Politecnico di Torino

EURECOM

Master's degree in Electronic Engineering

Smart Object

LDPC 5G decoder implementation for EMBB project

Supervisor
prof. Maurizio Martina
prof. Renaud Pacalet

Author
Riccardo Torres

candidate's signature

Torino, 10 2022

To my mother Fabiola, for teaching me the
beauty of life and the power of a smile.

To my father Paolo, for teaching me
curiosity and the art of staying young.

To my sister Francesca, who completes
me by constantly showing me a world
that I forget to look at.

EURECOM
Sophia Antipolis

DECLARATION POUR LE RAPPORT DE STAGE
DECLARATION FOR THE MASTER'S THESIS

Je garantis que le rapport est mon travail original et que je n'ai pas reçu d'aide extérieure. Seules les sources citées ont été utilisées dans ce projet. Les parties qui sont des citations directes ou des paraphrases sont identifiées comme telles.

I warrant, that the thesis is my original work and that I have not received outside assistance. Only the sources cited have been used in this report. Parts that are direct quotes or paraphrases are identified as such.

À Biot, *in Biot*
Date : 08 March 2021

Nom Prénom : Riccardo Torres
Name First Name

Signature :

Abstract

[**EN**]

This paper of work describes the implementation of a library that aims to implement a device at the hardware level that is capable of decoding a 5G signal using the LDPC algorithm. During the implementation of the algorithm, various optimizations will be adopted to reduce the number of iterations required for the complete decoding of the received data and reduce power consumption.

In addition to the implementation of the algorithm, it will also be necessary to implement blocks that manage the transfer from an external block to the one in which the decoding will be processed.

[**FR**]

Ce document de travail décrit l'implémentation d'une bibliothèque qui vise à mettre en œuvre un dispositif au niveau matériel capable de décoder un signal 5G en utilisant l'algorithme LDPC. Lors de l'implémentation de l'algorithme, diverses optimisations seront adoptées afin de réduire le nombre d'itérations nécessaires au décodage complet des données reçues et de réduire la consommation électrique.

Outre l'implémentation de l'algorithme, il sera également nécessaire de mettre en œuvre des blocs qui gèrent le transfert d'un bloc externe vers celui dans lequel le décodage sera traité.

Contents

List of Figures

List of Tables

1 Introduction

This internship aims to implement a new library for the EMBB project. This project aims to group various libraries that aim to make the best use of the DSP blocks inside an FPGA and use them to process a specific algorithm. The library that will be implemented in this project will deal with implementing the LDPC algorithm for 5G decoding.

In particular, the block that will be worked on can be represented as in the following diagram:

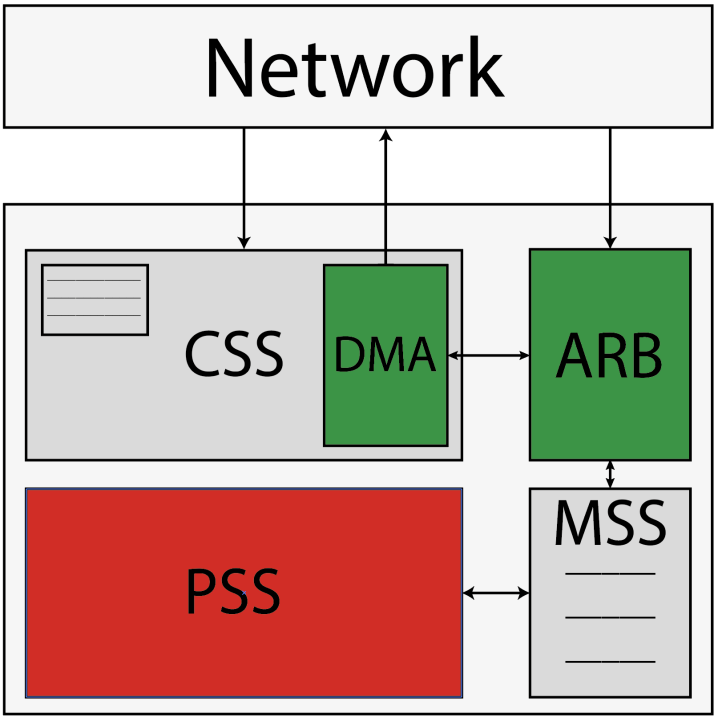

Figure 1: Generic library of EMBB project

In the image above, three distinct regions can be distinguished by colors. The grey region represents the blocks that will not be treated within the document. In fact, of these blocks, only the input and output signals and how to interact with them are known.

The green block refers to implementing a Direct Memory Access (DMA) block, which is entrusted by the Control Sub System (CSS) block with signals containing the information to perform data transfers. For now, what is essential to understand is that this block must be responsible for the correct transfer between its block of interest, i.e., the Memory Sub System (MSS) and the external system.
In addition to this block, there is also the ARBITER block. This block is needed to manage the various data transfer requests of the MSS; in fact, the memory system of the individual block can be accessed both from the reference block and from other external blocks representing other libraries. Therefore, this block deals with the management of possible conflicts in MSS access requests.

The second block that will be dealt with is the Processing Sub System (PSS), in which all

the hardware necessary for processing the 5G decoding using the LDPC algorithm will be implemented.

Before proceeding, what must be kept in mind is the correlation between these two blocks (green and red). In fact, the first will be used to transfer into memory the data to be analyzed by the second, during the latter, once decoding is done, must update the value of the data in the MSS, which can then be taken from another block for a next step.

It is now possible to proceed to the detailed discussion of how each of these blocks has been implemented and highlight the optimizations' criteria.

2 AXI4 Protocol

Before dealing with the implementation for the management of data transfer within the library, the main characteristics of the AXI4 protocol will be briefly explained, which must be kept in mind to proceed with the discussion. Specifically, in the following paragraphs, we will describe the handshake processes regarding reading and write operations, the response channels (both read and write), and finally, the burst.

Excluded from the following discussion are all the more controlling signals and will not be used in the subsequent implementation. Specifically, the signals that will not be treated are the following: xLOCK, xCACHE, xPROT, xQOS, xREGION, and xUSER.

Finally, it should be noted that the following is a quick explanation of some mechanics that will be used for understanding future chapters. These explanations should not be understood as exhaustive or a substitute for what is explained in the original document.

2.1 Specification

2.1.1 Handshake process

In this chapter, the protocols that will be dealt with interfere with a master component and a slave and treat in a very general manner. No information will be given on the channel in question or the type of data to be sent. Precisely because of this sense of generality, only three signals will be mentioned: *valid*, *ready*, and *information*, the latter in particular indicating both the sending of an address and a piece of data.

What is important to understand is that the valid signal is asserted by the block that is to send something, while the receiving block asserts the ready signal. The valid signal should only be asserted high when we are sure of the type of information we want to send. Finally, for communication between the two blocks, these two signals must be asserted high together.

Therefore, all that remains is to understand how to interface these two signals to establish communication.

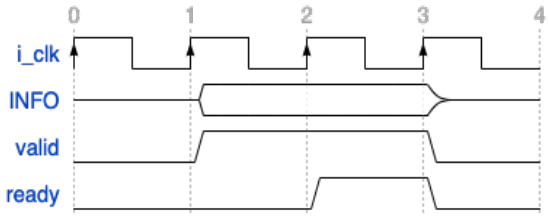

Figure 2: VALID before READY handshake

In this first case, the receiving block is always available, and as soon as the sanding block is sure of the information, the vali signal is also asserted. Communication between the two blocks takes place.

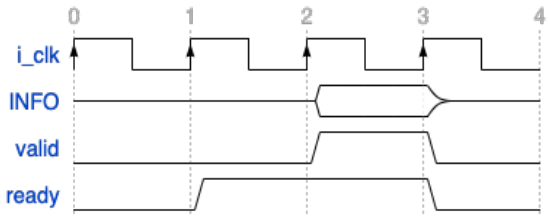

Figure 3: READY before VALID handshake

In this second case, the opposite occurs, so the sending block is ready to send the information but is waiting for the receiving block's signal.

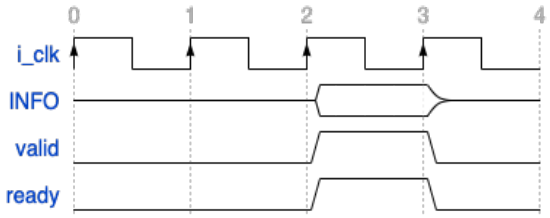

Figure 4: VALID with READY handshake

This last case shows the case in which the two segals are asserted simultaneously.

For the implementation shown here, the third case will never be used. In almost all blocks, communication will be as described in the first case, i.e., the valid signal will only be asserted when we are sure that there is a block on the other side that is ready to receive the information to be sent.
Only in the communication between the arbiter's block and the block that precedes it is used the communication shown in the second handshake process; in fact, it is necessary that each sending block immediately declares the need to send information. In this way, the arbiter will evaluate whether there is a case of conflict and decide which block to give precedence.

2.1.2 Burst

The burst is a method of performing several data transfers with a single instruction. The following three signals manage their use within the AXI4 protocol:

- SIZE: which manages the maximum number of bytes that can be transferred in a single data transfer. The values that this signal can assume are described in the following table:

AxSIZE[2:0]	Bytes in transfer
0b000	1
0b001	2
0b010	4
0b011	8
0b100	16
0b101	32
0b110	64
0b111	128

Table 1: Different encodings for transmission size.

- LEN: manages the maximum length of the transfer and the number of data sent in a single burst. The following formula describes the method of representation:

$$Burst\_Length = \mathbf{AxLEN[7:0]} + 1 \tag{1}$$

- TYPE: manages the type of burst that it is wanted to perform, the difference lies only in the method for calculating the next address, and there are three types of burst, although, for the case study, we will use only the first two:

  * FIXED: the address always remains the same. This type of burst is used when interacting with a FIFO memory, and therefore, the address is always the same, and then the device will manage the data inside it.
  * INCREMENTAL: the address is incremented from time to time following the law:

$$New\_Address = Old\_Address + SIZE \tag{2}$$

    This type of burst is used, for example, for a RAM-type memory where it is necessary to increment the address each time before reading or writing the data.

Before proceeding, it is necessary to underline that these three signals must be added, as information for the burst generation, the signal that indicates the starting address, which will remain as it is or will be increased according to the type of burst used.

However, the above parameters are subject to some limitations, which can be found in the specifications and are listed below:

- for wrapping bursts, the burst length must be 2, 4, 8, or 16,
- a burst must not cross a 4KB address boundary,

- early termination of bursts it not supported.

Since this project's applications do not require burst wrapping, only two essential rules remain, and only the second one is a limitation.
Given that in the chapter on the DMA, it will be necessary to generate a burst request, it is useful in this paragraph to analyze the limitations that exist for the generation of this and how they should be managed, for which three parameters must be taken into account:

- il massimo *address boundary* (4kB),
- il massimo valore di *lenght* ($256 \Rightarrow 2^8$),
- il massimo valore di *size* $\{1, 2, 4, 8, 16, 32, 64, 128\} \Rightarrow \{2^0, 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7\}$.

Among these three parameters, only the last one has variable values (the other two can also vary, but in this case, we are considering their maximum limits). What we want to study is the variation of the maximum value of the burst length as the length value varies, trying to obtain, when possible, a transfer of 4kB.
The link between these three values is as follows:

$$ADDR\_BOUND = LEN \cdot SIZE \tag{3}$$

From which we can deduce that:

$$max\_LEN = \frac{max\_ADDR\_BOUND}{SIZE} = \frac{4 \cdot 1024}{SIZE} = \frac{2^{12}}{SIZE} \tag{4}$$

Now all that remains is to vary the value of *SIZE* and study the trend of *max_LEN*:

$$max\_LEN = \frac{2^{12}}{SIZE} = \begin{cases} 32(2^5), & \text{if } SIZE = 128\ (2^7) \\ 64(2^6), & \text{if } SIZE = 64\ (2^6) \\ 128(2^7), & \text{if } SIZE = 32\ (2^5) \\ 256(2^8), & \text{if } SIZE = 16\ (2^4) \end{cases} \tag{5}$$

It is possible to deduce that the maximum limit to obtain the max_LEN is to have SIZE $\leq$ 16. Moreover, SIZE = 16 is the only combination with which it is possible to obtain both the max_LEN and the max_ADDR_BOUND, in fact, without showing it, for SIZE values lower than 16, it will not be possible to reach the max_ADDR_BOUND, since in this case the upper limit is set by the length.

These observations will be taken up during the description of the *burst_generator* block, which has the task of generating burst signals with the length and size parameters available.

2.1.3 Differences with the lite protocol

For the implementation of the following blocks the AXI4 lite protocol will also be used, which unlike the AXI4, removes all control signals and the possibility of generating a burst, focusing only on the signals used to manage transfers.

In concrete terms, as far as this document is concerned, since all the control signals have not been treated, the management of transfers remains the same since the handshake protocols are unchanged. It must be remembered that the management of the ID signal is eliminated. So, in the end, the only element that can no longer be used is the burst.

2.2 Implementation

This part of the project is based on the design of two entities: a *DMA* (Direct Memory Access) and a *arbiter*. The first element receives signals from the CSS and has to process them to manage transfers; in particular, its task is to retrieve data and perform read and write operations in the blocks described by the CSS. This block is responsible for the generation and management of the AXI4 protocol to generate signals that can optimize the data transfer process.

The *arbiter* is a block placed halfway between the DMA and the MSS. The need for this block arises from having several blocks trying to access the same memory bank. Specifically, the two blocks are the DMA and the SYS, so this block has the purpose of deciding, in case of conflict, which of the two requests takes precedence.

Before proceeding with a detailed description of the following blocks, it is worth saying a few words about an implementation choice that affects all the code to be described. During the block design, in order to have a more readable code, it has been decided to separate the five channels present in the AXI4 interface (*ar*, *r*, *aw*, *w* and *b*), so that each of them can be managed independently from the others. For this reason, in addition to the records referring to the public interface, the *axi_pkg.vhd* code also contains records, called *slice*, containing only the five channels, which in turn contain the signals connected to them.
In addition to making the code more readable, this choice further emphasizes the independence of these channels from each other.

We begin with describing all the blocks necessary for data transfer, starting with the DMA block and arriving at the block that precedes the MSS.

2.3 DMA

This block's primary function is the generation and management of burst signals; in fact, it receives parameters from the CSS to manage transfers that generally exceed the maximum size allowed for a burst transfer.
Therefore, we can divide the task of this block into two fundamental parts: *generation* of burst signals and *management* of transfers. The first task has been wholly entrusted to another block that is allocated inside the DMA, but that, unlike it, is purely combinatorial and is called *burst generator*.
After acquiring the CSS signals, the first task to be carried out is the generation of the burst signals.

2.3.1 Burst Generator

This block has the purpose of generating burst requests. This need arises from the fact that the CSS's length parameter is greater than the length that can be requested with a single burst, which makes it necessary to generate many intermediate bursts.
For the generation of these bursts, the block in question is based on the previous paragraph's theory. Taking as input the total length and size parameters, we try to define the maximum value of *burst_len* that can be set to obtain a maximum transfer of 4kB.

For this purpose, a *case* is carried out on the size parameter and the value of max_LEN is deduced. Once this value has been defined, all that remains is to define whether or not the value of max_LEN is less than the total length. If this is not the case, the burst length value will be equal to the total length. Otherwise, it will be equal to the value of max_LEN, and the value of the total length will be updated, and the same analysis will be carried out at the subsequent burst request. The generation of a fresh burst is carried out by activating an input signal managed by the DMA.

In addition to generating the burst, this block generates the burst size, the burst type, and the address from which the burst must start. In particular, the new address calculation is based on the type of burst, which for our project can only be of fixed or incremental type. This block will continue generating new bursts until the value of max_LEN is greater than or equal to the value of total_LEN, in which case it will not only generate the last burst but will also set high the signal that warns that this is the last one and that all the read or write requests have been sent.
Having seen in detail how the burst signals are generated, all that remains is understanding how they are managed, returning to the DMA block. Given the previously mentioned division into channels, there will be two BURST GENERATORS in the DMA: managing read bursts and one for managing write bursts. The need to use two of these blocks arises from performing, in most cases, reading and writing operations simultaneously. Therefore, the best thing would be to separate the two channels and manage them independently, also from the point of view of burst generation.

At this point, we can concentrate on the second function of this block: transfer management. Through some input signals, the CSS can request two possible cases:

1. Write-only: this usually occurs when a memory is initialized, in fact, in this case, the data to be written passed directly from the CSS and will therefore remain the same for all memory addresses. In this particular case, the DMA will not use any read channel and will limit itself to writing consistent data through the AW, W, and B channels.

2. READ and WRITE: In this case, no data is passed to the DMA, but instructions are given to read them. In this case, all five channels will be used, and the read data will be stored inside a FIFO that will be managed in writing by the R channel and in reading by the W channel. It is also used to manage the rready signal that will be asserted high until the memory is full. In this case, the signal will be asserted low, and the read reception will be blocked until at least one data is read from the FIFO, freeing a new place to receive new data.

In addition to the separation into channels, a recent separation must be considered between signals that manage the request's sending (AR, AW, and W) and signals that manage the response (R and B).
Those in the first type are strictly related to the burst_generator. In fact, the axvalid signals are raised only when a new burst is requested and asserted to zero in all other cases. It is important to remember that the block that deals with burst generation is purely combinatorial, and therefore the signals (the new burst request) can be obtained in the same clock stroke in which the request is made.

However, it is necessary to highlight the difference between reading and write burst requests. In the first case, new burst requests will be made when the block being communicated with is ready to receive a new burst request. This happens because the new request must take place once several data equal to the length of the burst have been sent, and the management of these values will be entrusted to this channel. In contrast, the AW channel only understands when a new burst request takes place and asserts the awvalid signal.

The signals that deal with both read and write responses are used to ensure that there are no errors. The management of the status signal will finally tell the CSS how all transfers (both reads and writes) went. If errors occur, it is necessary to wait until all the burst signals that have already been started have finished and to acknowledge the done signal by updating the status signal with the type of error received. In case no error occurs, channel B will take care of the done signal since channel B responses will always arrive after channel R responses.

An essential difference between the read and write responses is that the rlast signal's assertion determines the end of the former's reading process. In contrast, there is no such signal for the B channel, so to understand when all the write operations have been carried out, it is necessary to use a counter that is used at the initial length passed by the CSS.

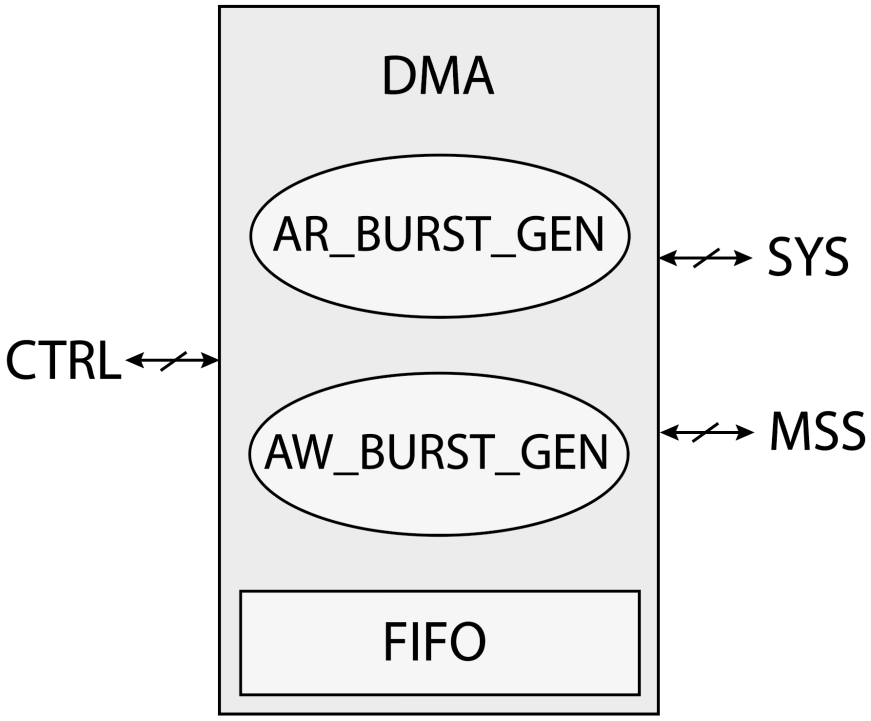

Figure 5: Final structure of the DMA

2.4 AXI2lite

This block is placed as a link between the DMA or SYS and the arbiter and has the purpose of receiving signals from an AXI4 protocol and converting them to AXI lite. As mentioned in the previous chapter, the difference we are interested in in this project between an AXI and its lite counterpart is essentially removing the burst and the ID segments.

This block takes care of receiving incoming burst requests and creating from them all the addresses and from time to time each new single request generated until the burst is exhausted and requesting a new one. For this purpose, similarly to what we have seen in the DMA block, a combinatorial process is allocated inside this block that takes care of generating these addresses: the addr_generator.

2.4.1 Addr_generator

This block receives all burst signals as input, and its purpose is to generate two outputs: the address to write or read the data to, and the signal that generates the last address referred to the received burst parameters.
The generation of a new address occurs only if the 'next_addr' signal is asserted high; in fact, once an address has been generated, before generating the next one, it is necessary to make sure that the previous address has been correctly sent to the arbiter.
This process continues until the counter linked to the length of the burst is zeroed, in which case the *last_addr* signal is also raised along with the new address, indicating the end of this burst management request
Returning to the AXI2lite block, the management of protocol conversion remains, in particular, the situation is very similar to that already studied in the case of the DMA, in fact also in this case, it will be necessary to have FIFO memories used for storing read and response data.
This block also considers the management of any errors; in the event of a reading or writing error, it is necessary to complete a burst request that has already been started. For this reason, in the event of an error, the generation of new addresses will continue until the burst request is completed. However, unlike the normal situation, no new burst will be requested, and instead, the error will be sent, so that the DMA can then inform the general control block.

2.5 Arbiter

This block can manage any conflicts that may occur when two blocks try to access the internal memory block in reading or write simultaneously.
This conflict is managed by the sel_machine block, which generates a signal to decide which of the two blocks should take precedence. This signal should consider the number of times a peripheral has gained access concerning the other and guarantee a certain regularity between transfers. However, for this project, the block in question has been managed so that the DMA block is always chosen.

Apart from managing the conflict employing the signal sent by the sel_machine, this block sends the requests received to the next block. In this case, there is no protocol conversion. Therefore the only influential factor is that the read or write requests are stored inside FIFOs, and also, in this case, the signal that determines when a FIFO is full will manage whether the block can accept new requests. In contrast, the signal that indicates whether or not the FIFO is empty manages the possibility of sending new requests.

As far as response management is concerned, the block has at its disposal other FIFOs to save the read data and response, in case of reading, or only the response in writing. The additional detail is that a bit is added to these responses to represent which of the two

blocks (DMA or SYS) the response should be sent to the arbiter must keep track of the order of the requests accepted and which block they came from to manage the sending of the response.

2.6 lite2RAM

This final block deals with converting the signals from the AXI4 lite protocol to the interface used for the MSS block. At this point, many of the mechanisms previously exposed are repeated. In particular, in this block, there are FIFOs to store the read or write requests, which manage the AR, AW, and W channels.

.
These are flanked by two other FIFOs that store the read data and the responses connected to them; these FIFOs are used to manage the R and B channels connected to the response. The B channel is of particular interest, since the way the interface with the memory is set up, there is no response in the case of a write request; the operation must be considered successful if the memory is ready for a new request at the next clock.

So it is now possible to review the totality of the blocks that make up the arbiter block:

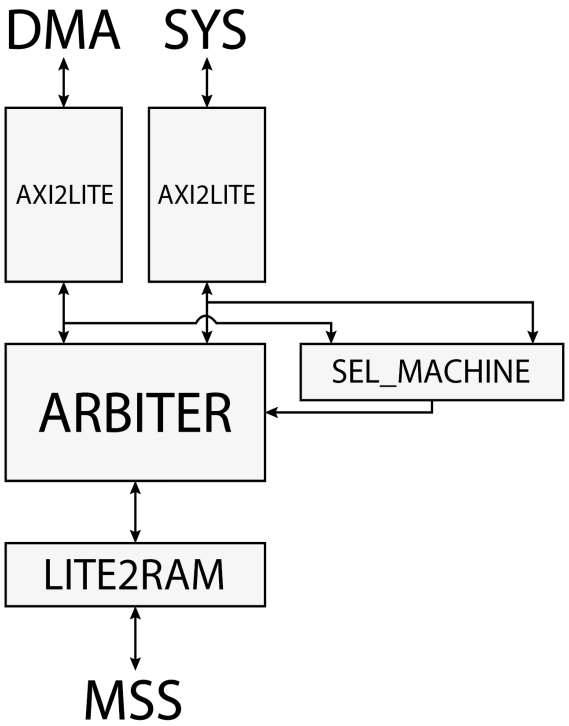

Figure 6: VALID with READY handshake

3 LDPC

3.1 Algorithm

This section will be exposed the criteria used for the implementation and optimization of the LDPC block.

Before starting with it, however, it is good to emphasize that in this document, there will be no mention of theories related to the LDPC algorithm, neither related to its historical importance nor related to how or why its use is gradually increasing over time. This choice is because the choices that follow are related only to implementing an algorithm that is created. All its optimization is related to hardware observations and not related to the theory of LDPC.
If it is interested in knowing more about this algorithm's theory, it is possible to read the thesis in the references section.

As for the information necessary to continue the discussion, it is exposed that the LDPC algorithm whose implementation is required is described as follows:

Algorithm 5 The memory-optimized SCMS algorithm with layered schedule

for $n = 0$ to $N - 1$ **do** ▷ For all columns
  $\tilde{\gamma}_n^{(0)} \leftarrow \ln(\Pr(x_n = 0|channel)/\Pr(x_n = 1|channel))$ ▷ LLRs
end for
for $m = 0$ to $M - 1$ **do** ▷ For all rows
  $r_m^{(0)} = \{0, 0, 0, \mathbf{+1}, +1, \boldsymbol{true}\}$
end for
for $l = 1$ to l_{max} **do**
  for $m = 0$ to $M - 1$ **do**
    $r_m^{(l)} \leftarrow \{+\infty, +\infty, 0, \mathbf{+1}, +1, \boldsymbol{false}\}$
    for $n \in \mathcal{H}(m)$ **do**
      $\alpha \leftarrow \tilde{\gamma}_n^{(l-1)} - \beta(r_m^{(l-1)}, n)$
      if $\mathrm{sgn}(\alpha) \neq r_m^{(l-1)}.\alpha_n^s$ and $(\mathrm{not}\, r_m^{(l-1)}.\alpha_n^e)$ **then**
        $\alpha \leftarrow 0$
        $r_m^{(l)}.\alpha_n^e \leftarrow true$
      end if
      $r_m^{(l)}.\alpha_n^s \leftarrow \mathrm{sgn}(\alpha)$
      $r_m^{(l)}.sp \leftarrow r_m^{(l)}.sp \times \mathrm{sgn}(\alpha)$
      if $|\alpha| < r_m^{(l)}.m_1$ **then**
        $r_m^{(l)}.m_2 \leftarrow r_m^{(l)}.m_1$
        $r_m^{(l)}.m_1 \leftarrow |\alpha|$
        $r_m^{(l)}.i \leftarrow n$
      else if $|\alpha| < r_m^{(l)}.m_2$ **then**
        $r_m^{(l)}.m_2 \leftarrow |\alpha|$
      end if
    end for
    for $n \in \mathcal{H}(m)$ **do**
      $\tilde{\gamma}_n^{(l)} \leftarrow \tilde{\gamma}_n^{(l-1)} + \beta(r_m^{(l)}, n)$
    end for
  end for
end for

Figure 7: LDPC Algorithm

The algorithm just seen is the result of various optimizations implemented on the original LDPC algorithm, and the definition of the Beta function mentioned within it is as follows:

Algorithm 4 Computing $\beta_{m,n}^{(l)}$ from $r_m^{(l)}$ compact data structure

1: **if** $n = r_m^{(l)}.i$ **then**
2: $\quad \beta_{m,n}^{(l)} \leftarrow r_m^{(l)}.\alpha_n^s \times r_m^{(l)}.sp \times r_m^{(l)}.m_2$
3: **else**
4: $\quad \beta_{m,n}^{(l)} \leftarrow r_m^{(l)}.\alpha_n^s \times r_m^{(l)}.sp \times r_m^{(l)}.m_1$
5: **end if**

Figure 8: Beta function

The only thing left to do is to understand how the block will receive the data needed for processing and if it is necessary to perform some information on them.

3.2 Algorithm implementation

In this session, we will discuss implementing the Process Sub System (PSS) and, therefore, the accurate datapath dealing with decoding using the LDPC algorithm for 5G. The figure below shows the final structure that the PSS will assume, and in the following chapters, each of the blocks contained in it will be examined in detail.

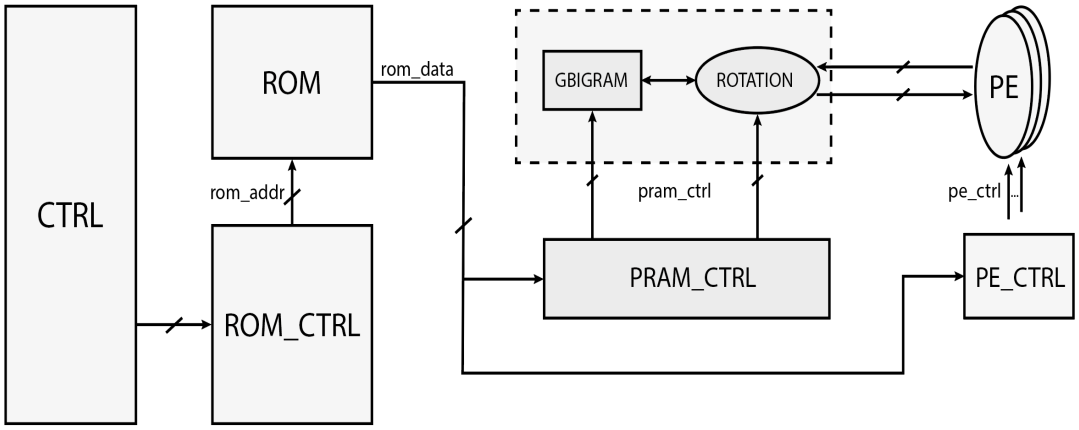

Figure 9: General overview of the PSS

Before proceeding, it is necessary to make a few notes about the first and second blocks, ROM and PRAM respectively, these two blocks are closely related to what was explained in the chapter on AXI4 since both these blocks take data from the MSS. In ROM's case, this type of representation is used for simplicity and to make the simulation of the PSS independent from that of the whole library. The ROM is used for storing all the control signals that will be read by the ROM control and sent to the various blocks.
In the final realization, instead, the control signals will be directly stored inside the MSS. So the above representation serves to show the purpose of the ROM block, storing the PSS control signals. However, its role remains the same since, in the final implementation, this information will be directly taken from the MSS.

There is an extensive memory inside it for the PRAM block, whose role and operation will be discussed later. All the values needed for decoding, taken from the MSS, will be stored in this GBIGRAM. This initialization will be carried out before starting decoding by the general control block, which will take the data being received from the memory and store them inside the GBIGRAM.

So far, what has been said has the purpose of emphasizing the link between the PSS and the MSS. Secondly, we want to bring to the reader's attention that a correct representation of the PSS block is as follows:

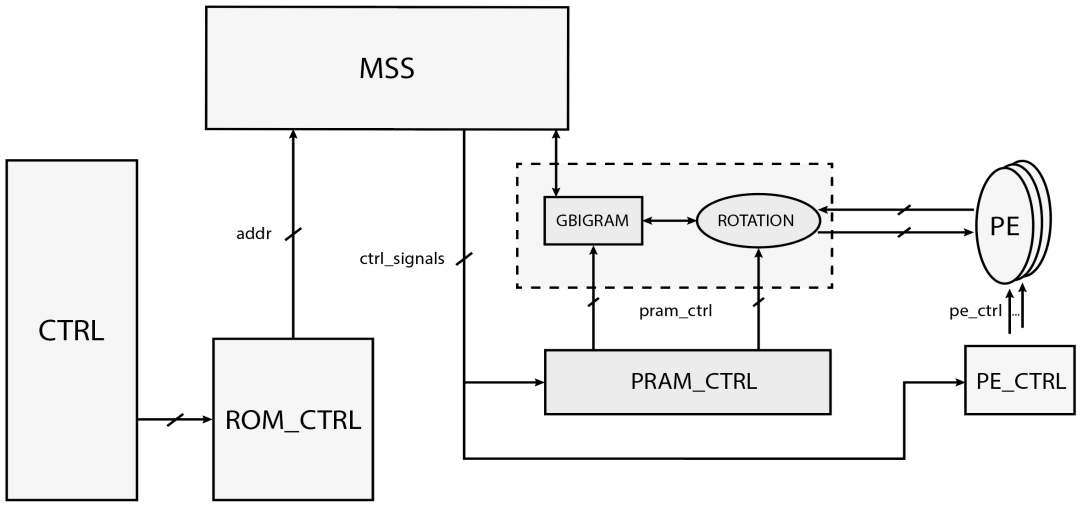

Figure 10: Overview of how the PSS interacts with the MSS

To simplify the discussion and make the simulation of the PSS block independent from that of the AXI4, we will continue to refer to the block in the TODO figure (link to the inserted figure), initializing the values in the ROM and GBIGRAM by reading the files.

At this point, the basis for implementing the LDPC decoding block for 5G has been laid. The discussion will follow the same Project flow, starting from the rightmost block (Processing Element) and going backward, identifying all the signals necessary for the block's correct functioning, which will be saved inside the ROM..

3.3 Implementation of the three main blocks

3.3.1 PE

The initial structure of this block, based on what the algorithm describes, is as follows:

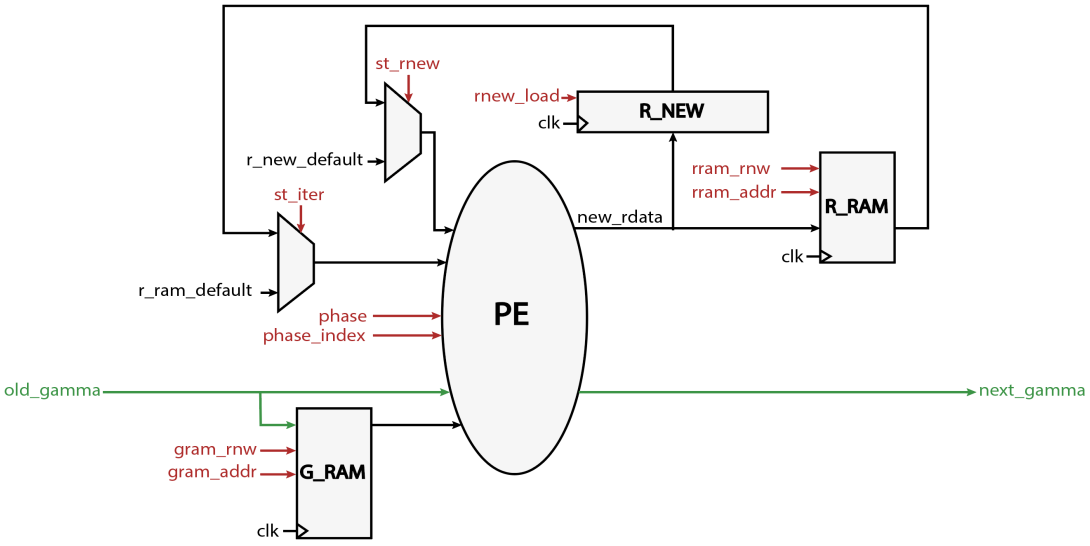

Figure 11: First Processing Element representation

The figure shows the simplest structure that can be thought of from the algorithm. This structure is controlled first by the phase control signal, which determines which type of phase the algorithm is in: the first phase, second phase, or zero phase (no data is being processed). This signal is followed by the relative index value, which is used within each phase for processing the Beta function and for reading or writing the information within the GRAM. Inside GRAM, the gamma values elaborated in the first phase are saved and will be used in the second phase for error correction and, therefore, for the calculation of the new gamma value.

Finally, two signals manage when to take the values to read in output from the R_NEW_REGISTER and the RRAM or when to adopt the default parameters. Specifically, the default value of the RRAM will be used for all the first iteration, therefore, until all the values to be analyzed. The first time is read, while the default value of R_NEW_REGISTER is used only at the beginning of each first phase since the value is calculated for each line.

In addition to the control signals that interact with the combinatorial part of the block, there are also control signals that act on the memory blocks.
0.3cm
The simplest of these is the R_NEW register, which stores all the algorithm's procedure values. The only control signal connected to it is the load signal, which must be asserted high whenever the first phase is active; during this phase, the values to be stored in this register are calculated.

The RRAM memory is a singular access RAM, and it is, therefore, possible to access it either in reading or write through the control signal rram_rnw (read not write signal). During the first phase processing, the algorithm needs to access it always in reading, except

for the last index of the first phase, in which the value calculated during the same phase must be written. Once this has been done, from the beginning of the second phase, it is possible to reaccess this memory in reading mode by setting the following line as the read address, thus having the data ready to process the next first phase.

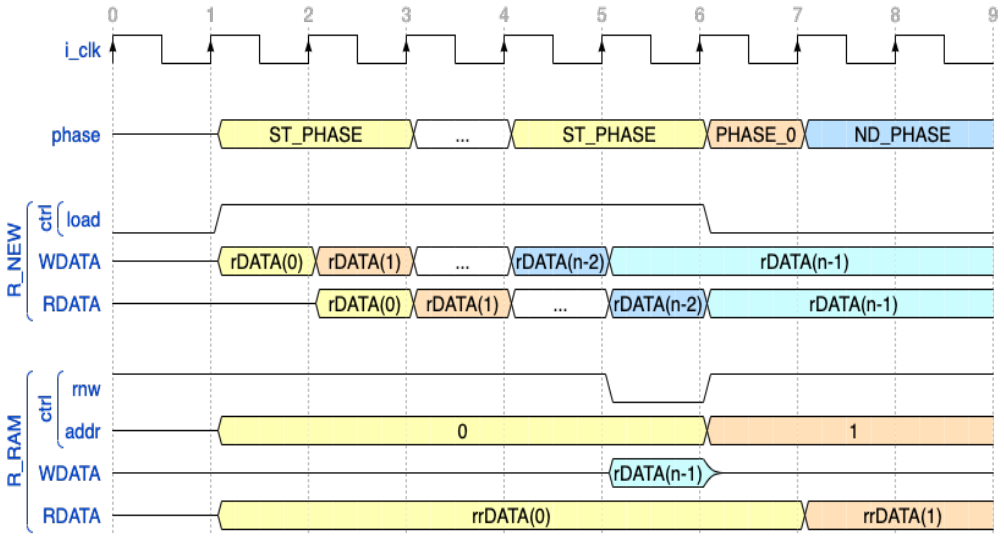

Figure 12: Timing Diagram showing the behavior of RRAM and RNEW at phase variation

The GRAM, which is also a single-access RAM, alternates between reading and writing phases. During the first phase, it is used to write all the values of the new ranges being analyzed, while in the second phase, the same memory is used to read these values. During the first phase, it is used to write all the new gamma values being analyzed, while in the second phase, the same memory is used to read these values,

Since we must always consider the latency time between the request to read a data and the actual reception, it is evident that the first and second phase cannot follow one another without the presence of at least one clock stroke in which no data is processed. The GRAM is accessed in reading mode. This stopped state, called phase zero, is needed because the GRAM memory is accessed in writing for the whole period of the first phase. Therefore, it is not possible to initialize the data to be read before the end of the first phase's processing.

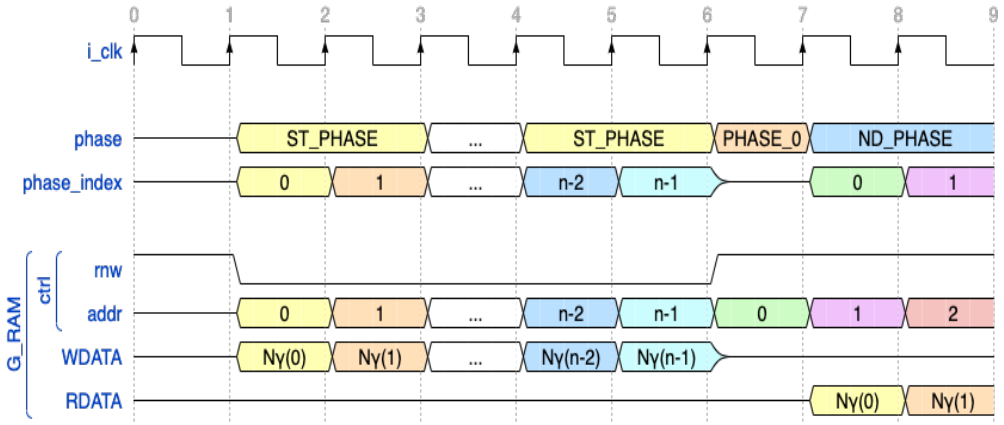

Figure 13: Timing Diagram of GRAM's behavior in phase variation

Optimisations The structure just described represents the basic implementation of the algorithm. The task now is to see how it can be optimized to make the decoding process as fast as possible.
The first possible optimization starts directly from GRAM's last observation and the limitation of having a single access memory. In fact, by replacing this memory with a dual-port memory, it will be possible to eliminate the zero-phase presence due to the wait when reading the gamma value from memory. Therefore, this memory will be replaced by the memory shown in the figure:

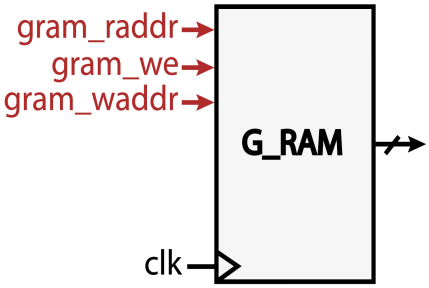

Figure 14: GRAM dual port

This first optimization eliminates phase zero and, from the GRAM point of view, makes the first phase independent of the second phase. With the introduction of a dual-port RAM, it would be possible to process a new first phase while processing the second phase of the previously processed line. The limitation of this optimization lies in the fact that it will never be possible to process several first phases while processing a second phase.
For example: once the first phase related to row n has been processed, it will be possible to process simultaneously a new first phase related to row n+1 and the second phase related to row n. This fact is due to the possibility of writing the new gamma value in the same address to which the second phase has already accessed in reading mode and has already read the data.

What said above implies that the limitation of this optimization resides in the impossibility to process a first phase linked to the line n+2 during the processing of the second phase linked to the line n. Since this process would lead to the overwriting of the gamma values saved during the first phase linked to the line n+1, the second phase has not yet been read since the processing linked to the line n is ending.

From this observation, we proceed with the optimization of the block in order to have the possibility to elaborate in parallel the first and the second phase, always keeping in mind the constraint according to which it is not possible to start a second phase at two rows away from the row elaborated by the second phase.
Therefore, taking up the example previously given, if the second phase is still processing the data relative to line n and the first phase should have finished the computation relative to line n+1, the system must suspend the activation of the first phase until the second phase reads the first value saved in the GRAM relative to line n+1.

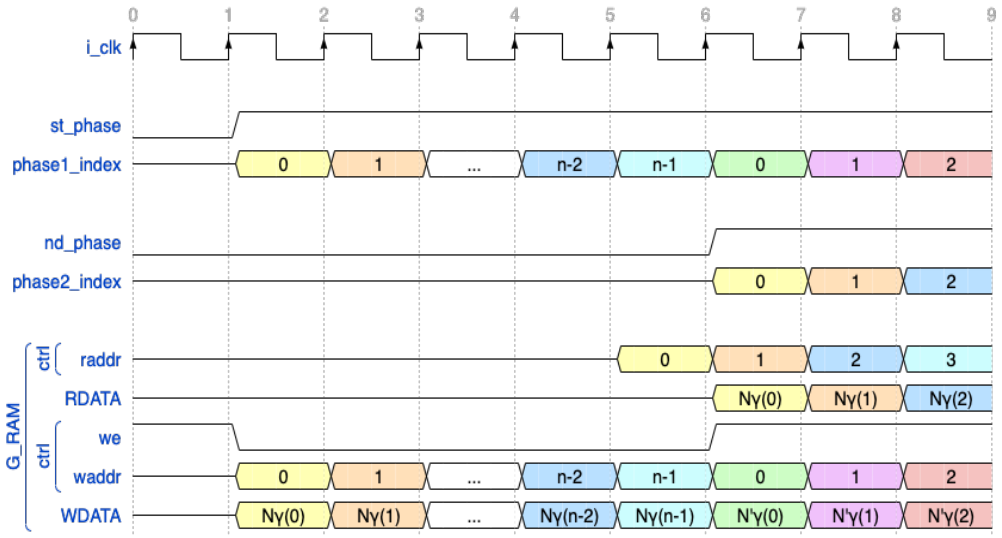

Figure 15: Timing Diagram showing the behavior of GRAM dual-port as the phases change

At this point, we must study how to extend this optimization to the rest of the block. First of all, instead of having a single block representing the processing of the first or second phase, managed by the control signals phase and phase index, it is possible to divide this block into two. This takes care of the first and second phase processing and respectively receives the control signals st_phase, phase1_index, and nd_phase and phase2_index as input.
At this point, two important observations remain to be made:

- Access to the RRAM memory, both for reading and writing, is managed during the first phase. However, the problem is that, while previously we thought of writing new data into this memory at the end of the first phase, since then we would have had all the time of the second phase to access the next data in reading, now this reasoning can no longer be carried out. Since at the end of the first phase, the next one will be immediately started, which needs to read the RRAM value relative to the following address. Therefore this reading phase must take place when the last index of the first phase is processed.
  Unfortunately, this is the same instant in which this memory is accessed to write the new processed value.

  As a first optimization, it would be possible to think about introducing, also in this case, a double-access memory, but unlike GRAM. This possibility of reading and writing at the same time would be used only at the end of each iteration, all this at the cost of having to occupy more hardware to implement a double-access memory than what is required only on a precise occasion.

  For this reason, the solution adopted consists of leaving the RRAM memory at single access and take the input data of it not from the input of the R_NEW_REG but its output. In this way, it will be possible to write the new value inside the memory at

the beginning of the new first phase, while at the end of the previous first phase, it will be accessed in reading.

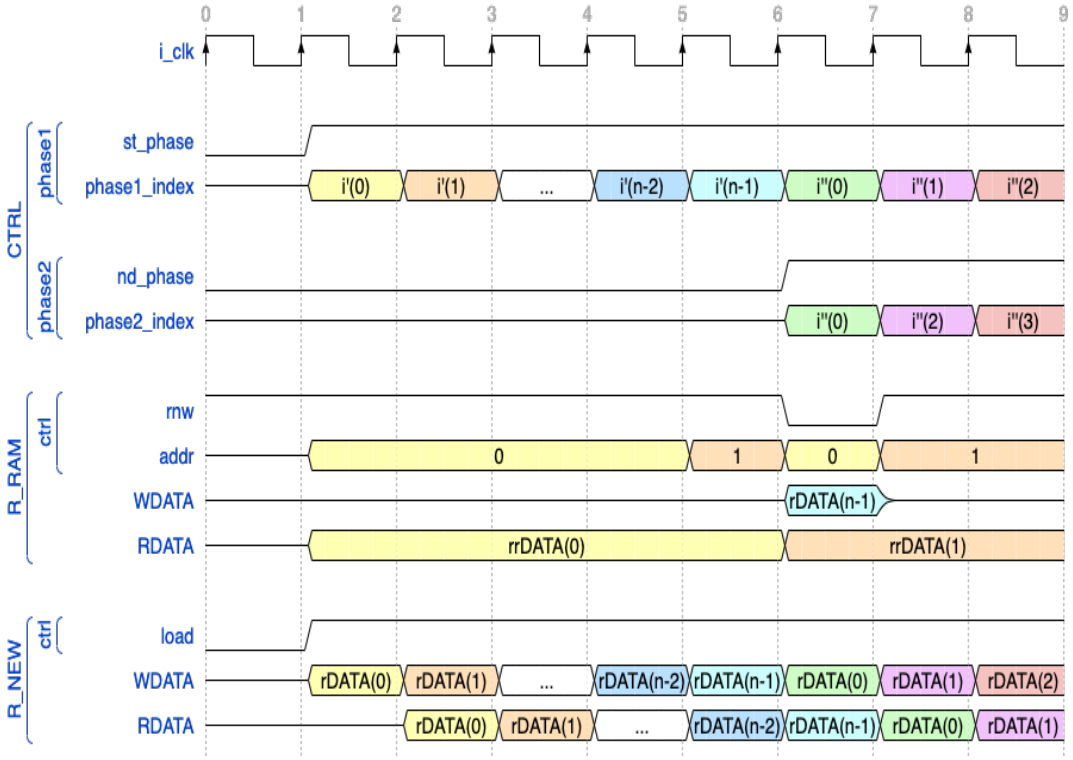

Figure 16: Timing Diagram showing the behavior of RRAM and RNEW at phase changes

- The second consideration must be addressed to the second phase and, more specifically, because it requires using the value saved in R_NEW_REG at the end of the first phase. In the previous block, this was not a problem, since, with the alternation of the first and second phase, the value calculated during the first phase was then taken from the second phase directly by the R_NEW_REG, which kept the value inside it unchanged for the duration of the first phase.
  This falls in the type of block that allows parallelization. During the second phase processing, that of a first phase relative to a subsequent line will take place in parallel, which will continue to vary the value inside this register. Therefore, it is necessary to find a way to save this value to use it during the second phase.

  The first thing you might notice is that this value is the same as the one we discussed in the previous point and that it must therefore be saved inside the RRAM, so it would be legitimate to take this data from the RRAM memory, but this would require the use of a dual-port for the RRAM.
  This hypothesis has already been ruled out in the previous point. The data in question must be ready at the beginning of the second phase of processing; while using the RRAM, it would be necessary first to write and then read this data. Thus bringing back the project from two points of view: the first would be reintroducing

a phase zero clock stroke for the RRAM reading, which implies a no longer total individuality between the first and the second phase.

To solve this problem, we proceeded with introducing a register similar to R R_NEW_REG, which takes the name of G_NEW_REG and which aims to save the value of r_new calculated at the end of each first phase.

This means that during the processing of the last index of each first phase, the block will be responsible for reading the RRAM read request and assert the high load signal for g_NEW_REG, while the first index of the new first phase must be asserted the value of gnew_load and a write inside the RRAM.

It is finally interesting to note that technically it is not necessary to receive as input to the block the signals st_phase and nd_phase. They can be used in the control block to determine which types of signals should or should not be raised, but they have no use inside the final block. The above is justified by choice of maintaining, in the case where the first or second phase is not active, the same output of these previously processed, without resetting an output in the case of no processing.
In this way, it is only necessary to make sure not to change the registers' values during the inactive phases. This consideration must be addressed to the first phase. Through this phase, the parameters for writing or loading new data into the memories or registers are activated or deactivated.

Therefore, it will only be necessary to manage, within the control block, all the information needed to prevent unwanted writes to the memory elements of the block.

All the above observations lead to a distortion of the block initially shown in the figure, arriving at the latest and final representation of the PE block:

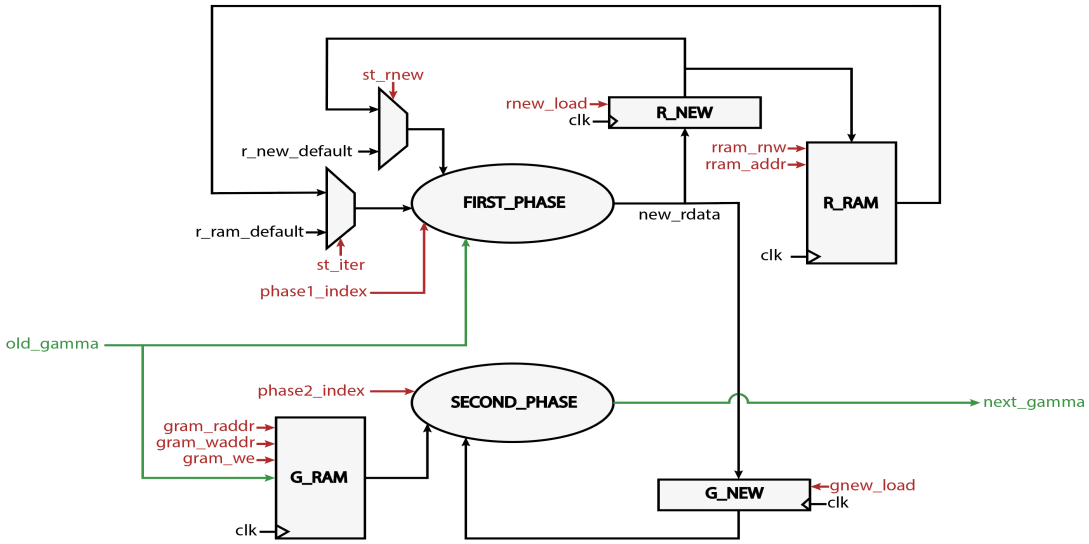

Figure 17: Final Processing Element schematic

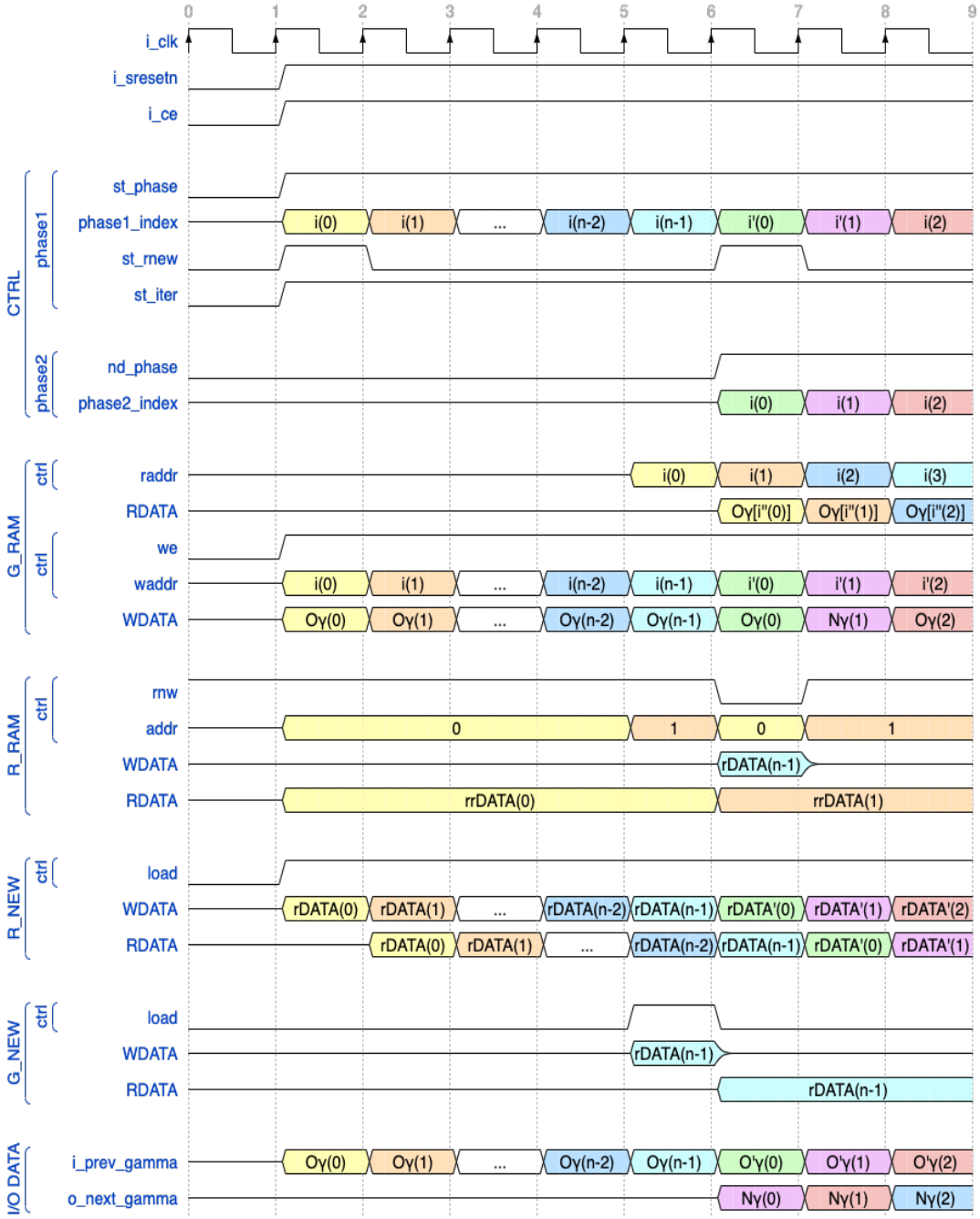

Figure 18: Timing diagram representing generic Processing element processing

3.3.2 PRAM

This block deals with the management of the inputs and outputs of the PE, in particular, all the data to be sent as input to the PE are stored inside the GBIGRAM, they must be taken from it, rotated, and then sent to the PE. This block's description is inevitably much simpler than that of the PE, but this does not make it any less suitable for optimization.

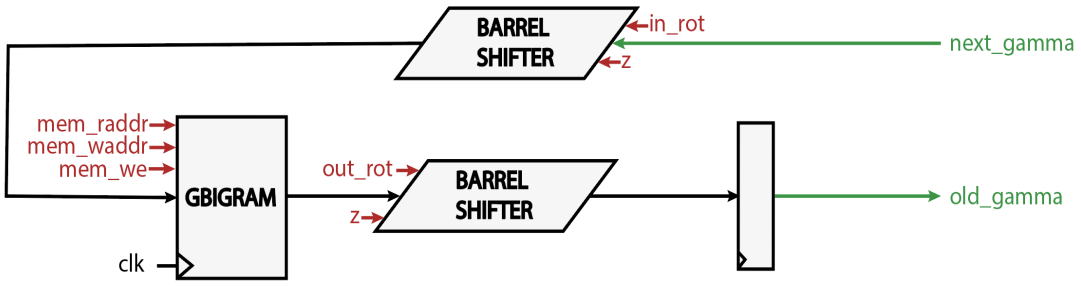

Figure 19: Schematic of PRAM block

The figure shows the connections made between the main blocks; the presence of two barrel shifter is necessary to guarantee the possibility of processing the first and second phases simultaneously, which would be impossible if only one barrel shifter were used.

Optimisations The only major optimization that can be applied to this block is the insertion of a bypass, i.e., introducing the possibility of taking the output data from the second barrel shifter and bringing it directly to the first input.
This optimization would make it possible not to rewrite the data in the GBIGRAM and then reread it, thus improving terms of power. The only thing needed to use the bypass is to introduce a MUX that a control block will then manage.

To conclude the observations on this block, it is necessary to remember that if a bypass operation is carried out, the barrel shifter that handles the PE output must carry out a rotation that considers both the old rotation and the new one. On the other hand, if bypass operation is not used, the data should be rotated to bring it back to its original state.
For this reason, it has been decided to use two barrel shifters with opposite rotations. The input barrel shifter to the PE will have a rotation to the right, while the output barrel shifter will have a rotation to the left. In this way, when the bypass is not used, the rotation to be used will simply be the same as that of the input, while in the case of bypass the new rotation will be given by the following formula:

$$BYPASS\_ROT = |NEW\_ROT - OLD\_ROT| \tag{6}$$

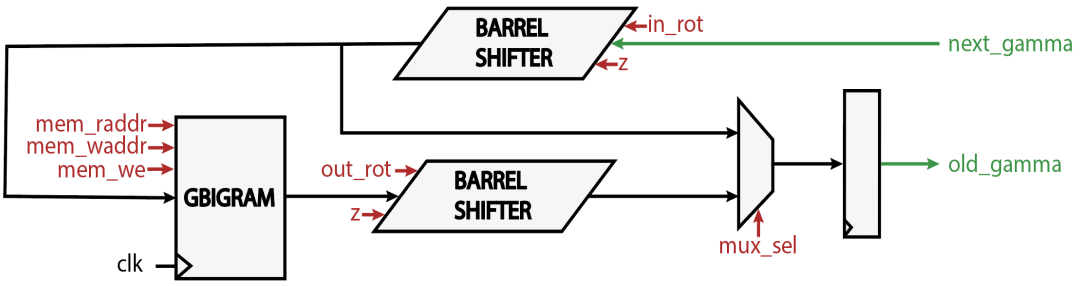

Figure 20: Schematic of PRAM block with bypass insertion

3.3.3 Generation of control signals

This chapter will not deal with the ROM hardware structure since it is a simple Read-Only Memory, which is used in the simulation phase and then removed from the final block since the information stored in it will be directly taken from the MSS. More interesting instead is the discussion behind generating the control signals, which will then be saved in the ROM.

The control signals generation will not be done by hardware, but it will need a software implementation, so we created the Python code rom_generator.py, which simulates the whole PSS and manages the input and output of each block.

When launching the code, it is possible to insert various parameters, including five parameters that determine whether or not registers are present in the structure. The meanings associated with these values are shown below, dividing them into two groups:

- Presence or absence of an output register: this is the case of the output of the GBIGRAM and the input and output of the PE. This variable's value can be chosen between zero and one to simulate the presence or absence of a register at these points is chosen. By default, the values are chosen to register at the output of the GBIGRAM since it is a synchronous block.
  A register is put before the PE to simulate the PRAM block's synchronicity, i.e., the delay between the request of data and obtaining the same. In contrast, no register has been inserted at the output of the PE. Therefore, any value calculated during the second phase will be directly sent to the left barrel shifter present inside the PRAM block.

- Pipeline presence: this case concerns the two barrel shifters since they have to manage a very high workload. Their combinatorial nature could be fatal at the throttle level, finding the critical path in the rotation process. For this reason, the possibility of inserting pipelines in the barrel shifters have been considered in the code.
  The value of delay that can be inserted for these barrel shifters goes from 0 to 3. For the simulation has been chosen to consider barrel shifters without pipelines.

To summarise what has been said so far, the figure that the code will simulate and that will therefore generate the control signals is as follows:

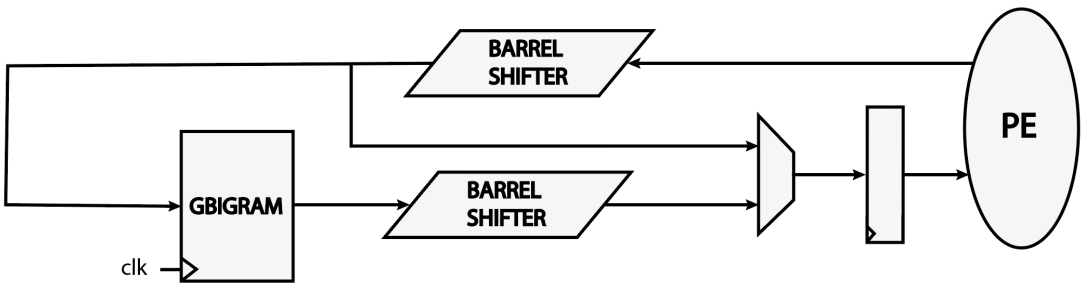

Figure 21: Simulation diagram for generating control signals

It is essential to underline that the simulation carried out by this code considers that all the signals generated arrive simultaneously to both blocks (PRAM and PE). Therefore, the control signals reading must be read simultaneously by both PRAM_CTRL and PE_CTRL since it contains information that can not be delayed.

The remaining parameters must define the type of file to be read to extrapolate the data from the matrix, which will provide the values to be processed and the value of the rotation to be performed. Finally, it is possible to choose the name of the file in which all the control values generated during the simulation will be written.

Having defined the parameters to be inserted during the launch of the code, it is now possible to move on to the simulation description and the various optimizations inserted throughout its duration.
First of all, the code reads the file containing the reference matrix, and from it, it will extrapolate. Only the columns with a value different from zerowill be considered associated with the rotation value for each row.

There will be a first phase processing and a second phase waiting for the end of the latter in the initial situation. So we start by reading columns with non-zero values, and the values sent will then be the read addresses for the GBIGRAM. Every time a data item is read in the GBIGRAM, it is necessary to remember that it will be present in the output only after one clock stroke.
Therefore the rotation value must be delayed by one clock stroke. Moreover, another register is placed between the output of the right barrel shifter and the PE input. Therefore, compared to the data's reading, the signal that asserts the effective start of the first phase will be delayed by two clock strokes.
In particular, the situation that occurs during the processing of the first line is the following:

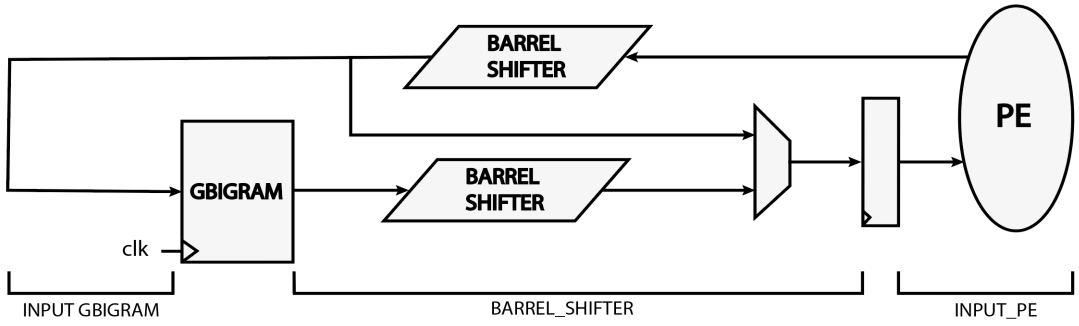

Figure 22: Figure for the study of data propagation within the simulation

INPUT_GBIGRAM	BARREL_SHIFTER	INPUT_PE
0	—	—
1	0	—
2	1	0
3	2	1

The figure shown in the figure emphasizes the concept relating to delays discussed at the beginning of this paragraph and will then be continued throughout the simulation. Once

this element of delay has been assimilated, the first line's simulation proceeds in the same way without any particular point of interest. The noteworthy facts reappear during the elaboration of the second line; in this case, always taking into account the delays, we have the actual termination of the first phase and the beginning of the second phase. This second phase is referring to values relative to the first line, while the first phase refers to values of the second line.

This precise situation offers several points of observation that can be converted into optimization and on which it is indeed worth dwelling.

- The first point to consider is the possibility, discussed above, of carrying out the first phase during a second phase. This decision optimizes the system by far, but in turn, lends itself to further optimization. Considering the matrix in the appendix is possible to note that the various rows that follow each other have several different columns with non-regular values other than zero; sometimes, these numbers are close to each other. In contrast, there is a drastic increase or decrease in the number of columns in other cases.
  The most obvious case is the transition from the third row to the fourth row. We pass from a situation of three rows containing all nineteen columns each to a row containing only three columns.

3	0	121	276	220	201	187	97	4	128
	1	89	87	208	18	145	94	6	23
	3	84	0	30	165	166	49	33	162
	4	20	275	197	5	108	279	113	220
	6	150	199	61	45	82	139	49	43
	7	131	153	175	142	132	166	21	186
	8	243	56	79	16	197	91	6	96
	10	136	132	281	34	41	106	151	1
	11	86	305	303	155	162	246	83	216
	12	246	231	253	213	57	345	154	22
	13	219	341	164	147	36	269	87	24
	14	211	212	53	69	115	185	5	167
	16	240	304	44	96	242	249	92	200
	17	76	300	28	74	165	215	173	32
	18	244	271	77	99	0	143	120	235
	20	144	39	319	30	113	121	2	172
	21	12	357	68	158	108	121	142	219
	22	1	1	1	1	1	1	0	1
	25	0	0	0	0	0	0	0	0
4	0	157	332	233	170	246	42	24	64
	1	102	181	205	10	235	256	204	211
	26	0	0	0	0	0	0	0	0

Figure 23: Row number three and four of the matrix

This fact will inevitably weigh on performance since it is essential to remember that it is not possible to finish the first phase until the second phase is finished since, at the end of the first phase, the value is saved inside the G_NEW_REG. Therefore, it is not possible to alter that value until the second phase that is using it is finished.

This fact will inevitably lead to a system block until all the second phase relative to the fourth line is processed. Moreover, in the following line, the number of colons to analyze increases again, leading to some states in which the second phase relative to the fourth line will be completed. Therefore for some clock strokes, no second phase will be processed and then recovered in the future.

4	0	157	332	233	170	246	42	24	64
	1	102	181	205	10	235	256	204	211
	26	0	0	0	0	0	0	0	0
5	0	205	195	83	164	261	219	185	2
	1	236	14	292	59	181	130	100	171
	3	194	115	50	86	72	251	24	47
	12	231	166	318	80	283	322	65	143
	16	28	241	201	182	254	295	207	210
	21	123	51	267	130	79	258	161	180
	22	115	157	279	153	144	283	72	180
	27	0	0	0	0	0	0	0	0

Figure 24: Row number four and five of the matrix

The above reasoning leads us to look for a solution to avoid this situation and not have, or at least limit, the cases in which the System does not use the Maximum Performance.
For this case, the solution is straightforward. It consists of reordering the rows according to a decreasing criterion, starting from those with the maximum number of columns (19) up to those with the lowest number (3). From this point onwards, we consider that the order of the columns is ordered in descending order at the beginning of the algorithm to apply this optimization.

Before moving on to the following optimization, it is worth pointing out that it reduces partial non-use of the PE block by 50% but does not eliminate these cases. The second phase will always remain unused during the first row, and every now, and then the first phase will need to be stopped when the number of columns to be analyzed decreases from one row to another.

- A further improvement is a possibility of using the bypass, i.e., a System that allows the PE output to be used as input directly, provided that the correct value rotates it. Apart from the possibility of using this bypass or not, the fact remains that many times two consecutive rows need to take data from the same Column. Therefore, it is necessary to wait for that Column to be correctly processed by the second phase. In this case, we take the first two rows as an example since the previously discussed optimization will not change their order since they both have nineteen columns to be analyzed.

0	0	250	307	73	223	211	294	0	135
	1	69	19	15	16	198	118	0	227
	2	226	50	103	94	188	167	0	126
	3	159	369	49	91	186	330	0	134
	5	100	181	240	74	219	207	0	84
	6	10	216	39	10	4	165	0	83
	9	59	317	15	0	29	243	0	53
	10	229	288	162	205	144	250	0	225
	11	110	109	215	216	116	1	0	205
	12	191	17	164	21	216	339	0	128
	13	9	357	133	215	115	201	0	75
	15	195	215	298	14	233	53	0	135
	16	23	106	110	70	144	347	0	217
	18	190	242	113	141	95	304	0	220
	19	35	180	16	198	216	167	0	90
	20	239	330	189	104	73	47	0	105
	21	31	346	32	81	261	188	0	137
	22	1	1	1	1	1	1	0	1
	23	0	0	0	0	0	0	0	0
1	0	2	76	303	141	179	77	22	96
	2	239	76	294	45	162	225	11	236
	3	117	73	27	151	223	96	124	136
	4	124	288	261	46	256	338	0	221
	5	71	144	161	119	160	268	10	128
	7	222	331	133	157	76	112	0	92
	8	104	331	4	133	202	302	0	172
	9	173	178	80	87	117	50	2	56
	11	220	295	129	206	109	167	16	11
	12	102	342	300	93	15	253	60	189
	14	109	217	76	79	72	334	0	95
	15	132	99	266	9	152	242	6	85
	16	142	354	72	118	158	257	30	153
	17	155	114	83	194	147	133	0	87
	19	255	331	260	31	156	9	168	163
	21	28	112	301	187	119	302	31	216
	22	0	0	0	0	0	0	105	0
	23	0	0	0	0	0	0	0	0
	24	0	0	0	0	0	0	0	0

Figure 25: Row number zero and one of the matrix

As it is possible to see, these rows have many columns in common; specifically they are the following: [0, 2, 3, 5, 9, 11, 12, 15, 16, 19, 21, 22, 23] while the new columns are [4, 7, 8, 14, 17, 24].

It is possible to notice the disparity between old and new columns and that most of the first columns to be analyzed are all part of the columns already analyzed in the previous column. Therefore, it still needs to be processed by the second phase of the PE to be used again.
If this order of columns is maintained, there will inevitably be new wait statements in the system, since even to use the bypass, it is necessary to wait for the second phase to be processed. Moreover, there would be clock strikes in which an old Column is accessed, possibly losing the possibility of using the bypass to follow the order initially chosen.

At this point, it is also possible to reorder the columns; this reordering assumes that

the simulation of this code takes into account the columns processed previously and those processed in this new row and creates an order for the processing of the second phase so that the columns that are to be used in this new row are processed first, and then those that will not be used in this row are rewritten in the bigram.

Unlike the previous reordering, the one proposed now has consequences that need to be managed. The main consequence is that before this change, during the first phase, the gram was accessed by merely increasing the index value by one and resetting it to zero for each new row, and this method would also be used for the second phase. Now it is no longer possible to do this since, during the second phase, the values are not accessed in consecutive order but by processing first the columns that can be used for the bypass and then the others.
Without taking this into account, there is a risk of writing a value into the gram during the first phase that has not yet been processed in the second phase, as shown in the figure below.

RADDR	WADDR	WDATA	DATA(0)	DATA(1)	DATA(2)	RDATA
0	0	O'γ(0)	Oγ(0)	Oγ(1)	Oγ(2)	—
2	1	O'γ(1)	O'γ(0)	Oγ(1)	Oγ(2)	Oγ(0)
3	2	O'γ(2)	O'γ(0)	O'γ(1)	Oγ(2)	Oγ(2)

This problem can be solved by assigning the same index as the write address of the gram in the first phase, as the value used for reading in the second phase. In this way, it is always confident that the value to be overwritten has been read previously, thus obtaining the following situation:

RADDR	WADDR	WDATA	DATA(0)	DATA(1)	DATA(2)	RDATA
0	0	O'γ(0)	Oγ(0)	Oγ(1)	Oγ(2)	—
2	2	O'γ(1)	O'γ(0)	Oγ(1)	Oγ(2)	Oγ(0)
3	3	O'γ(2)	O'γ(0)	Oγ(1)	O'γ(1)	Oγ(2)

Finally, thanks to the reordering of the columns in the second phase, the situation will be such that when using the bypass, it will continue to be used until all the columns in common between the new and old rows have been processed, and then proceed with processing the columns not present in the previous row. Simultaneously, the new values of the columns present in the old row but not in the new one are saved in the GBIGRAM.

- A final consideration must be made regarding the use of the bypass: what must be borne in mind is that, at the moment in which the bypass is used, the entire structure that is usually used for the PE input must remain frozen for the entire duration of the bypass.
  A first optimization consists of not changing the GBIGRAM reading address's value, which allows starting from where you left off as soon as the bypass is complete and

reduces any power consumption you might have had by setting the value to zero. It is also essential to think about the case where the right barrel shifter needs to be pipelined. In such a situation, the data inside the right barrel shifter would be lost, as shown in the figure below:

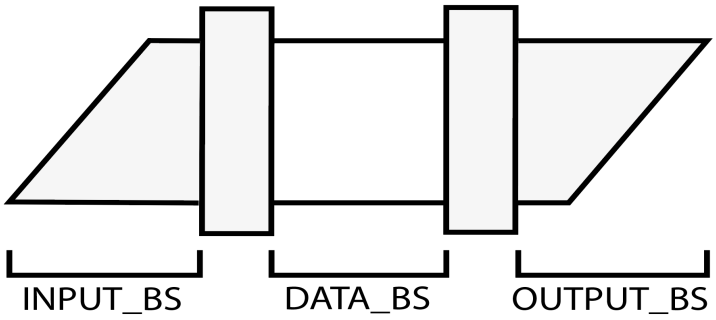

Figure 26: Data propagation in a pipelined barrel shifter

BYPASS	INPUT_BS	DATA_BS	OUTPUT_BS
0	Oγ(0)[rot(0)]	—	—-
0	Oγ(1)[rot(1)]	Oγ(0)[rot(0)]	—
0	Oγ(2)[rot(2)]	Oγ(1)[rot(1)]	Oγ(0)[rot(0)]
0	Oγ(3)[rot(3)]	Oγ(2)[rot(2)]	Oγ(1)[rot(1)]
1	Oγ(4)[rot(4)]	Oγ(3)[rot(3)]	Oγ(2)[rot(2)]
1	Oγ(4)[rot(4)]	Oγ(4)[rot(4)]	Oγ(3)[rot(3)]
1	Oγ(4)[rot(4)]	Oγ(4)[rot(4)]	Oγ(4)[rot(4)]

Table 2: Example of execution with bypass, data is lost.

To prevent this from happening, it is necessary to disable the enable chip of this barrel shifter for the duration of the bypass so that no data would be lost, and once the bypass is over, you can proceed from the same point where you left off.

BYPASS	CE	INPUT_BS	DATA_BS	OUTPUT_BS
0	1	Oγ(0)[rot(0)]	—	—-
0	1	Oγ(1)[rot(1)]	Oγ(0)[rot(0)]	—
0	1	Oγ(2)[rot(2)]	Oγ(1)[rot(1)]	Oγ(0)[rot(0)]
0	1	Oγ(3)[rot(3)]	Oγ(2)[rot(2)]	Oγ(1)[rot(1)]
1	0	Oγ(4)[rot(4)]	Oγ(3)[rot(3)]	Oγ(2)[rot(2)]
1	0	Oγ(4)[rot(4)]	Oγ(3)[rot(3)]	Oγ(2)[rot(2)]
1	0	Oγ(4)[rot(4)]	Oγ(3)[rot(3)]	Oγ(2)[rot(2)]

Table 3: Thanks to the enable chip, there is no more data loss.

Having introduced these optimizations, it is now possible to proceed with the simulation and arrive directly at the final phase. The processing of all the other lines will follow the behavior described up to this point, making them unnoticeable. At the end of the process, instead, it has been chosen to have a waiting situation in which all the processed values from the second phase of the PE relative to the last analyzed line are stored again following the new ordering.

At this point, you have all the signals you want, in fact, during the whole simulation, you have kept track of all the signals, and at the end, there is the possibility to choose which of them are considered necessary to store inside the ROM to control the whole PSS.
The choice of the signals to be extrapolated is divided as follows:

- Signals to be sent to the PE_CTRL:
  * **last_index**: this signal is necessary to let the block know when the line related to the first phase still under analysis ends, and therefore the load value of the G_NEW_REG must be raised.
  * **st_phase**: this value is needed because, as discussed above, during the decoding phase, it is necessary to know when the first phase is active or not. In case of inactivity, it is necessary to disable we of the GBIGRAM and the load of the R_NEW_REG.
  * **phase1_index**: this index must be passed by the control because of the second optimisation described.
  * **gram_raddr**: the control must always pass this address because of the second optimization described. It will be delayed by one clock stroke and used as an index of the second phase; in fact, the read address's value must arrive one clock stroke earlier, as the data will be read with one clock stroke delay.
- Signals to be sent to PRAM_CTRL:
  * **gbigram_raddr**: to manage the reading of columns

* **gbigram_we**: to manage the writing of data processed by the PE
* **gbigram_waddr**: to manage the writing of data processed by the PE
* **right_rot**: to manage the rotation value of the right barrel shifter
* **left_rot**: to manage the rotation value of the left barrel shifter
* **bypass**: to manage the sel of the mux which decides which data to pass, and its negated value is used to manage the enable chip of the right barrel shifter.

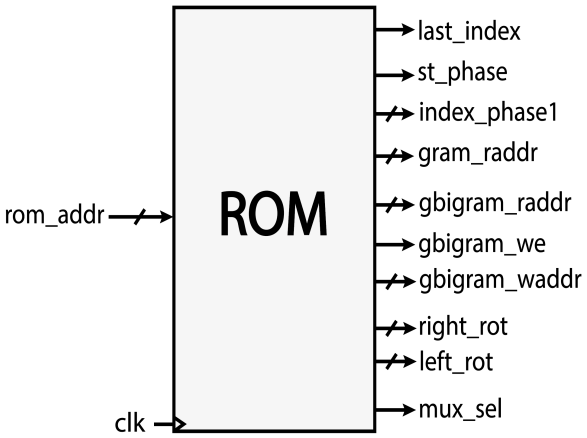

Figure 27: List of signals generated by the control software

It is concluded the discussion of the three main blocks and proceeds to describe their three control blocks.

3.4 Implementation of control blocks

Unlike the description of the three main blocks, which started from the last block and went up to the first, the control blocks' description will be in order from right to left. Before, it was interesting to ask what kind of signals are needed to generate them; now, the PSS flow is directly shown.

3.4.1 ROM_CTRL

This block is initially in an IDLE state and waits for the start signal from the general control block; in addition to the start signal, different signals will be sent to it, containing information about the starting address at which to read the ROM and its length. Another signal determines the value of the lifting size chosen and, finally, a feedback signal that warns the block when decoding is complete.

Therefore, this block's task is twofold: firstly, it must send information about the value of the lifting size (z) to the pram_ctrl block. Secondly, it must manage the ROM address increase until the last address is read and then proceed to start again until the feedback signal is received.

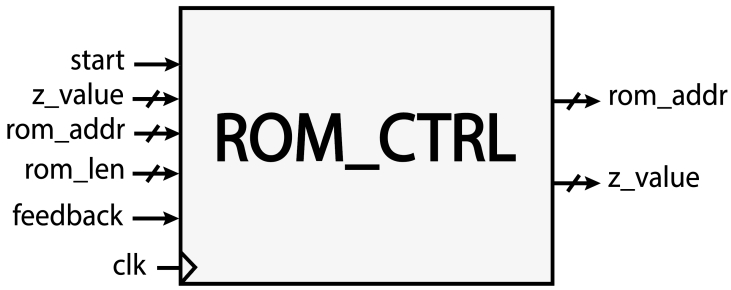

Figure 28: Schematic of ROM_CTRL

3.4.2 PRAM_CTRL

This block receives as input the control signals coming from the ROM and the lifting size value signal coming from the ROM_CTRL, the output of this block consists in picking up among all the control signals only those with information about the PRAM block and adding the lifting size value.

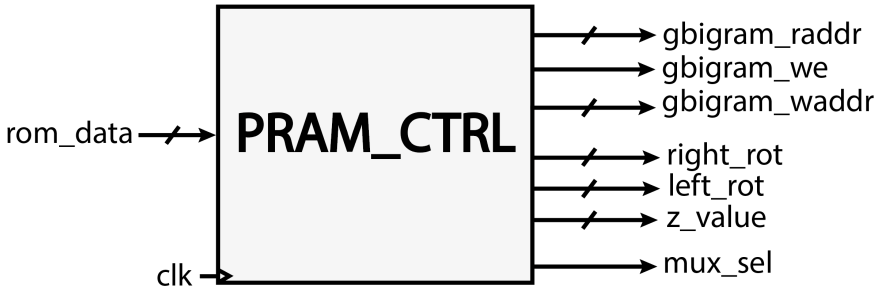

Figure 29: Schematic of PRAM_CTRL

3.4.3 PE_CTRL

This control is the most complicated of the three; it receives the ROM output vector as the only input and extrapolates from it the four values relative to its block. It will be necessary to generate the values for all the signals not managed by the software.

The gram_raddr signal must be sent as it is received for the value of the gram_raddr. Its value must also be delayed by one clock stroke and sent later as the value for the second phase index. These two signals and the value stored inside the G_NEW_REG represent all that is necessary to manage the second phase inside the PE.

This leaves the first phase management and, for this purpose, the last_index, st_phase, and phase1_index signals. The last of these values will be sent to the PE as the index value of phase one and will then be used for the gram_waddr.

At this point, the two remaining signals flanked by a line counter will handle all remaining parameters. The important points to keep in mind are the following:

- **Last index**: the PE must take care of the correct storing of the last R_NEW value calculated in G_NEW_REG and, therefore, the value of gnew_load must be asserted

high. It is also necessary to read the following RRAM address's value, which must be incremented by one unless it is at the last address. Therefore, it is necessary to set the following to zero.

- **First index**: The value of gnew_load has to be set to zero again, while the value calculated in the previous step has to be written in the RRAM, so the value of RRAM_RNW has to be set high, and the signal st_rnew has to be asserted.
- **Others index**: the RRAM must only be read, and the value of gnew_load must be set to zero.

In general, if the phase is active, gram_we is enabled; otherwise, it is set to zero. In addition to managing the RRAM addresses, the counter is used to understand when a new iteration occurs and, therefore, set the st_iter signal to zero.

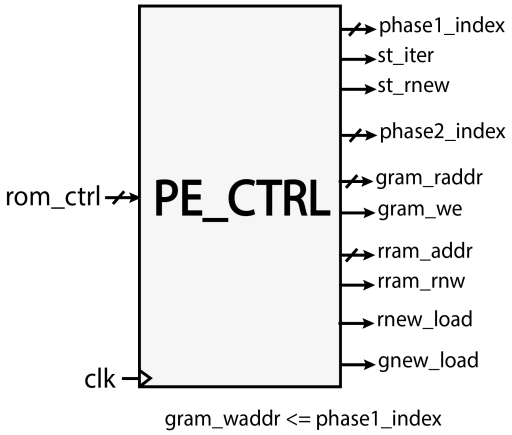

Figure 30: Schematic of the PE_CTRL

3.4.4 Timing diagarm

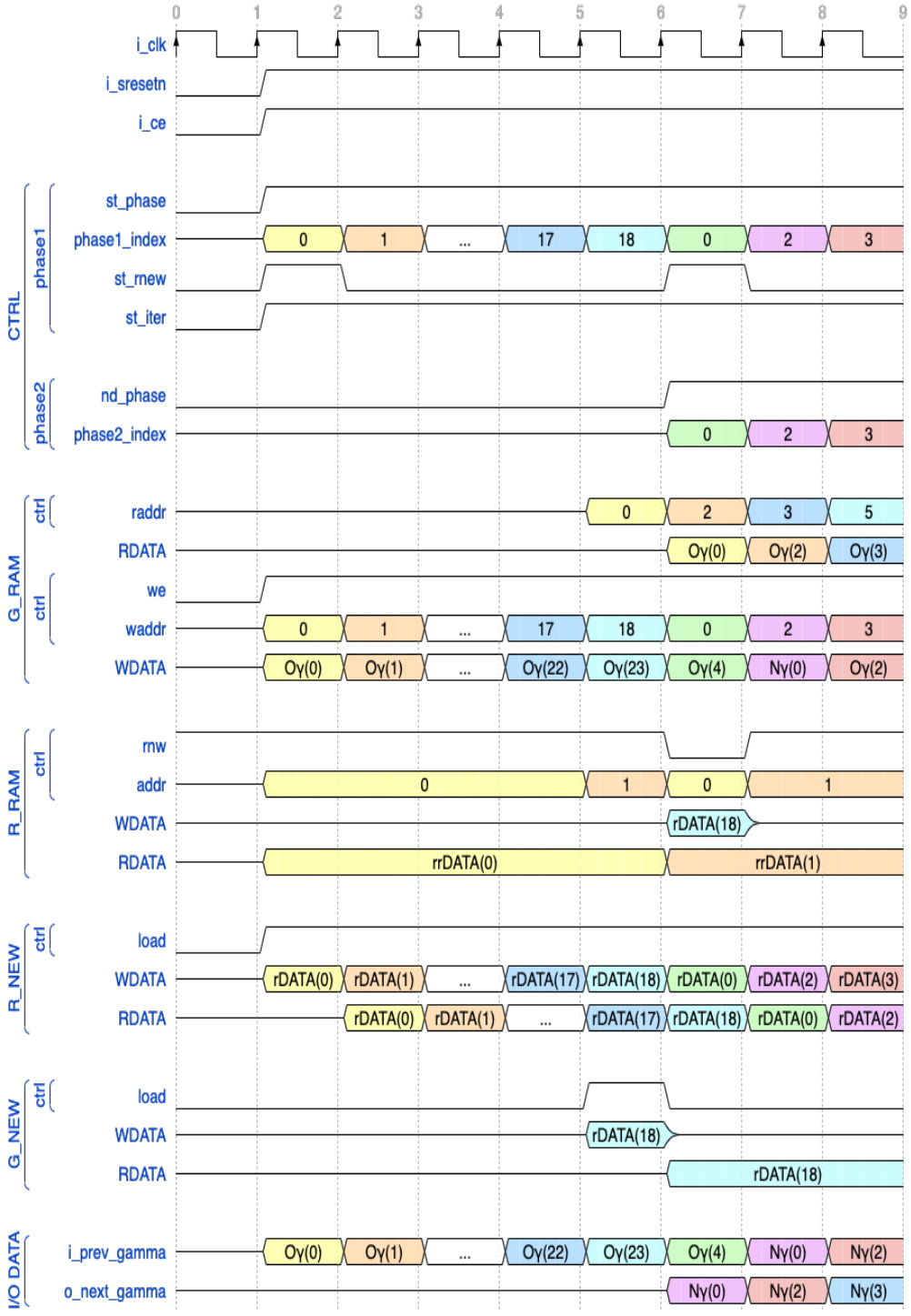

Figure 31: Example of the execution of the first two lines by the PE

The figure above shows the processing of the first and second columns of the matrix inside the PE. What is described in this example is sufficient to understand the general functioning of the PE. In fact, it contains all the peculiarities, except for the case in which, when passing from one row to the next, the number of columns to be processed in it is lower than in the previous row; however, as previously explained, this will only disable the first phase until the second phase is finished, and then it will resume the behavior shown in this example.
Once again, disabling the first phase disables all signals that modify the contents of memory elements so that the following signals will be set to zero: gram_we, r_new_load, g_new_load, and the value of rram_rnw will be set to one.

Once again, the peculiarities to be noted are all present in the area of the change between two columns, in fact, what happens every time we switch to a new column, thus ending the first phase and moving on to a new first phase and the processing of the second phase following the first just processed. During this phase of change, it is good to note the following elements:

- The management of reading and writing of the RRAM memory: as explained above, at the end of a first phase and the beginning of the next, it is necessary first to update the value read from the RRAM and then, at the first index of the new phase, to write the value previously calculated to the previous address.
- GRAM memory management: this confirms the choice of a dual-port memory for storing gamma values; in fact, it can be seen that not considering the first phase, this memory is constantly accessed in both reading and writing. It is also worth noting that the GRAM read address's value is always provided one clock stroke earlier than the index of the second phase.
- The management of the G_NEW register: the value in this register is modified at the end of the first phase. The new value contained in it is used during the second phase.
- The management of the register R_NEW: which continually updates the value inside it as long as the first phase is active.
- The value of the indices of the first and second phase. As discussed after the first line, the value of the first phase's indices will be set by the second phase and will no longer follow a simple increment of them. This will ensure that old range values are not overwritten before being processed by the second phase.

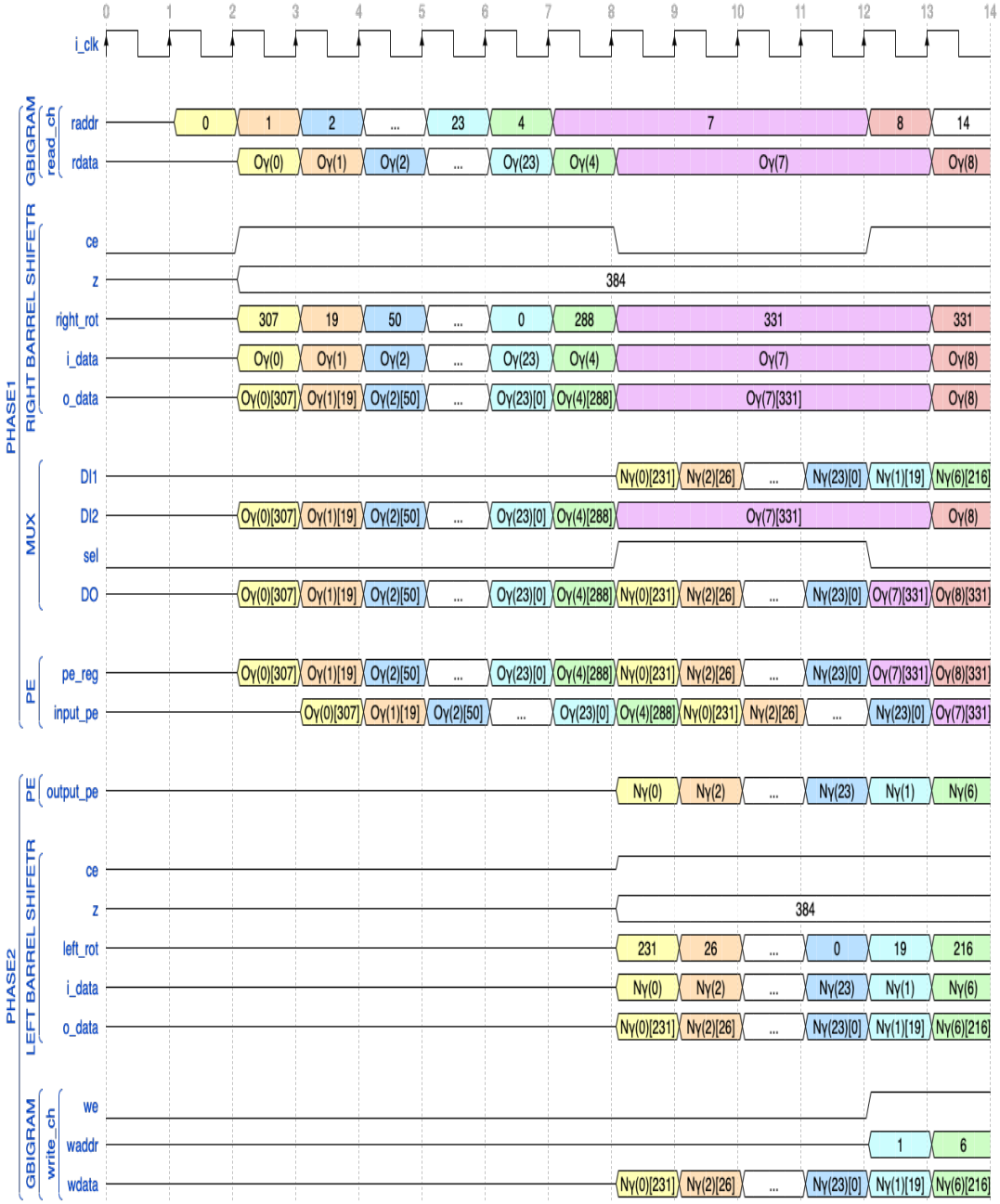

Figure 32: Example of PRAM execution of the first two lines

This diagram shows the PRAM trend during the execution of the first two lines. Again the points of interest are in the passage from the first line to the second line. In particular, the data of interest related to the possibility of using the bypass and studying how the system reacts to it.

When the bypass is used, the reading address of the GBIGRAM remains constant so that the same data can continue to be read. This choice is because instead of putting a standard address for the bypass cases and then going back to the execution once it is finished. Keep a constant address to avoid useless switching activity and have the data ready when the bypass is no longer active.

Another factor to be considered is the rotation value that is used for all the columns that benefit from the bypass. In fact, it should be noted that this value is not present in the matrix since it must take into account both the rotation value that should have been executed to restore the value of the data plus the new rotation value required for the current row. At the end of the bypass process, the block continues to take the data from the GBIGRAM and send them to the PE, saving in memory the values calculated on the previous line.

4 Conclusions

Finally, it is fair to conclude what has been done and highlight which points have not been touched and could be future implementations to increase performance. Unlike what has been done so far, in this last chapter, the two parts of the project will not be treated separately since the considerations to be made on each of them, and the possible implementations are valid for both projects.

For both projects, no simulations have been carried out that have led to concrete results. For this reason, the first thing to do is to generate a simulation that can guarantee the actual functioning and, in case of negative results, investigate what needs to be changed in the code.

As far as the various optimizations made are concerned, they have been set out in the best possible way during the process of creating these blocks; what we can recommend introducing in a future version of this project is the implementation of a channel coding for transmissions between the various blocks that make up the arbiter, in fact in these blocks in the case of a read or write request with an incremental burst, different addresses will be accessed successively, and therefore it is possible to introduce a coding that allows the access address not to be changed as long as the addresses are successive. This small optimization would avoid all the power consumption due to the switching activity on the address bus.

It is different for the PSS, where different optimization techniques have been used, both for power, such as not changing the data value when in a wait statement, and the various optimizations made when generating the control signals. In this type of block one could think rather than an optimization at the bus level to an introduction of clock gating levels, this because during the use of the library implemented in this document can be divided into three parts:

- Transfer of data to be analyzed inside the MSS and initialization of GBIGRAM
- processing the various iterations for decoding and writing the data inside the MSS.
- Transfer the data inside the MSS to another library op as output

With this subdivision in mind, it is possible to see that the PSS will be active for a little more than a third of these activities (it must be considered that in the first phase, there is also the initialization of GBIGRAM). Therefore, it would be a significant saving to disable the clock during the data transfer phases.

Finally, it is worth noting that two important blocks still need to be added to the PSS implementation:

- Feedback: This block has the role of evaluating, during the processing, the value of the new product ranges and also monitoring the changes of the values during the first phase. This monitoring has the purpose of understanding when the code has been successfully decoded. Therefore, it is possible to terminate the processing and save all the data in the GBIGRAM in the MSS by compressing each sample from eight bits to a single bit.

- General control: This block deals with the interaction of the PSS with the memory. Specifically, it deals with the initialization of the GBIGRAM. This start signal starts the processing, and once successfully decoded all the code, rewrite in memory the values correctly decoded inside the GBIGRAM.

5 References

[1] *AMBA® AXI™ and ACE™ Protocol Specification*, ARM.

[2] *Aspects of Energy Efficient LDPC Decoders*, Erick Amador, 2011.

[3] *Architecture générique de décodeur de codes LDPC*, Frédéric Guilloud, 2004.

[4] *Efficient Hardware Implementations of LDPC Decoders, through Exploiting Impreciseness in Message-Passing Decoding Algorithms*, Thien Truong Nguyen Ly, 2018.

[5] *3GPP TS 38.212 V16.2.0*, 2020.

[6] *Low Density Parity Check decoder (LDPC) documentation*, 2020.

6 Annex

H_{BG}		$V_{i,j}$							
Row index	Column index	Set index i_{LS}							
i	j	0	1	2	3	4	5	6	7
0	0	250	307	73	223	211	294	0	135
	1	69	19	15	16	198	118	0	227
	2	226	50	103	94	188	167	0	126
	3	159	369	49	91	186	330	0	134
	5	100	181	240	74	219	207	0	84
	6	10	216	39	10	4	165	0	83
	9	59	317	15	0	29	243	0	53
	10	229	288	162	205	144	250	0	225
	11	110	109	215	216	116	1	0	205
	12	191	17	164	21	216	339	0	128
	13	9	357	133	215	115	201	0	75
	15	195	215	298	14	233	53	0	135
	16	23	106	110	70	144	347	0	217
	18	190	242	113	141	95	304	0	220
	19	35	180	16	198	216	167	0	90
	20	239	330	189	104	73	47	0	105
	21	31	346	32	81	261	188	0	137
	22	1	1	1	1	1	1	0	1
	23	0	0	0	0	0	0	0	0
1	0	2	76	303	141	179	77	22	96
	2	239	76	294	45	162	225	11	236
	3	117	73	27	151	223	96	124	136
	4	124	288	261	46	256	338	0	221
	5	71	144	161	119	160	268	10	128
	7	222	331	133	157	76	112	0	92
	8	104	331	4	133	202	302	0	172
	9	173	178	80	87	117	50	2	56
	11	220	295	129	206	109	167	16	11
	12	102	342	300	93	15	253	60	189
	14	109	217	76	79	72	334	0	95
	15	132	99	266	9	152	242	6	85
	16	142	354	72	118	158	257	30	153
	17	155	114	83	194	147	133	0	87
	19	255	331	260	31	156	9	168	163
	21	28	112	301	187	119	302	31	216
	22	0	0	0	0	0	0	105	0
	23	0	0	0	0	0	0	0	0
	24	0	0	0	0	0	0	0	0
2	0	106	205	68	207	258	226	132	189
	1	111	250	7	203	167	35	37	4
	2	185	328	80	31	220	213	21	225
	4	63	332	280	176	133	302	180	151
	5	117	256	38	180	243	111	4	236
	6	93	161	227	186	202	265	149	117
	7	229	267	202	95	218	128	48	179
	8	177	160	200	153	63	237	38	92
	9	95	63	71	177	0	294	122	24
	10	39	129	106	70	3	127	195	68
	13	142	200	295	77	74	110	155	6
	14	225	88	283	214	229	286	28	101
	15	225	53	301	77	0	125	85	33
	17	245	131	184	198	216	131	47	96
	18	205	240	246	117	269	163	179	125
	19	251	205	230	223	200	210	42	67
	20	117	13	276	90	234	7	66	230
	24	0	0	0	0	0	0	0	0
	25	0	0	0	0	0	0	0	0
3	0	121	276	220	201	187	97	4	128
	1	89	87	208	18	145	94	6	23
	3	84	0	30	165	166	49	33	162
	4	20	275	197	5	108	279	113	220
	6	150	199	61	45	82	139	49	43
	7	131	153	175	142	132	166	21	186
	8	243	56	79	16	197	91	6	96
	10	136	132	281	34	41	106	151	1
	11	86	305	303	155	162	246	83	216
	12	246	231	253	213	57	345	154	22
	13	219	341	164	147	36	269	87	24
	14	211	212	53	69	115	185	5	167
	16	240	304	44	96	242	249	92	200
	17	76	300	28	74	165	215	173	32
	18	244	271	77	99	0	143	120	235
	20	144	39	319	30	113	121	2	172
	21	12	357	68	158	108	121	142	219
	22	1	1	1	1	1	1	0	1
	25	0	0	0	0	0	0	0	0
4	0	157	332	233	170	246	42	24	64
	1	102	181	205	10	235	256	204	211
	26	0	0	0	0	0	0	0	0
5	0	205	195	83	164	261	219	185	2
	1	236	14	292	59	181	130	100	171
	3	194	115	50	86	72	251	24	47
	12	231	166	318	80	283	322	65	143
	16	28	241	201	182	254	295	207	210

H_{BG}		$V_{i,j}$							
Row index	Column index	Set index i_{LS}							
i	j	0	1	2	3	4	5	6	7
15	1	96	2	290	120	0	348	6	138
	10	65	210	60	131	183	15	81	220
	13	63	318	130	209	108	81	182	173
	18	75	55	184	209	68	176	53	142
	25	179	269	51	81	64	113	46	49
	37	0	0	0	0	0	0	0	0
16	1	64	13	69	154	270	190	88	78
	3	49	338	140	164	13	293	198	152
	11	49	57	45	43	99	332	160	84
	20	51	289	115	189	54	331	122	5
	22	154	57	300	101	0	114	182	205
	38	0	0	0	0	0	0	0	0
17	0	7	260	257	56	153	110	91	183
	14	164	303	147	110	137	228	184	112
	16	59	81	128	200	0	247	30	106
	17	1	358	51	63	0	116	3	219
	21	144	375	228	4	162	190	155	129
	39	0	0	0	0	0	0	0	0
18	1	42	130	260	199	161	47	1	183
	12	233	163	294	110	151	286	41	215
	13	8	280	291	200	0	246	167	180
	18	155	132	141	143	241	181	68	143
	19	147	4	295	186	144	73	148	14
	40	0	0	0	0	0	0	0	0
19	0	60	145	64	8	0	87	12	179
	1	73	213	181	6	0	110	6	108
	7	72	344	101	103	118	147	166	159
	8	127	242	270	198	144	258	184	138
	10	224	197	41	8	0	204	191	196
	41	0	0	0	0	0	0	0	0
20	0	151	187	301	105	265	89	6	77
	3	186	206	162	210	81	65	12	187
	9	217	264	40	121	90	155	15	203
	11	47	341	130	214	144	244	5	167
	22	160	59	10	183	228	30	30	130
	42	0	0	0	0	0	0	0	0
21	1	249	205	79	192	64	162	6	197
	5	121	102	175	131	46	264	86	122
	16	109	328	132	220	266	346	96	215
	20	131	213	283	50	9	143	42	65
	21	171	97	103	106	18	109	199	216
	43	0	0	0	0	0	0	0	0
22	0	64	30	177	53	72	280	44	25
	12	142	11	20	0	189	157	58	47
	13	188	233	55	3	72	236	130	126
	17	158	22	316	148	257	113	131	178
	44	0	0	0	0	0	0	0	0
23	1	156	24	249	88	180	18	45	185
	2	147	89	50	203	0	6	18	127
	10	170	61	133	168	0	181	132	117
	18	152	27	105	122	165	304	100	199
	45	0	0	0	0	0	0	0	0
24	0	112	298	289	49	236	38	9	32
	3	86	158	280	157	199	170	125	178
	4	236	235	110	64	0	249	191	2
	11	116	339	187	193	266	288	28	156
	22	222	234	281	124	0	194	6	58
	46	0	0	0	0	0	0	0	0
25	1	23	72	172	1	205	279	4	27
	6	136	17	295	166	0	255	74	141
	7	116	383	96	65	0	111	16	11
	14	182	312	46	81	183	54	28	181
	47	0	0	0	0	0	0	0	0
26	0	195	71	270	107	0	325	21	163
	2	243	81	110	176	0	326	142	131
	4	215	76	318	212	0	226	192	169
	15	61	136	67	127	277	99	197	98
	48	0	0	0	0	0	0	0	0
27	1	25	194	210	208	45	91	98	165
	6	104	194	29	141	36	326	140	232
	8	194	101	304	174	72	268	22	9
	49	0	0	0	0	0	0	0	0
28	0	128	222	11	146	275	102	4	32
	4	165	19	293	153	0	1	1	43
	19	181	244	50	217	155	40	40	200
	21	63	274	234	114	62	167	93	205
	50	0	0	0	0	0	0	0	0
29	1	86	252	27	150	0	273	92	232
	14	236	5	308	11	180	104	136	32
	18	84	147	117	53	0	243	106	118
	25	6	78	29	68	42	107	6	103
	51	0	0	0	0	0	0	0	0
30	0	216	159	91	34	0	171	2	170
	10	73	229	23	130	90	16	88	199

Figure 33: First part of the matrix from which the data will be extrapolated for the generation of checkmarks

	21	123	51	267	130	79	258	161	180
	22	115	157	279	153	144	283	72	180
	27	0	0	0	0	0	0	0	0
6	0	183	278	289	158	80	294	6	199
	6	22	257	21	119	144	73	27	22
	10	28	1	293	113	169	330	163	23
	11	67	351	13	21	90	99	50	100
	13	244	92	232	63	59	172	48	92
	17	11	253	302	51	177	150	24	207
	18	157	18	138	136	151	284	38	52
	20	211	225	235	116	108	305	91	13
	28	0	0	0	0	0	0	0	0
7	0	220	9	12	17	169	3	145	77
	1	44	62	88	76	189	103	88	146
	4	159	316	207	104	154	224	112	209
	7	31	333	50	100	184	297	153	32
	8	167	290	25	150	104	215	159	166
	14	104	114	76	158	164	39	76	18
	29	0	0	0	0	0	0	0	0
8	0	112	307	295	33	54	348	172	181
	1	4	179	133	95	0	75	2	105
	3	7	165	130	4	252	22	131	141
	12	211	18	231	217	41	312	141	223
	16	102	39	296	204	98	224	96	177
	19	164	224	110	39	46	17	99	145
	21	109	368	269	58	15	59	101	199
	22	241	67	245	44	230	314	35	153
	24	90	170	154	201	54	244	116	38
	30	0	0	0	0	0	0	0	0
9	0	103	366	189	9	162	156	6	169
	1	182	232	244	37	159	88	10	12
	10	109	321	36	213	93	293	145	206
	11	21	133	286	105	134	111	53	221
	13	142	57	151	89	45	92	201	17
	17	14	303	267	185	132	152	4	212
	18	61	63	135	109	76	23	164	92
	20	216	82	209	218	209	337	173	205
	31	0	0	0	0	0	0	0	0
10	1	98	101	14	82	178	175	126	116
	2	149	339	80	165	1	253	77	151
	4	167	274	211	174	28	27	156	70
	7	160	111	75	19	267	231	16	230
	8	49	383	161	194	234	49	12	115
	14	58	354	311	103	201	267	70	84
	32	0	0	0	0	0	0	0	0
11	0	77	48	16	52	55	25	184	45
	1	41	102	147	11	23	322	194	115
	12	83	8	290	2	274	200	123	134
	16	182	47	289	35	181	351	16	1
	21	78	188	177	32	273	166	104	152
	22	252	334	43	84	39	338	109	165
	23	22	115	280	201	26	192	124	107
	33	0	0	0	0	0	0	0	0
12	0	160	77	229	142	225	123	6	186
	1	42	186	235	175	162	217	20	215
	10	21	174	169	136	244	142	203	124
	11	32	232	48	3	151	110	153	180
	13	234	50	105	28	238	176	104	98
	18	7	74	52	182	243	76	207	80
	34	0	0	0	0	0	0	0	0
13	0	177	313	39	81	231	311	52	220
	3	248	177	302	56	0	251	147	185
	7	151	266	303	72	216	265	1	154
	20	185	115	160	217	47	94	16	178
	23	62	370	37	78	36	81	46	150
	35	0	0	0	0	0	0	0	0
14	0	206	142	78	14	0	22	1	124
	12	55	248	299	175	186	322	202	144
	15	206	137	54	211	253	277	118	182
	16	127	89	61	191	16	156	130	95
	17	16	347	179	51	0	66	1	72
	21	229	12	258	43	79	78	2	76
	36	0	0	0	0	0	0	0	0
15	0	40	241	229	90	170	176	173	39

	13	120	260	105	210	252	95	112	26
	24	9	90	135	123	173	212	20	105
	52	0	0	0	0	0	0	0	0
31	1	95	100	222	175	144	101	4	73
	7	177	215	308	49	144	297	49	149
	22	172	258	66	177	166	279	125	175
	25	61	256	162	128	19	222	194	108
	53	0	0	0	0	0	0	0	0
32	0	221	102	210	192	0	351	6	103
	12	112	201	22	209	211	265	126	110
	14	199	175	271	58	36	338	63	151
	24	121	287	217	30	162	83	20	211
	54	0	0	0	0	0	0	0	0
33	1	2	323	170	114	0	56	10	199
	2	187	8	20	49	0	304	30	132
	11	41	361	140	161	76	141	6	172
	21	211	105	33	137	18	101	92	65
	55	0	0	0	0	0	0	0	0
34	0	127	230	187	82	197	60	4	161
	7	167	148	296	186	0	320	153	237
	15	164	202	5	68	108	112	197	142
	17	159	312	44	150	0	54	155	180
	56	0	0	0	0	0	0	0	0
35	1	161	320	207	192	199	100	4	231
	6	197	335	158	173	278	210	45	174
	12	207	2	55	26	0	195	168	145
	22	103	266	285	187	205	268	185	100
	57	0	0	0	0	0	0	0	0
36	0	37	210	259	222	216	135	6	11
	14	105	313	179	157	16	15	200	207
	15	51	297	178	0	0	35	177	42
	18	120	21	160	6	0	188	43	100
	58	0	0	0	0	0	0	0	0
37	1	198	269	298	81	72	319	82	59
	13	220	82	15	195	144	236	2	204
	23	122	115	115	138	0	85	135	161
	59	0	0	0	0	0	0	0	0
38	0	167	185	151	123	190	164	91	121
	9	151	177	179	90	0	196	64	90
	10	157	289	64	73	0	209	198	26
	12	163	214	181	10	0	246	100	140
	60	0	0	0	0	0	0	0	0
39	1	173	258	102	12	153	236	4	115
	3	139	93	77	77	0	264	28	188
	7	149	346	192	49	165	37	109	168
	19	0	297	208	114	117	272	188	52
	61	0	0	0	0	0	0	0	0
40	0	157	175	32	67	216	304	10	4
	8	137	37	80	45	144	237	84	103
	17	149	312	197	96	2	135	12	30
	62	0	0	0	0	0	0	0	0
41	1	167	52	154	23	0	123	2	53
	3	173	314	47	215	0	77	75	189
	9	139	139	124	60	0	25	142	215
	18	151	288	207	167	183	272	128	24
	63	0	0	0	0	0	0	0	0
42	0	149	113	226	114	27	288	163	222
	4	157	14	65	91	0	83	10	170
	24	137	218	126	78	35	17	162	71
	64	0	0	0	0	0	0	0	0
43	1	151	113	228	206	52	210	1	22
	16	163	132	69	22	243	3	163	127
	18	173	114	176	134	0	53	99	49
	25	139	168	102	161	270	167	98	125
	65	0	0	0	0	0	0	0	0
44	0	139	80	234	84	18	79	4	191
	7	157	78	227	4	0	244	6	211
	9	163	163	259	9	0	293	142	187
	22	173	274	260	12	57	272	3	148
	66	0	0	0	0	0	0	0	0
45	1	149	135	101	184	168	82	181	177
	6	151	149	228	121	0	67	45	114
	10	167	15	126	29	144	235	153	93
	67	0	0	0	0	0	0	0	0

Figure 34: Second part of the matrix from which the data will be extrapolated for the generation of checkmarks

Acknowledgement

The drafting of this thesis has been very demanding as it comes as the completion of a long professional but above all personal journey that I have not always been able to sustain on my own. I therefore occupy this space with the specific intention of thanking the people who have supported me, helped me not to give up or even just helped me to distract myself so that I could recharge my energy.

A special “Thank You” goes to my family to whom, in addition to dedicating this work, I dedicate all my gratitude for having put up with me and supported me over the years.

Thanks go to my grandmother, Maria Teresa, for always bringing a smile to my face and for funding my hobbies on a monthly basis.

Thanks to Andrea and Daniel, two friends with whom I can always feel at home and with whom I have shared my passions and my life for more than ten years now.

Thanks to Matteo, a dear friend with whom I have shared the beginning of my career path since my first year at university and with whom, I hope, I will be able to spend many more years together. He is always able to give me a feeling of freshness and provide me with a glimpse into a world far removed from my own.

Thanks to Nicolò, in him, I found a sincere friend in a place and experience that was new and alienating for me, allowing me to have a serene memory of a dark period. Without him my whole journey would have been very different, and I therefore owe much of the satisfaction I have had to him.

Thanks to Angelo, with whom I share everything, the family degrees transcend when life puts you in front of a person in whom you see so much of yourself again. I thank you for the happiness you give me in each of our conversations.

Thanks to Arianna, a friend with whom I have shared so much, thank you for growing together.

Thanks to Lorenzo and Giulia, two excellent housemates and friends, with whom I have always enjoyed chatting over meals and sharing domestic space. In my time in Antibes, it was always a pleasure to come home.

Thanks to Professor Pacalet, with whom I have had the honor of working closely, giving me a glimpse of the person I would like to become in the future, both professionally and personally.

Thanks to Professor Martina, with whom I collaborated on the final draft of this paper. Thank you for your patience.

Finally, a thank you to all the people I have met throughout this journey, each of them has given me something and will be a part of me forever.