, which can work in dual mode to perform memory read/write and bit-line computing.

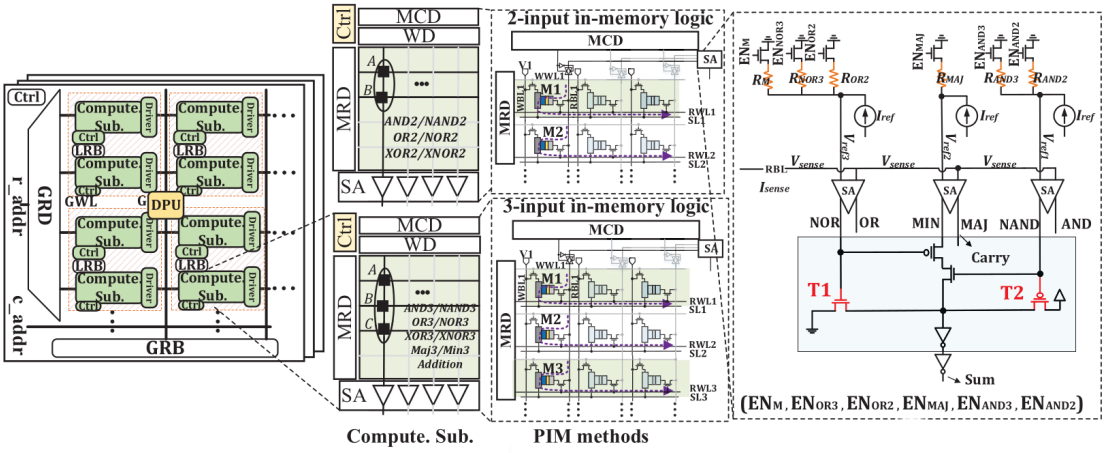


Figure 3.11. GraphS Architecture

The architecture's working process distinguishes between two modes: **Memory** Mode and **Bit-Line Computing** Mode.[12] The **Memory Mode** is the standard load and store operation.[12] In contrast, the **Bit-Line Computing Mode** consists of executing operations between two or three operands located on the same bit-line.[12] This is facilitated by the computational sub-arrays, which can select and sense multiple bits simultaneously using the Memory Row Decoder **(MRD)**.[12] The re-configurable Sense Amplifier

**(SA)** plays a crucial role in this mode, as it can be adjusted to perform various logic functions by selecting different reference resistances and enabling specific sub-SAs.[12]

## 3.10 Architecture X: CRISP

The **CRISP** architecture is designed to compute matrix multiplication and is based on **spin-tronics-assisted logic-in-memory** (SLIM) cells.[13] The SLIM cell leverages the properties of **SOT-MRAM** to perform basic logic operations, such as NAND, within the memory itself.[13] This **integration** of logic and memory reduces the number of memory cycles required for operations, enhancing energy efficiency and throughput.[13] The SLIM cell operates by **initializing** output cells to a known state and then **applying** inputs to generate currents that, when accumulated, can flip the output cells based on the logic operation being performed.[13]
The architecture features a **weight array**, two **computing arrays**, and **peripheral circuits** to support in-memory operations (as shown in Figure 3.13).[13] The weight array stores weights in STT-MRAM cells, while the computing arrays consist of SLIM cells for partial product generation and addition.[13] The architecture is designed to allow for the parallel processing of multiple matrix multiplications by connecting memory cells in the weight array and computing arrays via **computing lines** (CLs).[13] The **CRISP subarray** includes **wordline** (WL) **drivers**, **bias voltage drivers** for the VCMA effect, **sense amplifiers**, and a **shift-adder** to complete the multiplication process.[13]
The described **Cell Architecture** can be observed in the following Figure:

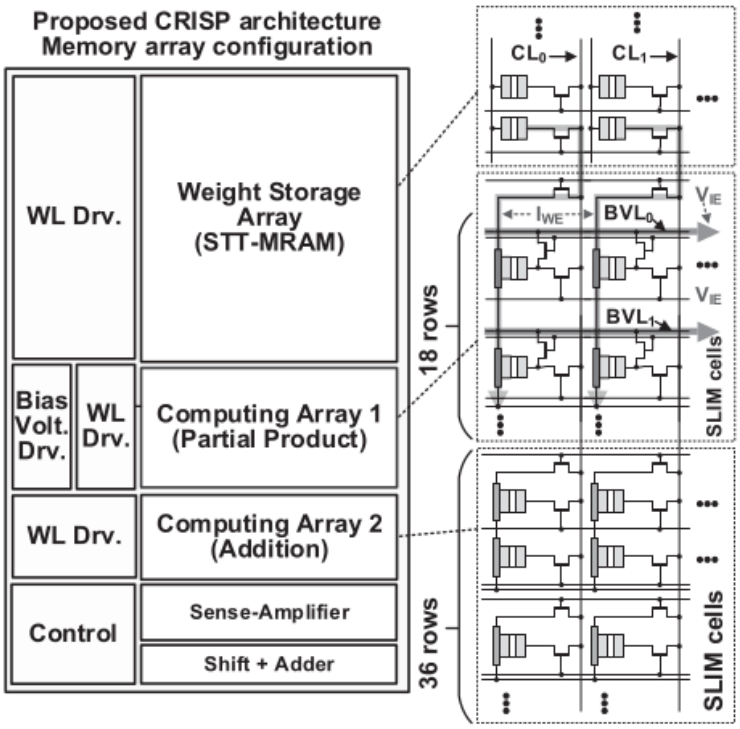


Figure 3.12. CRISP Memory Array Architecture Overview

Furthermore, the architecture exploits **intra-memory block pipelining** to further improve throughput.[13] This technique involves overlapping operations within the memory blocks to maximize the utilization of memory cells and logic gates, reducing idle time and increasing the overall efficiency of the computation process.[13] The overall architecture of the CNN processor features **processing elements** (PEs) and an external **I/O interface**.[13] Each PE contains multiple **CRISP subarrays** for performing matrix multiplications, an **input storage array**, and a **global function unit** for additional CNN-related computations, such as max-pooling and activation functions.[13] The weights for the CNN layers are partitioned and stored in the **CRISP sub-arrays**.[13] The computation process involves loading inputs into the input storage array and then deploying them to the **input buffers** inside the PEs.[13] The intermediate results from the CRISP subarrays are grouped in the **global function unit** for final processing.[13] The **Overall Architecture** is shown in Figure 3.13.

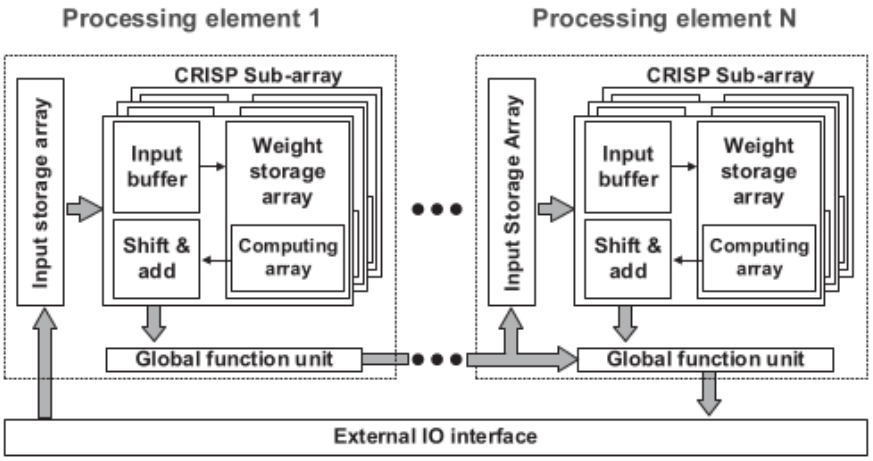


Figure 3.13. CRISP Overall Architecture Overview

## 3.11 Architecture XI: ParaPIM

The last architecture under analysis is the **ParaPIM** accelerator, which is designed to efficiently perform inferences for Binary-Weight Deep Neural Networks (BWNNs) using **Processing-in-Memory** (PIM) techniques with **Spin-Orbit Torque Magnetic Random Access Memory (SOT-MRAM)** sub-arrays. [14] The **SOT-MRAM** cells within these sub-arrays can perform logic operations by comparing the resistance of the cells with reference resistances.[14]
The **ParaPIM** accelerator architecture includes **Image** and **Kernel Banks** for respectively storing input feature maps and kernels.[14] These banks feed into **SOT-MRAM-based computational sub-arrays** where the bulk of the processing occurs.[14] A **Digital Processing Unit** (DPU) is also part of the architecture, containing dedicated units for binarization, batch normalization, and activation functions.[14] The DPU ensures that the inputs and outputs are properly formatted for the computational sub-arrays and the neural network operations.[14]

The **computational sub-arrays** are the core of the ParaPIM accelerator, featuring a design that allows for both memory operations and local bit-line computing.[14] The sub-arrays consist of several components (as shown in Figure 3.14):

- **Write Driver** (WD);

- **Memory Row Decoder** (MRD);
- **Memory Column Decoder** (MCD);
- **Reconfigurable Sense Amplifier** (SA).

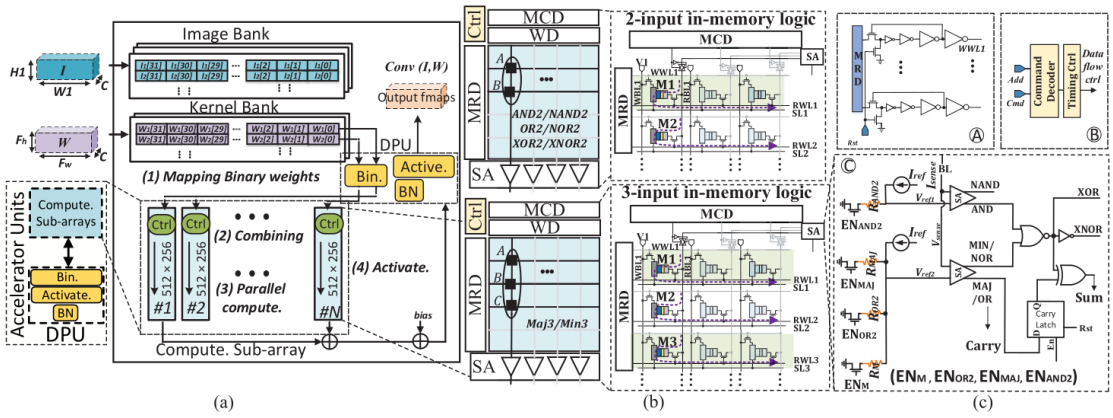


Figure 3.14. (a) ParaPIM accelerator architecture, (b) Computational sub-array of ParaPIM and its 2-input and 3-input local logic operations, (c) Peripherals of SOT-MRAM computational sub-arrays to support computation

The SOT-MRAM sub-arrays can function in both **Memory** and **Bit-line Computing** Modes.[14] In Memory Mode, the sub-arrays perform traditional read and write operations.[14] In contrast, the Bit-Line Computing Mode consists of local data-parallel computation directly within the memory cells.[14] This is achieved by setting different reference resistances to perform various logic functions (e.g., AND, OR, MAJORITY) on operands located in the same bit-line.[14]

Furthermore, the architecture features the **in-memory convolver**, which is crucial for BWNN operations.[14] The convolver can handle two types of convolution operations: massive binary-weight convolution and bit-wise convolution.[14] The architecture supports these operations efficiently, further enhancing the accelerator's performance.[14]

# Chapter 4

# Conclusions on Literature Implementations

The literature analysis presented several possibilities for implementing **Logic in Memory Architectures** at different levels: technological and architectural.
However, all the implementations reviewed are **highly optimized** for specific algorithms or benchmarks, making them challenging to adapt to other environments. Moreover, the most significant results and data reported in the analyzed papers are primarily based on **physical simulations** of the designs, with less emphasis on the performance of the benchmarks used to test these architectures.
This lack of performance data makes it difficult to adopt these models for future work, as it becomes impossible to select an architecture based on the benchmark to be run and the desired speedup. Therefore, the aim of this thesis is to develop and design a **High-Level Architectural Model** for Logic in Memory that supports running multiple algorithms for diverse purposes while incorporating most of the features implemented in existing architectures.
In addition to the model, this thesis seeks to provide **performance** data and **implementation** insights using currently available technologies. To achieve this, the design will be set at a higher level compared to the previously described implementations, focusing on the primary components and overall functionality without delving into the physical, low-level design details.
The main **specifications** of the proposed model are outlined in Chapter 5, while the architectural model is described in detail in Chapter 6.

# Part II

# The birth of LiMpire

# Chapter 5

# High-Level Architectural Model Design - Phase I: Preliminary Steps

## 5.1 Architecture Specifications

Following an in-depth investigation of **LiM Architectural Implementations** in the literature, defining the **Design Specifications** became necessary. The resulting main specifications of the design are as follows:

- The Model must have its own **Instruction Set**;
- The Model must support all major mathematical operations: **AND, NAND, OR, NOR, XOR, XNOR, NOT, ADD, SUB**, and **MUL**;
- **Instructions** length should be kept as short as possible;
- The Model should feature multiple **memory banks**, each consisting of 32 rows of 32-bit words;
- The Model must support operations across data from multiple **banks** and among **address ranges**, up to eight **operands**;
- The Model must support **parallel data** computation across banks;
- The **Model** must support **full-word** and **partial-word** operations.

All these specifications set a challenge throughout the entire design phase, due to the difficulty of integrating multiple features while ensuring high performance.

## 5.2 Instruction Set Design: Custom Assembly

The architecture design begins with the **Instruction Set Design**. The overall working principle of the architecture is to store a batch of **instructions** inside an **Instruction Memory**, allowing one instruction to be fetched every clock cycle. Therefore, designing or adopting a language to translate them is essential. The chosen approach is to adopt a structure inspired by the **RISCV Assembly Language** [1] [6], which is subsequently customized. The basic **RISCV Instruction** is as follows:

**OPERATION DEST, OP0, OP1**

Due to the multiplicity of **instruction types** to implement, a preliminary distinction was made based on the number of operands. The resulting distinction is:

- **One Operand** instruction;
- **Two Operands** instructions, where addresses are not necessarily consecutive;
- **Three Operands** instructions, where addresses are not necessarily consecutive;
- **Address Range** Instructions, in which all consecutive addresses in the range should refer to the same **bank**.

Unfortunately, **Address Range Instructions** and **Two-Operand Instructions** share the same layout in assembly, so the "R" flag has been added as a suffix to the operands to distinguish between them (e.g., "32R" for Address Range and "32" for standard Two-Operands). Additionally, **parallel computation** is implemented by adding the "A" flag as a suffix to the Destination; while **partial-word** computation is supported through the implementation of "H" and "L" flags next to the operands and destination, specifying the most significant 16 bits or least significant 16 bits, respectively.

If partial-word suffixes are not present, the instruction operates on the full 32-bit word.

All instruction types support partial-word and full-word computation. However, Global Parallel Computation and Address Range Computation are supported only if all operands and the destination refer to the same memory bank.

The resulting assembly language includes:

- **One Operand** instruction:

  **OPERATION DEST, OP0**

- **Two Operands** instruction:

  **OPERATION DEST, OP0, OP1**

- **Three Operands** instruction:

  **OPERATION DEST, OP0, OP1, OP2**

- **Address Range** instruction:

  **OPERATION DEST R, OP0 R, OP1 R**

  where OP0 and OP1 can be the starting and ending addresses of the range (or vice versa). Address decoding and range dimension estimation are handled by the **Decode Unit**.

The resulting assembly language is simple and fast to write, despite the absence of selection (e.g., CMP operations) or loops. An example of code is:

```
AND_LIM 32, 27, 32
OR_LIM 25AH, 1H, 2L
XOR_LIM 20AL, 30L, 29H, 19H
NOT_LIM 24A, 31
AND_LIM 44, 46, 39
OR_LIM 25AR, 1R, 5R
OR_LIM 25AR, 2R, 7R
AND_LIM 30AR, 1R, 8R
NOR_LIM 30AR, 1R, 3R
NOR_LIM 30AR, 1R, 4R
OR_LIM 55, 54, 64, 97
XNOR_LIM 127, 1, 32, 140
```

```
NOR_LIM 40L, 106L, 72H
AND_LIM 30, 29, 32, 28
NOT_LIM 22, 21
NOT_LIM 23, 46
NOT_LIM 22L, 46H
AND_LIM 20, 19, 18, 17
OR_LIM 17, 63, 18, 128
XOR_LIM 32R, 33R, 38R
NOR_LIM 0AR, 1R, 7R
XOR_LIM 32R, 33R, 38R
```

## 5.3 Instruction Set Design: Machine Code

The custom assembly language offers a relatively **high level of abstraction** for programming the LiM Architecture, rather than writing binary words made of "0" or "1". However, the **Instruction Memory** in the architecture supports only binary codes, so each assembly instruction needs to be translated into binary.

Like the assembly language, the **RISCV Instruction Layout** was chosen as a starting point for the **LiM Instruction Set**. **RISCV ISA** distinguishes six types of instructions, each with different bit allocations for fields [1]:

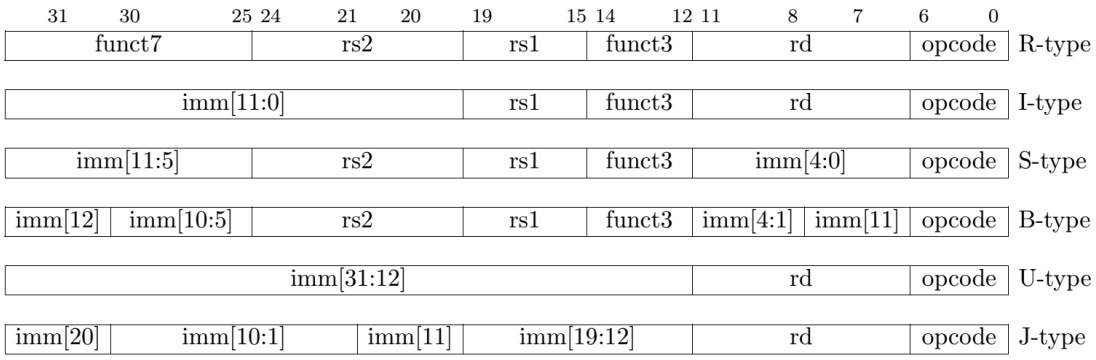


Figure 5.1. RISCV Instruction Set

Initially, the goal was to keep the instruction length at 32 bits, as most buses are powers of 2 (including those in X-Heep), but this approach proved unworkable, so the instructions had to be expanded. The proposed instruction set features **43-bit instructions**.

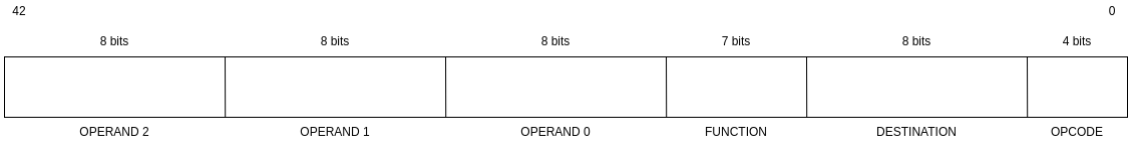


Figure 5.2. Custom LiM Instruction Set

As shown in Figure 5.2, the instruction word is divided into six main blocks:

1. **OPCODE**: bits 0-3. This section indicates the operation to perform. Operations are coded as:
   - **NOP** → 0000;
   - **AND** → 0001;
   - **NAND** → 0010;
   - **OR** → 0011;
   - **NOR** → 0100;
   - **XOR** → 0101;
   - **XNOR** → 0110;
   - **ADD** → 1001;
   - **SUB** → 1010;
   - **MUL** → 1011;
   - **NOT** → 1100.
2. **DESTINATION**: bits 4-11. This field specifies the 8-bit address of the **destination**.
3. **FUNCTION**: bits 12-18. This 7-bit word is divided as follows:

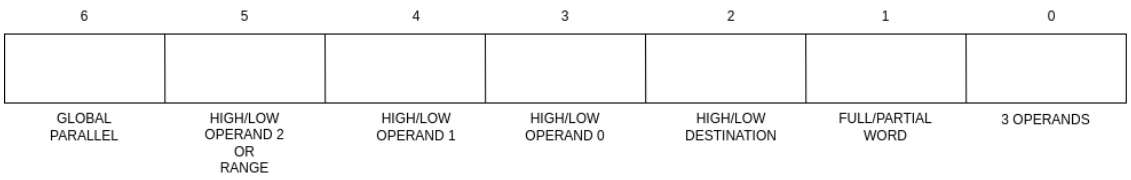


Figure 5.3. Custom LiM Instruction Set

The function of each bit is:

- **3 Operands**: If set, it's a **Three Operand** instruction, so bits 35-42 must be considered a valid address for the third operand, even if they are 0;
- **Full/Partial Word**: If set, the instruction works with partial-word operands.
- **High/Low Destination**: If Full/Partial Word is set, it becomes significant. If set, the result is written to the 16 MSBs; otherwise, to the 16 LSBs.
- **High/Low Operand 0**: If Full/Partial Word is set, it becomes significant. If set, the 16 MSBs of **Operand 0** are used; otherwise, the 16 LSBs.
- **High/Low Operand 1**: If Full/Partial Word is set, it becomes significant. If set, the 16 MSBs of **Operand 1** are used; otherwise, the 16 LSBs.
- **High/Low Operand 2 or Range**: If both Full/Partial Word and 3 Operands are set: if this bit is set, the 16 MSBs of Operand 2 are used; otherwise, the 16 LSBs. If Full/Partial Word is set but 3 Operands is clear, and this bit is set, it's a Partial Word Range Instruction. Otherwise, it's a Two-Operands instruction.
- **Global Parallel**: If set, it's a Global Parallel Instruction; otherwise, it's a standard operation.

Originally, this block was 8 bits long, but it was shortened by combining the **High/Low Operand 2** and **Range** bits, as they are mutually exclusive.

4. **OPERAND 0**: bits 19-26. This field specifies the 8-bit address of **Operand 0**.
5. **OPERAND 1**: bits 27-34. This field specifies the 8-bit address of **Operand 1**.
6. **OPERAND 2**: bits 35-42. This field specifies the 8-bit address of **Operand 2**. If **FUNCTION[0]** is clear, it has no meaning since it's a **Two-Operands Instruction**. In Two-Operands Instructions, this field is set to all 0s.

Notably, regardless of the instruction type, the instruction set has a **fixed layout**. This design simplifies the decode and control tasks by splitting the instruction word into smaller, easier-to-process blocks.

## 5.4 Instruction Set Design: Compilation

The **compilation** of Assembly Instructions translates the assembly code, written by the user in our custom language, into 43-bit binary words. This process is crucial for the entire architecture to function. **Automating** it became essential through a dedicated **Python Script**, which takes the target `.txt` file written in assembly, reads each line, checks for **flags** (R, A, H, and L), sets the correct bits in the **FUNCTION** field, converts the addresses of the **operands** and **destination** from decimal to binary, and translates the **operation** (e.g., NOT_LIM) into the 4-bit **OPCODE**.

Finally, all the resulting words are appended to a 64-bit word (with only 43 meaningful bits), split into two 32-bit words, and written to a file called `Instructions.txt`. The decision to write 32-bit words is related to the implementation in **X-Heep**, as its internal bus is 32 bits wide. For further information about the X-Heep Integration, see Chapter 8.
For initial tests, the built-in **SystemVerilog function** `readmemb` was used to **load** the words from the file into the Instruction Memory. For **X-Heep**, an additional script was required to convert the content of each line into base 10, which was then written to a separate file called `Instructions_Decimal.txt`.
This step was necessary due to X-Heep's lack of support for file import functions, which required the instructions to be written into a dedicated header file. The process is explained in more detail in Chapter 8. For example, this is the assembly file written by the user:

```
MUL_LIM 9, 0, 160
MUL_LIM 10, 1, 161
MUL_LIM 11, 2, 162
MUL_LIM 12, 3, 163
MUL_LIM 13, 4, 164
ADD_LIM 14R, 9R, 13R
MUL_LIM 20, 29, 14
```

Once converted into binary, it will look like this:

```
00000000000000000000000010011011
00000000000000000000000000000101
00001000000010000000000010101011
00000000000000000000000000000101
00010000000100000000000010111011
00000000000000000000000000000101
00011000000110000000000011001011
00000000000000000000000000000101
00100000001000000000000011011011
```

```
00000000000000000000000000000101
01101000010010100000000011101001
00000000000000000000000000000000
01110000111010000000000101001011
00000000000000000000000000000000
```

Finally, it is converted to base 10 representation:

```
155
5
134742187
5
269484219
5
404226251
5
538968283
5
1749680361
0
1894252875
0
```

# Chapter 6

# High Level Architectural Model Design - Phase II: Full Model Design

The designed Custom Assembly and Machine Code Languages set the foundation for the **Architectural Model Design**. The purpose of the architecture is to efficiently execute all instructions while respecting the constraints specified in Chapter 5.1.

## 6.1 The Birth of LiMPire

The design of the model started with the establishment of the **Visual Identity** of the project, leading to the name "**LiMPire**" (a reference to the "Galactic Empire" from Star Wars media) and the creation of the following logo:

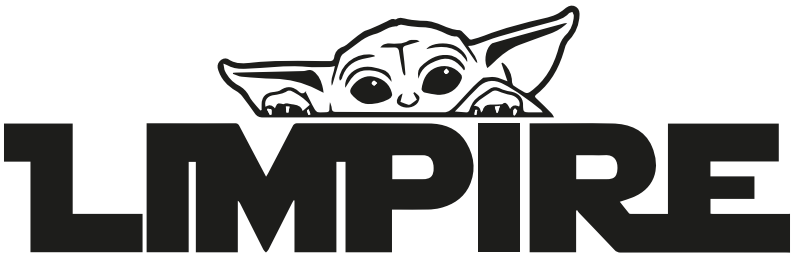


Figure 6.1. LiMPire Logo

The developed architecture is structured into three hierarchical levels:

1. Level 1 $\longrightarrow$ **Top Level**;

2. Level 2 $\longrightarrow$ **Datapath and Control Unit Level**;

3. Level 3 $\longrightarrow$ **Bank Level**.

For the implementation on **X-Heep**, a fourth level (called "**Level 0**") was introduced above the proposed scheme. It will be described later in Chapter 8. This section presents a detailed description of all the blocks composing the model, from the **Top Level** to the **Bank Level**. All blocks are written in **System Verilog** Hardware Description Language.

## 6.2 Level 1: Architecture

The **goal** of the LiMpire Architecture is to execute mathematical operations within memory by implementing computational elements inside the memory block. The model operates in two modes: **LiM Mode** and **Memory Mode**.

In **LiM Mode**, the architecture executes a specific set of instructions stored in the **Instruction Memory**, while in **Memory Mode**, it performs traditional **LOAD** and **STORE** operations. The procedure for switching modes is summarized as:

- From **LiM Mode** to **Memory Mode** $\longrightarrow$ Write a "1" in the **START** Register and a "0" in the **READY** Register;

- From **Memory Mode** to **LiM Mode** $\longrightarrow$ Write a "0" in the **START** Register and a "1" in the **READY** Register.

The architecture combines **asynchronous** and **synchronous** components.

The synchronous components update their signals on the rising edge of the **Clock Signal**, and the **Reset** is Active High. The highest level of the architecture contains only the **Datapath** and the **Control Unit**. The **Datapath** fetches and executes instructions, while the **Control Unit** provides control signals to the Datapath through a complex **Finite State Machine**. Both blocks can be connected to external units through an interface. This **Layout** can be observed in Figure 6.2.

For the full diagram, see Appendix A. It is important to mention that all blocks communicate **bidirectionally** with each other.

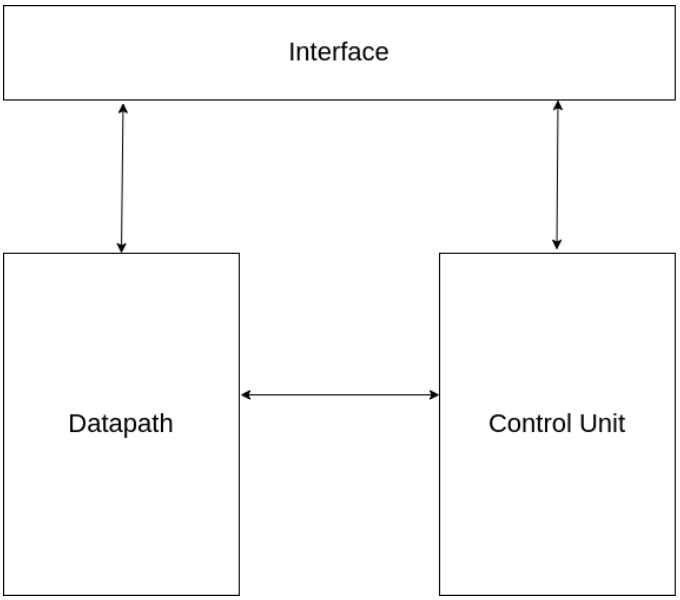


Figure 6.2. LiMPire Top Level

## 6.3 Level 2: Datapath

The **Datapath** is the execution block of LiMPire, where computation takes place. The LiMPire Datapath, shown in Figure 6.3, consists of the following components:

- **Memory Interface**;
- **Instruction Memory** (including **Program Counter Register**);
- **Instruction Decode Unit**;
- **Bus**;
- **Six Memory Banks**.

These components are connected as shown in Figure 6.3.
The Full Diagram of the Datapath can be observed in Appendix A.1.
To increase throughput, the following five pipeline stages were implemented:

- **Instruction Fetch (IF)**;
- **Instruction Decode (ID)**;

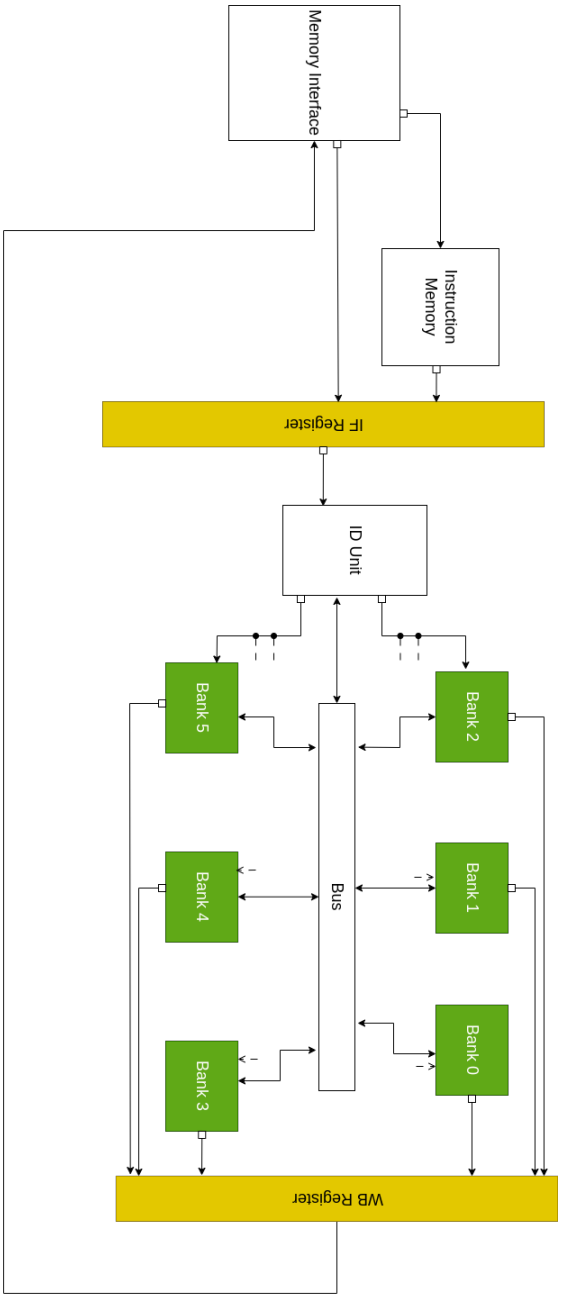


Figure 6.3. LiMPire Datapath

- **Execute (EX)**;
- **Memory (MEM)**;
- **Write Back (WB)**.

The **Pipeline Stages** are implemented using dedicated registers, which are categorized as either **Internal** or **External** with respect to each memory bank. The **Internal Pipeline Registers** are located inside the **Memory Bank Block**, while the **External Pipeline Registers** can be observed at the Datapath Level. The former category includes the **EX** and **MEM Registers**, while the latter includes the **IF** and **WB Registers**.

Although this version of the architecture features six memory banks, the design is fully customizable, allowing for an increase or decrease in the number of banks. Each **Memory Bank** contains a memory array of 32 words, each 32 bits wide, composing the overall **Data Memory**. In total, the Data Memory occupies 192 rows in the memory space. Despite the physical division among memory arrays due to the bank layout, the entire **Memory Space** is seen as unique and contiguous, featuring both **Data Memory** and **Instruction Memory**.

Depending on the working mode (**LiM Mode** or **Memory Mode**), the destination address provided to the memory banks is computed in three different ways:

- **LiM Mode & Parallel Computation Instruction** $\longrightarrow$ The destination register is equal to the one provided by the **ID Unit**;
- **Memory Mode** or any **LiM Instruction in LiM Mode** except **Parallel Computation** $\longrightarrow$ The destination address for each bank can be computed as:

$$\textbf{DEST\_ADDR = ID\_ADDR - (32 * BANK\_NUMBER)}$$

Where **ID_ADDR** is the destination address provided by the **DEST** Port of the **ID Unit**, **DEST_ADDR** is the computed destination address, and **BANK_NUMBER** is the bank number (e.g., for Bank 0, **BANK_NUMBER** is 0).

The overall workflow of the **Datapath** is:

1. In **Memory Mode**, the CPU sends **address**, **data in**, **dvalid**, **req**, and **write enable**. All this information passes through the **Memory Interface** and is attached to the **IF Register**. On the next clock cycle, this information is decoded, and the correct memory bank receives the information. If the current instruction is a **Load**, the bank pulls out the requested data, which is attached to the **WB Register**. After one clock cycle, it is selected by a multiplexer and forwarded back to the

CPU through the **Memory Interface**. In the case of a **Store**, data is written inside the bank.

2. In **LiM Mode**, the memory computation is triggered by writing into the **Status Registers** (START and READY). The maximum iteration counter of the **Instruction Memory** is written with the maximum iteration number, and the Instruction Memory starts fetching instructions by updating the **Program Counter**. The **Instruction Word** is attached to the **IF Register**, and after one clock cycle, it is sent to the **ID Unit**, responsible for decoding the information and sending the appropriate signals to the addressed bank. In the case of a parallel computation instruction, it communicates with all the banks. If the operands are external, it directs the banks to fetch the operands and provide them to the destination bank through the bus. Once all operands are in the same bank, the computation occurs, and the final result is written inside the addressed register. The process is repeated for each instruction stored inside the Instruction Memory until the Program Counter hits the **maximum iteration counter** value. If the current instruction is a memory instruction, the **Memory Mode** procedure is followed.

### 6.3.1 Level 2: Memory Interface

The **Memory Interface** serves as the link between the **Accelerator** and the external environment, such as the **CPU**. It operates as an **asynchronous interface**, directing signals to the appropriate blocks within the Datapath, as well as to other peripherals (such as Peripheral Registers) and the CPU. Input signals are transmitted from the **CPU** via a series of interface layers and the **Control Unit**. This component is purely **combinational** and includes the following ports, as shown in Figure 6.4:

- `CLK`: Standard clock signal.
- `RST`: Active High Reset.
- `RST_COUNTER`: This signal is used to reset the internal counter of the **Instruction Memory**.
- `DATA_IN`: Data coming from the **CPU** to be written to a memory array. It is connected to `DATA_IN_OUT`. The width is 32 bits.

- `DATA_WB_IN`: Data coming from the **WB Register**. It is directly connected to `DATA_OUT` and is used for **Load Operations**. The width is 32 bits.
- `MAX_CNT_IN`: This signal is connected to `MAX_CNT_OUT` and represents the maximum number of instructions to execute in **LiM Mode**. The width is 32 bits.
- `DONE_IN`: This signal is connected to `DONE_OUT`. It communicates that there are no more instructions to execute in **LiM Mode**.
- `ADDR`: This signal provides the address for any **Load** or **Store operation**. It is connected to `ADDR_OUT`. The width is 8 bits.
- `START_IN`: This signal is connected to `START_OUT` and is used for switching to **LiM Mode**.
- `WR_EN_IN`: This signal is used only to enable the write port of the memory array during **Store operations**.
- `READY_IN`: This signal is connected to `READY_OUT` and allows switching to **Memory Mode**.
- `DATA_OUT`: Used for **Load Operations**. The width is 32 bits.
- `DATA_IN_OUT`: Used for **Store Operations**. The width is 32 bits.
- `ADDR_OUT`: For **Memory Operations**. The width is 8 bits.
- `START_OUT`: This signal is used for mode switching.
- `WR_EN_OUT`: Used only for **Store Operations**.
- `READY_OUT`: For mode switching.
- `MAX_CNT_OUT`: Contains the maximum working address of the **Instruction Memory**. It defines how many instructions will be executed during the entire **LiM Mode** session. The width is 32 bits.
- `RST_COUNTER_OUT`: For resetting the counter.
- `DONE_OUT`: Used to signal the end of **LiM Mode**.

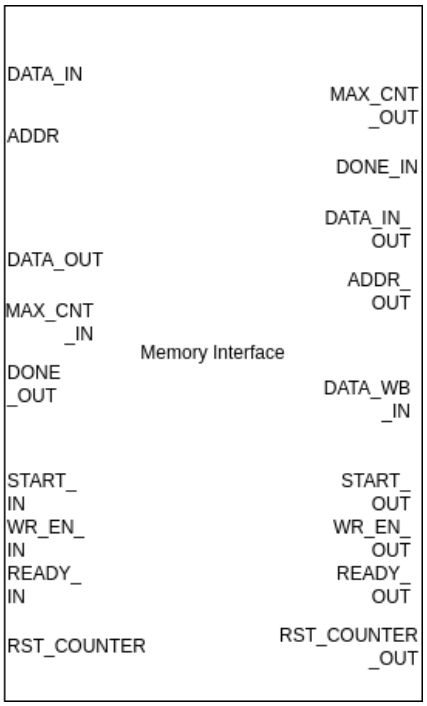


Figure 6.4. LiMPire Memory Interface

### 6.3.2 Level 2: Instruction Memory

The **Instruction Memory** stores all the **LiM Instructions** to be executed in **LiM Mode**, and is unused in **Memory Mode**. It has 1024 memory rows, each 32 bits wide. The total **dimension** is 4 KB (precisely 4096 Bytes).

Prior to the integration of the **LiM** into the **X-Heep Microcontroller** (see Chapter 8), this block consisted of an array of 512 signals, each 43 bits wide (a total of 2752 bytes). These signals were written using the `readmemb` function from the **SystemVerilog Libraries**. This function imports the contents of a text file into a specific array of logic. This modification was required because the **CPU** in the **X-Heep** writes instructions at startup, and its bus is 32 bits wide, which is standard for **RISC-V**.

To resolve the issue, two solutions were proposed: either **overhaul** the entire architecture of the **X-Heep** (a lengthy and complex process) or **divide** the instruction word into two blocks. Ultimately, the latter option was chosen. However, this configuration results in a memory **overhead** of 1344 bytes (approximately 1 KB) due to the unused bits.

This block consists of the following signals:

- `CLK`: Clock signal from the outside.
- `RST`: Active High Reset. The content of this port results from an **XOR** between the system reset and the `RST_COUNTER_OUT` from the **Memory Interface**.
- `LIM_MEM`: Selects the working mode ("1" for **LiM Mode**, "0" for **Memory Mode**).
- `INCREMENT_CNT`: Instructs the **Instruction Memory** to increment its internal address counter to update the **Program Counter** and proceed to the next instruction. Not used in **Memory Mode**.
- `MAX_CNT`: Sets the maximum number of instructions to execute in **LiM Mode**. Written at startup by the **CPU**. Not used in **Memory Mode**. The width is 32 bits.
- `REQ_I`: Compliant with the Open Bus Interface and used to request writing part of an instruction.
- `WE_I`: Compliant with the Open Bus Interface and used for writing instructions.
- `ADDR_I`: Address provided by the **CPU** for writing an instruction. Width is 10 bits.
- `WDATA_I`: Provides the instruction to store in memory. Width is 32 bits.
- `SET_RETENTIVE_I`: Currently unused, but can be connected if the internal 1024x32 bits array is replaced with an internal **SRAM**.
- `RDATA_O`: Currently unused, but can be connected to an internal **SRAM**. The width is 32 bits.
- `INSTR`: This 43-bit signal is the **fetched instruction**.
- `DONE`: Signals the CPU that the LiM Computation is **finished**.

All these ports can be observed in Figure 6.5.

The Memory operates in two main modes: **Memory Mode** and **LiM Mode**. In **Memory** Mode, the **CPU** writes instructions. According to the OBI Standard, writing to the memory is only allowed if:

- `REQ_I` is set to "1".

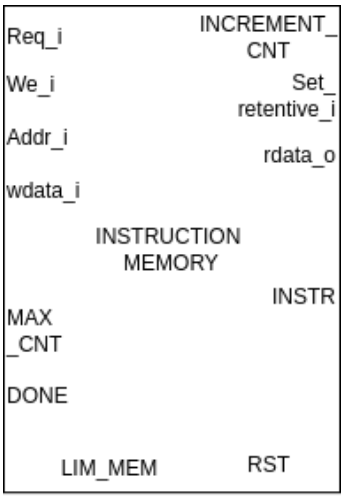


Figure 6.5. Instruction Memory Overview

- `WE_I` is set to "1".
- `ADDR_I` contains a valid address.
- `WDATA_I` is valid.

When these conditions are met, the **Instruction Memory** can be written.
In **LiM Mode**, it can only be read. To fetch the next instruction, the Instruction Memory features an internal **row counter**. When `INCREMENT_CNT` equals "1", the row counter increments by 1, moving to the next instruction and updating the **Program Counter**.

Each memory row is 32 bits wide, so two rows are fetched simultaneously using the following formulas:

$$\textbf{ADDRESS\_LSB} = \textbf{INT\_COUNTER} * \textbf{2}$$

$$\textbf{ADDRESS\_MSB} = (\textbf{INT\_COUNTER} * \textbf{2}) + \textbf{1}$$

Here, `ADDRESS_LSB` refers to the least significant 32 bits of the instruction word, and `ADDRESS_MSB` refers to the most significant 32 bits (later truncated by 21 bits), where `INT_COUNTER` is the internal row counter.

As shown in Figure 6.6, **even** addresses refer to the least significant 32 bits, while **odd** addresses refer to the most significant bits. Unfortunately, the 21 truncated bits (shown in red) are not used and are discarded.

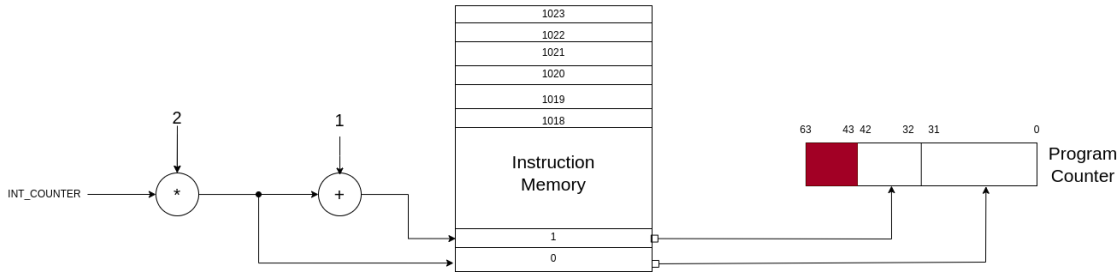


Figure 6.6. Instruction Memory Fetch Overview

The Instruction Memory continues **updating** the Program Counter until the internal row counter matches the maximum counter value. At that point, the unit writes "1" into the **Done** Register and informs the CPU that the computation is finished.

Within the architecture's memory space, which spans 4864 bytes, the Instruction Memory occupies the **address range** from 192 to 1215.

The **Instruction Word** is connected to all other blocks of the datapath through the **IF Register** and the **Control Unit**, containing all information about the instruction to execute.

For the synthesis, this block has been modified (see Chapter 10).

### 6.3.3 Level 2: Instruction Decode Unit

The **Instruction Decode Unit** is the main unit of the **Instruction Decode Pipeline Stage**. This entity is synchronous with respect to the input `CLK` signal. The **ID Unit** has different behaviors depending on the working mode:

- In **Memory Mode**, it receives `DATA`, `ADDRESS`, and `WR_EN`. `DATA` and `WR_EN` are considered valid only for a **Store**, while the `ADDRESS` is always used.

  When performing a **Store**, the `DATA` is forwarded by the ID Unit to the multiplexer, which inputs data to the bus.

- In **LiM Mode**, it decodes the instruction in the following steps:

  1. **Identify** the instruction **type**: 1 Operand, 2 Operands, 3 Operands, or Address Range.

2. If the instruction uses more than two operands, it **computes** the number of **iterations** needed to fully execute it.
3. It **decodes** the **addresses** for operands and destination and forwards them to the memory banks.
4. The ID Unit **identifies** the **operation** and sends the correct operation code to the banks.

When the **LiM instruction** involves at least three operands, the ID Unit sends the correct operand addresses to the banks through an **embedded iteration counter**, ensuring the architecture behaves accordingly.

In both modes, the received address is within a range between 0 and 191, but each memory bank has an address range between 0 and 31. To provide the correct address to each bank, the ID Unit subtracts the **offset** depending on the following cases:

- **0 <= Address <= 31** $\longrightarrow$ The final address is the same as the input address. **Bank 0** is activated.
- **32 <= Address <= 63** $\longrightarrow$ The final address is given by subtracting 32 from the input address and **Bank 1** is activated.
- **64 <= Address <= 95** $\longrightarrow$ The final address is given by subtracting 64 from the input address and **Bank 2** is activated.
- **96 <= Address <= 127** $\longrightarrow$ The final address is given by subtracting 96 from the input address and **Bank 3** is activated.
- **128 <= Address <= 159** $\longrightarrow$ The final address is given by subtracting 128 from the input address and **Bank 4** is activated.
- **160 <= Address <= 191** $\longrightarrow$ The final address is given by subtracting 160 from the input address and **Bank 5** is activated.

If the instruction requires **parallel** operations on all banks (and the operand addresses are less than 32), the ID Unit provides the **same addresses** to all the banks without subtracting any offset.

In summary, the ID Unit performs two main tasks: **decoding** and **in-loco control**. This reduces the workload of the Control Unit, which only handles a smaller set of signals.

The ID Unit has the following ports:

- `CLK`: This is the clock signal coming from the outside.
- `RST`: Reset signal, active high.
- `MODE`: This signal is connected to the memory interface and identifies the working mode. A "1" indicates **LiM Mode**, while a "0" indicates **Memory Mode**.
- `INSTRUCTION`: This signal represents the full 43-bit instruction word pointed by the program counter. It is decoded inside this unit.
- `MEM_ADDR`: This 8-bit signal is used only in **Memory Mode** and serves as the load or store address.
- `WR_EN_IN`: This signal is used only in **Memory Mode**, specifically for store operations.
- `DATA_IN_INT`: This 32-bit signal is the input data from the CPU, used only for store operations in **Memory Mode**.
- `COUNT_IN`: This 32-bit signal keeps track of the iteration number when working with multiple operands. It is managed by the **Control Unit**. Initially managed by the **ID Unit**, it was later turned into a port so the **Control Unit** could use it for state switching.
- `ADDR0`: This is the 8-bit address of Operand 0 in **LiM Mode** or the address for both load and store in **Memory Mode**.
- `ADDR1`: This is the 8-bit address of Operand 1 in **LiM Mode**. It is unused in **Memory Mode**.
- `FUNCT`: This 7-bit signal identifies the instruction type (depending on how many operands it operates on) and is crucial for the **ALU**'s operation.
- `DEST`: This is the 8-bit destination address.
- `DATA_OUT_BUS`: This 32-bit signal carries data provided by the CPU for performing the store operation.

The ports layout can be observed in Figure 6.7.

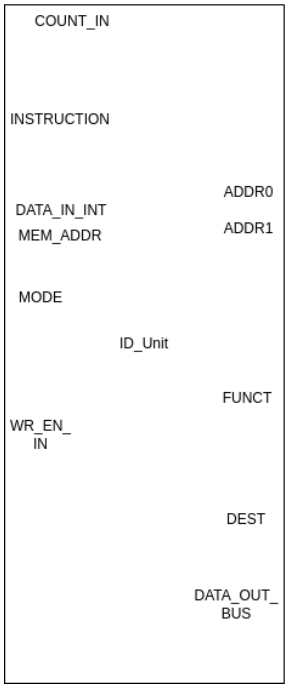


Figure 6.7. Instruction Decode Unit Layout

### 6.3.4 Level 2: Bus

The architecture presents six independent **Memory Banks** interconnected via a **Bus**. The **Bus** connects all the memory banks and the **ID Unit**, and it is used for load and store operations (in **Memory Mode**) as well as exchanging data among the banks (in **LiM Mode**). The **Bus** is synchronous with respect to the `CLK` signal and has the following ports:

- `CLK`: This is the clock signal coming from the outside.
- `RST`: Reset signal, active high.
- `BUS_ENABLE`: This signal enables the bus. When not in use, it is cleared.
- `DATA_IN_BUS`: This 32-bit signal receives input data from an 8-to-1 multiplexer, which provides data from the banks and the **ID Unit**. The inputs of the multiplexer are:

  0. **Bank 0**
  1. **Bank 1**

2. **Bank 2**
3. **Bank 3**
4. **Bank 4**
5. **Bank 5**
6. **ID Unit**

- `ADDRESS`: This 8-bit signal is used by the bus to send data to the addressed bank.
- `OUTPUT_ENABLE`: This signal allows to send data from the Bus to the banks. It is Active High.
- `DATA_OUT_BUS`: This matrix consists of 6 rows of 32-bit words. Depending on the selected bank, the corresponding row is filled with the input **data**, while the others remain zero.

The Bus and its input multiplexer present the following layout:

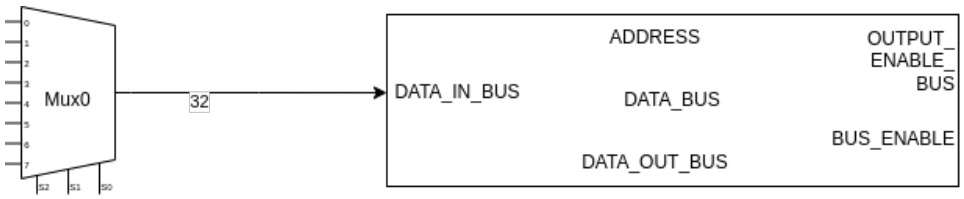


Figure 6.8. Bus with Input Multiplexer

## 6.4 Level 3: Memory Bank(s)

It was already mentioned in Section 6.3 that the High Level Datapath is composed of **Instruction Memory**, **Memory Interface**, **Instruction Decode Unit**, **Bus**, and six **Memory Banks**. Memory Banks are the **computational core** of LiMPire, as they contain all the computational and storage blocks. Moreover, they also act as dispatchers of operands, since some instructions require operands located in different banks. Thus, they are responsible for supplying the destination bank with the necessary operands.

All banks share the same structure, using the same model description, and they can operate in parallel if the current instruction doesn't need to fetch data from multiple banks. Furthermore, banks are interconnected via the

**Bus** for data exchange. The **Bank Model** includes the following components (as shown in Figure 6.9):

- **Bank Interface**
- **Input Register File**
- **Temporary Register File**
- **Memory Array**
- **Arithmetic Logic Unit (ALU)**

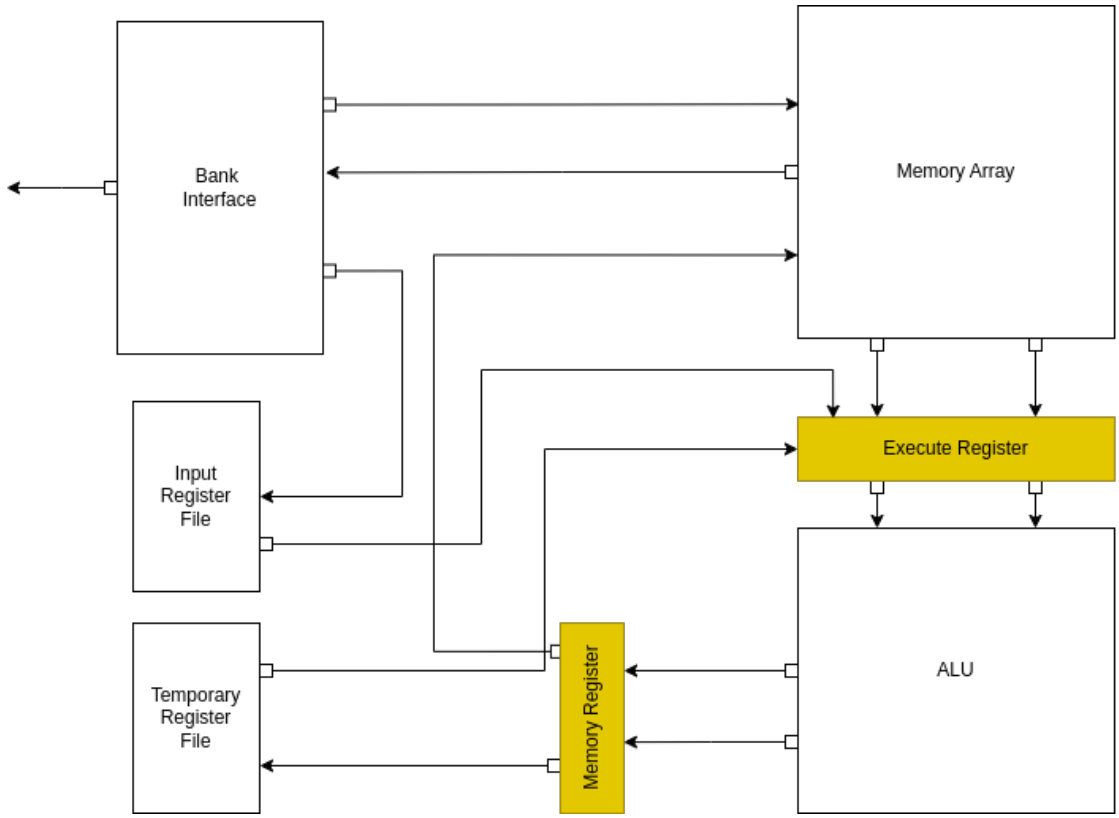


Figure 6.9. Memory Bank Layout

The Full Diagram of the Banks internal Datapath can be observed in Appendix A.1.

The initial project version included two additional blocks: one for computing the **Error Correction Code (ECC)** for data encryption and another

for data decryption using the **Hamming Code** method. These blocks were later removed due to their complexity and the need to add another **Pipeline Stage** for performance improvements. Anyway, it represents a valid challenge for Future Works (for more information, see Chapter 12). Furthermore, they would have complicated testing, as the stored data would have included **additional bits** for ECC, making it harder to read.
For further information on this method and a future implementation, see Chapter 12.
Earlier subsections mentioned **Pipeline Registers** inside Memory Banks. In Figure 6.9, you can see the two registers: the **Execute Register** and the **Memory Register**. The overall **workflow** of the Memory Bank can be summarized as follows:

1. **In Memory Mode:** If the current instruction is a **Store**, the data from the **Bus** is forwarded to the **Bank Interface** and written into the **Memory Array**. Alternatively, the Memory Array fetches the requested data, attaches it to the **MEM Register**, and, after one clock cycle, forwards it to the **Bank Interface**, which connects it to the **WB Register**.

2. **In LiM Mode:** Consider only the destination bank, as operand fetch and computation are controlled by the destination bank. If all operands and the destination bank match, the operands are fetched from the **Memory Array** and attached to the **EX Register**. After one clock cycle, the correct input is selected from the two multiplexers to feed the **ALU**. Once the ALU computes the result, it stores the result either in the **Temporary Register** or back in the **Memory Array**. If operands are external, they are fetched from other banks, passed through their **Bank Interface**, and attached to the **Input Register File** of the destination bank.

In Figure 6.9, the **ALU** can receive two operands at a time from the **Memory Array**, **Input Register File**, or **Temporary Register File**, with selection controlled by two multiplexers. Additionally, the Memory Array can be written to by the Memory Interface (during a **Store**) or the ALU (at the end of any **LiM Instruction**). The input data is selected by another multiplexer, similar to the strategy used for selecting data from the two read ports of the array during **Load** or **LiM Operations**. A complete diagram of the **Bank Layout** is provided in Appendix A.1, while individual components are described in the following subsections.

### 6.4.1 Level 3: Bank Interface

The **Bank Interface** provides data in four directions:

- **From Another Bank to the Current Bank** ⟶ Data passes through the input port and is written to the **Input Register File** (during **LiM Operations**).
- **From the Current Bank to Another Bank** ⟶ Data is fetched from the **Memory Array** and sent out via the output port of the **Bank Interface** (during **LiM Operations**).
- **From CPU to the Current Bank** ⟶ Data from the input port is written into the **Memory Array** (during **Store Operations**).
- **From the Current Bank to CPU** ⟶ Data is fetched from the **Memory Array** and sent to the CPU via the **WB Register** (during **Load Operations**).

The **Bank Interface** features the following ports (as shown in Figure 6.10):

- `CLK`: Clock signal from the outside.
- `RST`: Reset signal, active high.
- `DATA_IN`: Input 32-bit data from the **Bus**.
- `DATA_DEC`: 32-bit decrypted data from the **Memory Array** (from the original ECC design stage).
- `DATA_IN_SEL`: Select signal to store input data in the **Input Register File** or the **Memory Array**.
- `DATA_OUT_SEL`: Select signal to send data from the **Memory Array** to the **WB Register** or the **Bus**.
- `DATA_BUS_OUT`: 32-bit output data to the **Bus** (used in **LiM Mode**).
- `DATA_OUT_F`: 32-bit output data for memory writes.
- `DATA_OUT_INROWS`: 32-bit input data forwarded from the **Bus** to the **Input Register File** (used in **LiM Mode**).
- `DATA_BANK_OUT`: 32-bit output data from the **Memory Array** to the **WB Register**.

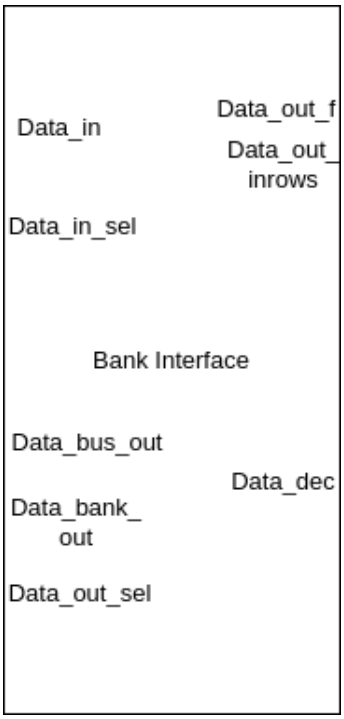


Figure 6.10. Bank Interface Layout

### 6.4.2 Level 3: Input Register File

Each Memory Bank has two distinct register files: the **Input Register File** and the **Temporary Register File**. The Input Register File stores data from other banks, ensuring all operands are in the same bank without affecting the Memory Array. This register has three 32-bit rows (corresponding to **12 Bytes** per RF), where each row corresponds to an operand number. It can read from two rows simultaneously, which is useful for fetching two external operands.

The component is synchronous with the clock's rising edge and has the following ports (as shown in Figure 6.11):

- `CLK`: Clock signal from the outside.
- `RST`: Reset signal, active high.
- `WR_ADDR`: 2-bit signal for the write address.
- `RD0_ADDR`: 2-bit signal for the read address for port 0.
- `RD1_ADDR`: 2-bit signal for the read address for port 1.

- `WR_EN`: 2-bit signal for enabling write operations (active high).
- `RD0_EN`: 2-bit signal for enabling reads from port 0 (active high).
- `RD1_EN`: 2-bit signal for enabling reads from port 1 (active high).
- `DATA_IN`: 32-bit input operand from another bank, provided by the **Bank Interface**.
- `DATA_OUT0`: 32-bit output operand from port 0.
- `DATA_OUT1`: 32-bit output operand from port 1.

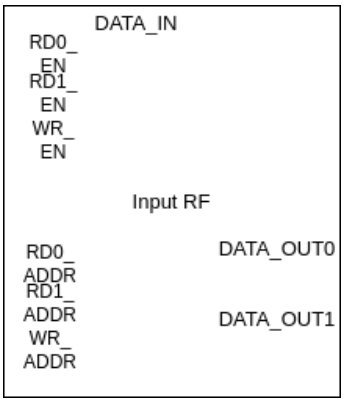


Figure 6.11. Input Register File Layout

### 6.4.3 Level 3: Temporary Register File

**Temporary Register Files** are crucial for storing temporary data during multi-operand operations, including both **Range** and **Three-Operand** operations. This **Register File** is composed of three rows of 32-bit words. It can read two rows simultaneously via two read ports, while it writes to one row at a time.

Similar to the Input Register File, the content of each row is **deterministic**, simplifying the job of the Control Unit when providing the correct row address for operand fetches or writes. The content of each row depends on the number of operands in the instruction. Given $OPx$ as the x-th operand and *op* as an operation, the row content is as follows:

- **Three Operands**:
  Row 0 $\longrightarrow$ OP0 *op* OP1

- **Four-Operands Range**:
  0. Row 0 $\longrightarrow$ OP0 *op* OP3
  1. Row 1 $\longrightarrow$ OP1 *op* OP2
- **Five-Operands Range**:
  0. Row 0 $\longrightarrow$ OP0 *op* OP4
  1. Row 1 $\longrightarrow$ OP1 *op* OP3
  2. Row 2 $\longrightarrow$ Row 0 *op* Row 1
  3. Row 2 $\longrightarrow$ OP2 *op* Row 2
- **Six-Operands Range**:
  0. Row 0 $\longrightarrow$ OP0 *op* OP5
  1. Row 1 $\longrightarrow$ OP1 *op* OP4
  2. Row 2 $\longrightarrow$ Row 0 *op* Row 1
  3. Row 0 $\longrightarrow$ OP2 *op* OP3
  4. Row 2 $\longrightarrow$ Row 0 *op* Row 2
- **Seven-Operands Range**:
  0. Row 0 $\longrightarrow$ OP0 *op* OP6
  1. Row 1 $\longrightarrow$ OP1 *op* OP5
  2. Row 2 $\longrightarrow$ Row 0 *op* Row 1
  3. Row 0 $\longrightarrow$ OP2 *op* OP4
  4. Row 2 $\longrightarrow$ Row 0 *op* Row 2
  5. Row 2 $\longrightarrow$ OP3 *op* Row 2
- **Eight-Operands Range**:
  0. Row 0 $\longrightarrow$ OP0 *op* OP7
  1. Row 1 $\longrightarrow$ OP1 *op* OP6
  2. Row 2 $\longrightarrow$ Row 0 *op* Row 1
  3. Row 0 $\longrightarrow$ OP2 *op* OP5
  4. Row 2 $\longrightarrow$ Row 0 *op* Row 2
  5. Row 0 $\longrightarrow$ OP3 *op* OP4

6. Row 2 $\longrightarrow$ Row 0 *op* Row 2

This **schedule** of multi-operand operations simplifies the Control Unit and the Instruction Decode Unit by tracking iterations and providing the correct signals accordingly.

This unit features the following ports, as shown in Figure 6.12:

- `CLK`: This is the Clock Signal coming from the outside.
- `RST`: Reset Signal, active High.
- `WR_ADDR`: This 2-bit signal is controlled by the **Control Unit** and corresponds to the Write Address.
- `RD0_ADDR`: This 2-bit signal is controlled by the **Control Unit** and corresponds to the Read Address for Port 0.
- `RD1_ADDR`: This 2-bit signal is controlled by the **Control Unit** and corresponds to the Read Address for Port 1.
- `WR_EN`: This 2-bit signal is controlled by the **Control Unit** and enables writing to the **Register File**. It is active High.
- `RD0_EN`: This 2-bit signal is controlled by the **Control Unit** and enables reading from Port 0 of the **Register File**. It is active High.
- `RD1_EN`: This 2-bit signal is controlled by the **Control Unit** and enables reading from Port 1 of the **Register File**. It is active High.
- `DATA_IN`: This 32-bit signal contains the input operand from the **ALU**.
- `DATA_OUT0`: This 32-bit signal corresponds to the operand read from the **Register File** to Port 0.
- `DATA_OUT1`: This 32-bit signal corresponds to the operand read from the **Register File** to Port 1.

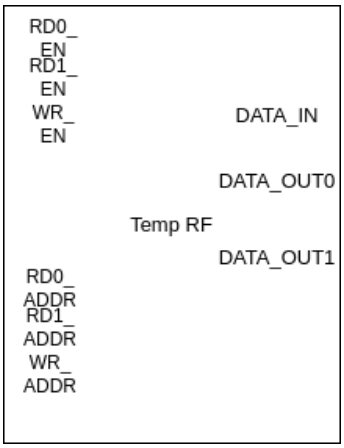


Figure 6.12. Temporary Register File Layout

### 6.4.4 Level 3: Memory Array

Each **Bank** has three components whose function is to store data: **Input RF**, **Temporary RF**, and **Memory Array**. The first two components keep **temporary** data, while the Memory Array keeps only the "**definitive**" data. Indeed, the Memory Array content is updated only with the final **result** of the operation in **LiM Mode** and with the input data for a **Store** in **Memory Mode**.

Each Bank includes a **Memory Array** composed of 32 rows of 32-bit words (128 Bytes per each, 768 Bytes in total). Every single Memory Array presents 8-bit addresses within the 0-31 range, while the CPU will see only a whole memory space. The **Address Translation** is performed with the following formula:

**ADDR_FINAL = INPUT_ADDR - (BANK_NUMBER * 32)**

Where **ADDR_FINAL** is the address of the memory row inside the addresses bank, **INPUT_ADDR** is the address provided by the CPU, and **BANK_NUMBER** is the Bank Number (e.g., Bank 0 has number 0, Bank 1 has number 1, etc.).

If the resulting address is within the desired range, it is possible to perform a **Write** by asserting the `WR_EN` signal, and a **Read** by enabling the `RD0_EN` or `RD1_EN` signals for the desired reading port. Furthermore, this component supports **Partial Word** writing.

This component presents the possibility to simultaneously read two memory rows thanks to the two read ports, while it can only write one row at a time. This model presents the following ports:

- `CLK`: This is the Clock Signal coming from the outside.
- `RST`: Reset Signal, active High.
- `DATA_IN`: This 32-bit signal is the input data coming from the **Memory Interface** or **ALU** (depending on the input selected by the **Multiplexer**).
- `WR_ENC`: If set, this signal indicates that the input data will only be partially written; otherwise, the input data will be treated as a whole.
- `HIGH_LOW`: If `WR_ENC` is set, there are two possibilities: if this signal is equal to "1", the input data will overwrite the 16 MSB of the information currently stored in the write address; otherwise, it will overwrite the 16 LSB. This signal is very similar to the "Byte Enable" of common SRAMs.
- `RD0_EN`: Active High, it enables Read Port 0.
- `RD1_EN`: Active High, it enables reading from Port 1.
- `WR_EN`: Active High, it enables writing.
- `WR_ADDR`: This 8-bit signal is the writing address for both **LiM** and **Store** operations.
- `RD0_ADDR`: This 8-bit signal is the read address for both **LiM** and **Load** operations on Port 0.
- `RD1_ADDR`: This 8-bit signal is the read address for both **LiM** and **Load** operations on Port 1.
- `RD0_ALU`: If set, the data read from Port 0 has to be sent to the **EX Register**, since it's going to be used by the **ALU** as Operand 0. If clear, the current operation is a **Load** or a **LiM** operation where at least one operand is located in a different bank than the destination.
- `RD1_ALU`: If set, the data read from Port 1 has to be sent to the **EX Register**, since it's going to be used by the **ALU** as Operand 1. If clear, the current operation is a **Load** or a **LiM** operation where at least one operand is located in a different bank than the destination.

- `RD0_OUT`: This 32-bit output port provides the data from Port 0 of the **Memory Array** to the **Memory Interface**, which is forwarded to the **WB Register** or to another **Bank** through the **Bus**.

- `RD1_OUT`: This 32-bit output port provides the data from Port 1 of the **Memory Array** to the **Memory Interface**, which is forwarded to the **WB Register** or to another **Bank** through the **Bus**.

- `OP0_DEC`: This 32-bit signal is used for sending data to the **ALU** through the **EX Register**. The name refers to the old implementation, where data had to be decrypted before being provided to the **ALU**.

- `OP1_DEC`: This 32-bit signal is used for sending data to the **ALU** through the **EX Register**. The name refers to the old implementation, where data had to be decrypted before being provided to the **ALU**.

All these ports can be observed in the following figure:

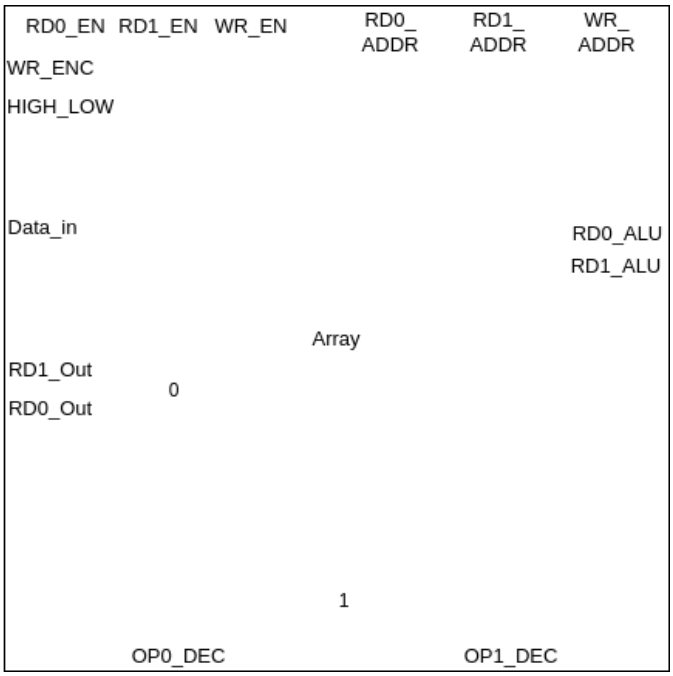


Figure 6.13. Memory Array Layout

For synthesis, the **Memory Array** was modified by replacing the memory cells with a real SRAM. For further information, see Chapter 10.

### 6.4.5 Level 3: Arithmetic Logic Unit

The **Arithmetic Logic Unit** is the unit in charge of performing **computations** given the two inputs, the operation code, and the status signals.

#### Phase 1: Preliminary Checks on Operands

The workflow of the unit starts with the **preliminary checks** on the operation type:

1. **Partial Word** Operations;
2. **Full Word** Operations.

When performing **Partial Word** Operations, it is necessary to extract the correct set of bits from each operand, depending on the `OP_STATE` signal. It is possible to distinguish three main cases:

1. Last Iteration of **Three Operands Instruction**: In this case, the last operands are aligned to the set of bits to replace in the destination, such that they can be easily attached to the **Memory Array**. For example, if the destination operand's LSBs are supposed to be replaced, all the operands will be shifted 16 bits to the right. This was originally supposed to prepare the result to be written to the destination, but in the end, this function was implemented inside the Memory Array logic, since it would have required one more operand fetch, leading to slower performance.
2. Middle Iteration of **Range** or **Three Operands** or **Standard Two Operands Instruction**: This case covers the standard operations alignment, so operands are prepared depending on the set of bits of the destination to overwrite. In the last version of the model, this case is the same as the previous one, but it was originally meant only for the standard operands. It was kept as a legacy for further optimizations (see Chapter 12).
3. One Operand Instruction: This case is exactly the same as the others. It was originally used for preparing the only operand for **One Operand Instructions**, depending on the destination bits set.

For **Full Word** Operations, this procedure is not necessary, since the final result will fully replace the content of the destination address.

### Phase 2: Computation

The second step of the flow is to **compute the result** given the operands. The **ALU** can perform the following operations:

- **NOP**: Coded as 0000;
- **AND**: Coded as 0001;
- **NAND**: Coded as 0010;
- **OR**: Coded as 0011;
- **NOR**: Coded as 0100;
- **XOR**: Coded as 0101;
- **XNOR**: Coded as 0110;
- **ADD**: Coded as 1001;
- **SUB**: Coded as 1010;
- **MUL**: Coded as 1011;
- **NOT**: Coded as 1100.

Originally, this component was designed to include **MAJOR** and **MINOR** operations (coded respectively as 0111 and 1000), but they were later **discarded** since these instructions need to simultaneously work with three operands, while the ALU supports only two. However, they can be implemented in software by means of simpler **LiMPire Assembly Instructions**.

It's worth mentioning that "composite" **Range** and **Three Operands Instructions**, made of a NOT and another Boolean operation (such as NAND, NOR, etc.), are treated as the non-negated operations (AND, OR, etc.) for all iterations but the last one, which is treated as the negated one (NAND, NOR, etc.). The output of the ALU is sent to the **Temporary Register File** for intermediate results, while the final results are sent to the **Memory Array** for writing. The destination is specified by the signal `ALU_OUT_SEL`.

The ports list of the **ALU** is:

- `CLK`: This is the Clock Signal coming from the outside.
- `RST`: Reset Signal, active High.

- `ALU_OUT_SEL`: This signal is used for **selecting** the desired **output port**, depending on the iteration number.

- `OP0`: This 32-bit signal is the Operand 0 of the **ALU**. Each operand can be provided by: **Input RF**, **Temporary RF**, or **Memory Array**.

- `OP1`: This 32-bit signal is the Operand 1 of the **ALU**. It's not used for the `NOT`.

- `FUNC`: This 7-bit signal is the **FUNC** Field of the Instruction Word, therefore it specifies the **Instruction Type** along with all the necessary information about the operands.

- `OPCODE`: This 4-bit signal specifies the **operation** to perform (e.g., AND, OR, etc.). If the current instruction is not composite, it matches with the OPCODE Field of the Instruction; otherwise, it is internally managed by the Control Unit.

- `OP_STATE`: This 2-bit signal was originally used to **prepare operands**. It could be simplified to one bit: if set, there is a Partial Word Instruction; otherwise, it is a Full Word. Additionally, it can be further optimized by checking bit 13 of the Instruction Word, which specifies if the instruction is Partial or Full Word.

- `ALU_READY`: This signal was originally used to communicate to the Control Unit the **end of the computation**. It was supposed to be set when the **ALU** finished the computation and cleared while it was busy.

- `ALU_OUT_MEM`: This 32-bit signal is the output port of the **ALU** connected to the Memory Register. After one clock cycle, this information will reach the input port of the **Memory Array**.

- `ALU_OUT_TEMP`: This 32-bit signal is connected to the Memory Register, and its purpose is to provide the temporary result of an operation to the **Temporary Register**.

The **ALU** Ports Layout is:

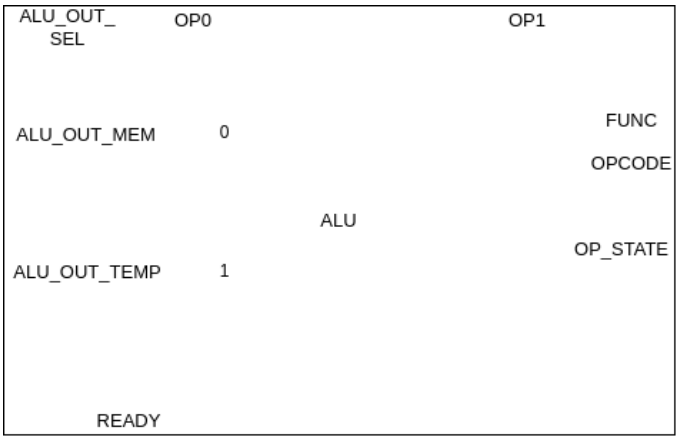


Figure 6.14. ALU Layout

## 6.5 Level 2: Control Unit

The **Control Unit** is the main **Control Block** of the LiMpire Architecture. There are three approaches to implement a **Control Unit**:

1. **Finite State Machine**;
2. **Hardwired**;
3. **Micro-programmed**.

The final choice for the **Control Unit** is the **FSM** because it is easy to read, modify, and debug.

The **Control Unit** is directly connected to the **Datapath**, so it is located at **Level 2** of the Architecture. This block has access to all the components described in this chapter, thus synchronizing and regulating their behavior. Figure 6.15 shows the overall control flow of the **CU**.

The **Architecture** starts from the **IDLE** state, which is reached through **Reset** or at the **end** of each **Instruction**. During this state, the architecture waits for a stimulus for a state transition, depending on the following combinations of signals:

1. `READY` = 1, `WR_EN` = 1, and `DVALID` = 1 $\longrightarrow$ The **Architecture** has to execute a **Store Instruction**, so the next state becomes **STORE**.
2. `READY` = 1, `WR_EN` = 0, and `DVALID` = 1 $\longrightarrow$ The **Architecture** has to execute a **Load Instruction**, so the next state becomes **LOAD**.

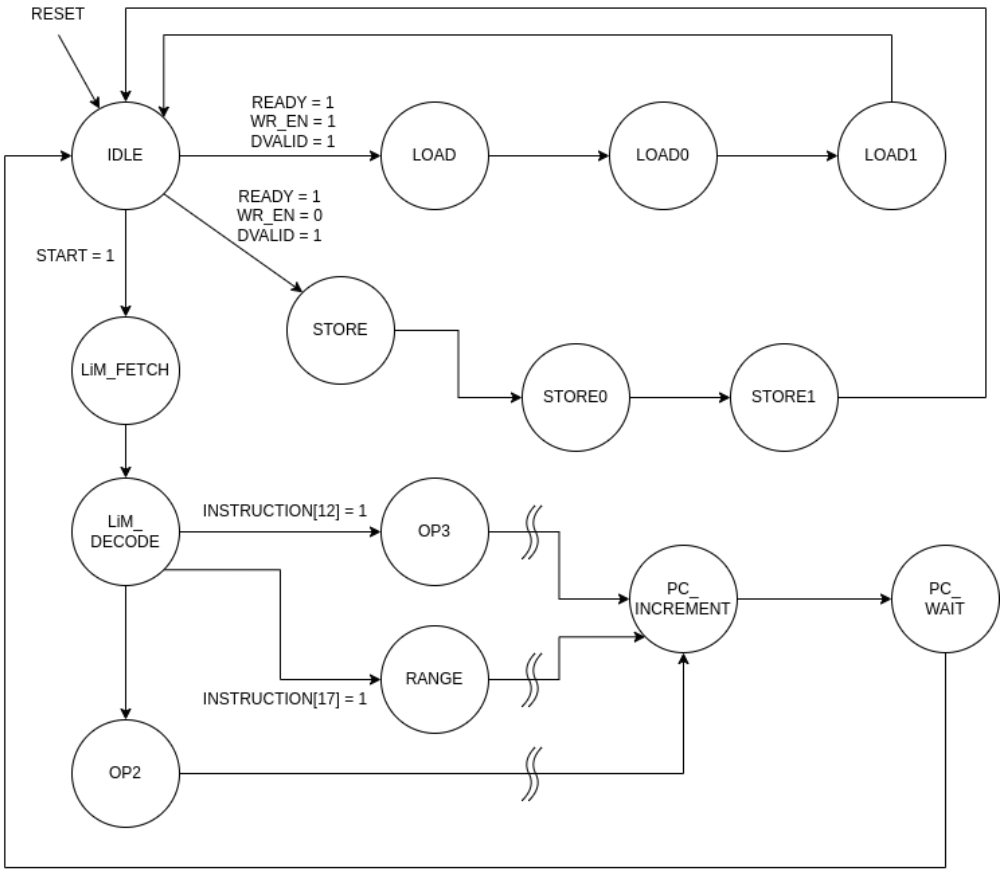


Figure 6.15. Summary of Control Unit Flow

3. `START` = 1 $\longrightarrow$ The **Architecture** has to execute a **LiM Instruction**, so the next state becomes **LIM_FETCH**.

### 6.5.1 Memory Mode: Load Instruction

The first two cases are covered in **Memory Mode**, which is triggered if `READY` and `DVALID` are both set while `START` is clear. When performing a **Load Operation**, the first state is **LOAD**, where the **CU** enables the **Memory Array** by activating the **Read Port 0** of the desired bank, while the **ID Unit** forwards the **Address**. Furthermore, the **CPU** is informed that the request has been granted through the `GRANT_OUT` signal. The next state is **LOAD0**, where the addressed **Bank Interface**'s output port is activated. Thus, data and the correct **Selection Signal** for the **WB Multiplexer** are brought to the **WB Register**. Then, the **CU** moves to **LOAD1**, where the

RDVALID signal is set to tell the **CPU** that the input data is valid and can be read. In this state, all signals are deactivated to "reset" them. Finally, the architecture goes back to **IDLE**.

### 6.5.2 Memory Mode: Store Instruction

**Store Operations** are divided into three states, starting from **STORE**, where the **CU** informs the **CPU** that the request has been granted by setting the GRANT_OUT signal and then inspects the input address to identify the destination bank. Once the destination bank is defined, the **CU** enables the output of the **Bus** and enables the **write port** of the destination **Memory Array**. The next state is **STORE0**, where the input data is sent to the bus through a proper selection by the **Input Multiplexer**. The final state, **STORE1**, gives enough time to the memory array for writing stable input data. Finally, the architecture moves back to **IDLE**, where all signals are also cleared.

### 6.5.3 LiM Mode

**LiM Mode** is triggered if READY and DVALID are both clear while START is set. In order to execute an instruction, the control flow in **LiM Mode** is very similar to that of a **CPU**, starting from **LIM_FETCH**, where the instruction is fetched and forwarded through the **IF Register** by enabling the **Pipeline Register**. The next state is **LIM_DECODE**, where the operands are decoded by the **ID Unit** while the instruction type is decoded by both the **ID Unit** and the **Control Unit**. Depending on the type, it is possible to jump to three different states:

1. **Three Operands Operation** $\longrightarrow$ **OP3**;
2. **Range Operation** $\longrightarrow$ **RANGE**;
3. **1 Operand and 2 Operands Operation** $\longrightarrow$ **OP2**.

#### LiM Mode: Three Operands Operation

**Three Operands Operations** start from **OP3**. In this state, the operation is identified and sent to the **ALU** of the destination bank. If the operands are external to the destination bank, they have to be fetched. Depending on how many operands are external with respect to the destination bank, it is possible to distinguish the following cases:

- If Operand 0 is in another bank, the next state is **OP3_FETCH_OP0_0**;
- If Operand 0 is in the destination bank, while Operand 1 is external, the next state is **OP3_FETCH_OP1_0**;
- If Operand 0 and Operand 1 are in the destination bank, the next state is **OP3_ALU_1_0**.

The Operand 0 is fetched in five steps, each corresponding to a specific state:

1. **OP3_FETCH_OP0_0** $\longrightarrow$ The **Reading Port 0** of the **Source Array** of Operand 0 is activated while the **Address** is provided. Data goes to the destination bank through the **Bus**;
2. **OP3_FETCH_OP0_1** $\longrightarrow$ The **Reading Port 0** of the **Source Array** of Operand 0 is deactivated while the input data on the destination bank is directed to the input port of the **Input Register File**;
3. **OP3_FETCH_OP0_2** $\longrightarrow$ This state is used to stabilize the input data coming through the port of the **Input Register File**;
4. **OP3_FETCH_OP0_3** $\longrightarrow$ In this state, data is written at **Row 0** of the **Input Register File** by enabling the writing port and providing the address;
5. **OP3_FETCH_OP0_4** $\longrightarrow$ This state is used to clear all the signals of the **Input Register File** and disable the writing port.

Once Operand 0 is fetched, the **CU** checks if Operand 1 is external. If it is external, it needs to be fetched; therefore, it moves to **OP3_FETCH_OP1_0** state and follows the same procedure as the OP0 Fetch, with the difference that the **Read Port 1** of the **Source Memory Array** is activated instead of **Read Port 0**, and the input data is stored into **Row 1** of the **Input Register File**. The following states are identical to those for OP0, with the only difference in the name since they present "OP1" instead of "OP0". Otherwise, if OP1 is internal, the next state is **OP3_ALU_1_0**, whose goal is to wait for one clock cycle to stabilize data.

All the operands are ready to compute the first result, so the **Control Unit** moves to **OP3_ALU_1_1**, which enables the reading ports of the **Input RF** or the **Memory Array**. Now data is at the input of the **Execute**

**Register**. **OP3_ALU_1_2** has the same goal as **OP3_ALU_1_0**, therefore it will just move to **OP3_ALU_1_3**, where the **input multiplexer**'s inputs are selected, and the result is computed and consequently stored to the **Temporary RF** by selecting the correct output port of the **ALU**. Furthermore, this state checks if the operation is composite and subsequently provides the **ALU** with the non-negated or the original operation. **OP3_ALU_1_4** state is in charge of deactivating the used reading ports and clearing the address port content. **OP3_ALU_1_5** is used to wait for one clock cycle so data is stable. During **OP3_ALU_1_6** and **OP3_ALU_1_7**, the writing port of the **Temporary Register File** is activated to write the temporary result in row 0.

Once the result is stored in the **Temporary Register File**, the **CU** checks if the third operand is internal to the destination bank or if it is external. In the former case, the next state is **OP3_ALU_2_0**, while in the latter it is **OP3_FETCH_OP2_0**.

The fetching of Operand 2 is very similar to that of Operand 0 and Operand 1, although it needs seven states instead of five because data takes longer to be fetched due to the load of the previous states. Anyway, the states are named as **OP3_FETCH_OP2_x**, where "x" is the number from 0 to 6. The main difference stands on the reading port used since it uses port 1, and the address of the **Input Register File** that is going to host this operand, which is **Row 2**.

Finally, the final result is computed in nine states. The procedure is very similar to the computation of the first result, except for the operands selection, since the **ALU** should receive the **Temporary Result** at Operand 0 Port and Operand 2 at Operand 1 Port. Finally, this state moves to **PC_INCREMENT**.
If the current instruction is a **Global Computation Instruction**, the procedure is the same as the one described before, with the exception of enabling the needed ports of each memory bank instead of one since the computation has to happen in parallel in all the banks. Furthermore, the only constraint for having parallel computation is to have all three operands inside the same bank as the destination, so basically it can be performed only in the best-case scenario. The control flow can be observed as:

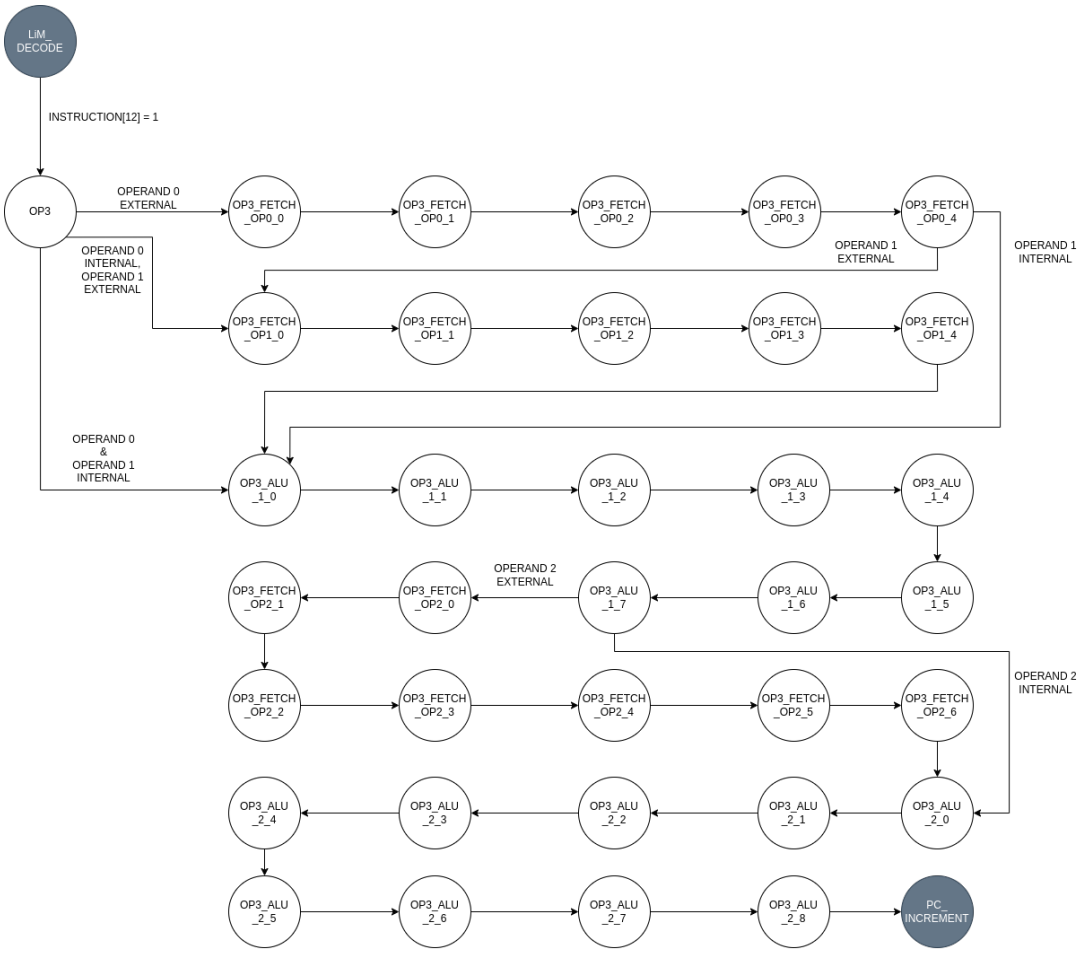


Figure 6.16. Summary of Control Unit Flow in Three Operands Instructions

### LiM Mode: Two Operands Operation

**Two Operands Operations** are the simplest to manage, since their working principle is the same as **Three Operands Instructions**, with the only exception that the **Temporary Register File** will not be used at all. The flow can be therefore summarized as:

1. **OP2** $\longrightarrow$ Check if operands are located in the same bank as the destination. If they are, enable the reading ports of the array; otherwise, keep them disabled. If Operand 0 is external, the next state is **OP2_FETCH_OP0_0**; if Operand 0 is internal while Operand 1 is external, the next state is **OP2_FETCH_OP1_0**; otherwise, if both operands are internal, the next state is **OP2_ALU_0**.

2. From **OP2_FETCH_OP0_0** to **OP2_FETCH_OP0_4** ⟶ These five states are used for fetching Operand 0 if it's located in a different bank than the destination bank. The operand is stored at **Row 0** of the **Input Register File**. Then it checks if Operand 1 is external to the destination bank as well. If it is, the next state is **OP2_FETCH_OP1_0**; otherwise, the **CU** moves to **OP2_ALU_0**.

3. From **OP2_FETCH_OP1_0** to **OP2_FETCH_OP1_4** ⟶ These states are used for fetching Operand 1 from another bank and storing it in **Row 1** of the **Input Register File**. The next state is **OP2_ALU_0**.

4. From **OP2_ALU_0** to **OP2_ALU_8** ⟶ These states are used for reading operands from the **Memory Array** (or the **Input Register File**), setting up the **ALU**, selecting the correct inputs of the **Multiplexers**, computing the result, and storing it back to the **Memory Array**. The next state is **PC_INCREMENT**.

As well as **Three Operands** and **Range Operations**, **Parallel Computation** and **Partial Word Computation** are supported, although the former can be performed only if both operands are internal. To summarize, the control flow for 2 Operands Operations is:

### LiM Mode: Address Range Operation

**Address Range** (or simply **Range**) Operations are the most complex instructions to control, due to the amount of operands they have to work with. Therefore, the **goal** of the design of the Control Unit States for managing these instructions is to optimize the amount of states in the flow, thus preventing any blowup. The final **result** leads to an eight-states **recursive control flow**, starting from the **RANGE** state. This state has the goal of checking if the instruction can be performed by verifying the locality of all the operands in the address range. Then it compares the current iteration number with the maximum amount of iterations for the current instruction to check if the current iteration is the last one.
Then it goes to **RANGE0**, where a "case" statement defines the signals for fetching the operands, depending on the value of the iterations counter (as well as all the other states for Address Range). It can potentially enable only the reading ports of the **Temporary Register File** for temporary computation among temporary results (or one temporary result and an operand from

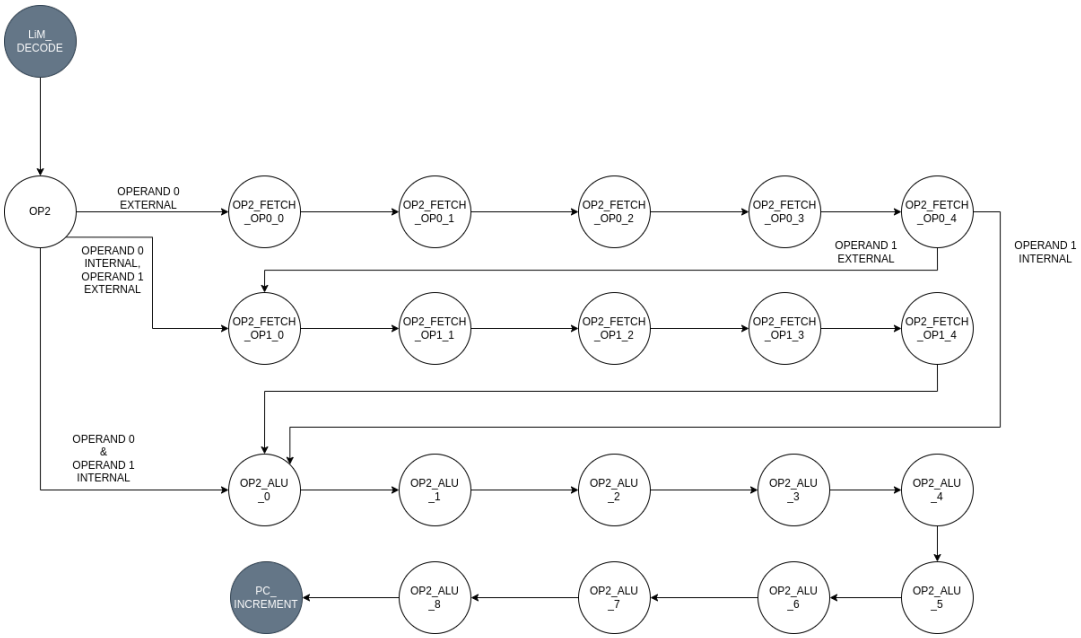


Figure 6.17. Summary of Control Unit Flow in Two Operands Instructions

the memory array); or the **Memory Array**. Furthermore, this state aims to communicate the **Memory Array** to use the Operand Output Ports for providing the Operands to the **ALU**.
In **RANGE1**, the CU **verifies** if the instruction requires a composite or non-composite operation and, depending on the iteration number, it provides the operation code to the **ALU**. It also selects the correct inputs of the **Multiplexer** connected to the Input Ports of the **ALU**.
**RANGE2** is used to **stabilize** the operands for the **ALU**.
**RANGE3** is the state where reading ports are deactivated and the result is eventually brought to the writing port of the memory (if it is the last iteration); otherwise, that port is kept disabled.
The next state is **RANGE4**, where the writing port of the **Temporary RF** and the memory array are kept deactivated.
**RANGE5** is used to write inside the **Memory Array** if the current iteration corresponds to the last one by providing the correct address and enabling the writing port. Otherwise, the writing port of the **Temporary Register File** is set, and the correct address is provided.
**RANGE6** is the state where the amount of iterations is increased, and **RANGE7** is the control state where the **CU** checks if it's the last iteration or not. In the former case, the next state is **PC_INCREMENT**; while in

the latter, it goes back to **RANGE**. Furthermore, all the ports are deactivated and addresses are cleared. This flow is repeated until the number of iterations is equal to the maximum number of iterations, which is computed as:

$$RANGE\_DUR = (OP0\_ADDR - OP1\_ADDR) - 1$$

or

$$RANGE\_DUR = (OP1\_ADDR - OP0\_ADDR) - 1$$

Where **RANGE_DUR** is the **Range Duration**, **OP0_ADDR** is the Address specified in the Operand 0 field, and **OP1_ADDR** is the Address specified in the Operand 1 field. The first formula is used if Operand 0 is the End of the Range and Operand 1 is the Beginning; while the latter applies in the opposite situation. The Range Operations can support up to eight operands to work with, and the flow of the operands selection is the same as the one specified in 6.4.3. Furthermore, it's worth clarifying that a Range Instruction can be performed only if the address range is included in the same bank, and it supports **Parallel Computation**. The flow of this type of instructions is illustrated in Figure 6.18.

### Ports Summary

The Control Unit presents the following Ports:

- `CLK`. This is the **Clock** Signal coming from the outside.
- `RST`. **Reset** Signal, active High.
- `DVALID` and `READY`. These signals are used for **switching** to Memory Mode and performing Load and Store Instructions when both are equal to "1".
- `START`. This signal is used for **switching** to LiM Mode and starting the execution of Instructions when it is equal to "1".
- `WR_EN`. In Memory Mode, it is used for **distinguishing** Load (when clear) from Store (when set) Instructions.
- `DONE`. This signal is used to **inform** the CPU that the LiM Computation is over; therefore, it has to clear the `START` signal.
- `INSTRUCTION`. This 43 bits signal is the **instruction**.

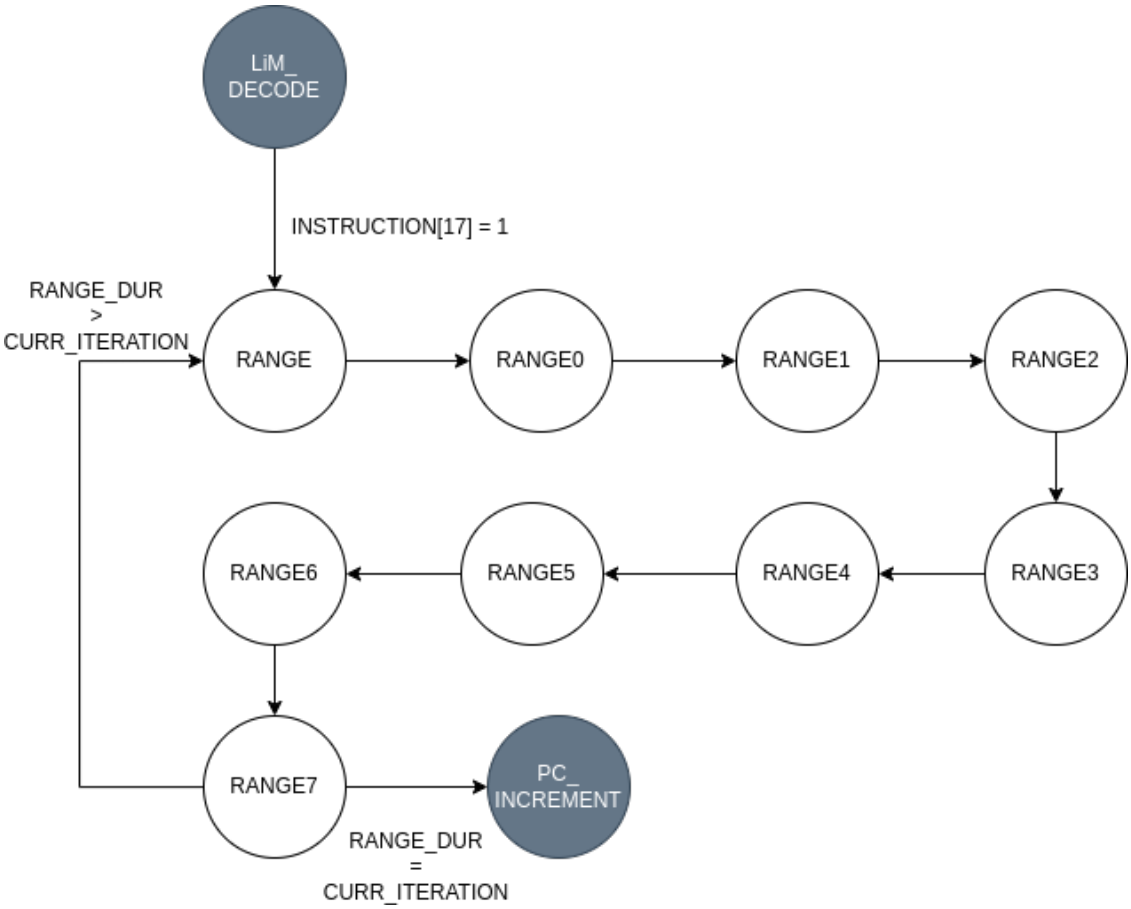


Figure 6.18. Summary of Control Unit Flow in Address Range Instructions

- `LDR_ADDR` and `STR_ADDR`. These 8 bits signals are the **addresses** for Load and Store in Memory Mode.
- `INT_ADDR`. This 8 bits signal is connected to the LiM Decoder and it decodes the address received by the CPU. This signal is used only in the X-Heep Implementation.
- `ALU_READY`. This 6 bits signal is not used anymore. It was originally used by the ALU to communicate the end of computation.
- `DATA_FIX`. This 6 bits signal is not used anymore. It is a legacy from the version of the architecture implementing the ECC Blocks.
- `STALL_ENABLE`. This 2 bits signal is used to **enable** and **disable** IF and WB Registers. These registers are enabled respectively once the new

instruction is fetched and when the output data is ready for the read, while they are disabled when the Architecture is in LiM mode and it is busy performing the current instruction and when no Load is performed.

- `INCREMENT_CNT`. This 32 bits signal is used to **increase** the internal address counter of the Instruction Memory.
- `OPERATION`. This 4 bits signal describes the ALU operation to perform.
- `COUNTER_ID_INCR`. This 8 bits signal is connected to the ID Unit for increasing its **internal** counter and providing the updated address in Range and Three Operands Instructions.
- `ALU_OUT_SEL`. This 6 bits signal selects the output ports of the ALU, depending on whether the result has to go to the Temporary RF or the Memory Array.
- `BANK_DATA_IN`. This 6 bits signal is not used anymore.
- `DATA_IN_SEL`. This 6 bits signal is used inside the Bank Interface to **forward** input data to the Memory Array through the Input Multiplexer or the Input Register File.
- `DATA_OUT_SEL`. This 6 bits signal is used for **directing** the output data from the banks to the Bus or the WB Register.
- `RD0_ALU` and `RD1_ALU`. These 6 bits signals are used for **sending** Operand 0 and Operand 1 from the Output Ports of the Array to the ALU or the Memory Interface through a dedicated Multiplexer. This Multiplexer performs arbitration by selecting one out of the two outputs of the Reading Ports, since only one data can be read at a time.
- `ARR_RD0_EN`, `ARR_RD1_EN`, and `ARR_WR_EN`. These three 6 bits signals enable respectively the Read Port 0, Read Port 1, and Writing Port of the Memory Array.
- `RFS_RD1_ADDR`, `RFS_RD0_ADDR`, and `RFS_WR _ADDR`. These 8 bits signals are respectively the **Reading Address** for Port 0 and 1 and the **Writing Address** of the Writing Port.
- `OP_STATE`. This 12 bits signal is used by the ALU to **prepare** the operands depending on the iteration status.

- `DEC_SEL`. This 6 bits signal is used to **select** among data coming from Reading Port 0 and Port 1 of the Memory Array to feed the Memory Interface through Memory Register.
- `LIM_MEM`. This 6 bits signal **selects** data coming from the Bank Interface and the Memory Register for writing the Memory Array.
- `WR_ENC`. This 6 bits signal, if set, indicates the Memory Array that the Result needs to partially replace the content of the destination address.
- `HIGH_LOW`. This 6 bits signal, if set, indicates to replace the upper 16 bits of the content of the destination address, while if it is clear, it replaces the lowest 16 bits. Along with the `WR_ENC`, they make a Byte Enabler.
- `OP0_SEL` and `OP1_SEL`. These 12 bits signals are used for selecting the correct input from the Execute Register for feeding the ALU with operands.
- `RFS_RD1_EN`, `RFS_RD0_EN`, and `RFS_WR_EN`. These 2 bits signals are used for **enabling** the respective **ports** of the Input Register File (if bit 0 is set) and Temporary Register File (if bit 1 is set). The two bits can only be alternatively set, while they can be both clear.
- `BANK_IN_SEL`. This 3 bits signal is used for selecting the input of the Bus in the Datapath.
- `OUTPUT_ENABLE_BUS`. This signal enables the output port of the Bus.
- `BUS_ENABLE`. This signal enables the Bus for read and write.
- `START_EN`, `ADDR_FF_EN`, and `DATA_IN_FF_EN`. These signals are not used at all.
- `RVALID_OUT`. This signal is used to tell the CPU that the read **data** is **valid**. It is set only for one clock cycle.
- `GRANT_OUT`. This signal is used for **communicating** with the CPU when performing any operation in Memory Mode, according to the OBI standard. It is set only for one clock cycle.
- `MUX_SEL_OUT`. This three bits signal is used for selecting the input of the WB Multiplexer. The inputs are:

  0. Data Out from Bank 0;

1. Data Out from Bank 1;
2. Data Out from Bank 2;
3. Data Out from Bank 3;
4. Data Out from Bank 4;
5. Data Out from Bank 5.

All the signals whose width is a multiple of 6 are connected bit by bit with the memory banks; therefore, they are only simultaneously set when performing **parallel computation** instructions. The ports layout of the Control Unit is:

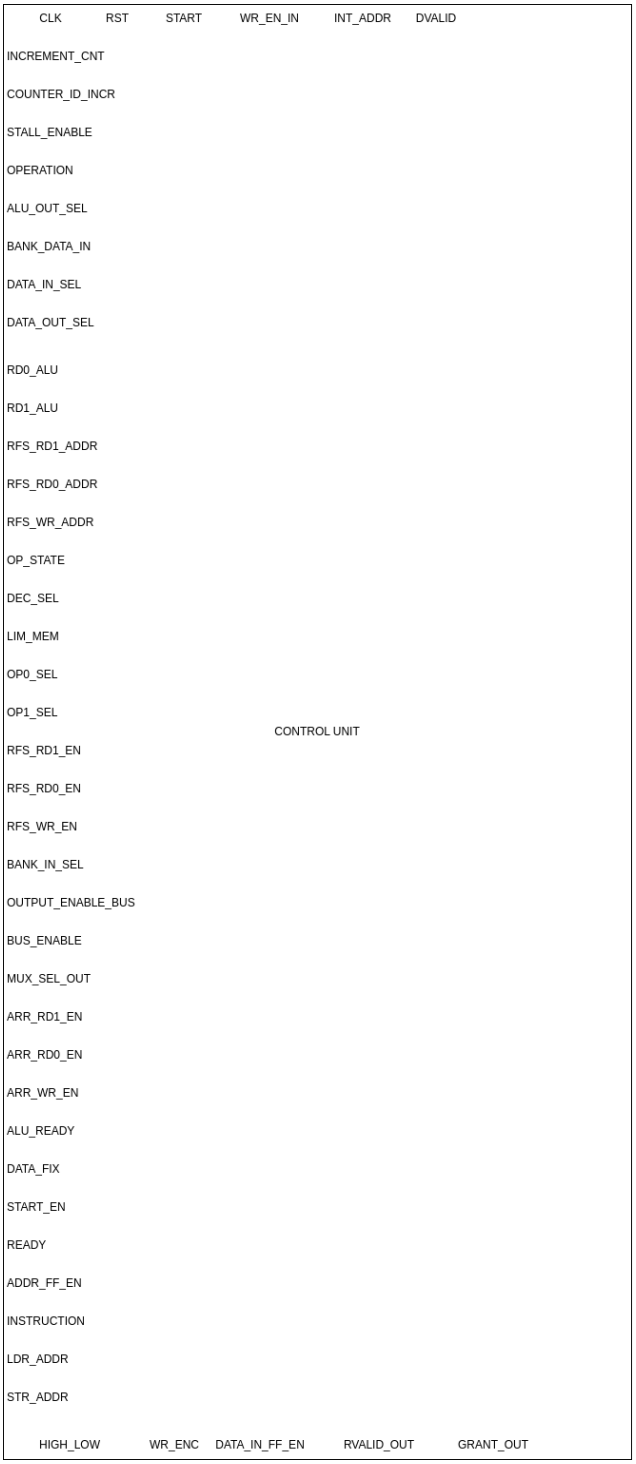


Figure 6.19. Control Unit Layout

# Chapter 7

# Model Testing - Part I: Testbenches

Each component's design phase was followed by a complex **testing phase**, which was carried out in several steps:

1. Individual Component Testing;
2. Bank Testing;
3. Datapath Testing;
4. Control Unit Testing;
5. Full LiMpire Architecture Testing.

To efficiently test all components, a series of **scripts** were developed to automate the entire process. The resulting Bash script followed these steps:

1. Execute the Python Compiler $\longrightarrow$ **Compile** the LiMpire Assembly code;
2. **Set up** the environment for QuestaSim;
3. **Run** .src Script on QuestaSim $\longrightarrow$ Compile all blocks under test;
4. Open .do Script on QuestaSim $\longrightarrow$ **Add** the required **waveforms** to the display;
5. **Run** the simulation.

The scripts used in this phase were written in **different languages**, including Bash, Python, and Tool Command Language (TCL).

Testing was conducted using **testbenches**, which included the following elements:

- **Clock Generator**: Provides a clock signal with a period of 2 ps;
- **Reset Generator**: Generates the reset signal;
- **Stimulus Generators**: Produces stimuli for the Unit Under Test (UUT);
- **Unit Under Test (UUT)**: The unit being tested.

The testbench layout is shown in Figure 7.1.

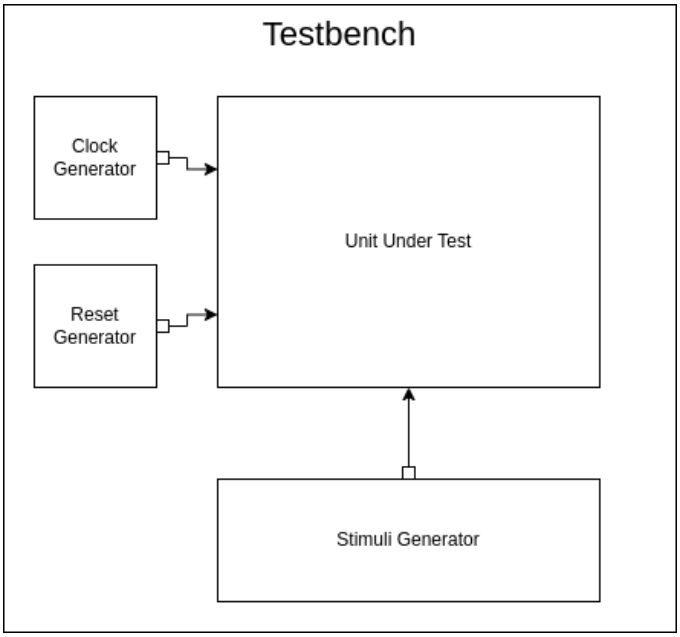


Figure 7.1. Testbench Layout

The Stimuli Generator operated in **two modes**: Random Stimuli Mode and Custom Stimuli Mode.

In **Random Stimuli Mode**, random signals were generated to test the UUT's behavior under various conditions and unexpected signal combinations.

In **Custom Stimuli Mode**, controlled input signals were provided to simulate specific scenarios or features. This mixed approach enhanced fault coverage, made the model more robust, and reduced the presence of bugs.

This methodology was applied across **all components**, starting with the smaller blocks and moving to the more complex ones, refining their behavior and internal communication. Additionally, the full architecture was tested using a **random** set of instructions to verify its behavior across all instruction types and operating modes.

An example of a random stimuli LiMpire Assembly program is shown below:

```
AND_LIM 32, 27, 32
OR_LIM 25AH, 1H, 2L
XOR_LIM 20AL, 30L, 29H, 19H
NOT_LIM 24A, 31
AND_LIM 44, 46, 39
OR_LIM 25AR, 1R, 5R
OR_LIM 25AR, 2R, 7R
AND_LIM 30AR, 1R, 8R
NOR_LIM 30AR, 1R, 3R
NOR_LIM 30AR, 1R, 4R
OR_LIM 25R, 1R, 7R
OR_LIM 25R, 1R, 5R
AND_LIM 30R, 1R, 8R
NOR_LIM 30R, 1R, 3R
NOR_LIM 30R, 1R, 4R
XOR_LIM 53R, 04R, 08R
NAND_LIM 87H, 21L, 55L, 69H
ADD_LIM 20L, 90L, 100H
MUL_LIM 20L, 31L, 99H
AND_LIM 0R, 1R, 4R
SUB_LIM 81L, 32L, 92H, 88H
XNOR_LIM 81L, 84L, 92H, 30H
NAND_LIM 76L, 30L, 28H, 77H
OR_LIM 55, 54, 64, 97
XNOR_LIM 127, 1, 32, 140
NOR_LIM 40L, 106L, 72H
AND_LIM 30, 29, 32, 28
NOT_LIM 22, 21
NOT_LIM 23, 46
NOT_LIM 22L, 46H
AND_LIM 20, 19, 18, 17
OR_LIM 17, 63, 18, 128
XOR_LIM 32R, 33R, 38R
NOR_LIM 0AR, 1R, 7R
XOR_LIM 32R, 33R, 38R
```

While this phase prepares the model for the real implementation, it ensures that the overall behavior is correct before integrating it into X-Heep, allowing

for only minor bug fixes later on. Finally, this phase was purely **theoretical**, since the tests were performed in a controlled environment, where the input combinations and the timing of their variations were relatively deterministic, while a real micro-controller may have a different behavior under **actual conditions**.

# Chapter 8

# LiMpire in a real environment: X-Heep Integration

X-Heep (**eXtendable Heterogeneous Energy-Efficient Platform**) is a **RISC-V** microcontroller described in **SystemVerilog** that can be configured to target small and tiny platforms as well as extended to support **accelerators**. [15] X-Heep presents a very high degree of **customization**, starting from the choice of the **CPU**, **peripherals**, and other devices, up to installing **hardware accelerators** by simply connecting them.[15] Furthermore, **compiling** and **testing** the whole system is very simple, since X-Heep features a set of **Makefile** and **scripts** that can be customized and expanded to add any additional design. The **architecture** of X-Heep can be represented as:

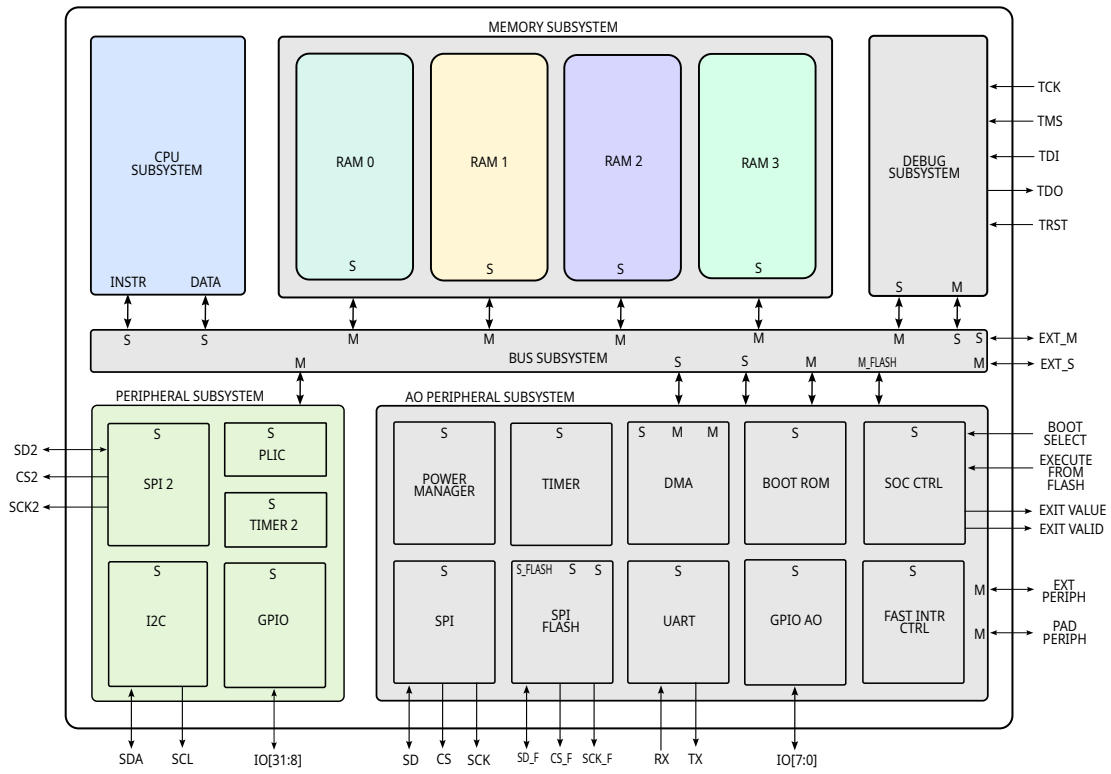


Figure 8.1. X-Heep **Architectural Layout**

From the **Software** Point of view, X-Heep offers the possibility of **coding** and running **software** as well as a physical microcontroller due to a system of **scripts** and **Makefile** to run **compilation** on the .c files written by the user.

## 8.1 LiMpire Integration

Integrating LiMpire inside X-Heep required some additional **models**, that are:

1. **Peripheral Registers**;
2. **Decoder**;
3. **LiMpire**.

All these **modules** are used to adapt the **signals** that interconnect the **CPU** with the **accelerator**, thus creating the **Level 0** of the Architecture. The Overall **System** is called:


Figure 8.2. Qui-Gon Heep **Logo**

The whole **system hardware** is compiled and simulated through the `run_hw.sh` script, which compiles the entire hardware and fixes a minor bug in X-Heep by calling the `finalize_build.py` script. This script ensures that the `Vtestharness.mk` file includes a missing string necessary to run the simulation.

### 8.1.1 Level 0: Peripheral Registers

The **Peripheral Registers** are extremely necessary for allowing the **CPU** to communicate with the **LiMpire** by enabling, disabling a **register** or writing a certain **value** into another **register**. These **registers** were described through the generation of **SystemVerilog code** by modifying a specific `.hjson` script. The proposed implementation features two main **Peripheral Registers**:

- **Status Register**;
- **Counter Max Register**.

#### Status Register

The **Status Register** is a 4-bit peripheral register that enables the **CPU** and the **LiMpire** to communicate for mode switching and resetting the internal counter of the **Instruction Memory**. This register has the following layout:

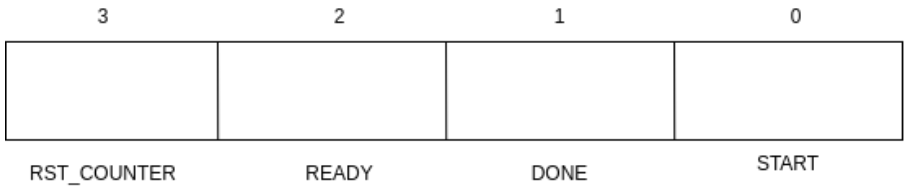


Figure 8.3. **Status Register**

Each bit has a distinct function, as summarized below:

0. `START` ⟶ This bit, along with the `READY` bit, is used for mode switching, and they should be set in complementary states. When `START` is set, the **LiMpire** begins executing the instructions stored in the **Instruction Memory** (described in Chapter 6.3.2). This bit is written by the **CPU** and read by the **LiMpire**.

1. `DONE` ⟶ This bit informs the **CPU** that the execution of **LiM Instructions** is complete, allowing the **CPU** to access the **memory** again. This bit is written by the **LiMpire** and read by the **processor**.

2. `READY` ⟶ This bit, used along with the `START` bit, is for mode switching and must be set in complementary states. When `READY` is set, the **LiMpire** enters **Memory Mode** and, upon `DVALID` assertion, performs a **Load** or **Store**. This bit is written by the **CPU** and read by the **LiMpire**.

3. `RST_COUNTER` ⟶ This bit is written by the **CPU** to reset the internal counter of the **Instruction Memory** to zero (discussed in Chapter 6.3.2). This signal is read by the **LiMpire** and then XOR-ed with the Reset of the **Instruction Memory** to achieve the desired effect.

In the original model, this function was achieved through logical signals.

### Counter Max Register

The **Counter Max Register** is a 32-bit register that stores the total number of instructions to execute in a complete LiM Computation Cycle. This register is written by the **CPU** and read by the **LiMpire**, which copies its content into the **Instruction Memory** (as described in Chapter 6.3.2). This register is essential during computation, as it determines when the **Accelerator** should stop computing.

### Component Layout

The **Peripheral Registers Component** features the following ports:

- `CLK_I`. This is the **Clock Signal** coming from the external source.
- `RST_N_I`. **Reset Signal**, Active Low.
- `REG_REQ_I`. This signal is used to write to the registers.
- `REG_RSP_O`. This signal is used to read from the registers.
- `REG_START_O`. This signal is used to read the **START Register** from the LiMpire.
- `REG_READY_O`. This signal is used to read the **READY Register** from the LiMpire.
- `REG_DONE_I`. This signal is used to write to the **Done Register** from the LiMpire.
- `REG_RST_COUNTER_O`. This signal is used to read the **RST_COUNTER Register** from the LiMpire.
- `REG_COUNTER_MAX_O`. This signal is used to read the **COUNTER_MAX Register** from the LiMpire.

This layout is shown in Figure 8.4.

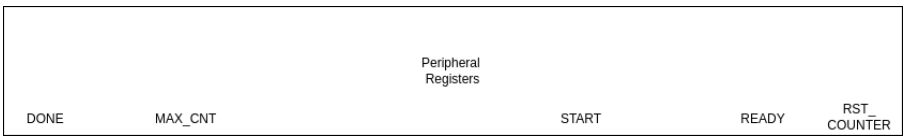


Figure 8.4. Peripheral Registers Layout

## 8.1.2 Level 0: Decoder

**X-Heep** includes various peripherals and external devices, each occupying a distinct portion of memory space. Consequently, when the **CPU** communicates with the **LiMpire** (e.g., to perform a **Store**), it needs to send an address with an offset related to the Accelerator's memory space location.

The LiMpire identifies two types of addresses:

- **Data Memory Address** $\longrightarrow$ From 0 to 191. This address range is divided among **Memory Banks** into six blocks of 32 addresses each, referring to the Data Memory.
- **Instruction Memory Address** $\longrightarrow$ From 192 to 1215. This range pertains to the **Instruction Memory** and specifically to the location of the Instruction Words.

The unit responsible for identifying the address type and subtracting the offset is the **Decoder**. To subtract the offset, it performs the following operation:

$$\texttt{LiM\_ADDR} = \texttt{ADDR} >> 2$$

Where `LiM_ADDR` is the address formatted for the LiMpire, and `ADDR` is the address provided by the CPU. Depending on the value of `LiM_ADDR`, two behaviors are possible:

1. `LiM_ADDR` $<$ 192 $\longrightarrow$ **Data Memory Address**. The CPU wants to perform a **Load** or **Store**; thus, it must forward the Request, Write Enable, Address, and Data In to the **Accelerator**. If the operation is a Load, the Write Enable and Data In are set to zero; otherwise, the Write Enable is activated.
2. 191 $<$ `LiM_ADDR` $<$ 1216 $\longrightarrow$ **Instruction Memory Address**. The CPU wants to write to the Instruction Memory, typically during startup. In this case, the Decoder sends the Request, Write Enable, Address, and Data In to the LiMpire's **Instruction Memory**, bypassing the Datapath.

If the address is out of range, the Decoder does not send any signal.
This component has the following ports:

- `LIM_REQ_I`. This signal is sent by the CPU to indicate a pending operation on the LiM, per the **OBI standard**.
- `LIM_WE_I`. This signal is **Active High** and is used to perform a **Store**.
- `LIM_ADDR_I`. This 32-bit signal is the full address provided by the CPU.
- `LIM_WDATA_I`. This 32-bit signal is the data to write to the LiMpire.
- `LIM_RDATA_I`. This 32-bit signal is the data read from the LiM to the CPU, connected to the LiMpire and CPU respectively.

- `LIM_REQ_O`. This signal tells the LiM that the CPU wants to access the **Data Memory**, per the OBI standard.
- `LIM_WE_O`. This signal indicates the CPU wants to write to the Data Memory.
- `LIM_RDATA_O`. This signal indicates the CPU wants to read from the Data Memory.
- `LIM_ADDR_O`. This 8-bit signal specifies the address for the Data Memory.
- `LIM_WDATA_O`. This 32-bit signal is the data to be written to the Data Memory.
- `INSTR_MEM_REQ_O`. This signal communicates to the **Instruction Memory** that the CPU wants to perform an operation.
- `INSTR_MEM_WE_O`. This signal indicates that the CPU wants to perform a Store operation on the Instruction Memory.
- `INSTR_MEM_ADDR_O`. This 10-bit signal specifies the address for the Instruction Memory.
- `INSTR_MEM_WDATA_O`. This 32-bit signal is the data to be written to the Instruction Memory.

The component layout is shown in Figure 8.5.

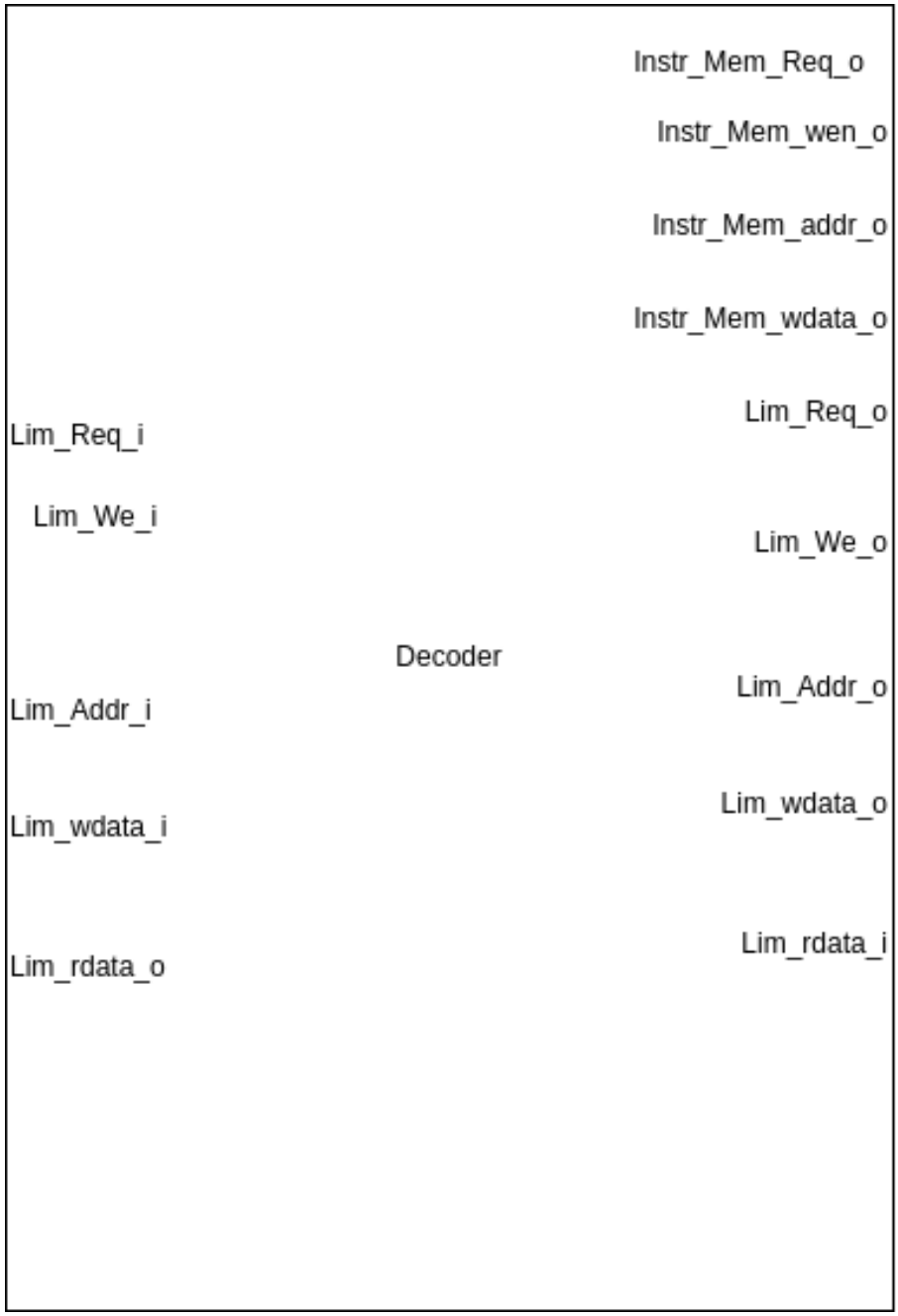


Figure 8.5. Decoder Layout

# Chapter 9

# QuiGon Heep Test: Benchmarks

All the components composing **QuiGon Heep** were previously tested in Chapter 7 through testbenches. Although this type of test effectively checks the **overall behavior** of a module or group of modules, it does not fully reflect the **realistic behavior** of the architecture. Each CPU has unique timing and synchronization constraints, making it essential to test the entire architecture using benchmarks.

This chapter is divided into two main blocks:

1. **Block I - Benchmarks Implementation**
2. **Block II - Software Interface and Drivers**

## 9.1 Block I - Benchmarks Implementation

This section provides an overview of the **algorithms** and **benchmarks** used to test the QuiGon Heep System. Most of the architectures discussed in Chapter 3 were designed to execute **cryptographic** algorithms, which involve sequences of operations with varying complexity. To evaluate QuiGon Heep, dedicated benchmarks based on commonly used algorithms were developed.

The list of algorithms includes:

1. **GEMM**
2. **GEMMVER**

3. **Keccak Round F**
4. **One Time Pad**
5. **SHA-1**
6. **XOR Cipher**

For each algorithm, two benchmarks were created: one where the **CPU** alone executes the algorithm, and another where the **LiM** (Logic in Memory) is responsible for its execution.

### 9.1.1 Algorithm 1: GEMM

The **General Matrix Multiply** (GEMM) algorithm was developed to efficiently utilize an OpenCL device for performing **matrix multiplication** on two dense square matrices. It is designed for devices with cache memory, such as Intel Xeon Phi and Intel Architecture CPU OpenCL devices. This implementation enhances the basic nested-loop matrix multiplication by applying an optimization technique known as **tiling** (or **blocking**), which divides the matrices into smaller blocks. This approach improves memory cache usage by maintaining better data locality during the multiplication of matrix blocks.

The **pseudocode** for GEMM is:

```
for i from 0 to size-1
    for j from 0 to size-1
        c = 0
        for k from 0 to size-1
            c = c + A(k, i)*B(j, k)
        end for
        C(j, i) = alpha*c + beta*C(j, i)
    end for
end for
```

### 9.1.2 Algorithm 2: GEMMVER

The **GEMMVER** algorithm is a computational kernel commonly used in high-performance computing, particularly in the context of linear algebra. GEMMVER, which stands for "General Matrix Multiply and Vector Addition with Extra Rank," serves as a **benchmark** for performance testing in mathematical software libraries and hardware systems.

GEMMVER performs a sequence of **matrix and vector operations**, including matrix multiplication (GEMM) and vector additions, common in various scientific and engineering applications, especially those requiring large-scale computations.

The GEMMVER algorithm typically involves the following steps:

1. **Matrix Multiplication (GEMM)**:
$$C = \alpha * A * B + \beta * C$$
where $\alpha$ and $\beta$ are scalars, $A$ and $B$ are matrices, and $C$ is the resultant matrix. This operation is a general matrix multiplication.

2. **Vector Addition and Outer Product**:
$$C = C + \text{outerproduct}(u, v)$$
where $u$ and $v$ are vectors. This step involves adding the outer product of two vectors to the matrix $C$.

3. **Vector Scaling and Addition**:
$$w = \gamma * (C * x) + \delta * y$$
where $\gamma$ and $\delta$ are scalars, $C$ is the matrix from the previous step, $x$ and $y$ are vectors, and $w$ is the resulting vector. This operation involves scaling and adding vectors.

The pseudocode for GEMMVER is:

```
// Function to compute GEMVER: y := alpha * A * x + beta * y
function gemver(alpha, A, x, beta, y, n):

    // Initialize temporary vector
    tmp = new vector of size n

    // Compute A * x and store result in tmp
    for i = 0 to n-1:
        tmp[i] = 0
        for j = 0 to n-1:
            tmp[i] += A[i][j] * x[j]

    // Compute alpha * (A * x) + beta * y and store result in
    y
    for i = 0 to n-1:
        y[i] = alpha * tmp[i] + beta * y[i]

    // Return resulting vector y
    return y
```

### 9.1.3 Algorithm 3: Keccak Round-f

**Keccak** is a family of hash functions based on the **sponge construction**, making it a sponge function family [16]. The core of Keccak is a **permutation** function chosen from a set of seven `Keccak-f` permutations, denoted as `Keccak-f[b]`, where $b \in \{25, 50, 100, 200, 400, 800, 1600\}$ represents the width of the permutation. This width also defines the **state size** in the sponge construction, which is structured as a $5 \times 5$ array of lanes, each with a length $w \in \{1, 2, 4, 8, 16, 32, 64\}$, with $b = 25w$. On a 64-bit processor, a lane of `Keccak-f[1600]` corresponds to a 64-bit **CPU** word. The `Keccak[r, c]` sponge function, with parameters **capacity** $c$ and **bitrate** $r$, is obtained by applying the sponge construction to `Keccak-f[r + c]`, along with a specific **message padding** scheme.

The pseudocode describing this algorithm is:

```
Keccak-f[b](A) {
    for i in 0...n-1:
        A = Round[b](A, RC[i])
        return A
   }

Round[b](A,RC) {
    # \theta step
    for x in 0...4:
        C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[
   x,4]

    for x in 0...4:
        D[x] = C[x-1] xor rot(C[x+1],1)

    for (x,y) in (0...4,0...4):
        A[x,y] = A[x,y] xor D[x]

    # \rho and \pi steps
    for (x,y) in (0...4,0...4):
        B[y,2*x+3*y] = rot(A[x,y], r[x,y])

    # \chi step
    for (x,y) in (0...4,0...4):
        A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y])

    # \iota step
    A[0,0] = A[0,0] xor RC
    return A
}
```

Since there is no suitable compiler that can optimize data placement and translate instructions from a high-level programming language (such as `C`) into LiMpire Assembly language, this algorithm was executed only three times (`for i in 0..2`).

### 9.1.4 Algorithm 4: One-Time Pad

The **One-Time Pad** (OTP) is a cryptographic algorithm that provides theoretically **unbreakable encryption** when used correctly. It was invented by Gilbert Vernam in 1917 and later mathematically formalized by Claude Shannon, who proved that it offers **perfect secrecy**. The algorithm involves three main steps:

1. **Key Generation**: A key (a random string of bits or characters) is generated that matches the length of the message to be encrypted. This key must be truly random, not pseudo-random.

2. **Encryption**: The message is converted into binary or numerical form. The key is then combined with the message using bitwise exclusive OR (**XOR**) for binary data or modular arithmetic for text.

   - For binary data: Each bit of the message is **XOR**-ed with the corresponding bit of the key.
   - For text: Each character of the message is converted into a number (e.g., using `ASCII`), and the key's corresponding character (also converted to a number) is **added** to the message's number, modulo 26 for letters, or modulo 256 for binary data.

   This process produces the encrypted message, or **cipher-text**.

3. **Decryption**: The recipient, who has the same key, **reverses** the process by XORing the cipher-text with the key (or using the same modular arithmetic). This restores the original message.

The pseudocode is as follows:

```
1 //Encryption
2 function OneTimePadEncrypt(plaintext, key):
3     if length(plaintext) != length(key):
4         raise Error("Plaintext and key must be of the same
    length")
5
```

```
    ciphertext = ""
    for i from 0 to length(plaintext) - 1:
        # Convert characters to their ASCII values
        plaintext_char = ord(plaintext[i])
        key_char = ord(key[i])

        # Perform bitwise XOR between the characters
        cipher_char = plaintext_char XOR key_char

        # Convert the result back to a character and append
        to ciphertext
        ciphertext = ciphertext + chr(cipher_char)
    return ciphertext

//Decryption
function OneTimePadDecrypt(ciphertext, key):
    if length(ciphertext) != length(key):
        raise Error("Ciphertext and key must be of the same
   length")

    decrypted_text = ""
    for i from 0 to length(ciphertext) - 1:
        # Convert characters to their ASCII values
        cipher_char = ord(ciphertext[i])
        key_char = ord(key[i])

        # Perform bitwise XOR between the characters
        plain_char = cipher_char XOR key_char

        # Convert the result back to a character and append
        to decrypted_text
        decrypted_text = decrypted_text + chr(plain_char)

    return decrypted_text
```

In this code, the `key` length must match the `plaintext` length, and the `ord()` function converts characters into their ASCII values. To maintain a high level of security, the `key` must be **randomly generated** and **never reused**.

### 9.1.5 Algorithm 5: SHA-1

**SHA-1** (Secure Hash Algorithm 1) is a **cryptographic hash function** that takes an input and produces a 160-bit (20-byte) hash value [6]. It is widely used for integrity verification, digital signatures, and other cryptographic

applications. SHA-1 operates on blocks of 512 bits, and the resulting output is a 160-bit hash value [6].

SHA-1 processes input data by breaking it down into chunks and applying a series of **bitwise operations**, **logical functions**, and **modular additions** [6]. The algorithm processes the input message in the following steps:

1. **Padding**. The message is padded so that its length is congruent to 448 bits mod 512. Padding is done by appending a "1" bit followed by enough "0" bits, so the total length is congruent to 448 mod 512. The original message length (in bits) is then appended as a 64-bit integer [6].

2. **Message Parsing**. The padded message is divided into 512-bit blocks. [6] Each block is then split into sixteen 32-bit words $W_0, W_1, \ldots, W_{15}$, which serve as the input to the main processing loop [6].

3. **Message Expansion**. The sixteen 32-bit words are expanded to eighty 32-bit words, using the following relation for $t \geq 16$:

$$W_t = (W_{t-3} \oplus W_{t-8} \oplus W_{t-14} \oplus W_{t-16}) \lll 1$$

   where $\oplus$ is the bitwise XOR and $\lll 1$ represents a left circular shift by 1 bit.

4. **Initial Hash Values**. **SHA-1** uses five 32-bit hash values, initialized as follows:

$$H_0 = 0x67452301, \quad H_1 = 0xEFCDAB89, \quad H_2 = 0x98BADCFE,$$
$$H_3 = 0x10325476, \quad H_4 = 0xC3D2E1F0$$

5. **Main Loop**. For each 512-bit block, SHA-1 performs **80 rounds** of operations using the following formulas:

$$T = (a \lll 5) + f_t(b, c, d) + e + W_t + K_t$$
$$e = d, \quad d = c, \quad c = b \lll 30, \quad b = a, \quad a = T$$

   The non-linear function $f_t(b, c, d)$ and the constant $K_t$ depend on the current round $t$:

$$f_t(b, c, d) = \begin{cases} (b \wedge c) \vee (\neg b \wedge d) & \text{for } 0 \leq t \leq 19 \\ b \oplus c \oplus d & \text{for } 20 \leq t \leq 39 \\ (b \wedge c) \vee (b \wedge d) \vee (c \wedge d) & \text{for } 40 \leq t \leq 59 \\ b \oplus c \oplus d & \text{for } 60 \leq t \leq 79 \end{cases}$$

$$K_t = \begin{cases} 0x5A827999 & \text{for } 0 \leq t \leq 19 \\ 0x6ED9EBA1 & \text{for } 20 \leq t \leq 39 \\ 0x8F1BBCDC & \text{for } 40 \leq t \leq 59 \\ 0xCA62C1D6 & \text{for } 60 \leq t \leq 79 \end{cases}$$

6. **Hash Value Update**. After processing each message block, the initial hash values are updated:

$$H_0 = H_0 + a, \quad H_1 = H_1 + b, \quad H_2 = H_2 + c,$$
$$H_3 = H_3 + d, \quad H_4 = H_4 + e$$

7. **Final Output**. Once all the message blocks have been processed, the final hash value is the **concatenation** of the updated values:

$$\text{Hash} = H_0 \| H_1 \| H_2 \| H_3 \| H_4$$

This produces a 160-bit output that uniquely represents the input message.

The pseudo-code can be written as:

```
function sha1(message):
    // Constants for SHA-1
    K = [... constants ...]

    // Initialize hash variables
    H0 = initial value
    H1 = initial value
    H2 = initial value
    H3 = initial value
    H4 = initial value

    // Pre-processing
    padded_message = pad_message(message)
    blocks = split_into_blocks(padded_message, 512 bits)

    // Process each block
    for each block in blocks:
        words = expand_block(block)     // Expand block to 80
    words

        // Initialize working variables
        A = H0
        B = H1
```

```
        C = H2
        D = H3
        E = H4

        // Main loop
        for i from 0 to 79:
            if 0 <= i <= 19:
                F = (B AND C) OR ((NOT B) AND D)
                K = constant

            else if 20 <= i <= 39 or 60 <= i <= 79:
                F = B XOR C XOR D
                K = constant

            else if 40 <= i <= 59:
                F = (B AND C) OR (B AND D) OR (C AND D)
                K = constant

            temp = LEFT_ROTATE(A, 5) + F + E + words[i] + K
            E = D
            D = C
            C = LEFT_ROTATE(B, 30)
            B = A
            A = temp

        // Update hash values
        H0 += A
        H1 += B
        H2 += C
        H3 += D
        H4 += E

    // Concatenate hash values
    hash_result = concatenate(H0, H1, H2, H3, H4)

    // Truncate to 128 bits (if needed)
    truncated_hash = take_most_significant_bits(hash_result,
   128)

    return truncated_hash

// Example usage
message = "Your 128-bit message here"
sha1_hash = sha1(message)
print("SHA-1 hash:", sha1_hash)
```

Since there is **no** suitable **compiler** that can optimize data placement and

translate instructions from a **high-level programming language** (such as C) into **LiMpire Assembly language**, this algorithm was only executed fifteen times (for `i` in 0..2).

### 9.1.6 Algorithm 6: XOR Cipher

The **XOR cipher** (Exclusive OR cipher) is a simple encryption algorithm that operates on binary data. It relies on the **XOR bitwise operation** to **encrypt** and **decrypt** messages. The XOR operation works by comparing two bits: if the bits are the same, the result is 0; if the bits are different, the result is 1. The flow consists of two main phases:

1. **Encryption**. Encryption is the process of converting plaintext, or readable data, into ciphertext, which is an unreadable, encoded form. The goal is to **protect** the original message from unauthorized access, ensuring that only individuals with the correct key can decrypt and read the message. Encryption involves using an **algorithm** and a **key**, where the algorithm applies a series of mathematical operations to transform the plaintext into ciphertext. The key is a piece of information that dictates how the transformation is performed, making the encrypted data appear random without it.

2. **Decryption**. Decryption is the **reverse** process, where the ciphertext is converted back into its original plaintext form using the same algorithm and a corresponding key. The same key is often used for both encryption and decryption in **symmetric encryption**, while **asymmetric encryption** involves two different keys: a public key for encryption and a private key for decryption. Decryption requires having the correct key; without it, the encoded message remains unintelligible.

XOR is often used in **stream ciphers** and **cryptographic protocols** where lightweight encryption is needed and in various bitwise manipulation techniques in programming. This algorithm can be represented with the following pseudo-code:

```
// The same function is used to encrypt and
// decrypt
void encryptDecrypt(char inpString[])
{
    // Define XOR key
    // Any character value will work
    char xorKey = 'P';
```

```
    // calculate length of input string
    int len = strlen(inpString);

    // perform XOR operation of key
    // with every character in string
    for (int i = 0; i < len; i++){
        inpString[i] = inpString[i] ^ xorKey;
        printf("%c", inpString[i]);
    }
}

// Driver program to test above function
int main(){
    char sampleString[] = "ExampleString";

    // Encrypt the string
    printf("Encrypted String: ");
    encryptDecrypt(sampleString);
    printf("\n");

    // Decrypt the string
    printf("Decrypted String: ");
    encryptDecrypt(sampleString);

    return 0;
}
```

## 9.2 Block II - Software Interface and Drivers

The procedure outlined in this section can be applied to any benchmark compatible with the QuiGon Heep System. The process involves:

1. Writing **drivers** for the system;
2. Writing **instructions** into a 1024-word array, with each word being 32 bits;
3. Writing the "**Main**" function in the `main.c` file;
4. Enabling **interrupts** (or polling);
5. Loading the **instruction array** into the instruction memory;
6. **Simulating** the system while recording performance data;
7. **Comparing** the performance results.

### 9.2.1 Instruction Array Setup

All algorithms mentioned in Section 9 can be executed by the CPU and memory. To run **benchmarks** on the **CPU**, data needs to be stored in memory banks, loaded when necessary, processed, and the final results stored back in the memory banks.

When the execution relies on **memory**, the approach changes slightly: data is still stored in the memory banks, but the entire process is run through instructions stored inside the Instruction Memory. Thus, it's essential to load the **LiMpire Assembly file** content into the Instruction Memory array. The steps to do this task are:

1. **Write** the LiMpire Assembly file;
2. Use a Python **compiler** to convert the assembly file into a 32-bit machine code file (by running `asm_to_bin_64.py` and `bin_to_dec_64.py` scripts);
3. Create two C files: `instruction_array.h` (which **declares** an empty 1024-integer array) and `instruction_array.c` (which **fills** the array with the machine code). This array is called the "Instruction Array";
4. **Load** the instruction array into the Instruction Memory when the system starts.

The entire process is **automated** using Bash and Python scripts that follow the steps and configure the system. The **toolchain** can be executed by simply running the script `run_sw.sh`, which calls the compiling scripts and `instruction_memory_store.py` to operate on the Instruction Memory. Finally, the script compiles the C code, simulates with Verilator, and opens GTKWave for waveform analysis. Furthermore, the script prompts the user to select the algorithm to execute. Based on this selection, it chooses the appropriate LiMpire Assembly file and replaces the `main.c` file with a specific main file developed for the corresponding benchmark.

### 9.2.2 Drivers Setup and Implementation

The software implementation began with **setting up** all the drivers to enable interaction with the accelerator. This project is characterized by two different types of drivers:

- **System Drivers**;

- **Functional Drivers**.

### Type 1: System Drivers

System Drivers are responsible for executing **basic functions**, such as writing to or reading from a register. They are declared in `limpire.h` and defined in `limpire.c`. The main functions are:

- `void limpire_start_set()`: Writes a 1 to the **Start** Bit of the Status Register, allowing a potential switch to **LiM** Mode (depending on the Ready Bit).

- `void limpire_start_clear()`: Writes a 0 to the **Start** Bit of the Status Register, allowing a potential switch to **Memory** Mode (depending on the Ready Bit).

- `void limpire_ready_set()`: Writes a 1 to the **Ready** Bit of the Status Register, allowing a potential switch to **Memory** Mode (depending on the Start Bit).

- `void limpire_ready_clear()`: Writes a 0 to the **Ready** Bit of the Status Register, allowing a potential switch to **LiM** Mode (depending on the Start Bit).

- `void limpire_rst_cnt_set()`: Writes a 1 to the **Reset Counter** Bit of the Status Register, resetting the Internal Counter of the Instruction Memory.

- `void limpire_rst_cnt_clear()`: Writes a 0 to the **Reset Counter Bit**, reversing the previous action.

- `int32_t limpire_done_read()`: Reads the **Done** Bit and returns its value, used to trigger interrupts or exit polling.

- `void limpire_counter_ set(uint32_t max_index)`: Writes the passed value to the **Max Counter** Register, defining the maximum value of the Internal Counter of the Instruction Memory.

- `void limpire_counter_clear()`: Writes a 0 to the **Max Counter** Register.

**Type 2: Functional Drivers**

Functional Drivers are declared in `lim_ functions.h` and defined in `lim_ functions.c`. They are **higher-level functions** built on top of the System Drivers to simplify the C code. The main functions are:

- `int32_t lim_load(int32_t address)`: **Loads** data from a specified memory address into a variable by passing the return value.
- `void lim_store(int32_t address, int32_t data)`: **Stores** the input data in the specified memory address.
- `void instruction_ memory_fill()`: **Fills** the Instruction Memory with the content of the Instruction Array.
- `void limpire_counter_ set(int32_t num_instructions)`: Sets the **number of instructions** for the internal counter.
- `void lim_ startup()`: Combines `instruction_memory_ fill()` and `limpire_ counter_set(num_instructions)`, usually run at the start of the `Main` function.
- `void lim_ execute_instr_steps(int32_t unit_val)` and `void lim_ execute_range(int32_t max)`: Execute a **specific number** of instructions, e.g., from 0 to 100. These are useful for partial sets of LiM Instructions. They always start from 0 due to hardware limitations. The former repeats the execution based on `unit_val`, while the latter executes the range directly.
- `void lim_execute_instr()`: Executes the **full set** of instructions in LiM mode, starting by setting the Start Bit in the Status Register.

### 9.2.3 Pooling and Interrupt Mechanisms in QuiGon Heep

LiM computation is **triggered** by writing a "1" to the Start Bit of the Status Register while the Ready Bit of the Status Register is clear, and it **completes** when the Done Bit in the same register is set. There are two methods to inform the CPU about this event:

- **Pooling**: This technique involves the system **regularly checking** the status of a device or resource at set intervals to see if it requires attention.

Pooling is simple to implement and manage, though it requires the CPU to stay idle and continuously check the status.

- **Interrupt**: This method uses a **signal** that automatically stops the CPU's current activity and alerts the system when an **event** occurs. While more complex to implement, it allows the CPU to perform other tasks while the peripheral, handled by the interrupt, operates independently.

Both techniques can be implemented in the `main.c` file.

#### Pooling Implementation

Pooling can be easily implemented using the following code:

```
while(limpire_done_read() != 1){
  if(limpire_done_read() == 1){
    break;
  }
}
```

This simple `while` loop uses the Done Bit being set to 1 as the exit condition. While Done remains 0, the system remains stuck in the loop.

### 9.2.4 Interrupt Implementation

Implementing an interrupt requires several steps, as it is more complex:

1. **Write the Interrupt Request Handler Function**: This function is triggered only when the Done Bit is set. The implementation can be as follows:

```
    // Interrupt request handler
    static void handler_irq_quigon_heep( uint32_t int_id )
    {
      if (limpire_done_read() == 1)
      {
        finished = true;
        limpire_start_clear();
        limpire_rst_cnt_set();
        limpire_rst_cnt_clear();
      }
      printf("Interrupt raised!\n");

      CSR_READ(CSR_REG_MCYCLE, &cycles);
      is_limpire_term = true;
```

```
15     }
16
```

2. **Enable Interrupts**: This is done in the `main` function using the following instructions:

```
1     // Enable interrupt on processor side
2     // Enable global interrupt for machine-level
    interrupts
3     CSR_SET_BITS(CSR_REG_MSTATUS, 0x8);
4
5     // Set mie.MEIE bit to 1 to
6     // enable machine-level external interrupts
7     const uint32_t mask = 1 << 11;
8     CSR_SET_BITS(CSR_REG_MIE, mask);
9
10    if(plic_Init()){
11      return EXIT_FAILURE;
12    }
13
14    if(plic_irq_set_priority(EXT_INTR_0, 1)){
15      return EXIT_FAILURE;
16    }
17
18    if(plic_irq_set_enabled(EXT_INTR_0, kPlicToggleEnabled
    )){
19      return EXIT_FAILURE;
20    }
21
22    plic_assign_external_irq_handler(EXT_INTR_0,
23    &handler_irq_quigon_heep);
24
```

This procedure uses **built-in functions** from the C libraries `csr.h`, `handler.h`, `rv_plic.h`, `rv_plic_regs.h`, `rv_plic_structs.h`, `hart.h`, and `fast_intr_ctrl.h`.

### 9.2.5 Performances Comparison

The **purpose** of running two different benchmarks based on the same algorithm (executed by both the CPU and the Memory) is to demonstrate that the **proposed** model is fully compliant with the LiM Paradigm. This is highlighted by its faster performance for specific algorithms, allowing for the computation of **speedup** once the performance metrics are determined.

C language libraries provide the capability to implement a **Performance Counter**, which is enabled and reset at the beginning of the `main` function. This counter computes the number of clock cycles from the start of execution until the invocation of the function that reads the counter. This is accomplished using the `CSR_READ` function, which records the **timestamp** of the performance counter, along with `CSR_CLEAR_BITS` and `CSR_WRITE` to reset the performance counter.

The performance counter can be implemented independently of the choice between interrupts and polling. In a typical C program, the counter is set up in the `main` function as follows:

```
// Starting the performance counter
CSR_CLEAR_BITS(CSR_REG_MCOUNTINHIBIT, 0x1);
CSR_WRITE(CSR_REG_MCYCLE, 0);

CSR_READ(CSR_REG_MCYCLE, &start);
```

In this block, the counter is reset, and the starting clock cycle is computed. Once the computation is completed, the counter is read again using the same function, but the value is stored in another variable called `end`:

```
CSR_READ(CSR_REG_MCYCLE, &end);
```

If the implementation uses an interrupt, this function is invoked in the handler function instead of the `main`. Finally, the end value is subtracted from the start value as follows:

$$\texttt{cycles} = \texttt{end} - \texttt{start}$$

where `cycles` represents the number of clock cycles utilized for the computation. This procedure has been executed across all benchmarks, bringing the following data:

| **Algorithm** | **CPU Exec. Time (cc)** | **LiM Exec. Time (cc)** | **SpeedUp** |
|---|---|---|---|
| GEMM | 30,500 | 15,646 | 1.94x |
| GEMMVER | 8,296 | 3,795 | 2.18x |
| Keccak Round F | 58,051 | 25,179 | 2.30x |
| One Time Pad | 61,362 | 6,098 | 10.06x |
| SHA-1 | 55,492 | 26,200 | 2.11x |
| XOR Cipher | 71,608 | 9,011 | 9.46x |

Table 9.1. Execution times and speedup of various algorithms

Table 9.1 presents the **recorded durations** of all the benchmarks, which were initially executed by the CPU and later by the LiMpire Accelerator. It is worth noting that the LiM execution results in a **speedup** ranging from **1.9x** to **10.06x**, depending on the algorithm's complexity, as bitwise operations are generally faster than multiplications, additions, and subtractions.

Furthermore, the proposed performance can be enhanced through the development of a dedicated **LiMpire Compiler**, which can perform data location optimization based on the algorithm, thereby further reducing fetch times.

# Chapter 10

# Synthesis of LiMpire

**Logic synthesis** is the process of converting a high-level description of a **digital circuit** (typically written in a hardware description language, or HDL) into a gate-level representation that can be implemented in **physical hardware**, such as FPGAs (Field-Programmable Gate Arrays) or ASICs (Application-Specific Integrated Circuits). It is a crucial step in the digital design process, as it automates the creation of optimized digital logic from the designer's specification. The entire process was conducted using the **Synopsys Design Compiler**.

## 10.1 SAED EDK14 FinFET Overview

In the domain of logic synthesis, **libraries** are essential for defining the available **building blocks** for circuit creation. These libraries consist of predefined **logic gates** and other components utilized by synthesis tools to translate high-level circuit descriptions into gate-level representations. The careful selection and optimization of these gates, oriented toward specific technology and design requirements, are crucial for achieving the desired performance, power consumption, and area characteristics of the final hardware.

The **SAED EDK14 FinFET** library is specifically designed for digital integrated circuit development utilizing **14nm FinFET** technology. It provides advanced features for creating high-performance, low-power, and highly scalable circuits, making it ideal for applications in mobile devices, high-performance computing, IoT, and automotive systems.

The key **advantages** of selecting the SAED EDK14 FinFET library include:

1. **Power Efficiency**: FinFET transistors significantly reduce leakage currents and operate at lower supply voltages, minimizing both **static** and **dynamic power consumption**.

2. **High Performance**: Enhanced switching speeds and reduced parasitic effects contribute to improved **performance** and **faster clock rates**.

3. **Scalability**: The 14nm process technology allows for a higher density of transistors, making it suitable for **more complex** integrated designs.

4. **Leakage Reduction**: Better control over short-channel effects leads to decreased leakage currents, enhancing **power efficiency**.

5. **Reliability**: FinFET technology provides superior control over device variability, ensuring consistent performance and better **threshold voltage control**.

6. **Advanced Tool Support**: This library is compatible with widely used EDA tools, facilitating easy integration into existing digital design workflows.

7. **Customization**: It supports **multi-Vt** options and **low-power** design techniques, enabling optimization for power or performance according to application requirements.

8. **Design for Manufacturability (DFM)**: Incorporates DFM rules that enhance yield and manufacturability.

These factors contributed to the decision to use it for synthesizing the LiMpire Architecture.

## 10.2 Hardware Modifications for Synthesis

Despite the numerous advantages mentioned in Section 10.1, the selected library does not include a **Memory Compiler**. Consequently, it was essential to implement some **hardware modifications** by replacing the Memory Arrays (found in both the Banks and Instruction Memory) with existing SRAMs.

To include all possible implementations, three different synthesis **strategies** were executed:

1. Without substituting the Arrays with SRAMs, treating the memory cells as **Registers**.

2. By replacing the Memory Blocks as follows:

   - Instruction Memory: two **512x32 bits** Single Read Port and Single Write Port **SRAMs**.
   - Memory Array: two **32x16 bits** Double Read Ports and Double Write Ports **SRAMs**.

3. By replacing the Memory Blocks as follows:

   - Instruction Memory: two **512x32 bits** Single Read Port and Single Write Port **SRAMs**.
   - Memory Array: one **32x32 bits** Double Read Ports and Double Write Ports **SRAM**.

The SAED EDK14 FinFET Library offers **multiple** types of **SRAMs**, varying based on the number of Reading and Writing Ports and the dimensions concerning word count and bits per word. The only blocks that needed to be synthesized as Memories are the Instruction Memory and the Memory Array for each Bank.

### 10.2.1 SRAM1RW512x32 Overview

The Instruction Memory has a width of **1024 x 32 bits**. To implement this memory, the largest SRAM available in the library is the **SRAM1RW512x32**, which required the use of two such units.

#### SRAM1RW512x32 Ports Layout

This Memory Block is characterized by the following ports:

- `A`: A 10-bit port representing the input **address** of the Memory.
- `CE`: This signal serves as the Input **Clock** for the Memory.
- `WEB`: This signal indicates the **Write Enable**, which is Active Low.
- `OEB`: This signal is the **Output Enable**, allowing the memory to output data for reading. It is also Active Low.

- `CSB`: This port **enables** the Memory Block and is active low, being set to 0.

- `I`: This 32-bit data represents the **input data** for writing.

- `O`: This 32-bit data represents the **output data** for reading.

The Ports Layout can be visualized as follows:

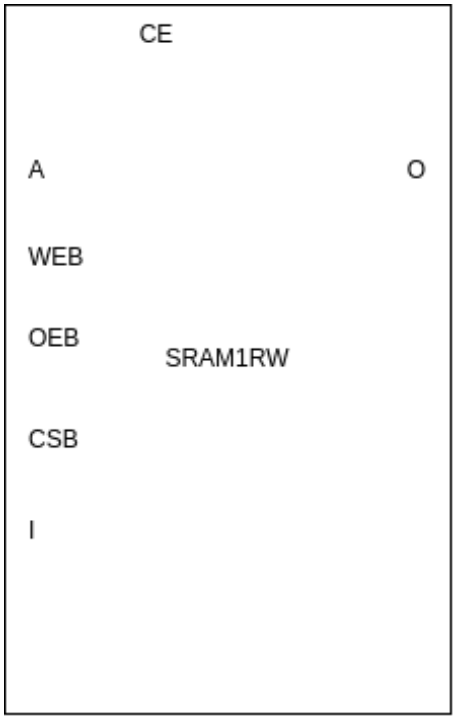


Figure 10.1. Single Port SRAM Layout

### SRAM1RW512x32 Timing Diagrams

The Timing Diagrams for this Memory Block can be observed in the following figures:

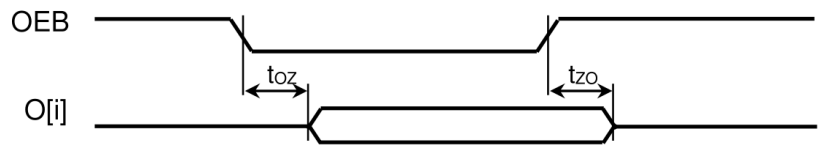


Figure 10.2. Single Port SRAM Output-Enable Timing Waveforms

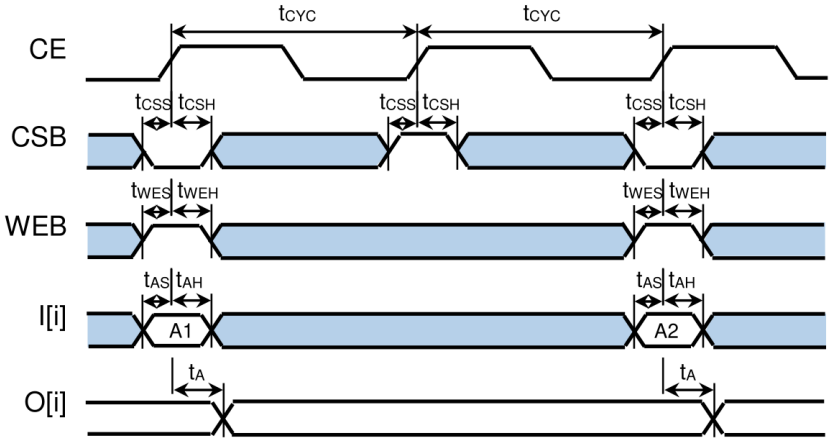


Figure 10.3. Single Port SRAM Read-Cycle Timing Waveforms

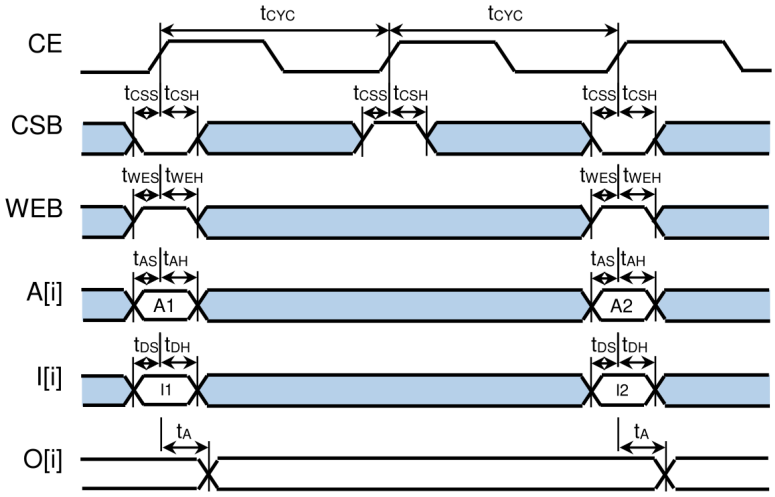


Figure 10.4. Single Port SRAM Write-Cycle Timing Waveforms

Where:

- $t_{OZ}$ is the time from `OE` to high-impedance (hi-Z);
- $t_{ZO}$ is the time during which `OE` is active;
- $t_{CYC}$ is the Cycle Time;
- $t_{CSS}$ is the Setup Time for `CSB`;
- $t_{CSH}$ is the Hold Time for `CSB`;
- $t_{WES}$ is the Setup Time for `WEB`;
- $t_{WEH}$ is the Hold Time for `WEB`;
- $t_{AS}$ is the Setup Time for `A`;
- $t_{AH}$ is the Hold Time for `A`;
- $t_{DS}$ is the Setup Time for `I`;
- $t_{DH}$ is the Hold Time for `I`;
- $t_A$ is the Access Time.

### SRAM1RW512x32 Implementation in Instruction Memory

The **Instruction Memory** consists of two distinct `SRAM1RW512x32` elements: one that contains the **Odd Memory Addresses** (e.g., 1, 3, 5, ...) and another that contains the **Even Memory Addresses** (e.g., 0, 2, 4, ...). When the input address is provided to the Instruction Memory, it determines whether the address is Even or Odd and computes the offset as follows:

$$\texttt{EVEN\_ADDR} = \texttt{ADDR} >> 1$$

$$\texttt{ODD\_ADDR} = (\texttt{ADDR} >> 1) + 1$$

where `EVEN_ADDR` is the input address for the Even SRAM and `ODD_ADDR` is the input address for the Odd SRAM. This distinction is utilized only during a Write operation; for reading, the Internal Counter Content is employed as the input address for both SRAMs.

This behavior can be depicted as follows:

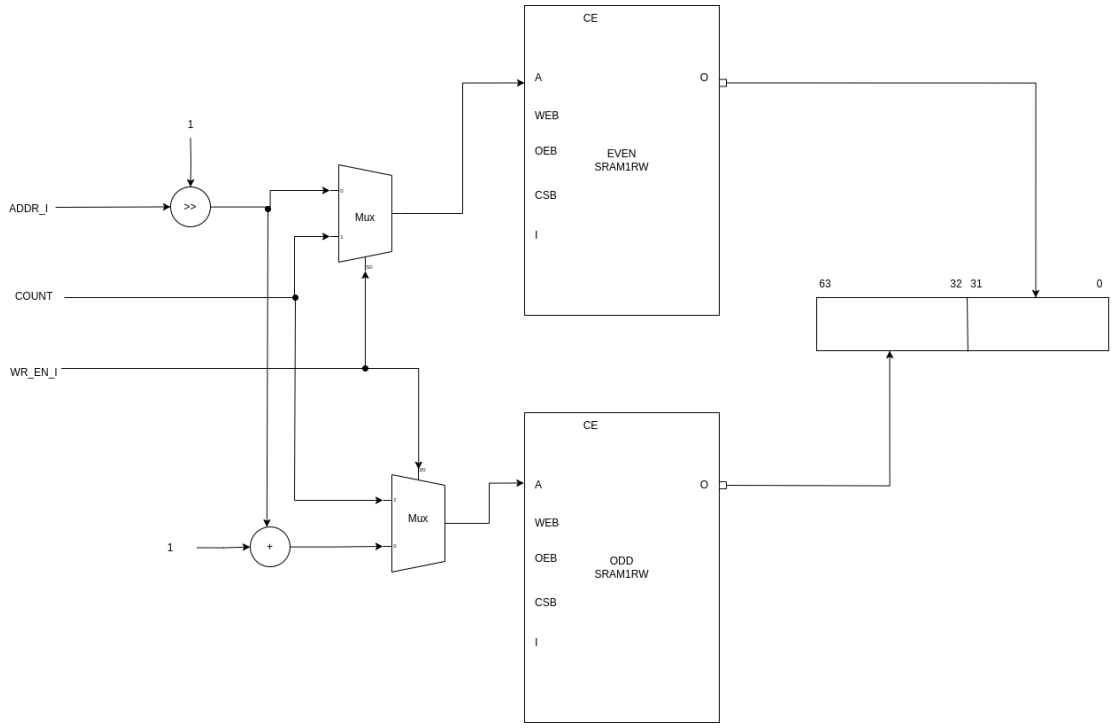


Figure 10.5. Instruction Memory with SRAM1RW Ports Layout

This design choice was made to allow the **concatenation** of two memory words to form a complete **64-bit Instruction Word**. Therefore, the Odd SRAM provides the Most Significant 32 Bits, while the Even SRAM supplies the Least Significant 32 bits (as illustrated in Fig. 10.5).

### 10.2.2 SRAM2RW32x32 and SRAM2RW32x16 Overview

The Memory Array consists of 32 words, each 32 bits wide, and includes two Reading Ports. Due to the dual-port nature, it is not possible to use the same SRAM as in the Instruction Memory. Therefore, the `SRAM2RWnxm` model is required for this design.

#### SRAM2RWnxm Ports Layout

This component is characterized by the following ports:

- `Ax`: An 8-bit port representing the input **address** for Reading (or Writing) from Port x.

- `CEx`: This signal serves as the Input **Clock** for Reading (or Writing) from Port x.
- `WEBx`: This signal is the **Write Enable** for writing from Port x and is Active Low.
- `OEBx`: This signal is the **Output Enable**, which allows the memory to provide data for reading from Port x, and it is Active Low.
- `CSBx`: This port **enables** the Memory Block and restricts operations to the x-th port for Reading and Writing. It is Active Low and is set to 0.
- `Ix`: A 32-bit **data input** for writing from Port x.
- `Ox`: A 32-bit **data output** for reading from Port x.

The layout of the component can be seen as:

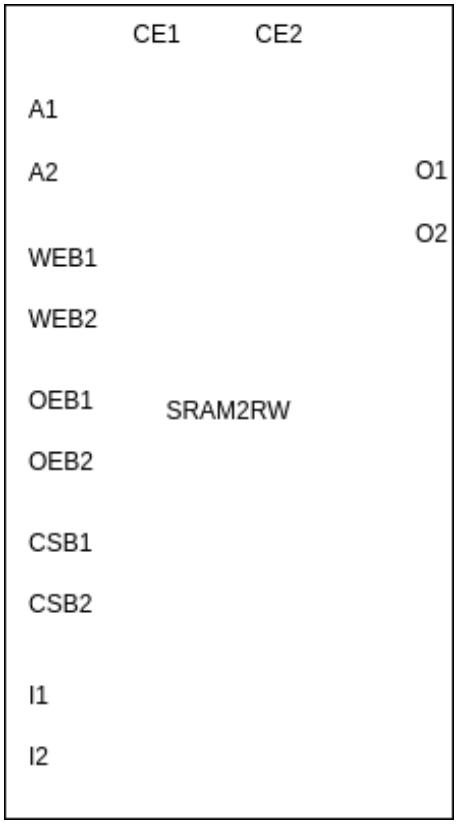


Figure 10.6. Dual-Port SRAM Layout

### SRAM2RWnxm Timing Diagrams

The Timing Diagrams for both the 32x16 and 32x32 RAMs are as follows:

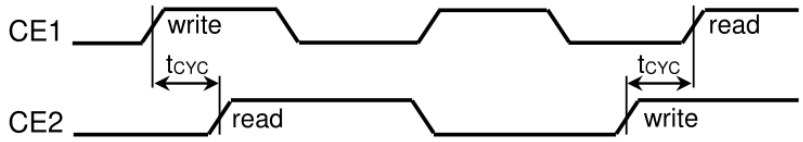


Figure 10.7. Dual-Port SRAM Write-Read Clock Timing Waveforms

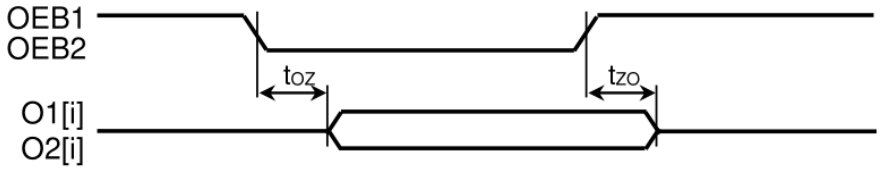


Figure 10.8. Dual-Port SRAM Output-Enable Timing Waveforms

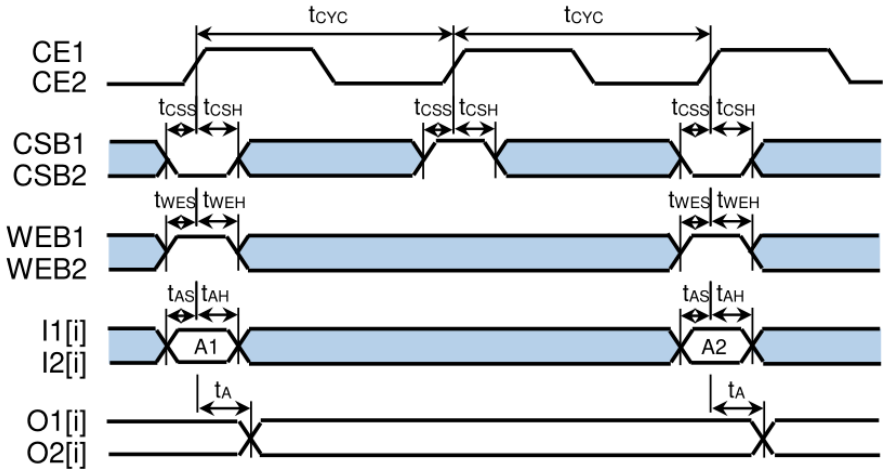


Figure 10.9. Dual-Port SRAM Read-Cycle Timing Waveforms

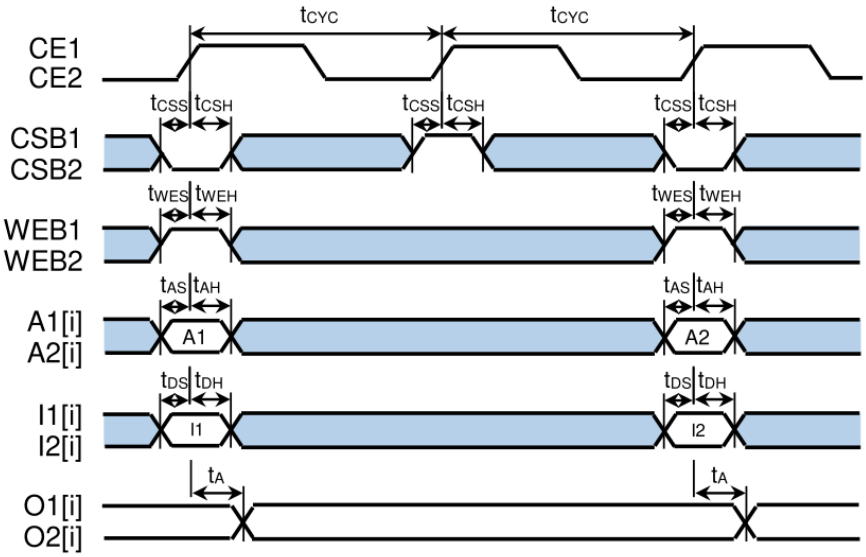


Figure 10.10. Dual-Port SRAM Write-Cycle Timing Waveforms

Where:

- $t_{OZ}$ is the time from `OE` to high-impedance (hi-Z);
- $t_{ZO}$ is the time while `OE` is active;
- $t_{CYC}$ is the Cycle Time;
- $t_{CSS}$ is the Setup Time of `CSB`;
- $t_{CSH}$ is the Hold Time of `CSB`;
- $t_{WES}$ is the Setup Time of `WEB`;
- $t_{WEH}$ is the Hold Time of `WEB`;
- $t_{AS}$ is the Setup Time of `A`;
- $t_{AH}$ is the Hold Time of `A`;
- $t_{DS}$ is the Setup Time of `I`;
- $t_{DH}$ is the Hold Time of `I`;
- $t_A$ is the Access Time.

### SRAM2RWnxm Implementations in Memory Arrays

One feature of the Accelerator is the ability to **overwrite** only **16 bits** of the word at the destination address rather than the full 32 bits. Unfortunately, the `SRAM2RW32x32` does not have any Byte Enable port, making partial bit overwriting impossible. One solution involves using a **masking system**, but this wastes clock cycles due to additional operations and memory words, as the intermediate results must be stored, potentially overwriting data needed elsewhere.

Alternatively, the Technology Library provides a **32x16 bits SRAM**, which is nearly identical to the 32x32 model and can be used similarly to the Instruction Memory. One SRAM stores the **higher 16 bits** of the Data Word, while another stores the **lower 16 bits**. This setup can be visualized as:

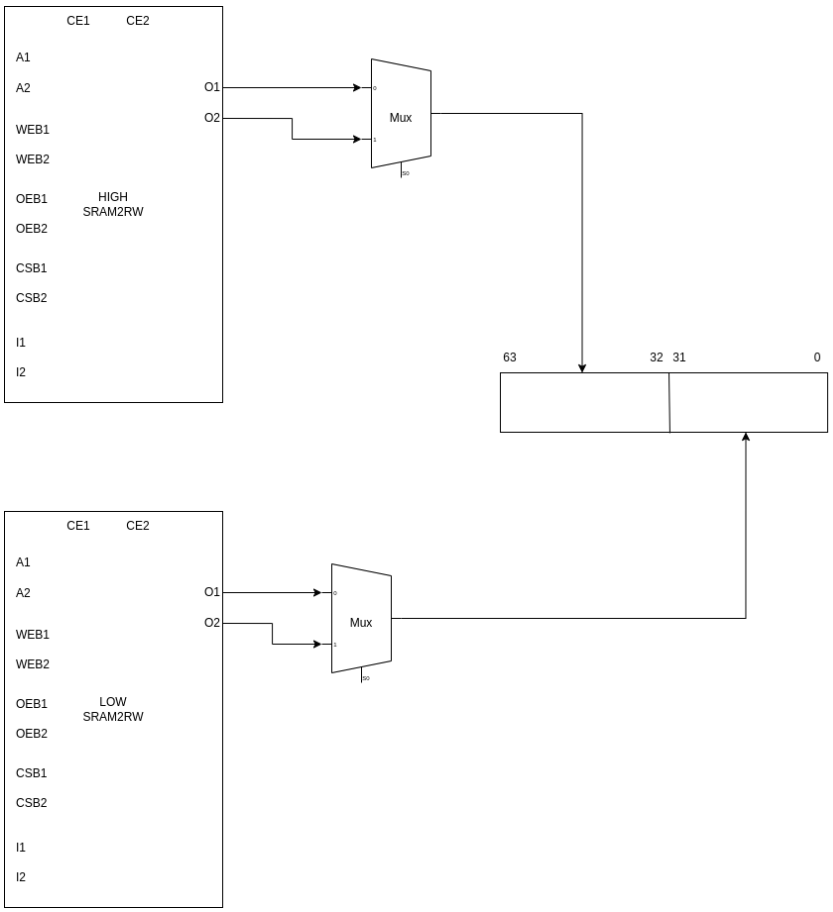


Figure 10.11. Memory Array with `SRAM2RW` Ports Layout

In this layout, the **selection** between the two Output Ports is based on which Enable is active—`RD0` for Port 1 and `RD1` for Port 2.

When **writing**, there are two cases:

- If `WR_ENC` is "1", the Memory Array Entity checks the `HIGH_LOW` signal: if set, data is written to the **High SRAM** (which is enabled); otherwise, it is written to the **Low SRAM** (which is enabled). In this case, the address is provided to the desired port of the enabled bank.

- If `WR_ENC` is "0", both Memory Arrays are written **simultaneously** with the 16 MSBs and 16 LSBs of the Data Word. Both SRAMs are enabled, and the address is provided to both.

When reading, both SRAMs are enabled, and the input address is provided to both blocks.

## 10.3 Synthesis Flow and Results

The entire synthesis process was automated using **scripts** to speed up the procedure and avoid errors. The first step involves setting up the `.synopsys_dc.setup` file to include paths to all necessary libraries. The resulting setup file is as follows:

```
1  define_design_lib WORK -path ./work
2
3  #define the path to the directories where libraries are
       stored
4  set search_path [list .
5  /eda/synopsys/2022-23/RHELx86/SYN_2022.12/libraries/syn
6  /eda/dk/SAED14nm/stdcell_lvt/db_nldm
7  /eda/dk/SAED14nm/SAED14nm_EDK_SRAM_v_05072020/lib/sram/
       logic_synth/single
8  /eda/dk/SAED14nm/SAED14nm_EDK_SRAM_v_05072020/lib/sram/
       logic_synth/dual]
9
10 #set the library names
11 set link_library [list "*" "saed14lvt_ff0p88v25c.db" "
       saed14sram_ff0p88v25c.db"
12 "/eda/dk/SAED14nm/SAED14nm_EDK_SRAM_v_05072020/lib/sram/
       logic_synth/dual/saed14sram_tt0p8vm40c.db" "dw_foundation.
       sldb"]
13
```

```
14 set target_library [list "saed14lvt_ff0p88v25c.db" "
      saed14sram_ff0p88v25c.db" "/eda/dk/SAED14nm/
      SAED14nm_EDK_SRAM_v_05072020/lib/sram/logic_synth/dual/
      saed14sram_tt0p8vm40c.db"]
15 set synthetic_library [list "dw_foundation.sldb"]
```

Once the DC setup file is configured, the next step is to **set up the environment** for the Synopsys Design Compiler and run the **analysis**, **compilation**, and **synthesis** processes. These steps have been automated using two TCL scripts, one for each task. This process was repeated for three different synthesis strategies at a frequency of 100 MHz to find the maximum frequency, which is 235.29 MHz.

The synthesis generated several key outputs:

- **Analysis, Elaboration, and Synthesis Reports**: These detail any errors encountered during the process.
- **Timing Report**: This provides critical path information, maximum operating frequency, and any timing violations.
- **Area Report**: This calculates the total cell area used in the design and provides a hierarchical area breakdown.
- **Resources Report**: This breaks down the design resources (e.g., logic cells, flip-flops, memory blocks) by design hierarchy.
- **Netlist Timing Information** (`.sdf`): This file details netlist delays for timing verification during simulation, allowing accurate behavior simulation with real timing characteristics.
- **Design Constraints** (`.sdc`): This file is essential for static timing analysis tools to evaluate timing paths and ensure the design meets performance requirements.
- **Verilog Synthesized Netlist**: This contains the Verilog representation of the netlist for further simulation or as input to place-and-route tools.

This procedure was performed twice: once defining a variable named `OLOAD` to hold the load capacitance value of a specific output buffer and apply this value to all output pins, and once without defining this parameter. The relevant command is:

```
1 #set the load of each output
2 set OLOAD [load_of saed14lvt_ff0p88v25c/SAEDLVT14_BUF_4/X]
3 #set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
4 set_load $OLOAD [all_outputs]
```

The synthesis results are summarized in the following table:

| **Strategy** | **Total Area ($um^2$)** | **Total Cell Area ($um^2$)** | **Frequency (MHz)** |
|---|---|---|---|
| **Without Load on Output Pins** | | | |
| Logic Cells | 297954.850150 | 72975.707484 | 100 |
| 2x16bits SRAM | 89648.854220 | 52481.787392 | 100 |
| 1x32bits SRAM | 83401.807173 | 46506.959459 | 100 |
| Logic Cells | 299393.609121 | 73379.303489 | 235.29 |
| 2x16bits SRAM | 89732.732745 | 52629.639390 | 235.29 |
| 1x32bits SRAM | 83605.818019 | 46682.739058 | 235.29 |
| **With Load on Output Pins** | | | |
| Logic Cells | 297954.850150 | 72975.707484 | 100 |
| 2x16bits SRAM | 89648.854220 | 52481.787392 | 100 |
| 1x32bits SRAM | 83401.807173 | 46506.959459 | 100 |
| Logic Cells | 299393.609121 | 73379.303489 | 235.29 |
| 2x16bits SRAM | 89732.732745 | 52629.639390 | 235.29 |
| 1x32bits SRAM | 83605.818019 | 46682.739058 | 235.29 |

Table 10.1. Area and frequency data for different synthesis strategies

From Table 10.1, we can see that despite the complexity of the developed model, it maintains a **compact area footprint** and achieves a **high operating frequency**. This data was recorded using a standard synthesis library based on common technologies. Using a **FeFET** or **MTJ**-based library would likely yield more accurate data, improved performance, and a smaller area footprint.

# Chapter 11

# Conclusions

In Chapter 5, we defined all the design specifications and outlined the primary objective of the architecture: to create a **high-level architectural model** for Logic in Memory that facilitates the execution of multiple algorithms for various purposes while incorporating many features found in existing architectures.

The proposed architecture **adheres** to all previously established specifications and objectives by providing a **fully operational model** that can be utilized and tailored to fit the specific algorithm being executed. As a result, the architecture can serve **diverse purposes** and **algorithms**, delivering performance advantages that surpass those of conventional CPU execution. This is attributed to its capability to perform a wide range of operations based on the number of operands and bits per operand, allowing for various types of computations, including:

- **One Operand** Computation;
- **Two Operands** Computation;
- **Three Operands** Computation;
- **Operands Range** Computation (up to eight operands per instruction);
- **Parallel** Computation;
- **Partial and Full Word** Computation.

These features enable an advanced and optimized **compiler** to potentially streamline workloads for both the CPU and memory, starting from a single file (potentially in `.c`). This facilitates proper **data allocation**

for each instruction and operation, delineating tasks for the CPU and the LiMpire Accelerator. Consequently, this approach simplifies the benchmark code writing process, as users only need to create one file. Depending on which component executes a specific instruction, it can be translated into the appropriate language for either the LiM or the CPU.

Moreover, users can **fully configure** the design depending on the complexity, which further simplifies the compiler's task in relation to the current architectural setup.

This strategy enhances **computation parallelism**, allowing the CPU to operate independently while memory processes additional data, particularly in **complex domains** like Machine Learning and Artificial Intelligence, where computation is crucial.

Additionally, the **synthesis results** presented in Chapter 10.3 underscore the compact footprint of the architecture, despite its complexity and the absence of suitable libraries for further optimization of area and power consumption. Unfortunately, the synthesis data available are merely **rough estimates** of the actual data that might be obtained with an appropriate technology library. Therefore, these figures should be viewed as a **preliminary reference** for future research and implementation efforts.

# Chapter 12

# Future Works

In the previous chapters, it was mentioned that the proposed design was developed to comply with most of the key features of the implementations described in Chapter 3 and to support a wide range of algorithms that can be easily executed on this system. To achieve this, the design may include some unused blocks or components, depending on the chosen algorithm and goal, which can potentially be eliminated. Therefore, the purpose of this chapter is to propose future works as a **set of tasks** to properly modify and expand the developed architecture.

## 12.1 Task I - Datapath Modifications

It was noted that the design approach is based on the **worst-case scenario**, where operands might not be located in the same bank, thus needing six memory banks. However, the datapath presents a parametric structure, allowing **customization** with a relatively high degree of freedom by adding or removing banks. Furthermore, some signals have been retained for legacy purposes, but they are not used in the current implementation and can be **removed** or **repurposed**.

Additionally, memory **partial word management** can be optimized by using the standard `byte_enable` signal, allowing for a finer amount of bytes to be overwritten instead of half of the data word. If all operands are located in the same bank as the destination address, the **input register files** can potentially be removed, as their only purpose is to store data coming from other banks. Similar considerations apply to the **temporary register files**, which store temporary data if the instruction has more than two operands; if the target application only requires two operands per instruction, these

would never be used.

The **interfaces** can be improved by renaming some existing signals to align with the Open Bus Interface Standard, as they are already compliant. Other signals and ports can be merged to simplify the overall **control flow** of the architecture or expanded to enhance their functionalities. All these modifications can be easily implemented depending on the target application.

## 12.2 Task II - LiMpire Compiler Introduction

Another crucial task is to develop a proper **compiler** based on the LiMpire architectural design. This compiler would perform the following tasks:

1. **Read** a `.c` file containing the instructions and operations to perform;
2. **Optimize** the **instructions** to be executed by removing those not relevant for computation and merging others;
3. **Translate** these instructions into a language comprehensible to the **LiMpire** architecture, thus exploiting the instruction word layout;
4. **Optimize data location** based on operations to avoid (or reduce) operand fetch control blocks, ensuring that operands are already present in the destination bank. This allows the main task to be simply computing the results and storing them in the memory array;
5. **Regulate** and **synchronize** the LiMpire execution cycle in LiM mode to prevent the CPU from getting stuck while waiting for the accelerator to complete its execution cycle.

## 12.3 Task III - Control Flow Simplification

The **control unit** was designed to consider all possible cases based on:

1. **Data location** (the banks in which operands are located);
2. **Operations** to perform;
3. Amount of **operands**.

In some cases, certain **states** could potentially be **removed** due to optimal data placement or unused instructions or modalities (e.g., partial word computation). This task is closely related to the other tasks in Sections 12.1 and 12.2, as modifications to the datapath will affect the application, reducing the need to check for certain signals or include states that are never used, thereby simplifying the control flow. Additionally, with the introduction of the compiler, the control flow would be further reduced and simplified due to the optimization of data location and instructions performed by the compiler.

## 12.4 Task IV - LiMpire Assembly Expansion

Another potential improvement involves modifying the **LiMpire assembly language**. The proposed language is relatively simple and does not currently support **selection** statements (e.g., if-else) or **loops** (e.g., while). These features can be added by expanding the capabilities of the **Python script** used for compilation and/or through the introduction of the compiler described in Section 12.2. However, implementing these additional statements requires a significant increase in the complexity of the compiler, leading to an extended design and implementation timeline.

Furthermore, **additional fields** in the instruction word could be incorporated, such as extra bits for replacing specific bits instead of two at a time in partial word computation.

## 12.5 Task V - Error Correction Code (ECC) Implementation

**Error Correction Code** (ECC) is a technique for detecting and correcting errors in data transmission or storage. It adds redundant parity bits to the original data, enabling systems to detect and correct errors during transmission or retrieval. Common types of errors include single-bit, multiple-bit, and burst errors. Techniques such as **Hamming Code** (for single-bit errors), **Reed-Solomon Code** (for burst errors in CDs/DVDs), **BCH Code** (for multiple errors in QR codes), and **LDPC** (used in modern communications like 5G) are widely employed. ECC is applied in areas like ECC RAM (for reliable memory), data storage (SSDs, hard drives), and wireless communication to ensure data integrity.

The key **advantages** of ECC include improved data reliability, error detection and correction, and system stability. However, it introduces overhead (additional data) and increases system complexity, especially in more advanced algorithms.

Initially, this technique was implemented in the LiMpire datapath through the introduction of an **encryption** and a **decryption block** inside the memory banks. The former received the input data (for a store) or the final result (for a LiM operation), encoded the word, and wrote it into the memory array. The latter was connected to the read ports of the memory array, whose output had to be forwarded to another bank (for a LiM operation) or to the `WB` register (for a load), and its job consisted of decrypting the output data from the array. However, these blocks were removed due to the lack of time to properly research the approaches and theories necessary to implement this technique.

# Appendix A

# Appendix

## A.1 Datapaths

This section contains all the **diagrams** for the QuiGon Heep System and LiMpire Architecture. Each diagram illustrates the following:

1. **QuiGon Heep Wrapper** Top View Layout;

2. **LiMPire Top** View Layout;

3. **LiMpire Datapath** Layout - Part 1;

4. **LiMpire Datapath** Layout - Part 2;

5. **LiMPire Bank** Layout.

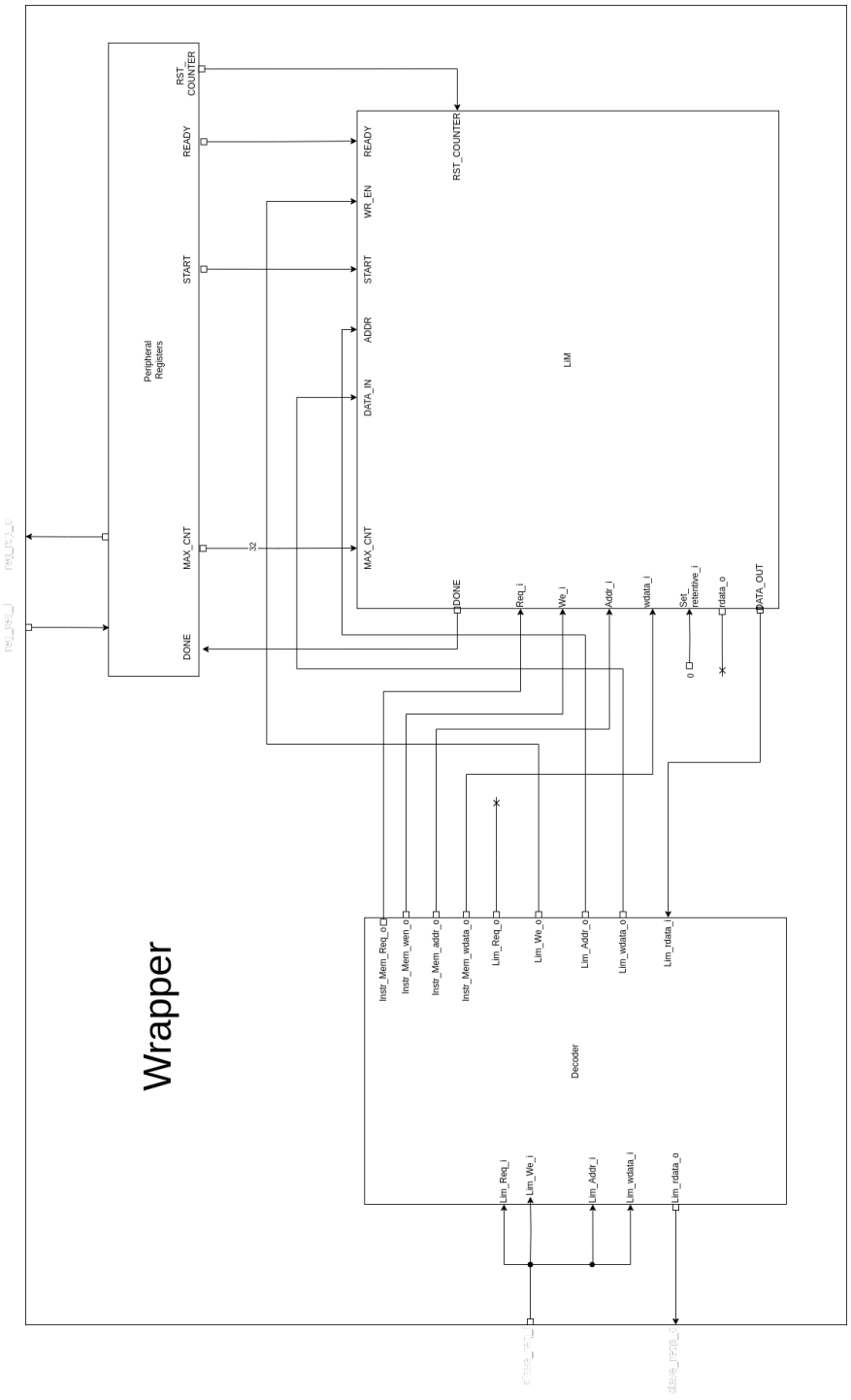


Figure A.1. QuiGon Heep Wrapper Top View Layout

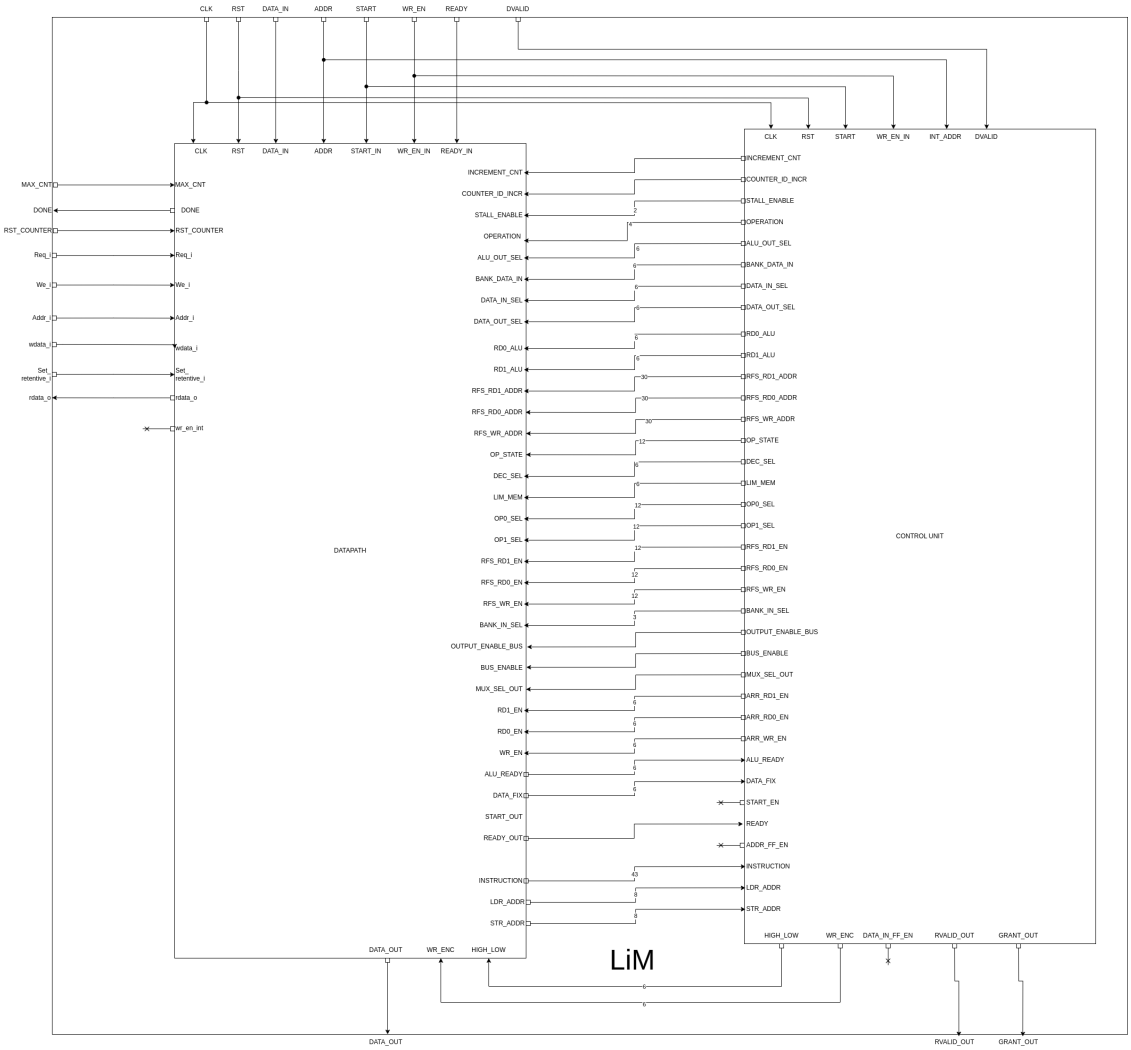


Figure A.2. LiMPire Top View Layout

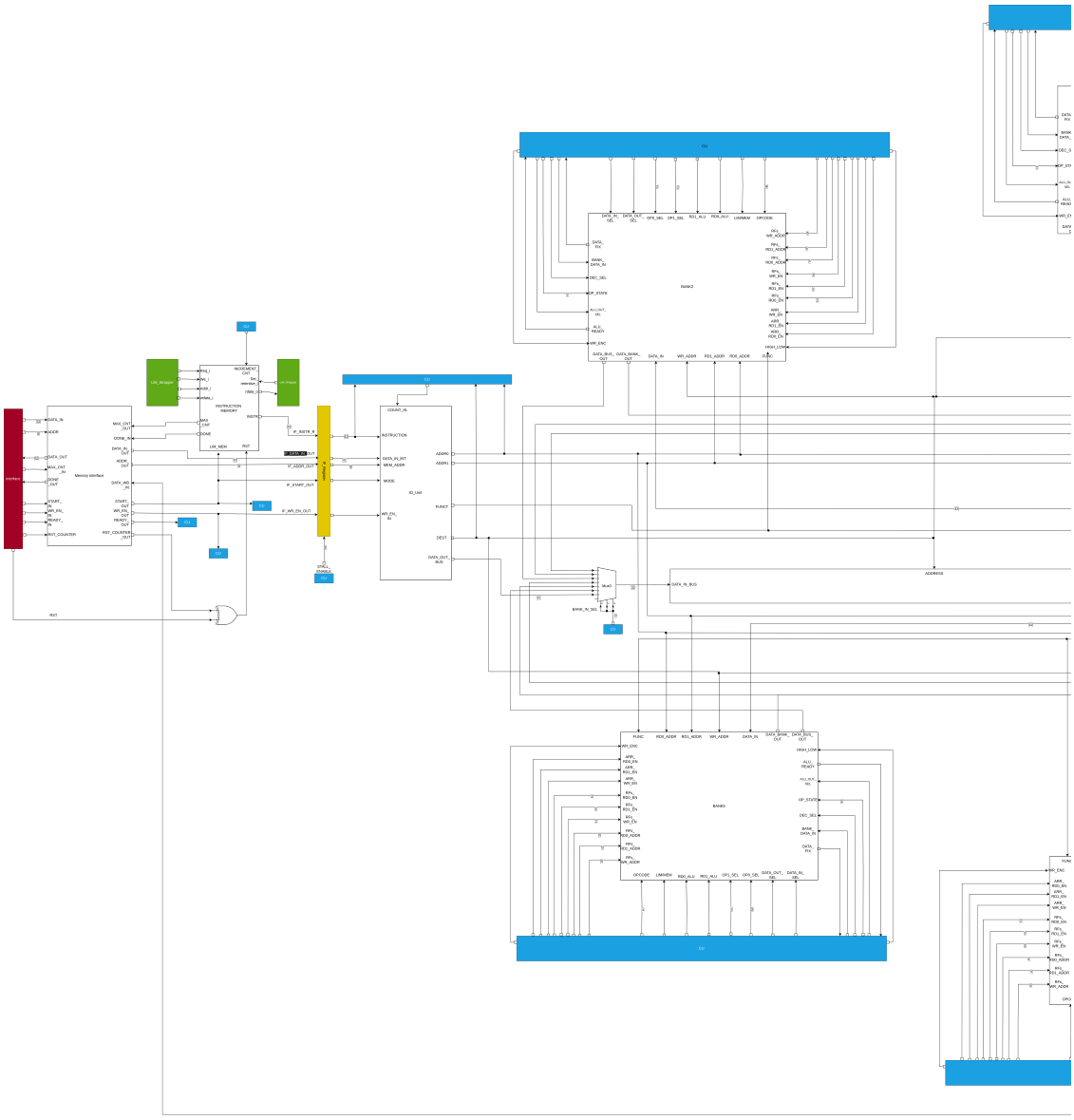

Figure A.3. LiMPire Datapath Layout - Part 1

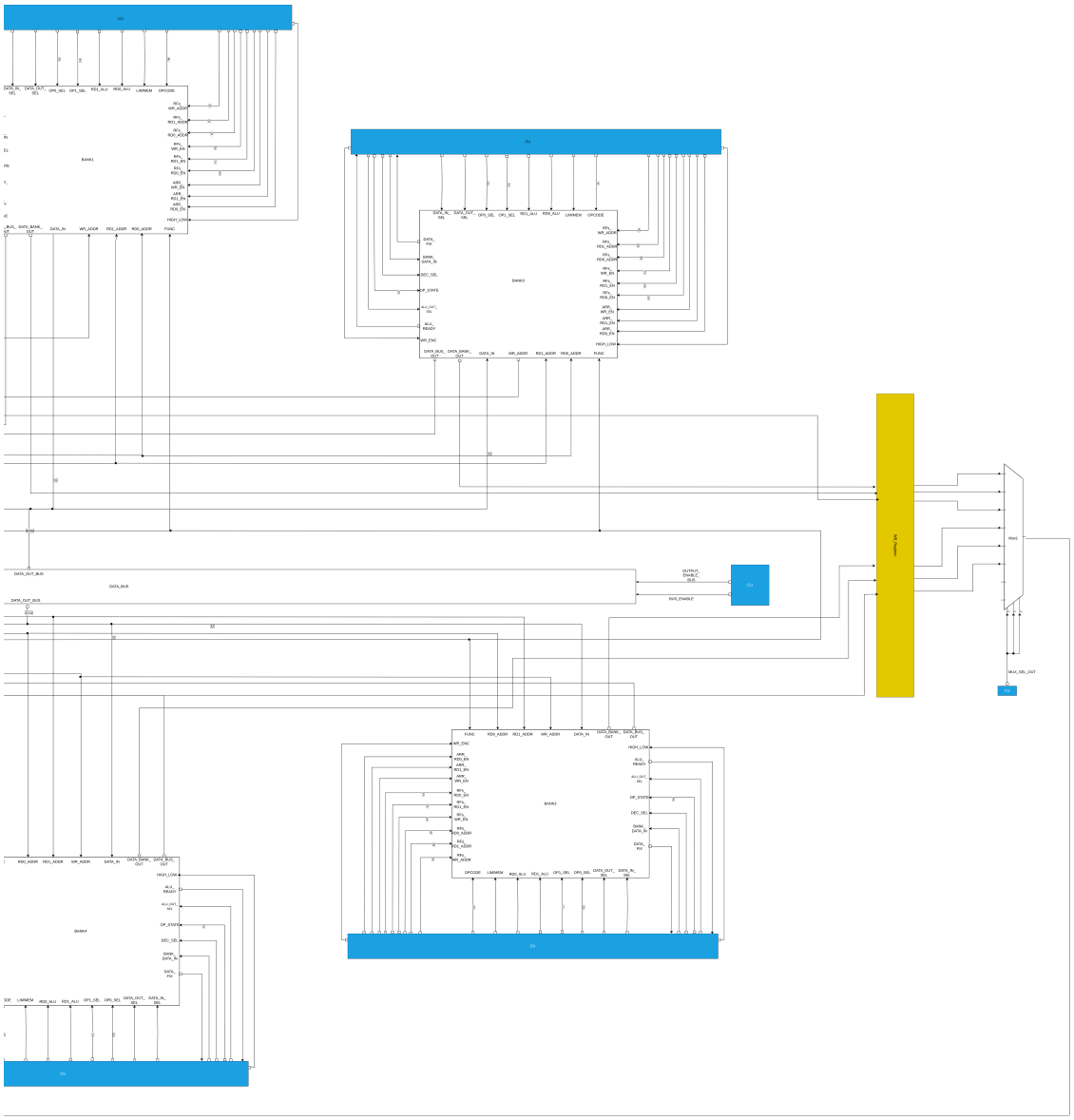

Figure A.4. LiMPire Datapath Layout - Part 2

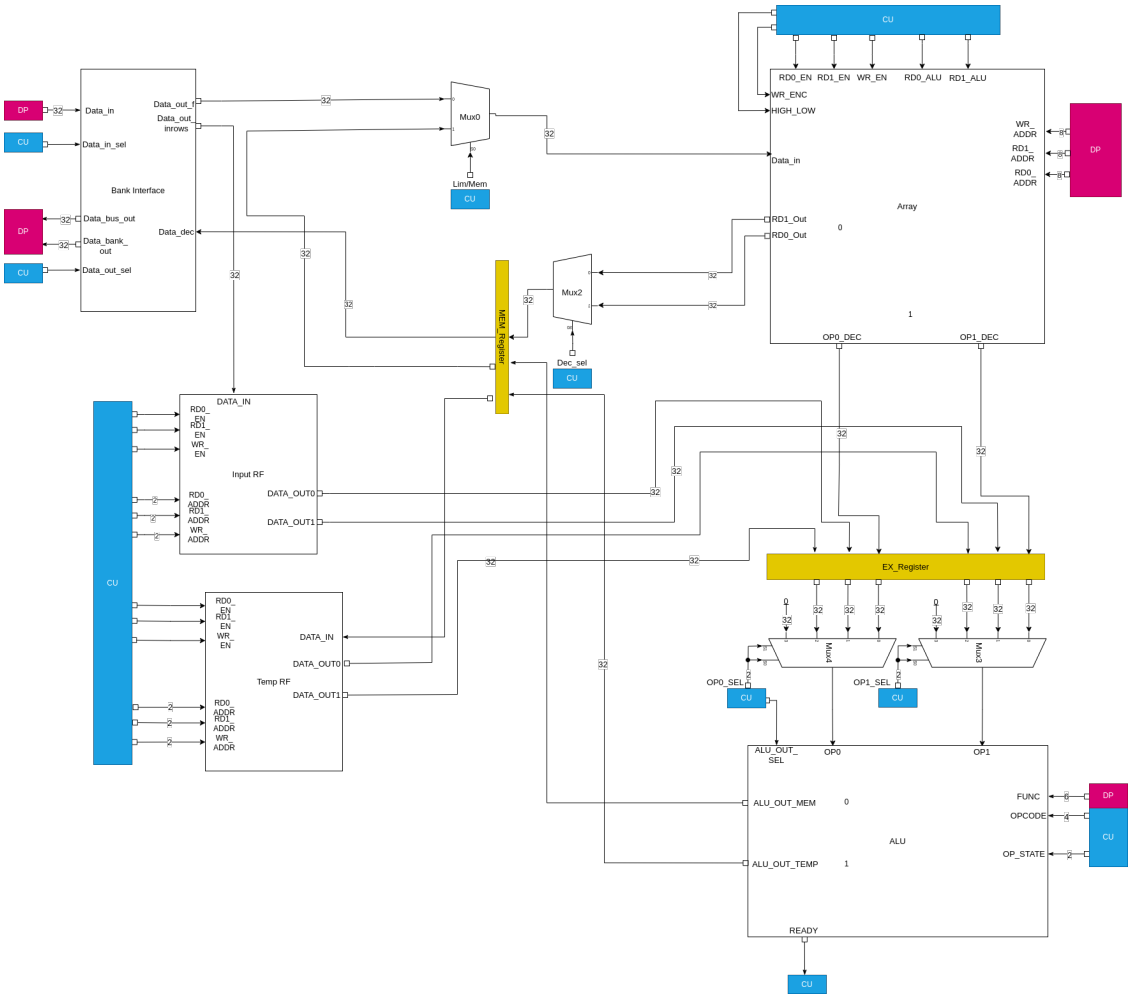


Figure A.5. LiMPire Bank Layout

## A.2 Control Unit Status Transition Flow

This section includes all the **flow charts** for the Control Unit. Each diagram addresses the following aspects:

1. **Basic** Flow;
2. Flow for **Three Operands** Instructions;
3. Flow for **Address Range** Instructions;
4. Flow for **Two Operands** Instructions.

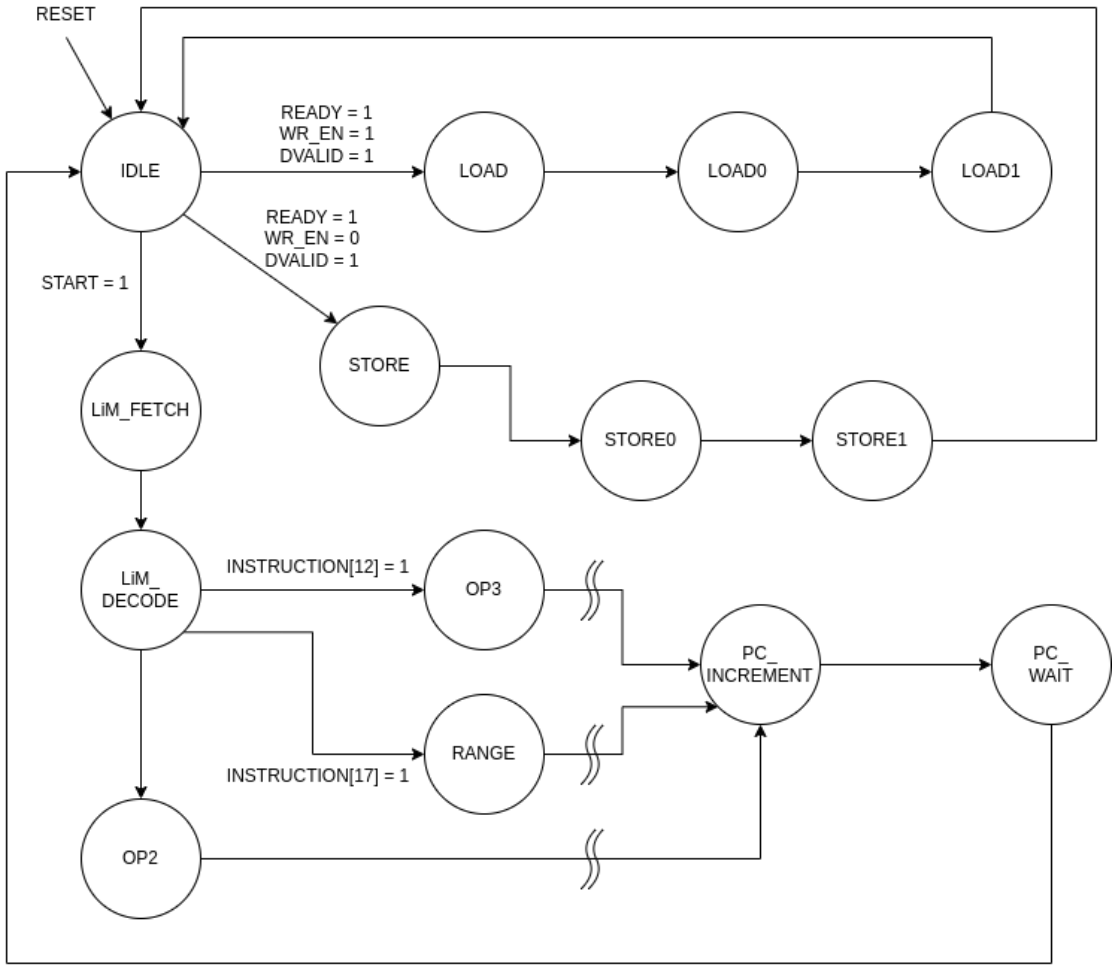


Figure A.6. Control Unit Basic Flow Chart

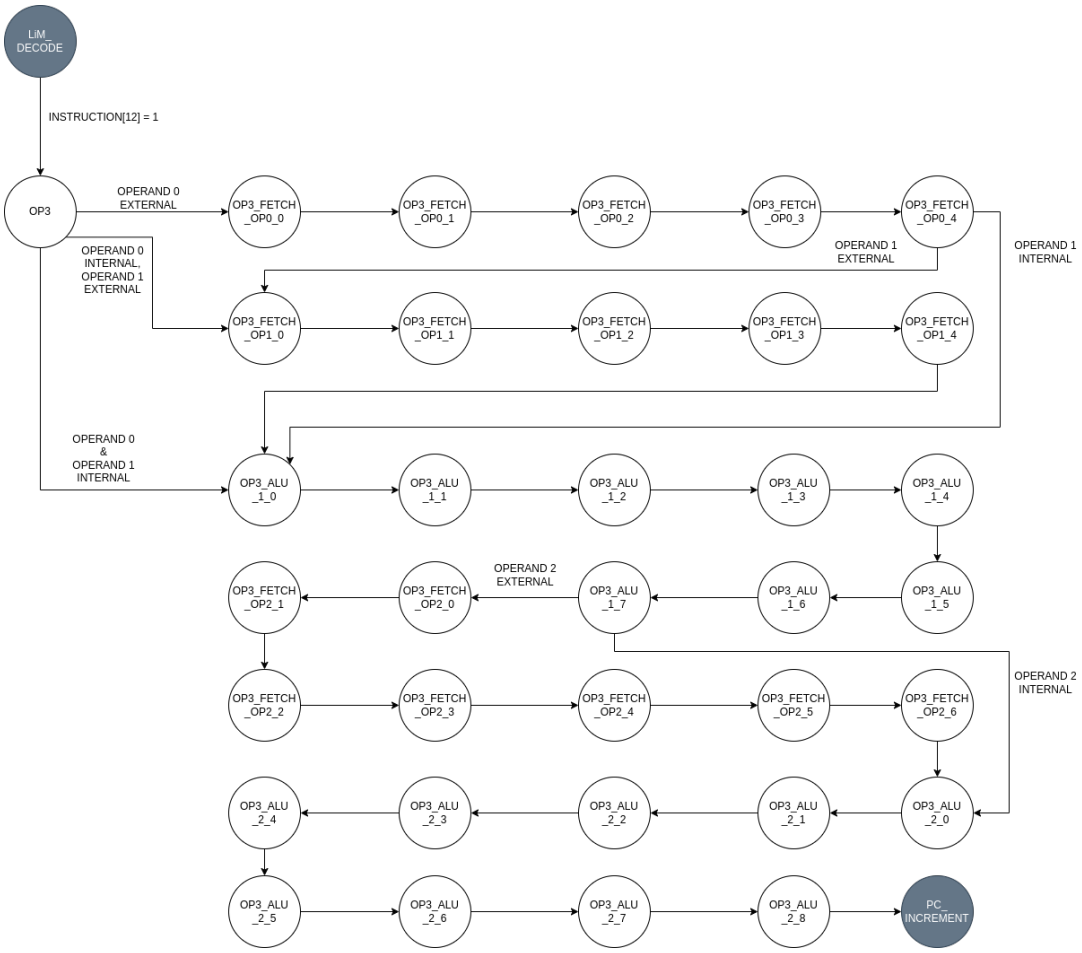


Figure A.7. Three Operands Instruction Flow Chart

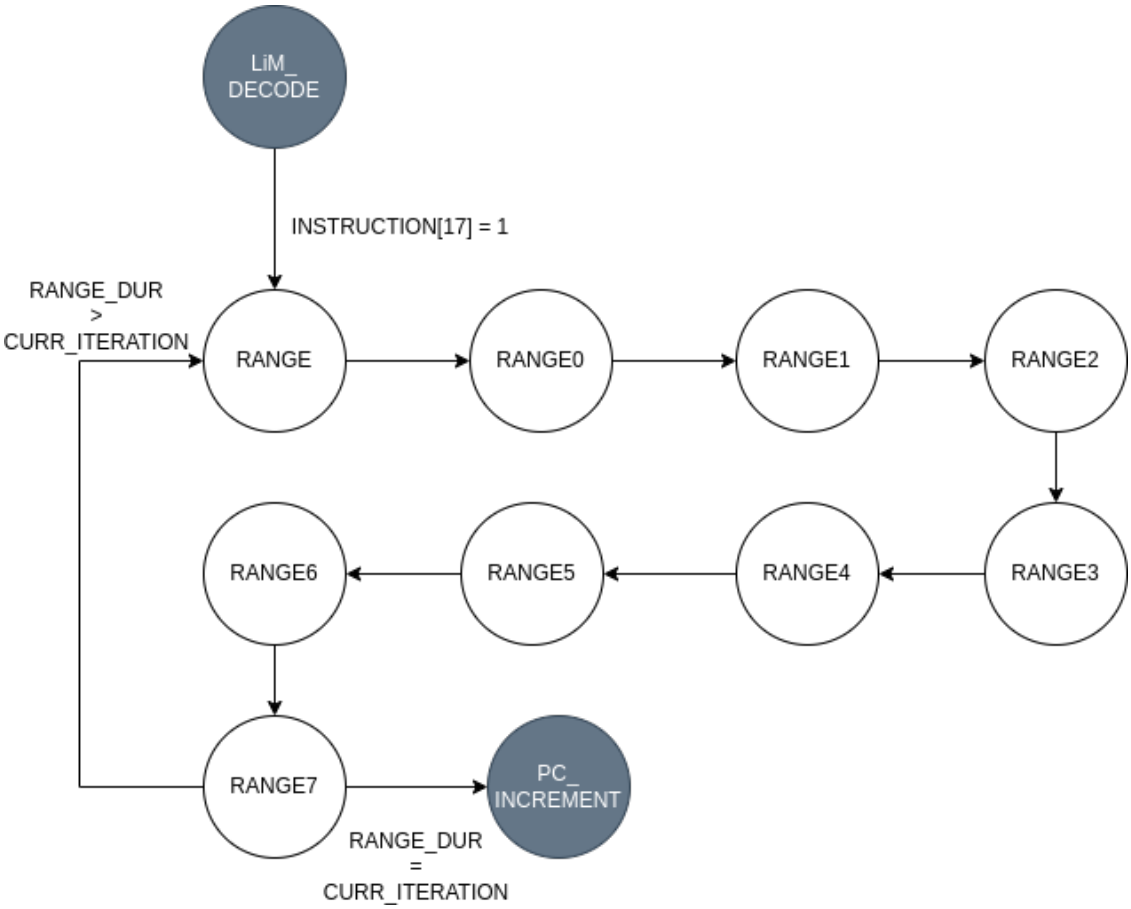


Figure A.8. Address Range Instruction Flow Chart

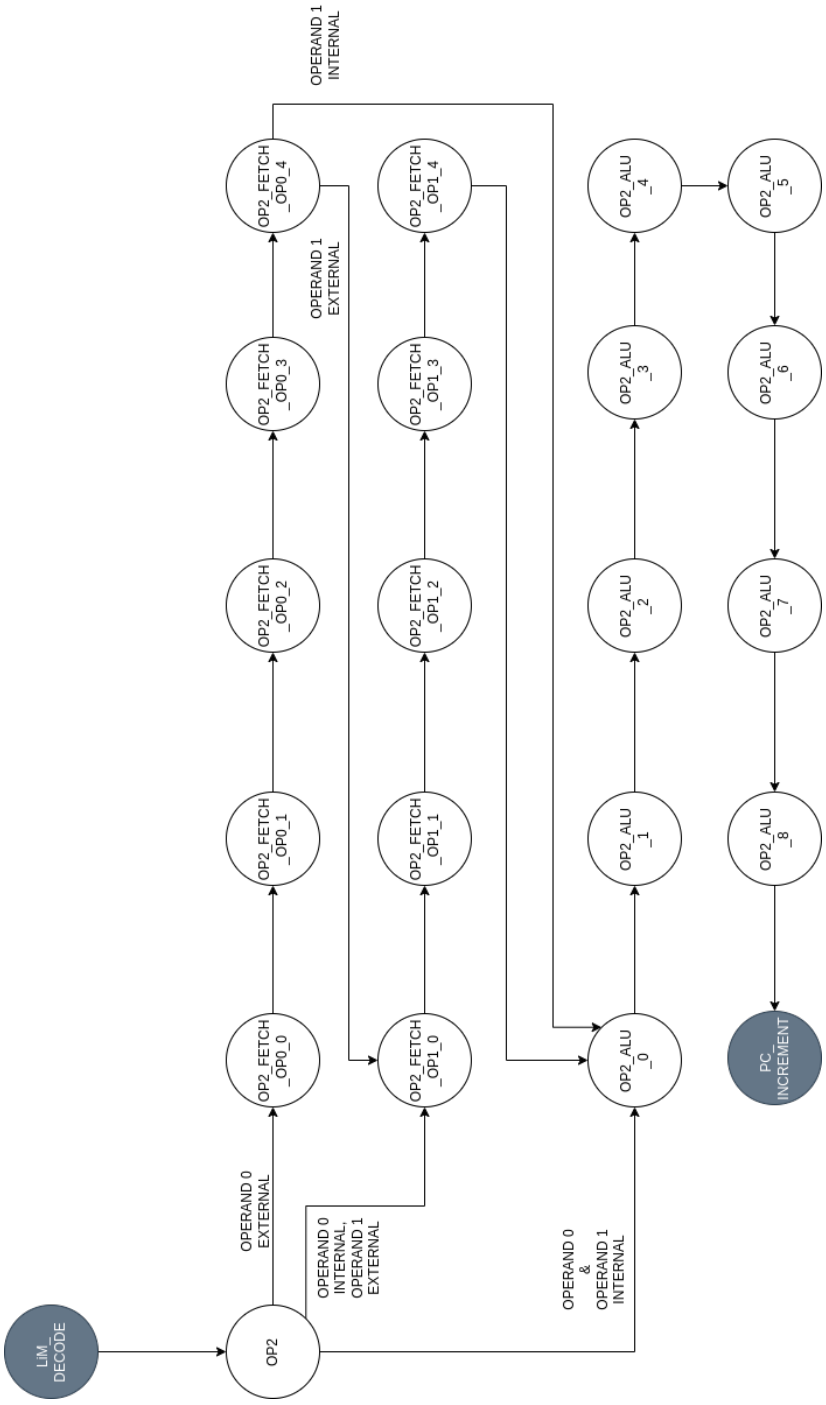


Figure A.9. Three Operands Instruction Flow Chart

# Bibliography


[1] Andrea Coluccio, Antonia Ieva, Fabrizio Riente, Massimo Ruo Roch, Marco Ottavi, and Marco Vacca. Risc-vlim, a risc-v framework for logic-in-memory architectures. *Electronics*, 11:2990, 09 2022. doi:10.3390/electronics11192990.

[2] Xiaoming Chen, Yuping Wu, and Yinhe Han. Fepim: Contention-free in-memory computing based on ferroelectric field-effect transistors. 2021. doi:10.1145/3394885.3431530.

[3] Xiaoyu Zhang, Xiaoming Chen, and Yinhe Han. Femat: Exploring in-memory processing in multifunctional fefet-based memory array. *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 11 2019. doi:10.1109/iccd46524.2019.00080.

[4] Dayane Reis, Michael Niemier, and X. Sharon Hu. Computing in memory with fefets. *Proceedings of the International Symposium on Low Power Electronics and Design*, 07 2018. doi:10.1145/3218603.3218640.

[5] Wenjun Tang, Mingyen Lee, Juejian Wu, Yixin Xu, Yao Yu, Yongpan Liu, Kai Ni, Yu Wang, Huazhong Yang, Vijaykrishnan Narayanan, and Xueqing Li. Fefet-based logic-in-memory supporting sa-free write-back and fully dynamic access with reduced bitline charging activity and recycled bitline charge. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 70:2398–2411, 06 2023. doi:10.1109/tcsi.2023.3251961.

[6] Rui Liu, Xiaoyu Zhang, Zhiwen Xie, Xinyu Wang, Zerun Li, Xiaoming Chen, Yinhe Han, and Minghua Tang. Fecrypto: Instruction set architecture for cryptographic algorithms based on fefet-based in-memory computing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 42:2889–2902, 09 2023. doi:10.1109/tcad.2022.3233736.

[7] Zhi Yang, Kuiqing He, Zeqing Zhang, Yao Lu, Zheng Li, Yijiao Wang, Zhaohao Wang, and Weisheng Zhao. A novel computing-in-memory

platform based on hybrid spintronic/cmos memory. *IEEE Transactions on Electron Devices*, 69:1698–1705, 04 2022. doi:10.1109/ted.2021.3137761.

[8] Bi Wu, Haonan Zhu, Dayane Reis, Zhaohao Wang, Ying Wang, Ke Chen, Weiqiang Liu, Fabrizio Lombardi, and Xiaobo Sharon Hu. An energy-efficient computing-in-memory (cim) scheme using field-free spin-orbit torque (sot) magnetic rams. *IEEE Transactions on Emerging Topics in Computing*, 11:331–342, 04 2023. doi:10.1109/tetc.2023.3237541.

[9] Rui Liu, Xiaoyu Zhang, Xiaoming Chen, Yinhe Han, and Minghua Tang. Femic: Multi-operands in-memory computing based on fefets. *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 01 2022. doi:10.1109/asp-dac52403.2022.9712498.

[10] Xiaoyu Zhang, Rui Liu, Tao Song, Yuxin Yang, Yinhe Han, and Xiaoming Chen. Re-femat: A reconfigurable multifunctional fefet-based memory architecture. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41:5071–5084, 11 2022. doi:10.1109/tcad.2021.3140194.

[11] Shubham Jain, Ashish Ranjan, Kaushik Roy, and Anand Raghunathan. Computing in memory with spin-transfer torque magnetic ram. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 26:470–483, 03 2018. doi:10.1109/tvlsi.2017.2776954.

[12] 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE). *GraphS: a Graph Processing Accelerator Leveraging SOT-MRAM*. IEEE, 03 2019.

[13] Taehwan Kim, Yunho Jang, Min-Gu Kang, Byong-Guk Park, Kyung-Jin Lee, and Jongsun Park. Sot-mram digital pim architecture with extended parallelism in matrix multiplication. *IEEE Transactions on Computers*, 71:2816–2828, 11 2022. doi:10.1109/tc.2022.3155277.

[14] Shaahin Angizi, Zhezhi He, and Deliang Fan. Parapim: a parallel processing-in-memory accelerator for binary-weight deep neural networks. 01 2019. doi:10.1145/3287624.3287644.

[15] Simone Machetti, Pasquale Schiavone, Thomas Müller, Miguel Peón-Quirós, and David Atienza. X-heep: An open-source, configurable and extendible risc-v microcontroller for the exploration of ultra-low-power edge accelerators. 2024. doi:10.1145/nnnnnnn.nnnnnnn.

[16] Chengmo Yang and Zeyu Chen. A processing-in-memory implementation of sha-3 using a voltage-gated spin hall-effect driven mtj-based crossbar. *Proceedings of the 2019 Great Lakes Symposium on VLSI*, 05 2019. doi:10.1145/3299874.3317972.

# Ringraziamenti

Ecco una versione più scorrevole e fluida del tuo testo, pur mantenendo il contenuto intatto:
Ho deciso di scrivere questo "capitolo" di getto, senza tornare indietro a rileggere, per rendere il processo di scrittura più naturale e immediato per me. Sono stati tre anni davvero intensi, durante i quali sono stato costantemente messo alla prova. Ho sfidato me stesso, la mia autostima, i miei limiti e la mia forza interiore. Questi tre anni mi hanno segnato profondamente, nel bene e nel male. Potrà sembrare banale, ma credo che, purtroppo o per fortuna, faccia parte del processo di crescita. Cambiare completamente vita, circa tre anni fa, è stata forse una delle cose più difficili che mi siano mai capitate. Lasciare la mia famiglia e i miei affetti in Sicilia, con la paura di non ritrovarli più o di trovarli cambiati, è stato estremamente difficile e complicato da gestire.
Voglio iniziare con un sincero ringraziamento alla mia famiglia, e in particolare ai miei genitori. Mi hanno sempre sostenuto, anche nei momenti più difficili, quando le cose non andavano come sperato. Mi hanno dato la spinta giusta quando ne avevo bisogno e mi hanno aiutato a rialzarmi nei momenti più bui, come quando ho fallito il mio primo esame. Quell'esperienza mi ha spinto a cambiare percorso, nonostante avessi dubbi su me stesso e sentissi di non essere abbastanza. In ogni caso, non mi hanno mai fatto mancare nulla, né affetto né supporto fisico.
I miei genitori sono stati il mio punto di riferimento, sempre al mio fianco sotto ogni aspetto: economico, mentale ed emotivo. Non hanno mai perso di vista i miei obiettivi, anche quando io stesso li avevo smarriti. Non credo che esistano parole che possano esprimere appieno la mia gratitudine per tutto quello che hanno fatto per me, e per avermi insegnato a ragionare con la mia testa, anche quando le mie opinioni divergevano dalle convenzioni.
Mi hanno sempre incoraggiato a lottare per le mie idee e a perseguire i miei sogni. Non mi hanno mai detto: "Non sei capace", ma piuttosto: "Prova,

e vediamo come va". Questo loro atteggiamento ha avuto un impatto fondamentale nel farmi diventare la persona che sono oggi. Ci sono stati tanti momenti di crisi, momenti in cui sembrava non esserci una via d'uscita, ma loro erano sempre lì, pronti a spronarmi, a dirmi: "Datti una svegliata, ce la puoi fare".
Per tutto questo, vi dico ancora una volta: grazie. Grazie di cuore. Mi considero estremamente fortunato ad avere genitori che sono anche i miei idoli, il mio modello di ispirazione.
Questo traguardo non è solo mio, ma anche vostro, perché una parte importante del merito è vostra. Non sarei mai arrivato a questo punto senza di voi, e nonostante la distanza fisica, la vostra presenza non è mai venuta meno. Ci sarò sempre per voi, costi quel che costi, perché siete davvero la cosa più preziosa della mia vita. Non trovo parole che possano esprimere quanto bene vi voglia e quanto mi senta fortunato ad avervi come genitori. Grazie di tutto.

Un ringraziamento speciale va alla mia famiglia in senso più ampio: ai miei nonni e ai miei zii, sia paterni che materni. In particolare, desidero dedicare un pensiero al nonno Nino, alla nonna Maria, alla nonna Agnese e al nonno Antonio, che mi hanno insegnato tanto e dato ancora di più nel corso degli anni. Grazie anche a zia Franca, zio Salvatore P, zia Maria Rosaria, zio Salvatore DS e zio Nazzareno. Nonostante la distanza e la difficoltà nel sentirsi quotidianamente, il vostro affetto non è mai venuto meno, ed è sempre un piacere vedervi e confrontarsi, anche quando le nostre idee sono diverse.
Un ringraziamento particolare va al mio nonno Nino, che è sempre stato orgoglioso di me e mi ha dato fiducia, anche quando io stesso non ne avevo. È una delle persone più importanti della mia vita, una figura con cui ho sempre potuto dialogare, nonostante la grande differenza di età. Un pensiero speciale va anche al nonno Antonio, che mi ha sempre coperto quando facevo danni e, insieme a mio padre, mi ha insegnato cosa significa essere Juventino. Nonostante le difficoltà, è sempre bello vederti sorridere.

Un grande grazie anche ai miei cugini: Mirko, Roberta e il "cugino acquisito" Arcangelo. Con voi ho condiviso tanti momenti belli e significativi, che hanno contribuito a formarmi come persona. Mirko, i meme che ci scambiamo ogni giorno sono un piccolo rituale che mi fa sempre sorridere e che mi ricorda quanto sia bello tornare a casa. A Roberta e Arcangelo, auguro ogni bene per la nuova avventura che stanno per intraprendere e spero che possiate vivere qualcosa di simile a ciò che i miei genitori hanno vissuto con me.

Infine, un pensiero speciale va alla piccola Sophia, che è appena entrata

nella mia vita. Anche se sei ancora piccola, hai già lasciato un segno positivo in me, portando emozioni che non provavo da tempo. Spero un giorno di poterti prendere in braccio e di diventare una figura importante per te, anche se, prometto, non ti darò mai i soldi per le sigarette!

A tutti voi, genitori, nonni, zii e cugini, voglio dire che ci sarò sempre. Non importa la distanza o le difficoltà: siete la mia famiglia e non credo ci siano parole sufficienti per esprimere quanto vi voglia bene e quanto siate stati fondamentali in questo mio percorso. Grazie di cuore a tutti voi, vi voglio tanto bene.

Dopo aver ringraziato la mia famiglia, vorrei dedicare un pensiero a tutte le persone esterne che ho incontrato in questi anni e che hanno fatto parte della mia vita.

Prima di tutto, un grande grazie a Delfino, Fabio e Noemi. Il nostro rapporto, costruito in questi tre anni, è stato segnato da alti e bassi. Non sempre le cose sono andate come avremmo voluto, ma, in un modo o nell'altro, siamo sempre riusciti a trovare il nostro equilibrio. Sono felice di avervi nella mia vita e di poter condividere con voi questo percorso, nonostante i momenti in cui ci siamo dati il meglio e il peggio di noi stessi, sempre sostenendoci a vicenda.
Un grande ringraziamento va anche agli amici del BEST: Edo, Max, Miriam, Fabi Pisellina, Piero, Albessio, Bianca, Robb, Mariloo e Giovannino. Ognuno di voi ha contribuito, in modi diversi, a rendere questi anni indimenticabili. Abbiamo condiviso tante esperienze, rischiato la salmonella più volte (senza successo, visto che siamo ancora qui) e creato mille ricordi che porterò sempre con me. Anche se le nostre vite ci hanno portato su strade diverse e non ci vediamo spesso, sono sicuro che l'affetto che ci lega resterà immutato nel tempo.
Un pensiero speciale va infine a Fede e Dave, i miei "baby", che continuano a darmi soddisfazioni sia come loro mentore (oltre che come amico), sia con tante risate. Anche nei momenti più difficili, sono riusciti a regalarmi positività, leggerezza e una buona dose di demenza a livelli esorbitanti. Sono orgoglioso di voi e felice che le nostre strade si siano incrociate, anche se solo verso la fine del mio percorso.
Un altro grazie va ai miei coinquilini, Matteo e Nata. Con Matteo ho condiviso tre anni della mia vita: tra lamentele, commenti su La Zanzara, tante risate e una buona dose di cringe. Sei stato letteralmente l'unica certezza in una casa che cambiava inquilini ogni anno, e una persona su cui ho sempre potuto contare. Con Nata, invece, convivo da un anno, ma la sua presenza

ha portato una nuova energia nella casa (e qualche dose di depressione da assignment). Non posso dimenticare gli sfoghi in cucina quando qualcosa andava storto, che sono stati un modo per liberarci dalla tensione. Grazie a voi ho imparato a gestire gli spazi comuni, a rispettare gli altri e a trovare un equilibrio nella convivenza. Questi anni sono stati indimenticabili anche grazie a voi.
Un ringraziamento va anche ai compagni e amici della triennale, Antonio, Piercy e Davide, con cui abbiamo affrontato esami, modalità d'esame a volte folli e tanto stress. Nulla sarebbe stato lo stesso senza di voi, sia prima che ora, nonostante gli anni che passano e la vita che cambia. Anche se non abbiamo avuto la possibilità di laurearci tutti insieme, è stato bello sostenerci a vicenda, nonostante le distanze e i percorsi diversi. E ancora più bello è poter gioire dei successi l'uno dell'altro. Per non dimenticare tutte le volte che mi sono lamentato con Davide sul Poli, ed ora le cose si sono capovolte.
Un altro pensiero va ai ragazzi di Empathy: Giacca, Manu, Donato, Vinz, Simone e Peppe. Anche se il percorso che abbiamo fatto insieme è stato breve, per me è stato fondamentale. Mi ha permesso di acquisire una maggiore consapevolezza di me stesso e degli altri intorno a me, facendomi capire quanto sia bello condividere esperienze e affrontare sfide insieme, supportandoci nei momenti più difficili.
Un grazie speciale anche ai ragazzi del "3+4": Eli, Jimmy, Selene, Silvia, Babak, Caro, Lello, Nico e Roberto. Quando sono diventato ufficialmente tesista, temevo che questa fase sarebbe stata lunga e complicata. Invece, grazie a voi, questi nove mesi sono volati, tra risate, sostegno reciproco, tanta complicità e piccoli momenti di relax tra un test e l'altro. Sono davvero felice di avervi conosciuto e di quello che abbiamo fatto insieme, e spero che anche voi porterete con voi un ricordo tanto bello quanto il mio.
Un altro grazie speciale va ai ragazzi di "Domenica montagna": Miki (non so proprio dove collocarti, sei veramente ovunque, AAAAA!), Chiara, Dodo, Marco, Elena e Salvo, per avermi fatto sentire subito parte del gruppo e per avermi regalato tanta allegria nei momenti di stanchezza e stress. Le risate, le avventure e il tempo passato insieme sono stati davvero preziosi, e spero di poterne condividere tanti altri con voi in futuro. E non dimentico i "commenti tecnici" sui programmi trash con Chiara e Dodo, che ogni volta mi fanno morire dalle risate: Chiara, con la sua ironia demenziale, e Dodo, con il suo umorismo tagliente.
Infine, un'altra esperienza fantastica è stata lavorare allo stadio. Non credo che avrei potuto trovare colleghi migliori di Claudio, Manuela, Paola e la lenta Rebecca. Ogni trasferimento è sempre un'avventura a sé, e adoro quei

momenti in macchina a sparlare dei tifosi stupidi e ridere fino alle lacrime, nonostante ci aspetti un freddo boia e una valanga di insulti da parte delle persone. Non potrei chiedere colleghi e amici di stadio migliori di voi.
Infine, un grazie a Stefano "Il Maestro" e Salvo, che hanno portato un po' di Messina qui a Torino. Dalla triennale alla magistrale, il nostro rapporto si è rafforzato, e sono felice di ciò che abbiamo costruito insieme. Sappiamo noi quanto sia stato difficile affrontare i sacrifici iniziali e quante sfide abbiamo superato quì a Torino (tra cui OS161, ho ancora gli incubi AAAAA), ma ce l'abbiamo fatta, e il nostro traguardo è una soddisfazione enorme.
Un pensiero davvero speciale va ad Alessandro Spataro, compagno di studi, di mille esami e, soprattutto, di mille progetti (il RISCV Lite AAAAA). Abbiamo fatto insieme tanti sacrifici e affrontatomomenti di sconforto, cercando sempre di trovare una soluzione ai problemi. Grazie a te, questi anni non sono stati solo studio e stress, ma anche amicizia e confronto, oltre che sfottò e sorrisi. Anche se non abbiamo concluso il percorso insieme, non vedo l'ora di gioire per il tuo successo, perché te lo meriti davvero.
Come posso non ringraziare la mia dottoressa preferita Elisabeth? Ovviamente non si può! Per cui, ecco, dopo 8 anni di amicizia, non posso non ringraziarti di essere mia amica, di zupparti il mio egocentrismo, i miei "vengo a Milano" detti a tappo, le mie lamentele, la mia polemica, il mio cuttigghio e il mio scazzo. Sappiamo entrambi quanto sia snervante la mia compagnia, ma in un modo o nell'altro tu ci sei sempre, e questa cosa mi riempie il cuore di gioia. Sono davvero fortunato ad averti come amica (anche se sei palesemente cringe, ma lo sono anche io alla fine), e sono ancora più fiero di averti nella mia vita (audio tuoi lunghi come podcast inclusi). Abbiamo molti valori in comune, e questa cosa ci unisce più che mai. Graize ancora, Doc, tivvibì.

Voglio inoltre ringraziare il mio supervisore Alessio per l'enorme pazienza dimostrata nei miei confronti, anche quando le mie domande erano estremamente semplici o ingenue. Con il sorriso e con grande disponibilità, mi hai sempre aiutato, fornendomi suggerimenti su come procedere, migliorare e affrontare al meglio questo periodo di tesi. In base alle esperienze di altri amici, mi rendo conto di quanto sia preziosa la presenza di qualcuno come te: un supervisore capace di aiutare e incoraggiare, pur mantenendo il ruolo di guida. Hai sempre saputo insegnarmi con equilibrio, lasciandomi la libertà di lavorare ma allo stesso tempo proteggendomi e supportandomi nei momenti difficili. È davvero raro e bello avere accanto una persona che è già passata attraverso queste sfide e che, ricordando le difficoltà, riesce a trasmettere il

giusto approccio. Per tutto questo, grazie di cuore. Ti auguro tutto il meglio in questa nuova avventura in Svizzera!
Un ringraziamento speciale va anche alla professoressa Graziano per la grande opportunità offertami durante la stesura della tesi. Mi avete permesso di mettermi alla prova con un progetto stimolante e affascinante, che non solo mi ha arricchito didatticamente, ma mi ha anche fatto crescere a livello personale. La scelta di intraprendere questo percorso con lei è stata dettata dal grande valore umano che ho avuto modo di apprezzare nei suoi insegnamenti, specialmente durante i corsi di Microelectronic Systems ed Engineering Empathy. Ciascuno di essi è stato fondamentale per la mia crescita, sia come studente sia come futuro ingegnere.

Un ringraziamento importante va anche al professor Vacca che, nonostante non fosse il mio primo relatore, è stato estremamente disponibile durante questi mesi. Nonostante ci siamo "identificati" a vicenda solo dopo più di un mese dall'inizio del lavoro, la sua conoscenza e il suo ruolo umano sono stati fondamentali, oltre alla calda ironia con cui affrontava le difficoltà. Il suo contributo è stato essenziale per rendere questa esperienza non solo piacevole, ma anche altamente formativa. Per questo motivo, mi ritengo estremamente grato per tutto il supporto che mi ha offerto.

Infine, devo ringraziare il professor Donato. Fin dalla triennale non è mai mancato il suo supporto: si è sempre reso disponibile e non ha mai smesso di insegnarmi qualcosa di nuovo. Ogni volta che torno all'Università di Messina, lo faccio con gioia, sapendo che sarò accolto con la stessa generosità e disponibilità che mi ha sempre dimostrato. I suoi consigli, sempre preziosi, sono stati una guida fondamentale per me, e di questo gli sarò sempre grato.

Dulcis in fundo, voglio ringraziare me stesso. Voglio ringraziarmi per aver creduto in me, per essermi tirato giù nei momenti in cui avevo bisogno di riflettere e per essermi tirato su in quelli di sconforto. È vero, le persone intorno a me hanno contribuito enormemente e mi hanno dato la forza necessaria per andare avanti, ma sono altrettanto certo di aver trovato dentro di me quella spinta interiore che mi ha permesso di arrivare fin qui. Sono fiero di aver avuto il coraggio e la maturità per affrontare un cambiamento netto nella mia vita, almeno dal punto di vista didattico, e di non essermi mai tirato indietro di fronte alle sfide, alle difficoltà o alle paure che si sono presentate. Anche se questi tre anni mi hanno dato tante preoccupazioni, posso dire con orgoglio che mi hanno restituito ancora più soddisfazioni.
Questi anni mi hanno aiutato a crescere: sono diventato più uomo e meno bambino – o, forse, una versione più matura di me stesso. Sicuramente, oggi

mi sento più preparato per affrontare la fase successiva della mia vita. Sono contento di essere rimasto fedele ai miei valori, di non aver permesso ad abitudini o ambienti di influenzarmi negativamente, e di aver acquisito nuovi principi lungo il cammino.
Tra le tante lezioni apprese, la pazienza è una di quelle su cui continuerò a lavorare, ma sono felice dei progressi fatti. Ho imparato a gestire meglio la frustrazione derivante dall'impossibilità di avere tutto sotto controllo, riducendo l'ansia che mi ha accompagnato in passato. Questo non significa che il lavoro sia finito: c'è sempre spazio per migliorare, e so che non smetterò mai di crescere.
Potrà sembrare egocentrico, ma sono davvero fiero di me. In questi anni, ho raggiunto la maggior parte degli obiettivi che mi ero prefissato, anche affrontando innumerevoli ostacoli e accettando le conseguenze delle mie scelte. Custodisco tutto ciò dentro di me, come un prezioso bagaglio di esperienza che mi accompagna ogni giorno.
Posso finalmente dirmi: Bravo, ce l'hai fatta! Ma ricordati: non hai ancora finito, stai solo iniziando. Il meglio deve ancora venire. Non dimenticare mai chi sono i tuoi punti di riferimento, le cose realmente importanti, perché saranno loro ad aiutarti a rialzarti quando cadrai.
E, soprattutto, ricordati della fame che ti ha spinto fuori dalla tua zona di comfort. Custodisci quella porta aperta verso il futuro e lasciati guidare dall'ambizione che ti ha portato fin qui.
In bocca al lupo per tutto ciò che verrà.
*Per aspera ed astra*