Politecnico di Torino

Master's degree in Electronic Engineering

MASTER THESIS

Study and Development of a Convolutional Architecture based on Distributed RAM

Thesis advisor
Luca Sterpone

Candidate
Marco Bella
S253144

Academic year 2019/2020

Summary

Innovation is the main law ruling the technological world. What is now introduced in the market is required to be unique, intuitive, powerful and smart. These requirements increase the design complexity for the new technologies, which usually must integrate a self-system intelligence.
Nowadays, the technological system intelligence is based on *machine learning* algorithms: in the same way humans learn from previous experiences, a machine improves its performance by learning from previous events or by surrounding environment. Among machine learning approaches, one of the most promising is based on *Convolutional Neural Network (CNN)*. CNN is particularly suited for *image processing* and *real time task completion* two very important issues in digital systems. Convolutional nets basically perform convolution product-based filtering operation on images, in order to map them and extract their features. Edge detection and downscaling are among the others typical techniques to be applied on images. Generally, these kinds of operation require a large dataset to be processed in hardware structure with top-level performances. Among the others, machine learning techniques are assuming a great importance in autonomous driving field of application.
In this thesis work, a new approach for convolution product implementation is proposed. It aims to merge the versatility offered by FPGA with the potentialities of systems based on distributed RAMs, in order to boost designed hardware performance and accomplish operations in a reduced processing time. This need for performance gains is particularly important in autonomous driving field, where a real time inference is required.
Distributed RAM-based structure consists on a different way of managing storing elements in a system. It deals with memory fragmentation to handle smaller and faster resources. Alleviating the need for Device-to-Host and Host-to-Device data transfer is one of its main tasks.
A *Field Programmable Gate Array* (FPGA) is a hardware component that can be reprogrammed by the user every time is needed, using a Hardware Description Language (HDL). Logics implemented on FPGA are automatically optimised in terms of timing, area and power saving. This FPGA features perfectly match with systems needs for optimisations and that's the reason why a FPGA based approach has been investigated.

In this thesis work, a core executing convolution products has been designed for implementation on FPGA. The proposed architecture executes convolution products on vectors. These are *feature vector* (the first input vector, representing data to be filtered through convolution operation), *filter* or *weight* (the second input vector, relating to the chosen filtering technique) and *feature map* (the output vector, containing convolution product results).
The project has been designed to be tested on a PYNQ-Z2 board, integrating an embedded processor (ARM Cortex-A9) and a programmable logic portion, provided by Xilinx. It contains 13,300 logic slices and up to 640 KB RAM resources to be allocated by the user. They are called block RAMs and represent the more dedicated resources for storage in FPGAs. An optimal integration of embedded processor, convolution core and block RAM resources is supposed to be achieved, in order to obtain better performance with respect to standard approaches. An architecture that melts together block RAM and convolution core has been proposed. It is integrated so that the core interfaces with three block ram units. Every unit is called BRAM36 and corresponds to a 36Kb RAM memory. Memory A contains feature vector to be filtered, memory W is filled with filtering data, while memory B storages feature map vector. Embedded processor role consists on writing input data in A and W memories and reading the related results from memory B. AXI4 interface has been used to manage communication between the processor and the designed hardware. Every components behaviour has been proved through *ModelSim* simulation environment, before the actual implementation. A part for the above-mentioned board, the testing environment has included *Vivado* synthesis and implementation tool for HDL designs, *Vitis* compiler for C written code relating to the embedded processor and *Tera Term* terminal emulator program, used to print on screen the test results.
Actually three kind of architecture have been tested. The *first* consists of ARM processor and convolution core only, it represents a typical FPGA based application and has been implemented as a yardstick for distributed RAM-based architecture. The *second* represent this thesis innovation and consists of ARM processor, convolution core and single ports BRAMs. Once A and W memories are written by the processor, the core starts computing feature maps vector. When B memory is completely written, the processor reads the stored data. The *third* is an optimised version of the second system. The optimisation primarily consists on a revised control of the core, so that its execution starts as soon as possible without waiting for the processor to finish writing operation. This task has been reached by instantiating dual ports BRAM instead of single ports ones.
According to the performed test, the distributed RAM-based architecture outperforms the standard FPGA based structure (Test 1), introducing an average speed up of 1,21 times (Test 2) and 1,43 (Test 3). Resource analysis has been conducted as well, showing that no resource overheads are introduced varying the test.

The innovation proposed in this thesis work proves to be promising both in terms of performances enhance and resource utilisation. A different and unusual way of managing storing elements and the possibility of fragmentation turns out to be an effective approach to boost convolution product operation. The obtained results are not only valid for this specific operation but also for those algorithms based on a sequential access to memory, just like the convolutional one.

Contents

List of Figures vii

List of Tables ix

1 Introduction 1
- 1.1 Machine Learning Techniques . . . 2
- 1.2 Algorithms Analysis . . . 5
  - 1.2.1 Convolution Products . . . 6
  - 1.2.2 Filters . . . 7
  - 1.2.3 Pooling Techniques . . . 9
- 1.3 Introduction To FPGA . . . 11
- 1.4 This Thesis Work Purpose . . . 13

2 Computation RAM: Backgrounds 14
- 2.1 DRAM/SRAM Based Technologies . . . 14
- 2.2 Resistive Memory Based Technologies . . . 16

3 A Distributed RAM Based Approach 20
- 3.1 FPGA Memory Resources . . . 21
  - 3.1.1 Block RAM Features . . . 25
  - 3.1.2 Block Ram Implementation . . . 26

4 Design Of Developed Architecture 32
- 4.1 Convolutional Core Design . . . 32
  - 4.1.1 Designed Hardware Description . . . 34
  - 4.1.2 Designed Hardware Simulations . . . 40
- 4.2 Convolutional Core And Distributed RAM Integration . . . 42
  - 4.2.1 System Simulation . . . 44
- 4.3 Interfacing Issues . . . 44

5 Experimental Results 48
- 5.1 Testing Enviroment . . . 48
  - 5.1.1 Preliminary Results . . . 50

5.2 Tests Results 51
5.2.1 TEST 1: Convolutional Core And Processor 51
5.2.2 TEST 2: Distributed RAM-Based System 54
5.2.3 TEST 3: An Optimisation For Distributed RAM-Based System 56
5.2.4 Comparisons 57

6 Conclusion **60**
6.1 Future works 60
6.1.1 Existing Architecture Improvements And Tests 60
6.1.2 New Architectures Realisation 61

References **63**

List of Figures

1.1 Machine Learning Basic Structure 1
1.2 Reinforced Learning Description 3
1.3 Tensor Representation 6
1.4 Convolution Product Application On Image 7
1.5 Robert's Filter Masks 8
1.6 Sobel's Filter Masks 8
1.7 Prewit's Filter Masks 9
1.8 Average And Max Pooling Working Principle 10
1.9 An Example Of FPGA Based System 12

2.1 CRAM Unit Cell 15
2.2 CRAM Based System 15
2.3 RAM And PEs Connection Possibilities 16
2.4 Memory Mapping Example For MRAM Based Devices 17
2.5 (a) Total Energy And (b) Total Cycle-Count Comparisons 18
2.6 Scouting Logic Memory Cell 18
2.7 Scouting Logic Based Adder 19

3.1 PYNQ-Z2 Board 21
3.2 RAMB36E1 Port Signals 25
3.3 SoC Resource Localisation 26
3.4 2KB BRAM Device Implementation 27
3.5 4KB AND 8KB Resource Utilisation 27
3.6 4KB AND 8KB BRAMs Device Implementation 28
3.7 From 16KB To 512KB Resource Utilisation 28
3.8 From 16KB To 512KB Device Implementation 29
3.9 BRAM36 Utilisation With Varying Memory Size 30
3.10 PIP Utilisation With Varying Memory Size 30
3.11 LUT(logic) Utilisation With Varying Memory Size 31

4.1 Vectors Convolution Product 33
4.2 Zero Padding Working Principle 34

4.3 Convolutional Core Block Diagram 35
4.4 SR_PISO Component Description 36
4.5 SR_SIPO_3N Component Description 37
4.6 MAC And Zero Padding System Description 37
4.7 Saturation System Description 38
4.8 SR_SIPO_32bit Component Description 39
4.9 Core Starting And Shifting Simulation 40
4.10 Core Convolution Operation Simulation 40
4.11 Core Zero Padding Mechanism Simulation 41
4.12 Core Entire Vector Mapping Simulation 41
4.13 Block Diagram Of Core And Distributed RAM Structure 43
4.14 Input Memories Initialisation Simulation 44
4.15 Output Memory Write Operation Simulation 44
4.16 AXI4 Read Channels 45
4.17 AXI4 Write Channels 46
4.18 AXI4 IP Creation 46

5.1 In Lab Tools 49
5.2 Preliminary Result Presentation 50
5.3 Test 1 Proposed System 51
5.4 Handshake-Like System For Input Data 52
5.5 Handshake-Like System For Output Data 52
5.6 Test 1: In Device Implementation 53
5.7 Test 2 Proposed System 54
5.8 Test 2: In Device Implementation 55
5.9 Test 3 Proposed System 56
5.10 Tested Systems Performance Comparison 58
5.11 Tested System PIP Utilisation Comparison 58
5.12 Tested System LUT Utilisation Comparison 59
5.13 Tested System FLIP FLOP Utilisation Comparison 59

6.1 Interleaved Structure For Convolution Product 61
6.2 RISCV-based Computational RAM 62

List of Tables

1.1 Machine Learning Techniques Comparison 5

3.1 2KB BRAM Resource Utilisation 27

5.1 Test 1: Performance Analysis 53
5.2 Test 1: Used Resources 54
5.3 Test 2: Performance Analysis 56
5.4 Test 2: Used Resources 56
5.5 Test 3: Performance Analysis 57
5.6 Test 3: Used Resources 57

CHAPTER 1

Introduction

Nowadays, building *smart* machines is the main concern in every branch of computer science. The key issue consists of designing new technologies able to carry out complex tasks, someway emulating the human intelligence.
Machine Learning (ML) is the fundamental background for this system intelligence implementation [1]. Its basic structure is given in figure 1.1.

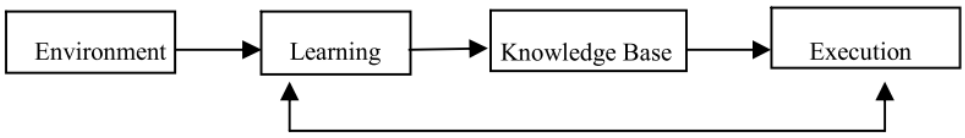

Figure 1.1: Machine Learning Basic Structure

The system intelligence implementation is based on a learning approach: in the same way humans learn from previous experiences, a machine improves its performance by learning from previous events or by surrounding environment. This is the contribution that machine learning techniques give to the technological world: providing and equipping with new functionalities and skills the brand new electronics systems, so that it's possible that they organise their knowledge structure and create a self-improving mechanism.
In general, ML has a great importance in the Artificial Intelligence (AI) context: a system not having learning capabilities cannot be called intelligent at all. Consequently, its application is spreading in various branches, such as automated reasoning, language understanding, pattern recognition, computer vision, intelligent robots realisation and so on.
In this thesis work, this learning system will be contextualised in the autonomous driving world, to give a practical example of its application and further understanding its potentialities. Making autonomous cars is a complex issue that car manufactures are facing in recent years. Their researches consist on developing software and hardware tool that give to a self-driving cars all the information they need to be autonomous.

The final task is to make cars *self aware* on the surrounding environment, also making them able to predict possible changes to those surroundings.
In a driving scenario, data incoming from internal and external sensors (i.e cameras, GPS, Light Detection and Ranging LIDAR, Laser Range Finder LRF etc.) have to be combined so that vehicles properly reach their tasks of object recognition, classification and movement actuation and predictions.
Machine learning algorithms are the theoretical background of these tasks implementation.

1.1 Machine Learning Techniques

Current state of art techniques for system intelligence implementation [2] can be categorised as follows:

- Supervised learning;
- Unsupervised learning;
- Reinforced learning;
- Deep learning.

Their differences consist on what and how the learning is actually carried out. In the next pages, these differences are going to be highlighted.

Supervised Learning

This technique tries to predict new outputs, knowing which results were obtained in the previous history for the same input. A more rigorous definition is here proposed: *given a certain number of input, usually called training set* $\boldsymbol{X}$*, a corresponding value of an output function* $\boldsymbol{f}$ *is elaborated. The final goal of this learning technique is to find the best hypothesis* $\boldsymbol{h}$ *that allows to find a close value of* $\boldsymbol{f}$ *for a given* $\boldsymbol{X}$*.*
Overall, this approach is applied for different purposes, such as driving lane changing and driver vigilance monitoring system. They are both important topics in automotive context.

Unsupervised Learning

From a theoretical point of view: *given a certain number of input, the training set* $\boldsymbol{X}$*, a corresponding value of the output function* $\boldsymbol{f}$ *is not given. The final goal is to try to create one.*
The main difference between the unsupervised and supervised approach is clear: the former lack of the knowledge of data contents and their interpretation, the latter has to be aware of data interpretation to realise a prediction. Supervised machine learning

technique has to learn and interpret what data cluster contains, describing them and their structure for the final interpretation.
This model can be applied to recognise, inside the environment, objects the machine does not have interpretation about, differentiating them and clustering data which deal to the same class of object (i.e. discerning different types of vehicles as bicycles, cars and bus without really knowing what they are). It can be used as a clustering method.

Reinforced Learning

It's a **reward** based machine learning technique. Who takes decision, the **agent**, in an environment, is not aware of which action is good or not but will learn it, knowing which decision give the highest reward. Moreover, a policy **p** is defined as the path of action that leads to a solution.
The mathematical model is based on Markov decision process (MDP) [3], a time discrete stochastic control process. Its key parameters are the following:

- State space **S**, whose constituting elements are current states, seen as known inputs;
- Action space **A**, made of the whole action taken in the current state;
- Probability **P** that an action in state s at time t will lead to the state s' at time t+1;
- Immediate reward **R**, following an action.

This learning techniques is graphically described in figure 1.2.

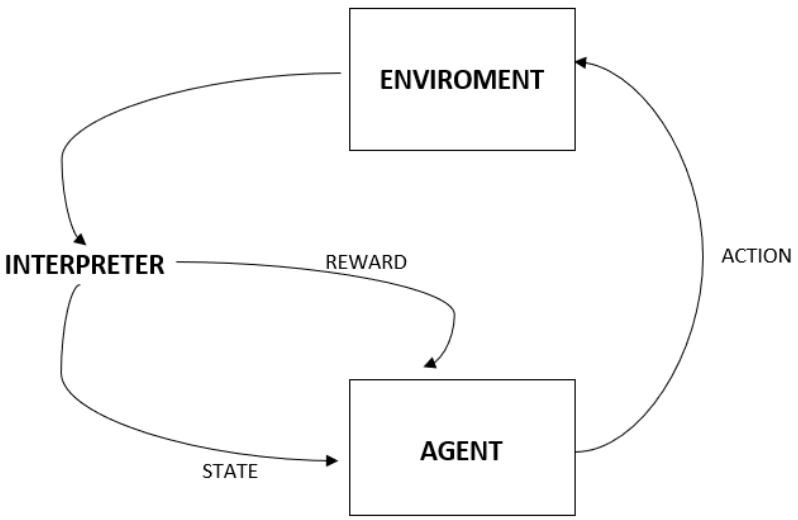

Figure 1.2: Reinforced Learning Description

In automotive field of application, reinforced approach is used mainly for parking assistant: the environment is the parking spot, the agent is the mechanism that controls wheels, reward is given when the car is perfectly positioned into the spot.

Deep Learning

Deep learning use artificial neural networks (ANN). Generally, a neural network is a structure that mimics biological neurons behaviour, whose excitation depend on the strength of input stimulus. Deep learning proved [4] to be the best choice for real time inference, a key issue in the autonomous driving field. Three models best suits deep learning approach:

- **Recurrent Neural Networks (RNN)**: they are designed to take a series input, understanding how every input is related to others and constituting a relationship system between them. Indeed, these networks work on input vectors and produce vectors as output. The final products are obtained not only by weight applied on input data (as a regular neural network) but also by "hidden" parameters extracted from prior inputs/outputs;

- **Convolutional Neural Networks (CNN)**: they are primarily used as image processing system, handling images as 4-D tensors.
  Tensors can be seen as arrays nested within arrays, infinitely. Convolutional nets performs basically filtering operation on images, to map them and extract their features. This filtering operation is based on convolution integration, which represent the fundamental tool for this operation completion.
  Image processing performed with CNN turned out to be less prone to error, with respect others Neural Network approach [5].

- **Deep Belief Networks (DBN)**: they apply a probabilistic model to latent variables (typically having binary values) organised in layers, having symmetric connections between them. DBN represent a layer-by-layer learning procedure in a top-down approach, creating relationship between layer according to specific weights.

Image processing is actually the field in which deep learning mechanisms prove to be really appealing. The ability to represent images by means of multiple layer allows clear analysis of specific parts of an image. These features are used in driving vigilance monitoring system, with the usage of cameras pointing to the driver's face. Deep learning is also used for obstacles, road lanes and pedestrian detection. These are fundamental tasks in automotive context and this is the reason why deep learning is assuming a starring role in autonomous driving field. For these reasons CNN algorithms, among others, are having great diffusion and application.

To sum up, in table 1.1 a comparison between just presented machine learning techniques is reported, quoting their methods and goal but also providing some example of their application in autonomous and assisted driving.

	METHOD	GOAL	EXAMPLE OF APPLICATION IN AUTOMOTIVE FIELD
SUPERVISED LEARNING	Input data and output data are given.	When new data are given, tries to predict the new output, learning from outputs produced in previous history by similar inputs.	Driving lane changing and driver vigilance monitoring system.
UNSUPERVISED LEARNING	Only input data are given.	Models hidden patterns or underlies structure in the given input data, in order to learn from them.	Clustering method, i.e discerning different types of vehicles into an environment without having previous history data on them.
REINFORCED LEARNING	Only input data are given.	Dynamically learn, by adjusting actions based in continuous feedback to maximise a reward.	Parking assistance.
DEEP LEARNING	Input data and output data are given.	Use artificial neural network tools to realise a prediction, establishing a self learning mechanism.	Driver vigilance monitoring system and obstacles, road lanes and pedestrian detection.

Table 1.1: Machine Learning Techniques Comparison

It has to be specified that this categorisation is not completely rigid, actually edges between techniques are not always clearly defined and intermediate approach between different techniques can be used as well.
Machine learning and, especially, deep learning work on huge datasets, requesting high computational power and top-level performances: a proper hardware selection is needed.
In the next section will be given an insight on some used algorithms in machine learning, selecting the ones that are supposed to have a large fields of application.

1.2 Algorithms Analysis

Convolution products, best pooling and filtering applications are among the most used and implemented operation in machine learning context, thus assuming a primary role in image processing area. Nowadays, image processing techniques are spreading in autonomous driving field of application: car must deal with data coming from cameras to autonomously move inside the environment. So, it's important to describe these algorithms in a more accurate way, under the mathematical point of view, for the sake of completeness.

1.2.1 Convolution Products

Convolution product [6] is the base operation when referring to CNN, already discussed in section 1.1. Image processing is the primary application where this operation assumes a key role. In this context, images are seen are as matrices with additional dimensions, forming a 4-D tensor whose possible representation is given in figure below.

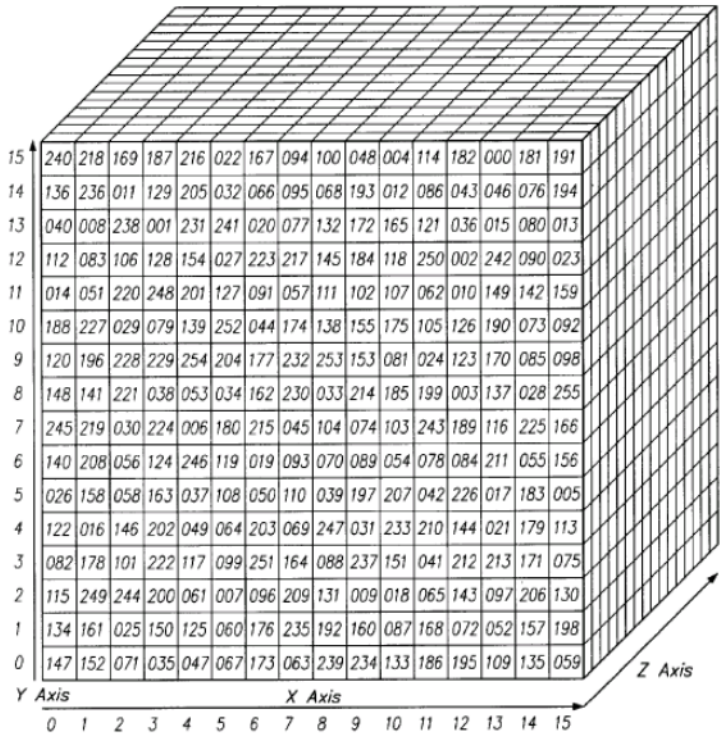

Figure 1.3: Tensor Representation

Human senses are common to visualise an image as a 2D figure but machines can better manage it, using these kind of 4D maps. Relating to this approach, images are defined in width, height and depth. Depth is specifically linked to image color encoding information, such that a Red-Green-Blue (RDB) picture, for example, will be three layers deep. Convolution operation is applied to every layer, giving an output (called *feature map*) that exists in the 4th dimension and consisting of details associated to the filtered layer.
From a mathematical point of view, the convolution is an integral measuring how much two function overlap, maintaining one function fixed and shifting in time the other one. In a more rigorous way:

$$(f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t-\tau)d\tau \tag{1.1}$$

In equation 1.1, g(t) is the shifted function, f(t) the fixed one.
In image processing context, the fixed function is the input image to be analysed, while

the moving function is known as the filter. It has the specific task of picking up signal or features in the image.
When moving to their actual implementation, a convolution net takes up square patches of pixels and examines them through a square matrix smaller than the image itself, and equal in size to the patch. Its job is to find patterns in the pixels. A graphical representation is given in figure 1.4 [1].

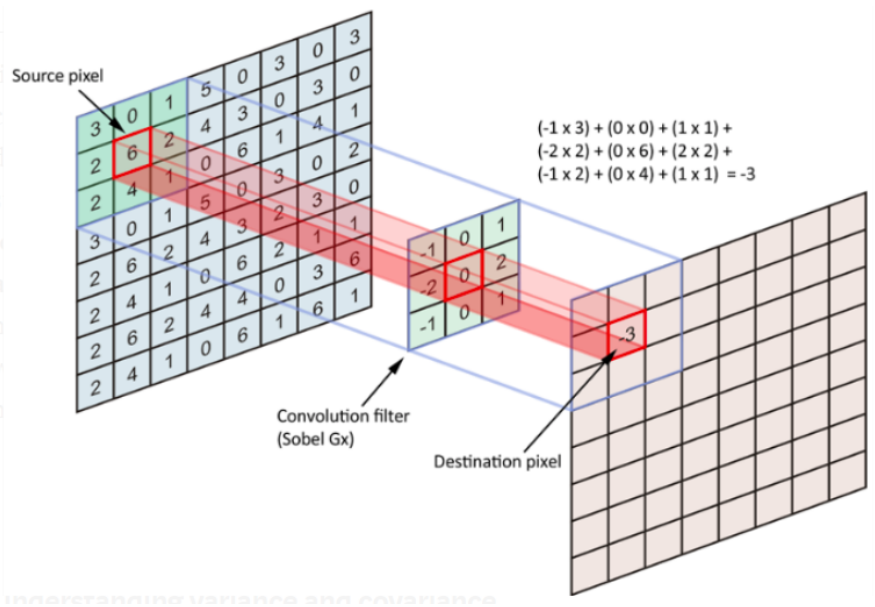

Figure 1.4: Convolution Product Application On Image

The results will be a map, generally of the same size of the input image, that focuses only on specific details of the source. This kind can to be used by the host system to carry out specific tasks, according to the field of application.

1.2.2 Filters

There is a great selection of filters applicable for images processing, among them Robert, Sobel, Prewitt and Marrs-Hildreth are to be considered [7]. They are specifically used for edge detection operations: they have to scan the image identifying sharp edges, which are discontinuities within the picture. These discontinuities bring changes in pixels intensities, defining the boundaries of an object. Edge detection techniques can be divided into two main groups: **gradient** and **laplacian** methods. The gradient method detects the edges by looking for the maximum and minimum in the first derivative of the image. The laplacian method searches for the zero crossings in the second derivative of the image to find edges. Robert, Sobel and Prewitt are gradient based filters, Marrs-Hildreth filter belong to laplacian category.

[1] credits for the image to Shuchen Du from medium.com

Robert

Robert method consists on calculating gradient in interpolation points between pixels. According to Robert's technique, the gradient magnitude can be expressed mathematically as follows:

$$G[f(i,j)] = |f(i,j) - f(i+1,j+1)| + |f(i+1,j) - f(i,j+1)| \tag{1.2}$$

Where i and j are specific points of the image.
Using proper convolution masks this operation can be simplified. Typical Robert's masks are presented in figure 1.5. They are two 2x2 convolutional kernel, in which one mask is simply the latter rotated by 90°. Every kernel is designed to evaluate maximally the gradient of edges placed along the two perpendiculars of the figure, G_x and G_y.

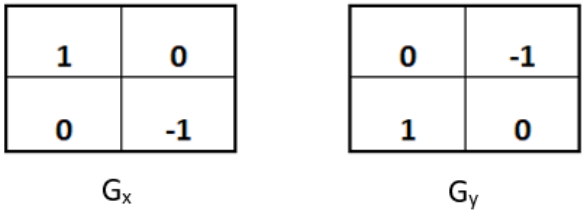

Figure 1.5: Robert's Filter Masks

The two masks can be applied separately to the images to finally merge together the results, using the proper formula:

$$G = \sqrt{G_x^2 + G_y^2} \tag{1.3}$$

Sobel

In Sobel's filters, the magnitude of the gradient is computed by:

$$S = \sqrt{S_x^2 + S_y^2} \tag{1.4}$$

S_x^2 and S_y^2 are partial derivatives to be applied around the pixel to be controlled. As for Robert's approach, these operations can be implemented using convolution masks:

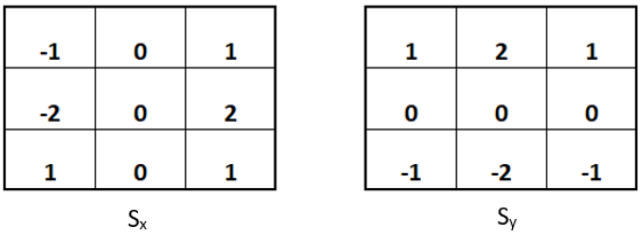

Figure 1.6: Sobel's Filter Masks

The Sobel operator is one of the most commonly used edge detectors and place emphasis on pixels that are closer to the center of the mask.

Prewit

The Prewitt's and Robert's operators shares the same mathematical equations, but differences can be noticed as regard the proposed masks. Prewitt's masks are given in figure 1.7.

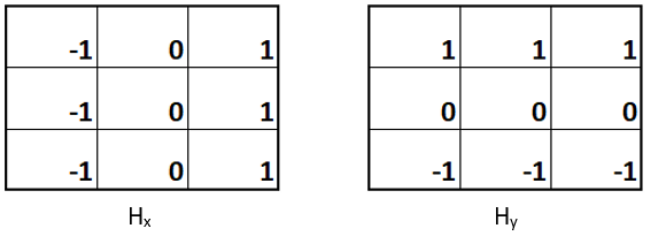

Figure 1.7: Prewit's Filter Masks

Unlike the Sobel operator, this operator does not place any emphasis on pixels that are closer to the center of the masks but at their edges.

Marrs-Hildreth

Marr-Hildreth uses the Gaussian smoothing operator to improve the response to noise. The Smoothing function is computed as I(i,j) and is used to denote the image. G(i,j,σ) has to be a Gaussian smoothing filter, where σ is the spread of the Gaussian and controls the degree of smoothing. The result of convolution of I(i,j) with G(i,j,σ) gives an array of smoothed data as:

$$S(i,j) = G(i,j,\sigma) * I(i,j) \tag{1.5}$$

It is then used to performs some noises evaluations, useful to make the edge detection operation more precise.

1.2.3 Pooling Techniques

Convolution product brings with it a great limitation: its output (i.e feature map) is the precise report of input features. This means that feature map characterisation is strongly dependent by the input image and small movements in it can produce different results. Rotating, shifting, cropping the input image is strongly affecting the convolution operation results. A way to solve this problem is the following: performing a sampling operation to the image. In this way a downscaled version of the input image will be produced, containing only the most important features.
Downsamples can be obtained by using convolution layers as a typical convolution

operation, but better results can be achieved by using specific pooling layers. A pooling layer is a new layer added after the convolutional layer. It operates upon each feature map so that a new set of output data is created. Actually, this is again a filtering operation but with the shrewdness of having filters whose size is smaller of feature map. Specifically, it is almost always 2x2 pixels map applied with a stride of 2 pixels. In this way, the pooling operation will reduce the size of each feature maps by a certain factor that depends on the pooling filter size. For example, if a 2x2 filter is applied the feature map will be reduced by a factor 2, overall each dimension will be halved, decreasing the number of pixels or values in each feature map to one quarter the size. The pooling operation is specified, rather than learned. Two common functions used in the pooling operation are:

- **Average Pooling**: calculate the average value for each patch on the feature map;
- **Maximum Pooling**: calculate the maximum value for each patch of the feature map.

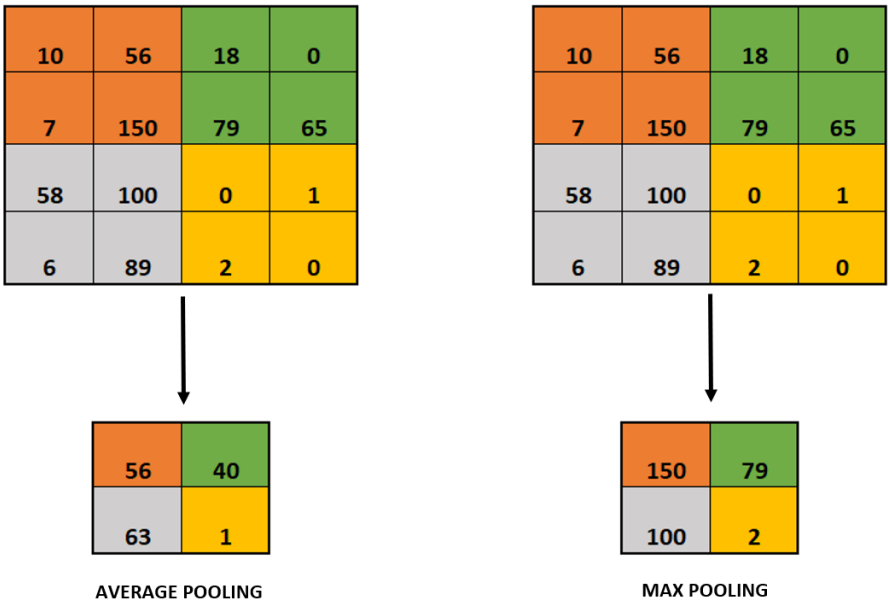

Figure 1.8: Average And Max Pooling Working Principle

An example of avarage and maximum pooling operation is given in figure 1.8. In this case a 2x2 filter is applied. The result of using a pooling layers, to create down sampled (or pooled) feature maps, is a summarised version of the features detected in the input image. Small changes in input image usually drive to different results in convolution layers contexts, meanwhile, the pooled image will not be affected by them, still giving the same results.

1.3 Introduction To FPGA

A Field Programmable Gate Array (FPGA) is an hardware component that can be re-programmed by the user every time is needed, using an Hardware Description Language (HDL). Flexibility, re-usability and easiness of hardware description make FPGA a widespread solution in almost all fields of interest, automotive is among them.
Every application implementation is made by several and different tasks to be carried to so that the final result is reached. Choosing an architecture that well suits all these tasks is not easy, considering that minimum latency, minimum storage occupation, communication interfaces and lots of others parameters have to be optimised as well. FPGAs introduce a large versatility that can be used advantageously for every task. These programmable components are usually integrated on boards (offering also processors, DSP slices, and a large number of ports and pins) with variable performance in terms of re-usable resource capability, power management and memory availability. Those boards are optimised according to the field of application, automotive is one field in which top-rated board are designed for maximising this sector applications and needs (i.e Xilinx XA Family FPGAs).

In literature is possible to find a large number of researches successfully implementing several applications on FPGAs, obtaining optimum results in terms of:

- Cost reduction with respect to non FPGA-based approaches;
- Real-time communication and high responsive systems realisation;
- Safe application implementation, in terms of data elaboration;
- Speed up of specific tasks compared to PC based implementations.

As previously said, autonomous driving application are based on machine learning techniques, requiring large amount of data. It's necessary collect data coming from sensors and cameras on disk, computing them as soon as possible to reach a real time inference. A possible block diagram of a FPGA based system realising automotive tasks is the one presented in figure 1.9. It gets inspired by previous researches [9]. The core of the entire device is the board, containing the processors and FPGA. USB, Ethernet and other peripheral component interconnects are also available, together with DRAM and SD-card ports presence. Communication in between processor and FPGA mapped logic happen thanks to AXI4 interface (a common communication standard used in these kind of boards).
The bottleneck is still represented by memory access, especially when targeting to writing and reading speeds of host PC disk.

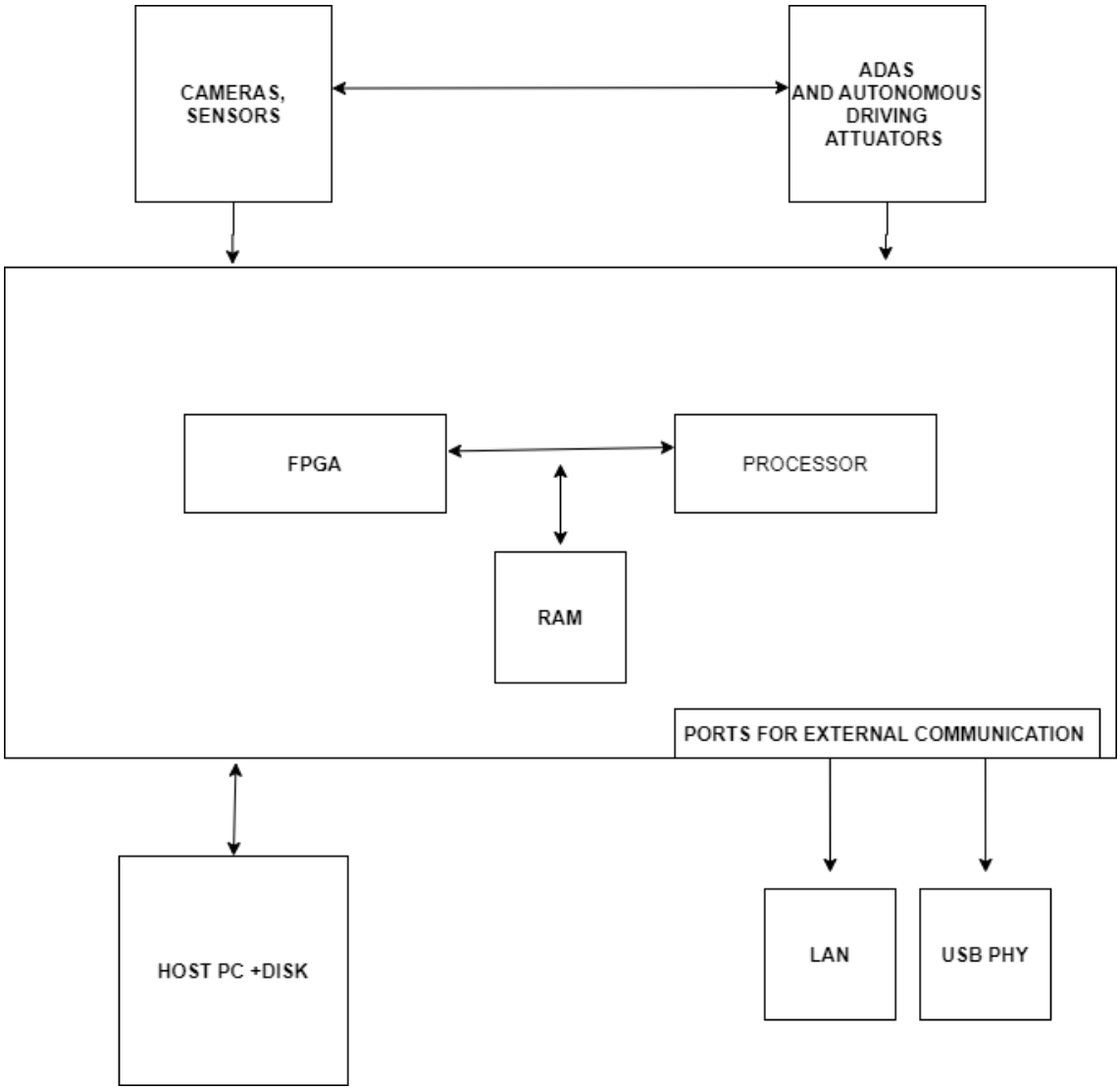

Figure 1.9: An Example Of FPGA Based System

1.4 This Thesis Work Purpose

As previously discussed, convolution algorithm is starring a primary role in machine learning field of application, hence assuming a great importance also in autonomous driving context. This is the reason why a new approach for convolution product implementation is proposed in this thesis work. It aims to merge the versatility offered by FPGA with distributed RAM concept, in order to boost the designed hardware performance and accomplish operations in a reduced processing time.

Representing the source of inspiration for this thesis work, a new technology called Computational RAM will be presented and discussed in chapter 2. After an introduction on distributed RAM (chapter 3), a new design approach, based on this different RAM managing, will be seen in chapter 4. It will also put the lights on the taken choices for a convolution core design implementation on the selected board. Chapter 5 will offer an insight on obtained results in terms of performances and resource utilisation, giving also space to some comparisons. In the last chapter, 6, new ideas for future development of the presented works will be discussed.

CHAPTER 2

Computation RAM: Backgrounds

Computational RAM (hereinafter called C-RAM) is a processing-in-memory structure, integrating storage elements and execution units in an architecture that exploit the features of memories in the best possible way.
Since Host-to-Device and Device-to-Host data transfer represent today the performance bottleneck for system performances, a C-RAM based design can alleviate this problem by allowing sequential portions of applications to run on the host without expensive data transfer: C-RAM in principle can be seen as host's memory, this means that data movement will be drastically reduced.
During the years, two main path for computational RAM applications have been developed: one is based on systems that exploit memories already available and used in current technologies, the other exploits new and experimental technologies.
The first path will be discussed in section 2.1, the latter in section 2.2.

2.1 DRAM/SRAM Based Technologies

Computational RAM has been firstly introduced to enhance the performance of Single Instruction Multiple Data (SIMD) operations. A key goal was to remain compatible with standard DRAMs and SRAMs in terms of cost, silicon area and packaging, but also exploiting the efficiencies resulting from a better utilisation of memory bandwidth and localisation.
A studied C-RAM structure [10] is the one in figure 2.1, it is based on a modified DRAM structure. The modification consists on pitch-matching a certain number of processing elements (PEs) to a small number of memory column. Using this structure, that shares a unique row address and providing a single instruction for every PE, an overall SIMD processor behaviour will be obtained.

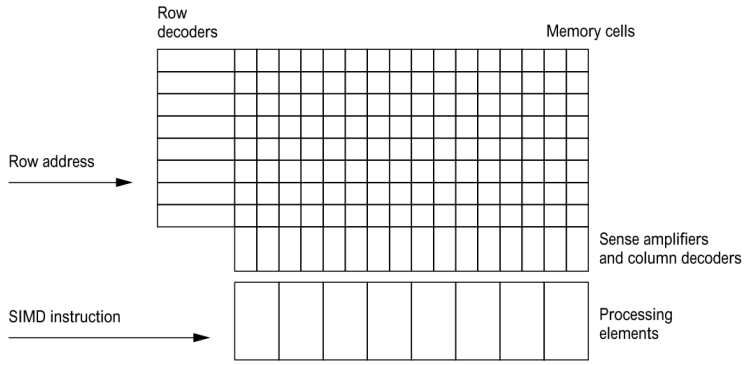

Figure 2.1: CRAM Unit Cell

The just presented unit cell is designed to be inserted in a more complex system, whose example is given in figure 2.2. This system has been developed so that C-RAM can be used as both CPU main memory or computational memory: during an operating cycle, all the processing memory execute the same SIMD instruction, when PE are not working, the host CPU can use DRAM resources as usual.

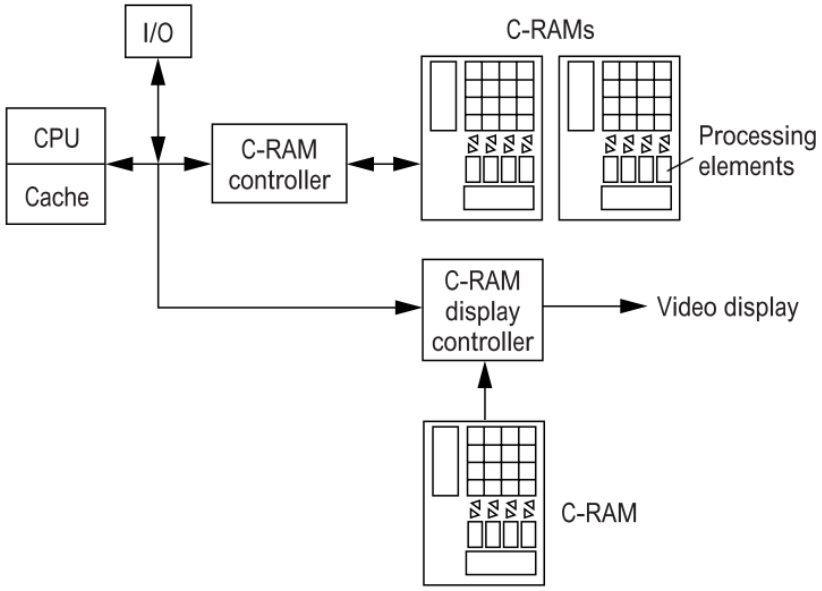

Figure 2.2: CRAM Based System

As previously said, C-RAM gains its key advantages from the fact that processing elements are integrated into the memory arrays, but inserting a PE close to DRAMs sense amplifier is not an easy task. Several approaches for memory and processors connections have been proposed, some of them are presented in picture 2.3. Choosing the best pattern for connections is an key issue for enhancing C-RAM performances and energy efficiency.

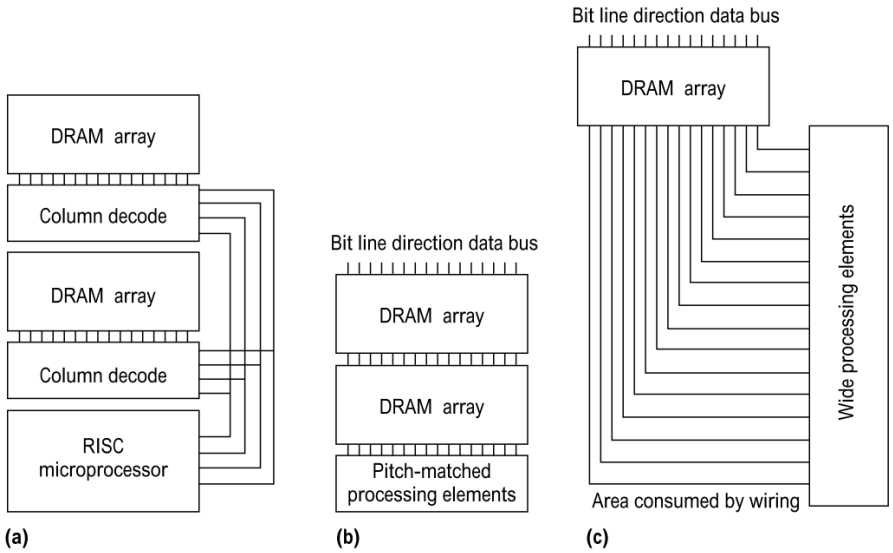

Figure 2.3: RAM And PEs Connection Possibilities

Overall, using a C-RAM approach presents advantages and disadvantages.

Among the **PROs** have to be accounted:

- High performance in terms of speed;
- Power saving, with the proper routing optimisation.

Among the **CONs** have to be cited:

- High cost for the actual design implementation;
- Complex control, requiring dedicated C-RAM controller architecture high in development time.

The efficiency of computational RAM processor has been evaluated in a motion estimation application [11]. Researchers have applied this approach to process one image frame of 352x288 pixel size, proving that this design strategy is very promising: the proposed architecture is 82 times faster than a non C-RAM based structure.

2.2 Resistive Memory Based Technologies

In recent years, the structure proposed in fig.2.1 has been modified with the introduction of non-volatile RAM. This technology is characterised by a typical RAM cell combined with magnetoresistive tunnel junctions (MTJ). MTJ-based RAMS are also called **magneto-resistive RAM** (MRAM).

In a non-volatile memory, even if the power supply is turned off, data will be not lost: a large power saving is achieved since the non used cells can be disconnected by the power line, allowing the static currents to tend to zero.
An MRAM based structure has been developed [12] for a convolution filtering operation. It has been applied to process an image consisting of 256x256 pixels, with a device designed so that the number of columns and processing elements is equal to that of horizontal pixels. Every pixel corresponds to a 8 bit data and it is associated to 8 different cells on the same bit line. Filter weights are stored in a constant value area. Overall, data memory mapping (reported in figure 2.4) is done so that a 3x3 image filtering process is executed at time.

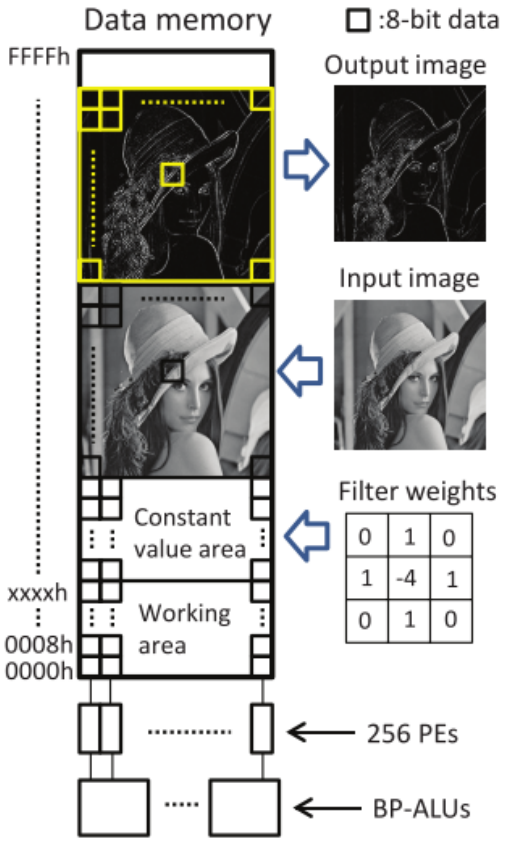

Figure 2.4: Memory Mapping Example For MRAM Based Devices

Performance related to this approach have been evaluated in terms of total cycle counts and total energy dissipation, results are presented in figure 2.5.
The current technology, compared to SRAM-based computational RAM and CPU+SRAM architectures, turns out to be extremely advantageous in terms of energy saving and required cycle number.

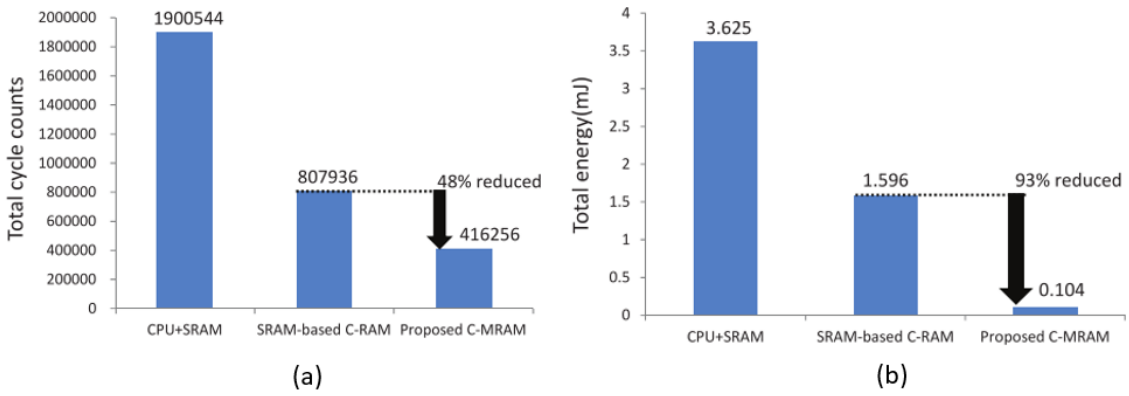

Figure 2.5: (a) Total Energy And (b) Total Cycle-Count Comparisons

Most recent researches have put a spotlight on **memresistor-based logic devices** [13]. A memresistor is a non-linear device characterised by a great scalability, high integration density and reduced stand-by power. Based on this device, several in memory computers architecture have been proposed, exploiting the so called *scouting logic*. This is a new logic design style, it has been introduced to solve short lifetime and high energy consumption related to other logic implementations. Scouting logic is actually based on read circuitry modification in typical memory cell (see fig. 2.6).

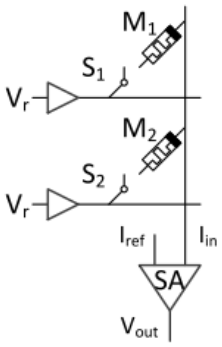

Figure 2.6: Scouting Logic Memory Cell

This technology is still under development and has been applied only for bitwise logic operation (i.e XOR, OR, AND) and a parallel adder realisation [14]. The one in figure 2.7 is an example of adder component exploiting this technology. The proposed adder has been tested, showing that the memristor-based architecture outperforms a similar multi-core implementation by at least two orders of magnitude in terms of energy efficiency and area efficiency but several problems have also to be solved, as regard interconnection network and peripheral circuit utilisation.

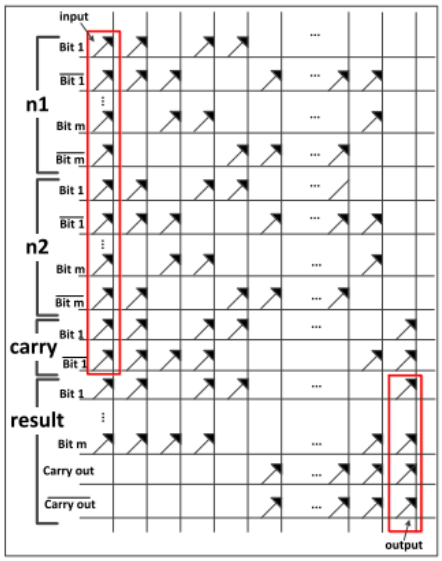

Figure 2.7: Scouting Logic Based Adder

Computational RAM researches represent a source of inspiration for this thesis work. The above proposed innovations deals with structural and technological changes on memory layout, aiming for an architecture that integrate storing and executing sources in a different ways. The proposed researches proved to be promising in terms of performance enhance and power saving for the tested technology. Similarly, in this thesis work an innovative way of integrating memories and executing core will be proposed, using the available technologies still in the market.

CHAPTER 3

A Distributed RAM Based Approach

Managing storing resources is not an easy task: memory access represent a bottleneck for modern systems. Several researches are still under development, searching for method to exploit and enhance memory performance in the best possible way. Among these there is the C-RAM approach, discussed in chapter 2. That path refers to physical changes to be introduced in memory utilities, as well as new technologies application. Computational RAM researches introduce changes in memory and processing elements integration to obtain a performance gain and power saving. Similarly, in this thesis work, a different and unusual memories and core integration will be presented, so that storing elements are fragmented and distributed according to application needs. Of course, some PROs and CONs are expected to be found because of this technology introduction.
Among the expected **advantages** there are:

- Better data managing, exploiting several memory resource independently.
- Speed improvements, due to smaller memory utilisation and better memory integration;
- Reduced Host-to-Memory and Memory-to-Host data transfer problem;
- Possibility to exploit principle of locality at its best;
- General performance increase.

Among the possible **disadvantages** there could be:

- Routing and logic resource increase;
- Larger area occupation, due to higher number of memory utilisation;
- Complex memories controller.

In general, this approach well suits machine learning needs for large datasets managing and reduced execution time. This is the reason why it will be tested with a typical ML algorithm. This innovation will be designed and tested for a FPGA implementation, according to the lab resources availability.

3.1 FPGA Memory Resources

In the following pages, the actual board used to carry out the thesis work is going to be analysed, describing its resource availability and highlighting the possibilities offered by it.
According to the lab availability, the used architecture consist of PYNQ-Z2 board. It integrates a SoC (i.e System On Chip) called Zynq 7020, produced by Xilinx. Figure 3.1 offers an insight on its structure.

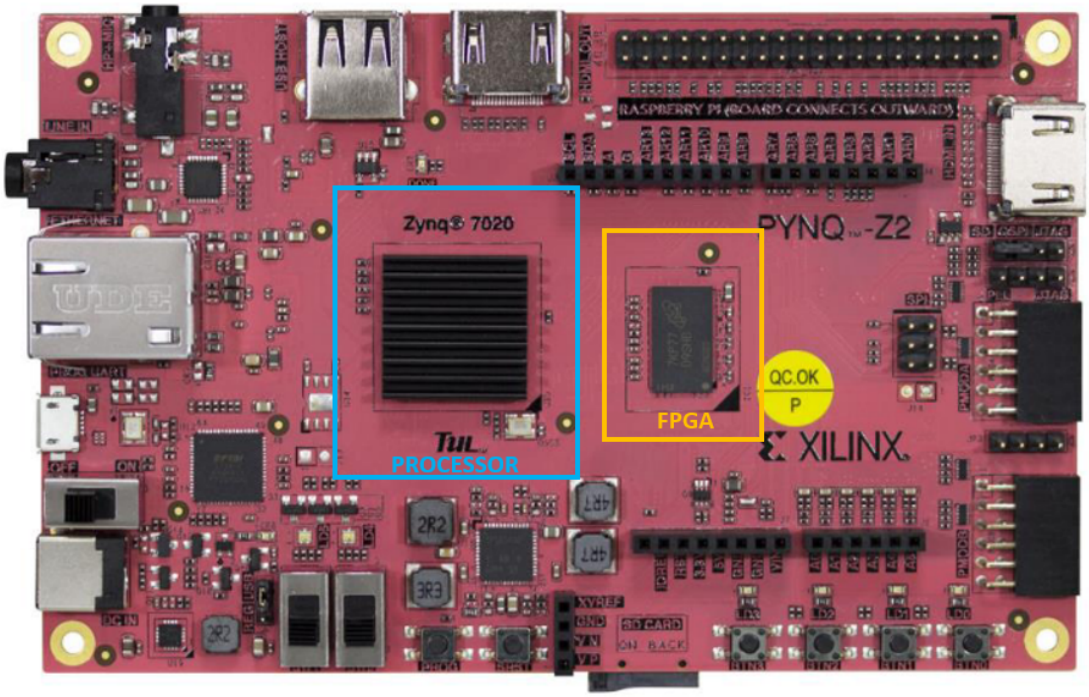

Figure 3.1: PYNQ-Z2 Board

Among this board features, the information related to processor performance and FPGA resources are to be highlighted.

From the board's system manual it's possible to learn the key feature relating to the SoC under exam:

- There is a 650MHz dual core Processor (ARM cortex A9);

- FPGA usable logic counts:
  - 13,300 logic slices, each with four 6-input LUTs and 8 flip-flop, with possibility to use it as LUTRAM (up to 140kB);
  - Up to 630 KB BLOCK RAM;
  - 220 DSP slices;
  - On chip Xilinx analog-to-digital converter (XASD);
- Internal clock availability of 125 MHz.

Other resources are not reported since not used in this work.
As a fact of matter, starting from these resource availability, a C-RAM like approach was tried to be implemented with the final purpose of boosting applications performance with respect to a standard approach. This purpose implies the necessity to exploit what the board offer in a smarter and different way.
This works focus on proposing an innovative memory utilisation to gain an improvement in applications performance, this is the reason why it's important to analyse what the programmable logic portion offers as memorisation tools. There are two memory structure it's possible to exploit in Xilinx FPGA: **Block RAM** and **LUT RAM**.
VHDL is the Hardware Description Language (HDL) used to carry out this research. The way the HDL code is written deeply affect the actual memory implementation.

LUT RAM

LUT RAMs are memories mapped in programmable logic portion of the FPGA. Xilinx software, used to physically implement what is described by the HDL code, is designed so that describing a memory with asynchronous read and synchronous write it will be implemented as a distributed RAM, by default. What follows is an example of VHDL code specifically written to map LUT RAM.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4
5  entity RAM_DISTR is
6    port(
7      CLK : in  std_logic;
8      WE  : in  std_logic;
9      ADDR   : in  std_logic_vector(6 downto 0);
10     D_IN  : in  std_logic_vector(31 downto 0);
11     D_OUT  : out std_logic_vector(31 downto 0)
12   );
```

```
13 end RAM_DISTR;
14
15 architecture STRUCTURE of RAM_DISTR is
16   type ram_type is array (127 downto 0) of std_logic_vector(31
       downto 0);
17   signal RAM : ram_type;
18
19 begin
20   process(CLK)
21   begin
22     if (CLK'event and CLK = '1') then
23       if (WE = '1') then
24         RAM(conv_integer(ADDR)) <= D_IN;
25       end if;
26     end if;
27   end process;
28
29   D_OUT <= RAM(conv_integer(ADDR));
30
31 end STRUCTURE;
```

Generally speaking, when not so much RAM is needed distributed RAM allocation could be an interesting choice. Moreover, this possibility must be considered when no more block RAMs can be allocated. But, as soon as a large amount of data has to be stored, a distributed approach could be risky since will translate in a large programmable logic module utilisation, making impossible to use it for other logic implementation. Moreover, since LUT RAM are not actual RAM structure, their speed will be reduced with respect the one of block RAMs when large amount of data are to be stored.

Block RAM

Writing a HDL code that describes a memory synchronous in both read and write will result in a block RAM implementation for Xilinx synthesis tools, by default. But, unlike the previous case, synchronous memories can be also forced to be mapped into distributed RAM thanks to a specific *RAM_STILE ATTRIBUTE*. To do so, these lines have to be inserted after signals declaration :

```
1   attribute_style: string;
2   attribute_style of RAM: signal is "distributed";
```

In this thesis, a Block RAM approach has been used to carry out the current researches. This choice has the aim of avoiding limitations on the amount of data needed by the

developed application, but also to use the best technology in terms of performance. The VHDL code related to BRAMs used in this project is the one reported below.

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.std_logic_unsigned.all;
4  use ieee.numeric_std.all;
5
6  entity BRAM is
7    generic (
8          SIZE       :integer := 1024;
9          ADDR_WIDTH :integer :=10;
10         COL_WIDTH  :integer :=32);
11   port (clk : in std_logic;
12         we  : in std_logic;
13         a   : in std_logic_vector(ADDR_WIDTH-1 downto 0);
14         di  : in signed(COL_WIDTH-1 downto 0);
15         do  : out signed(COL_WIDTH-1 downto 0));
16 end BRAM;
17
18 architecture syn of BRAM is
19  type ram_type is array (SIZE-1 downto 0) of signed (COL_WIDTH-1
       downto 0);
20  signal RAM : ram_type;
21  signal read_a : std_logic_vector(ADDR_WIDTH-1 downto 0);
22
23 begin
24   process (clk)
25   begin
26     if (clk'event and clk = '1') then
27      if (we = '1') then
28         RAM(conv_integer(a)) <= di;
29      end if;
30       read_a <= a;
31     end if;
32   end process;
33
34  do <= RAM(conv_integer(read_a));
35 end syn;
```

3.1.1 Block RAM Features

Every Block RAM in Xilinx 7th series FPGAs can be used as two separated 18Kb RAM (i.e **RAMB18E1**) or one 36Kb (i.e **RAMB36E1**) blocks.
Some interesting block RAM features are listed below:

- RAM blocks can be combined to form one 64K RAM without additional interconnect requirement;
- 18, 36, or 72-bit wide block RAM ports can have an individual write enable per byte;
- Every Block RAM can be configured as a **single port** or a **dual port memory**.

RAMB18E1 and RAMB36E1 (see figure 3.2 [1]), are the basic building blocks for every block RAM configurations. The input and output data buses are represented by two buses for 9-bit width (8 +1), 18-bit width (16 + 2), and 36-bit width (32 + 4) configurations. The ninth bit associated with each byte can store parity/error correction bits or serve as additional data bits.

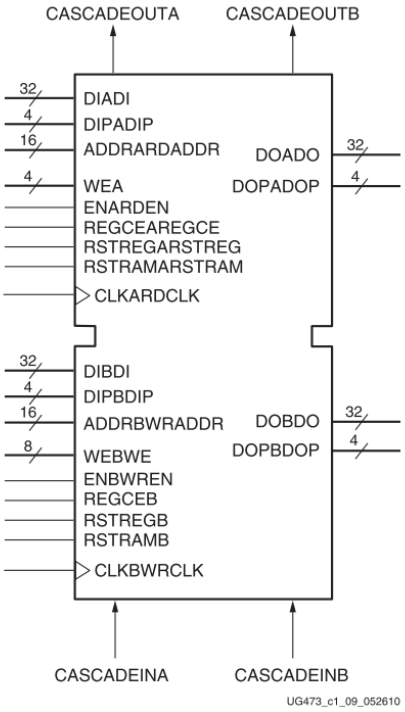

Figure 3.2: RAMB36E1 Port Signals

[1] figure is taken from *"7 Series FPGAs Memory Resources User Guide"* by Xilinx.

3.1.2 Block Ram Implementation

A specific study on memory synthesis and implementation through Xilinx tool (i.e *Vivado*) has been performed, with the purpose of comparing resource and block RAM utilisation with increasing memory capacity.
Before starting these analysis, a figure (3.3) describing the resource disposition in PYNQ-Z2 SoC space is presented. It is included to have a complete knowledge on the board resources localisation.

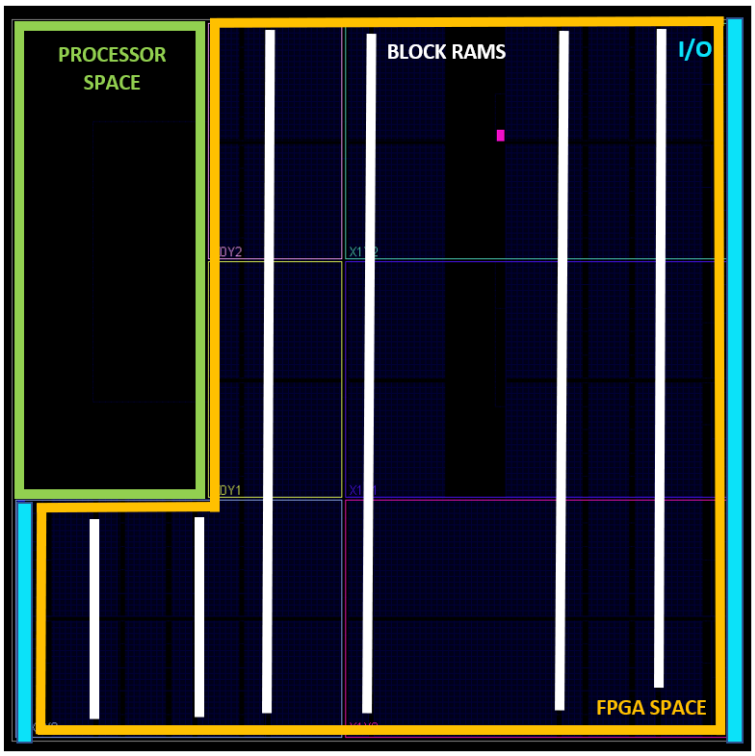

Figure 3.3: SoC Resource Localisation

It's important to consider that the used FPGA maximum availability include:

- **LUT as logic**: 53200;
- **LUT as memory**: 17400;
- **BRAM36**: 140 (so 280 BRAM18);

Memory sizes under analysis has been chosen so that a clear picture on RAM implementation can be derived, varying the RAM sizing from the minimum to the maximum possible capacity. Overall, the following memory sizes are investigated: 2kB, 4kB, 8kB, 16kB, 32kB, 64kB, 128kB, 256kB 512kB.

These memories will be characterised in terms of number of Programmable Interconnection Points (PIPs)[2], number of LUT used for logic and number, type and placement of used BRAMs.

2KB BRAM

Figure 3.4 allows to see how BRAM36 and BRAM18 are graphically represented in post implementation view on Vivado tool: it's obvious that BRAM18 is represented as one half of BRAM36, just as it's indicated also in resource table.

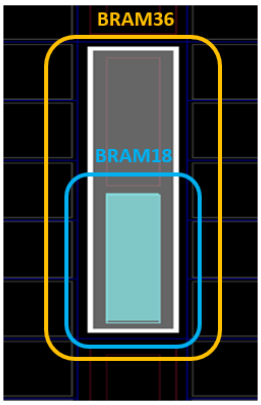

Figure 3.4: 2KB BRAM Device Implementation

	BRAM 2KB
PIPs	1146
LUT (LOGIC)	0
BRAM36	0.5

Table 3.1: 2KB BRAM Resource Utilisation

4KB AND 8 KB BRAMs

Figure resource utilisation for both 4KB and 8KB are reported in figure 3.5. It shows that not so much overhead is introduced when passing from one size to the other in terms of PIPs. It is an expected result since, in 8KB memory, simply two close BRAM36 are used, showing the area optimisation provided by the tool.

	BRAM 4KB
PIPs	1374
LUT (LOGIC)	0
BRAM36	1

	BRAM 8KB
PIPs	1863
LUT (LOGIC)	0
BRAM36	2

Figure 3.5: 4KB AND 8KB Resource Utilisation

[2]From Xilinx documentation: "Programmable interconnect points, or PIPs, provide the routing paths used to connect the inputs and outputs of IOBs and CLBs into logic networks.A PIP is a CMOS transistor switch that you can program to turn on or off".

It has to be observed that, in figure 3.6 and all the following Device Implementation images, the allocated memory is indicated by light blue rectangles meanwhile yellows lines refers to routing resources.

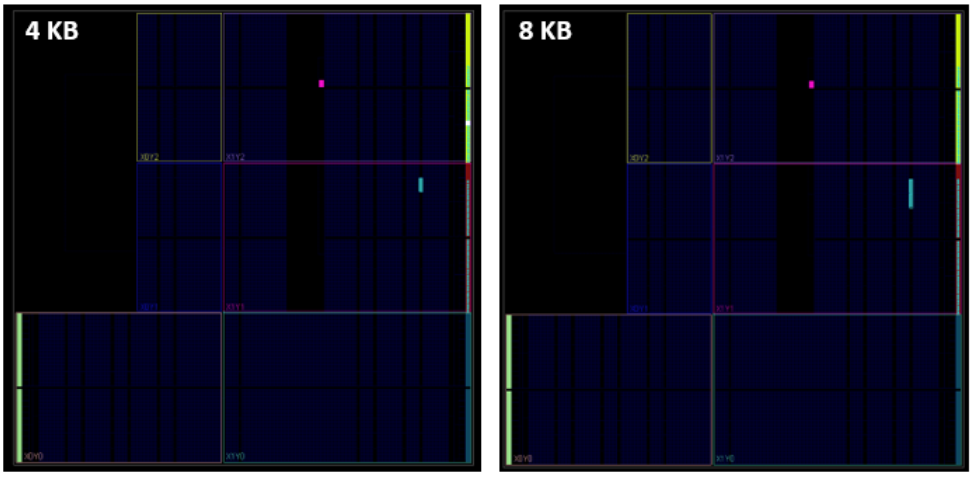

Figure 3.6: 4KB AND 8KB BRAMs Device Implementation

FROM 16KB TO 512KB BRAMs

In figure 3.7 and 3.8 resource utilisation and BRAMs implementation for the remaining sizing are given.

	BRAM 16KB
PIPs	2838
LUT (LOGIC)	0
BRAM36	4

	BRAM 32KB
PIPs	4700
LUT (LOGIC)	0
BRAM36	8

	BRAM 64KB
PIPs	8427
LUT (LOGIC)	0
BRAM36	16

	BRAM 128KB
PIPs	16048
LUT (LOGIC)	0
BRAM36	32

	BRAM 256KB
PIPs	31880
LUT (LOGIC)	32
BRAM36	64

	BRAM 512KB
PIPs	65167
LUT (LOGIC)	139
BRAM36	128

Figure 3.7: From 16KB To 512KB Resource Utilisation

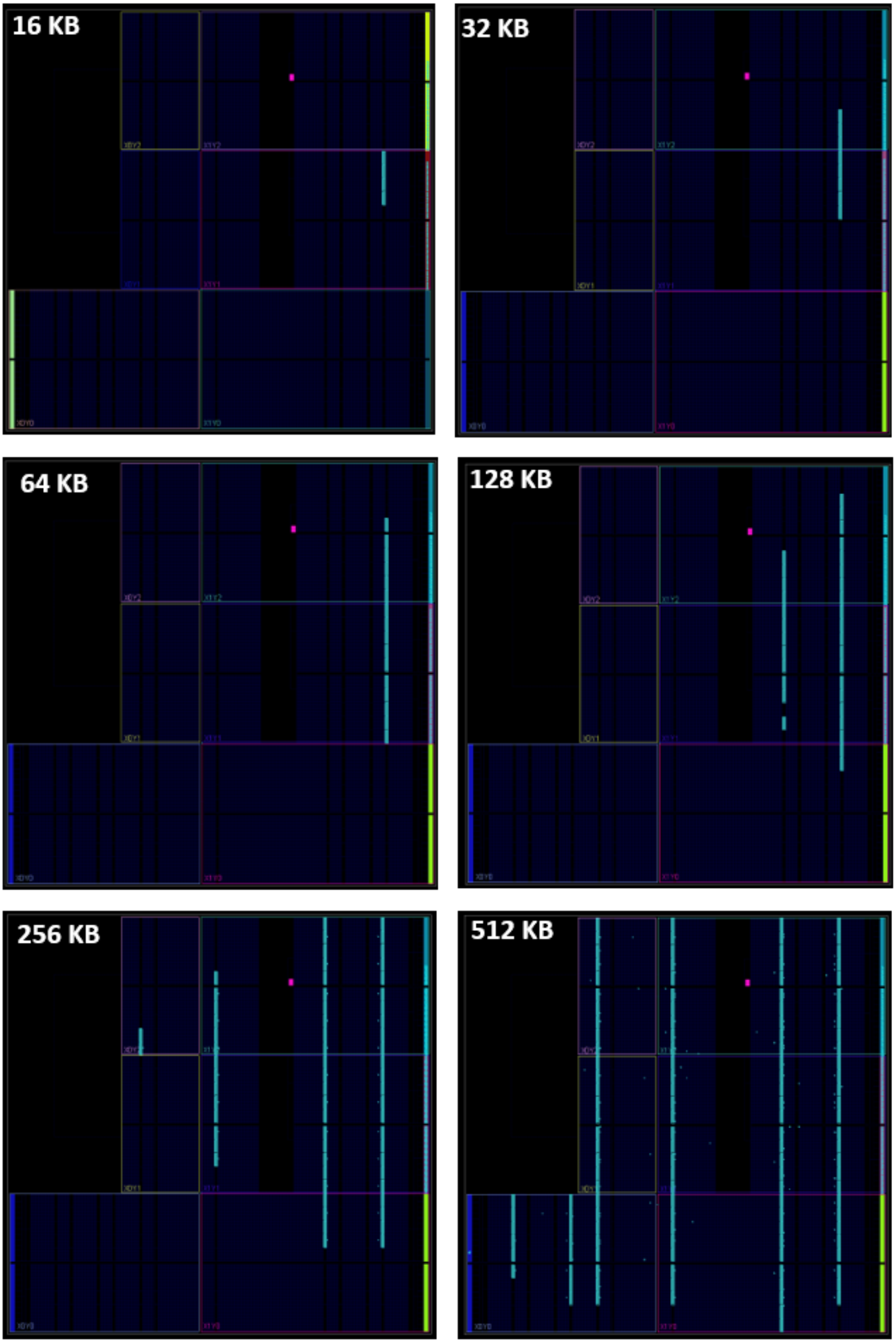

Figure 3.8: From 16KB To 512KB Device Implementation

This study is proving that resource utilisation are increasing linearly with the memory size. This is quite obvious when referring to the number of used BRAMs , but it was to be proved when referring to the number of PIP involved in the implementation. Graph that shows the BRAMs increases with varying sizing is presented in figure 3.9. Fig. 3.10, instead, gives an insight on the linear trend associated to the PIP number.

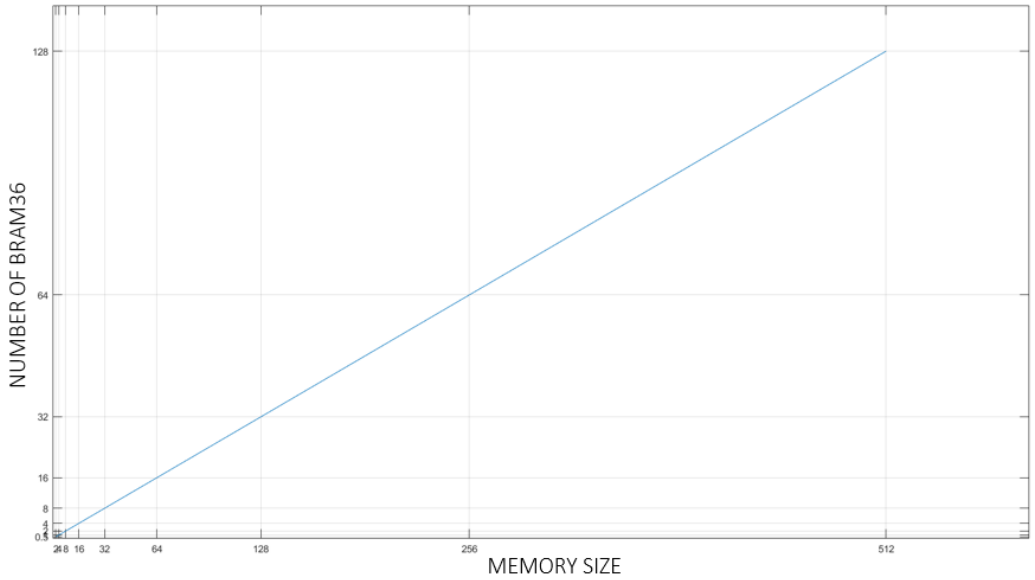

Figure 3.9: BRAM36 Utilisation With Varying Memory Size

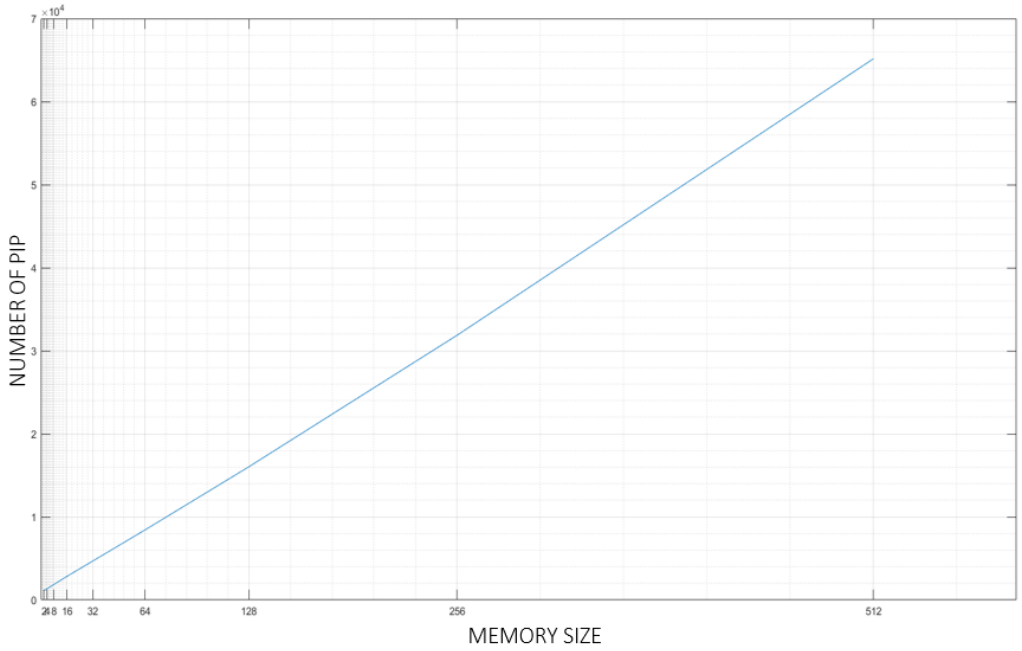

Figure 3.10: PIP Utilisation With Varying Memory Size

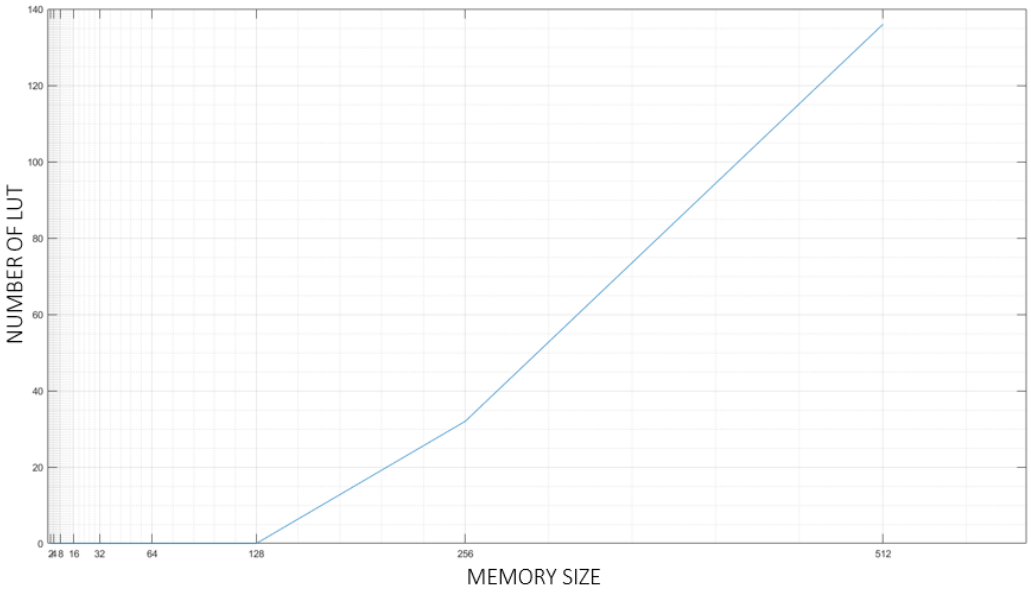

Figure 3.11: LUT(logic) Utilisation With Varying Memory Size

Unexpected results are linked to number of LUT (logic) data, presented in figure 3.11. According to the cascading rules stated in reference manual, logic resources should remain always at zero by varying the memory size but this is not completely true: with larger sizing (256 and 512 KB) some logic is actually used. The reason behind it could be linked to particular optimisation performed by the tool, forcing it to introduce some extra logic.

Overall, the linear trend characterising resource usage allow to exploit a specific memory size, managing and optimising its utilisation to obtain a performance enhance, with the possibility to generalise the results:

- Vivado tool space optimisation automatically allocates memories in a cascading way (almost) without inserting logic: working with a single memory or with more of them should not introduce performance loss because of spatial proximity of data.
- Resource management increase linearly with memory size, avoiding logic waste: number of used logic associated to memory is predictable at design time.

CHAPTER 4

Design Of Developed Architecture

In section 1.2.3 convolution product algorithm has been theoretically described and selected for the hardware implementation, considering its importance in machine learning. In this section, choices and methods for a Convolution Product Core design will be discussed.

4.1 Convolutional Core Design

The core under design is supposed to compute convolution products on vectors. It's important to clarify the technical terminology used from now on:

- **Feature vector**: it is one of the two input vectors, it actually represents the data to be filtered by this operation;
- **Filter** or **Weight**: the second input vector, it is usually given and relates to the chosen filtering technique;
- **Feature map**: Output vector, containing convolution products results.

The convolution is a shifting, multiply and accumulate operation, whose representation is given in figure 4.1. As it's possible to see, the feature vector and the weight vector are firstly multiplied term by term, then the resulting operators are added together constituting the first element of the feature map vector. At this point the weight vector is shifted and the entire operation in repeated until the whole feature vector has been filtered.
Feature map dimension change with feature vector and weight vector length. Being **W** the size of the feature vector, **F** that of weight and **S** the number of shift performed at time (usually 1), the output vector size **O** will be extracted by the following expression:

$$O = \frac{W - F}{S} + 1 \tag{4.1}$$

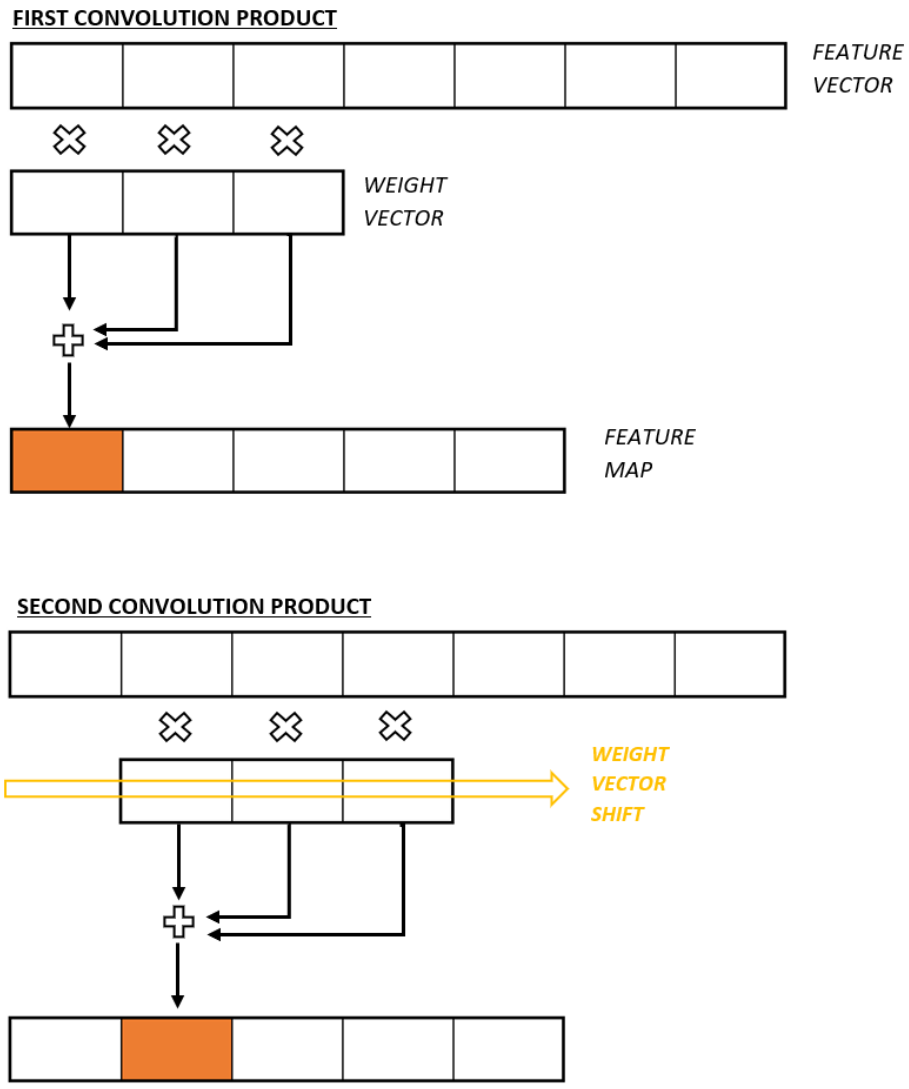

Figure 4.1: Vectors Convolution Product

To control the length of feature map vector, avoiding it reduces when weight vector length increases, it has to be introduced the concept of **zero padding**: it consists on virtual zeroes insertions in one or both sides of input feature vector, to compensate weight size enlarging. Figure 4.2 is introduced to better explain this concept, **P** refers to number of added zeros (considering only one side of the vector). This figure depicts an example of zero padding in which two zeros (one per side) are added to feature vector, so that an output vector of the same size of input (O=W) is obtained. In this case, expression 4.1 has to be changed accordingly:

$$O = \frac{W - F + 2P}{S} + 1 \tag{4.2}$$

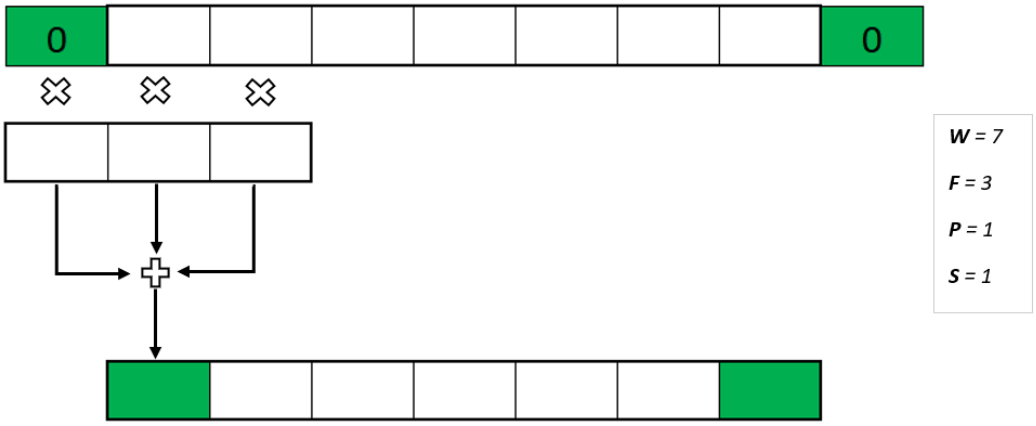

Figure 4.2: Zero Padding Working Principle

The output vector size must be determined according to the specific needs linked to results interpretation.

Core Technical Specifications

The designed core is expected to work under particular condition defined at designed time:

- Feature vector size **W**: it has to be chosen by the user;
- Weight vector size **F**: it is fixed and it's 3. It has been seen that convolution filters employ this kind of sizing. Moreover this choice make simpler the hardware design;
- Number of bit **N**: it is the parallelism of every vector element, it can be selected by the user up to 16. The system has been feed with 8 bit data;
- Zero Padding **S**: it has been fixed to 1, to get in output a vector of the same length of the input;
- Data are supposed to be integer number.

4.1.1 Designed Hardware Description

The overall designed core is described in figure 4.3, where:

- N= number of bit chosen by the user;
- A is a 32 bit word associated to the feature vector;

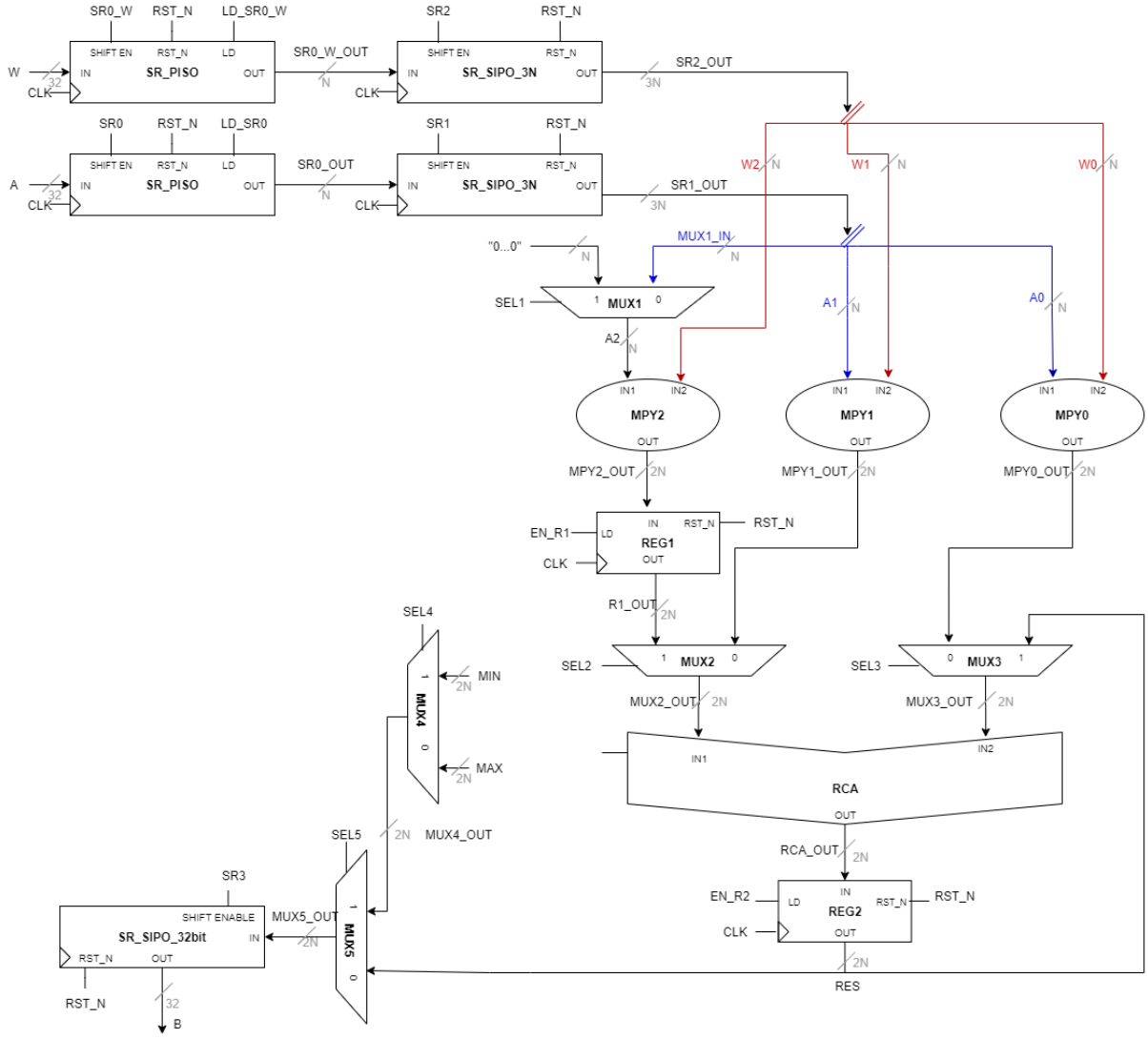

Figure 4.3: Convolutional Core Block Diagram

- *W* is a 32 bit word associated to weight vector;
- *B* is a 32 bit word related to feature map vector (outputs);
- Data coming from SR_SIPO_3N registers are defined on 3xN number of bits. This data is soon split in three portion so that, for example, *A0* is constituted by *SR1_OUT*(N-1 downto 0), *A1* come from *SR1_OUT* (2N-1 downto N) and *A2* from *SR1_OUT* (3N-1 downto 2N). These signals are emphasised by the blue arrows. The same mechanism has been adopted to create *W0*, *W1* and *W2* signals, highlighted by the red arrows.

The whole signals in the block diagram used to properly run the components are managed by a specific internal control unit. Only *Clock*, *Reset_N*, *A*, *W*, *B*, *Start* and *Done* signals are exposed toward the external world.
A detailed explanation on this system behaviour will be given in the following pages. Every component presented below has a corresponding VHDL description.

Shift Register Parellel IN Serial OUT

A shift registers with parallel input and serial output is inserted for every memory containing input data. When the corresponding *Load Enable* (LD) signal is asserted, a 32 bit word is loaded from the corresponding memory into the register and it is ready to be right shifted. *Shift Enable* (SHIFT_EN) signal allow to perform the word shift of N position, corresponding to the parallelism of every vector components (i.e if the weight vector has lenght F=3 and its components are defined on 8bit, a 8 position shift will be performed).

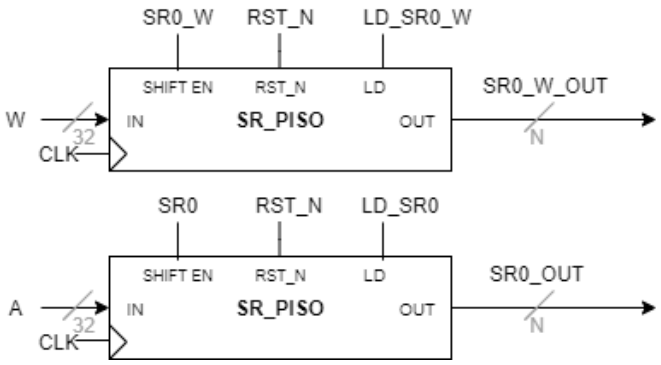

Figure 4.4: SR_PISO Component Description

Shift Register Serial IN Parallel OUT - 3N Version

Serial data coming from the shift register with parallel IN and serial OUT represent the input quantities of this component. It is inserted to allocate three separated data relating to feature vector and weight vector in the proper registers, at time. This

designed choice is introduced to gain a better control on data flowing toward MAC system, also allowing to have a shifting system that offers flexibility and can be easily managed, when the system control has to be arranged.

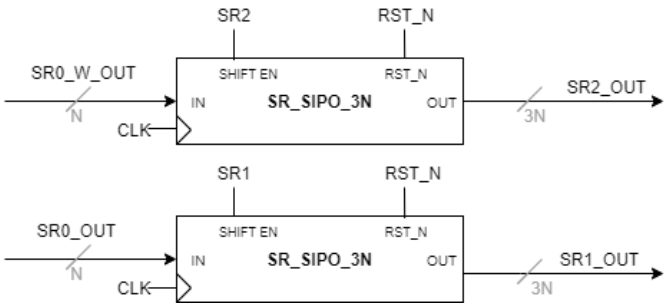

Figure 4.5: SR_SIPO_3N Component Description

Multiply And Accumulate Components

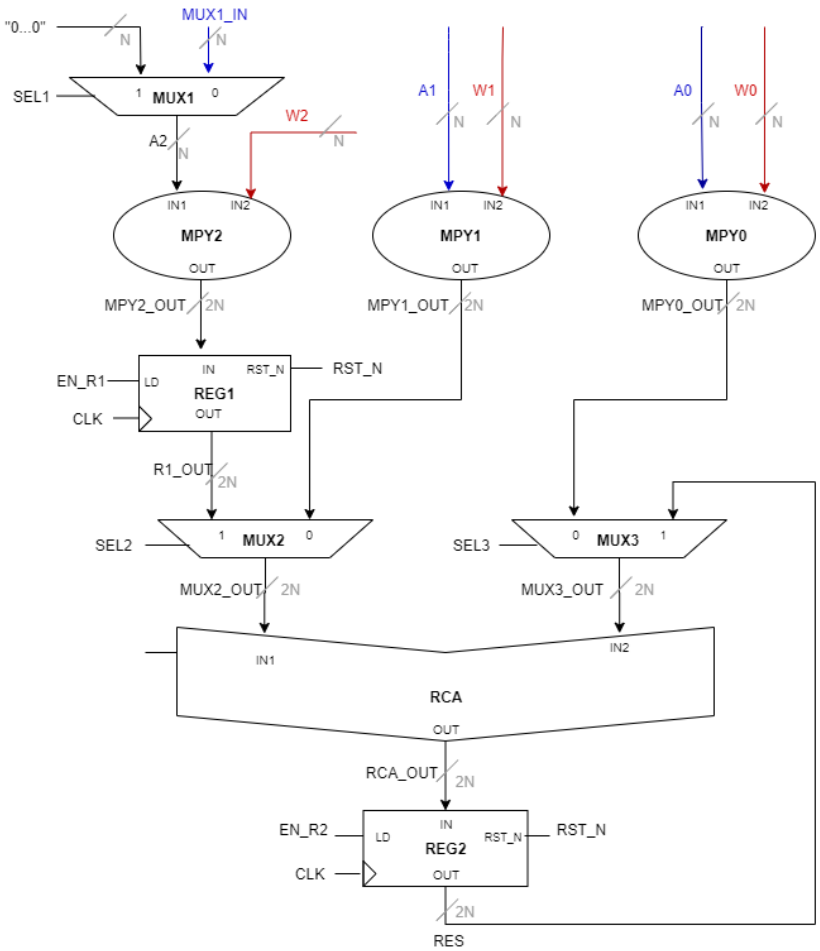

Figure 4.6: MAC And Zero Padding System Description

This is the real computing core of the whole design. Operation are performed according to a precise order. For sake of clarity, every step is enumerated below:

1. Three multiplications between three elements of feature vector and three of filter vector are performed together;

2. Results coming from multipliers 0 (MPY0) and 1 (MPY1) are directly sent to the Ripple Carry Adder (RCA), where sum operation on the first two data of input vectors is performed. Multiplication result coming from multiplier 2 (MPY2) is sent to register 1 (REG1).

3. First sum result is stored on register 2 (REG2), which is an accumulation register. After a clock cycle it is sent back to the RCA, where it is summed to the data previously stored in register 1. In this way the whole operation on three input vectors data is performed in just two clock cycle.

The RCA is an adder performing operation on signed number(i.e CA2) and it is described in a structural way, rather than in behavioural one, to maintain a total control on its operation completion, trying also to boost its performance. Only one adder is used to avoid using too large resources.
Multiplexer 2 and 3 are inserted to manage data flows through the adder.
It's actually important to highlight the importance on multiplexer 1, since it is responsible of the zero padding insertion mechanism. As previously said, zeros are not actual components of the feature vector but are virtually inserted, when requested by the control unit, forcing to 1 the *Select* signal of this mux. At the input corresponding to Select=1 is inserted the zero data defined on N bit, that will be handled as any other data managed by the MAC system.

Saturation Mechanism

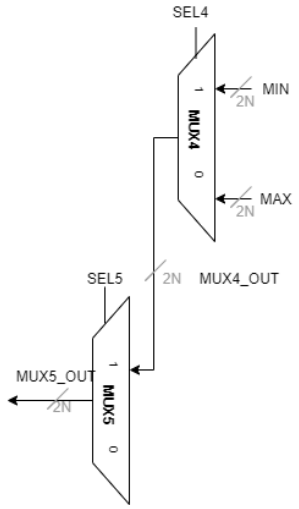

Figure 4.7: Saturation System Description

Working with integer numbers it's necessary to introduce a system that detect an erroneous result coming from RCA. A signalling system based on adder carry-out has been elaborated, so that the core is able to identify a data that is wrong because the maximum positive or negative number representable on the available dynamic has been exceeded. In this case multiplexer 4 and 5 will be controlled so that the right result takes the place of the erroneous one.

Shift Register Serial IN Parallel OUT- 32 Bit Version

This shift register host convolution products results. Every time a new result is produced a left shift happen so that, when a 32 bit word is collected it is given in output in parallel way. Every time a new 32 bit word bit the mechanism restart with the same behaviour.
These choices are been done to adapt the system to properly manage a proper memory interface, discussed in the following pages

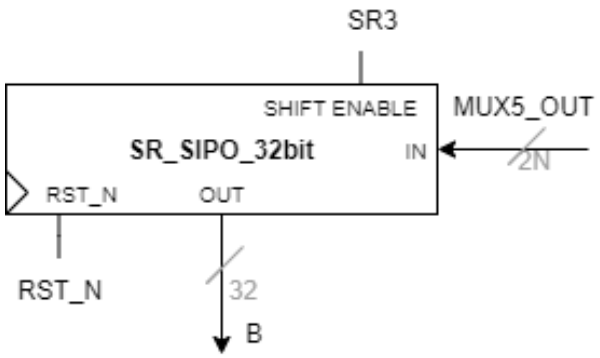

Figure 4.8: SR_SIPO_32bit Component Description

Control Unit

A proper control unit has been elaborated to manage the system resources in the best possible way, boosting its performance. A complete description of this unit is not give since it has been changed according to the tests needs and possibilities. These changes also include collateral module insertion for specific test settings and their associated control signals. Sometimes it has been necessary to mould the Convolution Core with more than one control unit, in order to carry out specific tasks.
It has to be specified that figure 4.3 is lacking of a few number of counters, inserted to accomplish some controls issues.

4.1.2 Designed Hardware Simulations

Before the actual implementation, the developed core has been simulated with an HDL simulation environment called **Modelsim**. Significant portions of simulations will be presented in this section.
Several testbench files have been elaborated according to the test needs and dealing with core architecture. Timing analysis performed on Vivado have denoted the necessity to trigger the system with a **100 MHz clock frequency** or lower, this is the reason why a dummy clock of 10 ns period have been implemented in the test bench. It is also providing to the system the whole control signals ruling the hardware components.
In figure 4.9 it's possible to see the core behaviour during system initialisation: when *Start* signal arrive, the designed system change its state, from IDLE to LOAD_SR0 and so on. During the starting states, the core performs all those shifting operation needed to properly setting up the machine and involving SRO_PISO and SR_SIPO_3N components previously described. In the presented picture it's possible to see that these shifts are happening in a correct way: in few clock cycles the system will compute the first results.

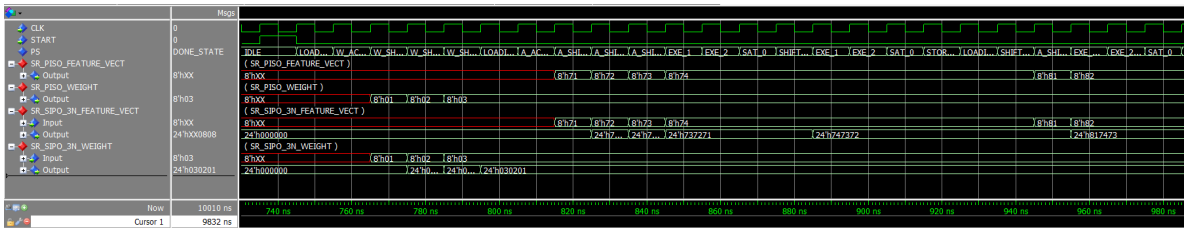

Figure 4.9: Core Starting And Shifting Simulation

Convolution Product execution on hardware is simulated in figure 4.10. A single convolution operation start in EXE_1 state and finish, just after one clock cycle, in EXE_2 state. During EXE_1 the three multiplications on input data happen, as well as the first sum, whose result is stored in register 2. This result is then resumed and added to the third multiplication result in EXE_2, in this way the overall result is obtained and stored again in reg 2. As previously said, in our tests weight vector is maintained fixed.

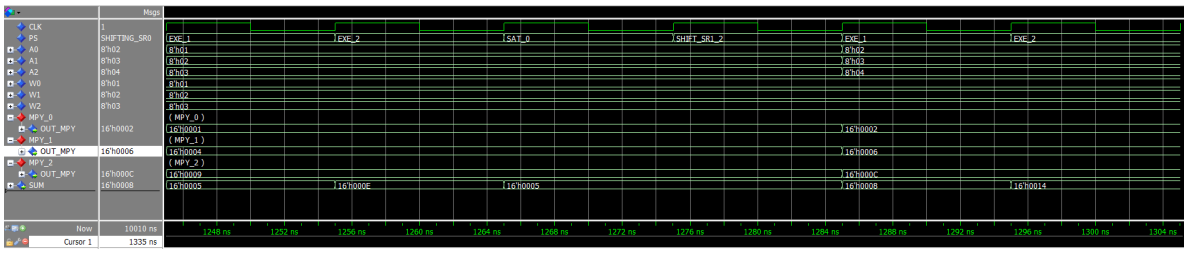

Figure 4.10: Core Convolution Operation Simulation

Another mechanism whose working behaviour has to be proved is that of zero padding. In figure 4.11 it's possible to see that, when ZERO_FILLING state comes, a zero is virtually inserted within the vector, so that the data shifted out from the related shift register is substituted with a 0. This is possible since in ZERO_FILLING, EXE_ZERO_FILLING_1 and EXE_ZERO_FILLING_2 states mux 1 selection signal *Sel1* is forced to 1. In the presented simulation fragment it's possible to see that during these states the A2 values is no more 8 but 0, returning to 8 after the convolution operation completion. EXE_ZERO_FILLING_1 and EXE_ZERO_FILLING_2 are substituting EXE_1 and EXE_2 states and are specifically defined for the zero padding operation.

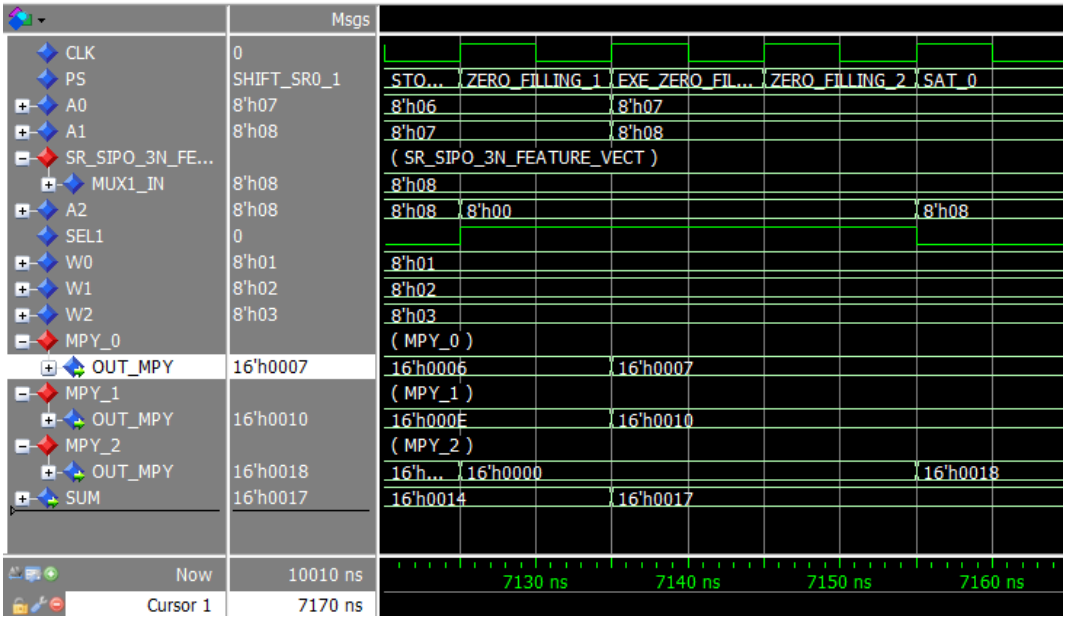

Figure 4.11: Core Zero Padding Mechanism Simulation

The last simulation fragment (fig. 4.12) allows to see the convolution operation performed on an entire vector composed of twelve elements. It has to be notice that, as expected, a *Done* signal is asserted when the core finishes computing.

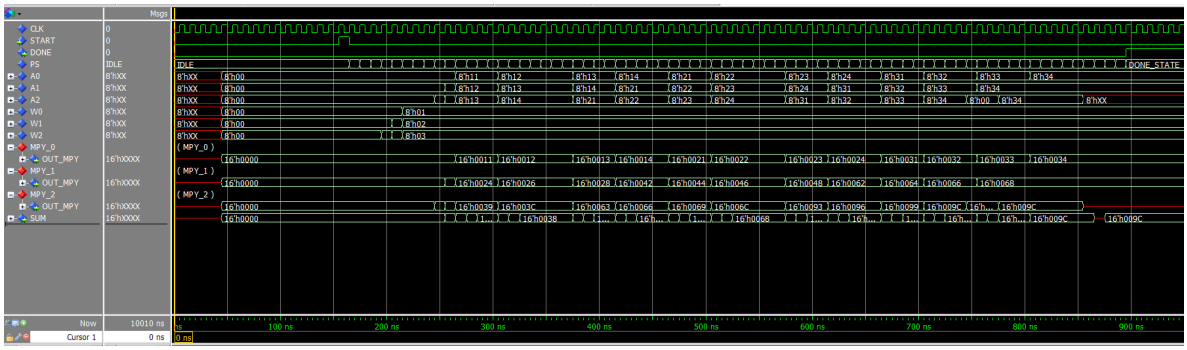

Figure 4.12: Core Entire Vector Mapping Simulation

4.2 Convolutional Core And Distributed RAM Integration

The just described and simulated Convolutional Core has to be interfaced with a memory structure, containing data on which executing this filtering operations and constituting the pool for the output data storage. This structure has been developed according to the C-RAM background given in chapter 2 but also considering the resource study performed on the board to be used (chapter 3).
The computational RAM background leads toward a structure where logic and memory implementation are closely connected to each other in space. Said so, the intention is to use a memories distribution, where storing system is placed physically near to the input/output ports of the logic net to which they refers. Controlling resource placement in FPGA is not an easy task but, using HDL synthesis tool like Vivado it's a great advantage since the tool itself realise these kind of optimisation automatically.
At design level, it was decided to use a single BRAM36 (allocating up to 4KB data) for every data cluster participating in the convolution operation, this is the reason why three memory have been interfaced with the current core:

- MEM_A is the block ram containing feature vector data;
- MEM_W will be filled with weight vector;
- MEM_B will represent the pool for output data storage.

The proposed architecture is presented in figure 4.13. Among the advantages related to this structure can be mentioned:

- Small memories are faster. According to our study (chapter 3) enlarging memory capacities should not affect too much the overall performance, but this is actually a result related to the PYNQ-Z2 board and cannot be generalised;
- Input vectors and output one can be managed in a completely separate way. This kind of data handling can be optimised for the application under test, demonstrating great versatility;
- Having three small memories instead of a larger one permit to boost performance since the core have not to be stopped when a new data among the input one (feature/ weight vector has to be read) or when a new output is produced. Data input request and output writing are operation that happen with a high frequency (on average, the time interval between these kind of operation 80 ns), so the advantages of using these structure are clear. Actually, this approach introduce the need for a more complex control and the necessity to introduce separated program counter, but this should not affect the validity of the proposed design.

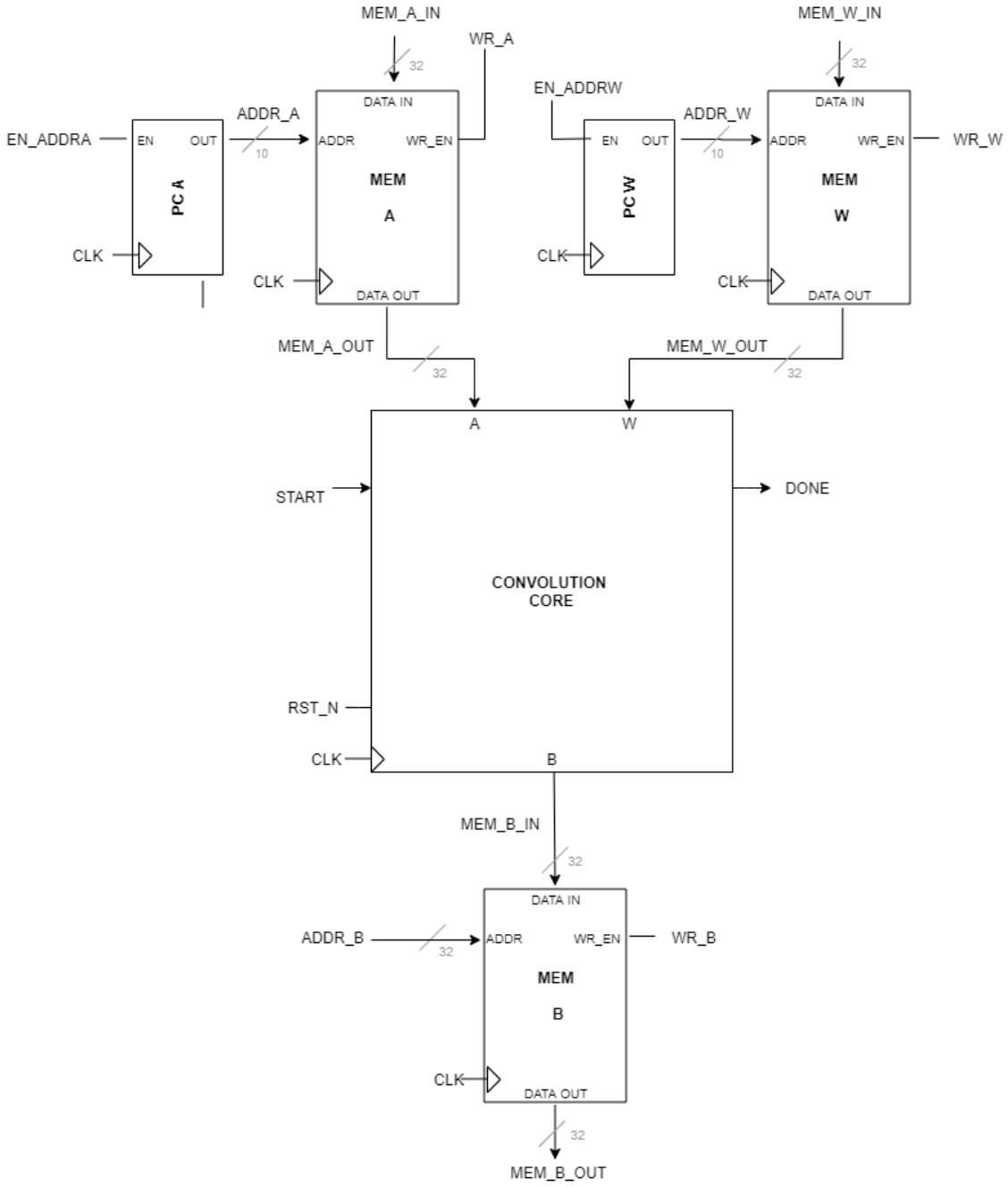

Figure 4.13: Block Diagram Of Core And Distributed RAM Structure

4.2.1 System Simulation

Convolution Core behaviour remains the same and has been previously analised. Now, it's important to present an insight on Core-BRAM integration.
For simulation sake, a dummy control unit has been developed with the specific task of writing input memories (MEM_A and MEM_W), starting from data contained in constant ROMs. These input writing operation is observable in figure 4.14. The dummy control unit starts charging the two memories in certain number of writing states. When the two memories have been completely initialised, the dummy CU drives to 1 the *Done_test* signal that, in the next clock cycle, will be translated as *Start* signal assertion for to the convolution core.

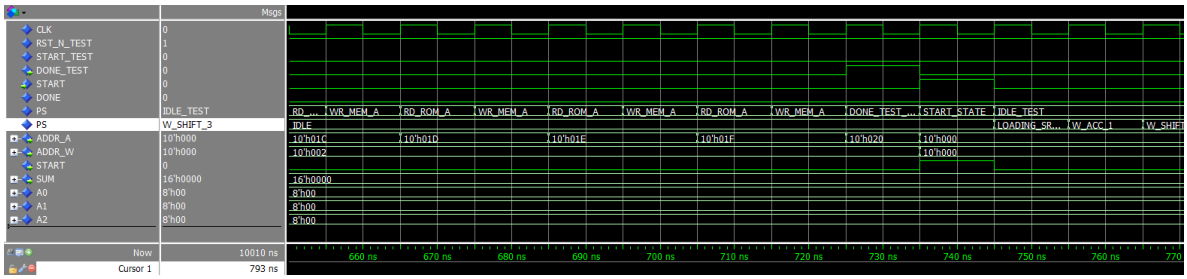

Figure 4.14: Input Memories Initialisation Simulation

In figure 4.15 writing operation in B memory is simulated. This is a distributed operation during the core computation operation. In this figure we can appreciate the last storage completion in STORAGE_B state and the *Done* signal assertion by the system.

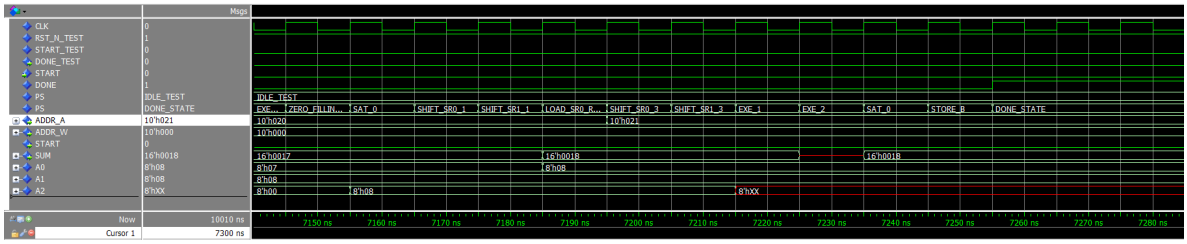

Figure 4.15: Output Memory Write Operation Simulation

4.3 Interfacing Issues

Designed architecture in both figure 4.3 and 4.13 have been previously tested using dummy control units and specific components, added for the sake of simulation. Now, it's important to make sure these architecture work in their actual physical implementation on hardware. To do so, it has been necessary the utilisation of the ARM processor on PYNQ-Z2 board. Interfacing logic implemented on FPGA together with

ARM processor is not an easy task, since these kind of communication on board happen through a specific interface protocol, called AXI4.

AXI4 is an interface used to connect a master and a slave block. Data transfer can happen choosing between three interfaces:

- **AXI4**: this protocol is used for specific memory mapped applications;
- **AXI4-LITE**: it is used for simpler application, not involving memory mapping but allowing the user to use an interface where control, status and data registers have to be managed;
- **AXI4-STREAM**: as for AXI4, this is used again for memory mapped application, with the further possibility to handle very high speed tasks.

Data can be sent in both directions at the same time, using different channels for both read and writes operations. Moreover, read/write data channels and read/write address channel are physically separated, allowing to transfer data and addresses simultaneously.
Figure 4.16 [1] introduces read AXI4 interface, showing a read transition that uses both read address and read data channels.

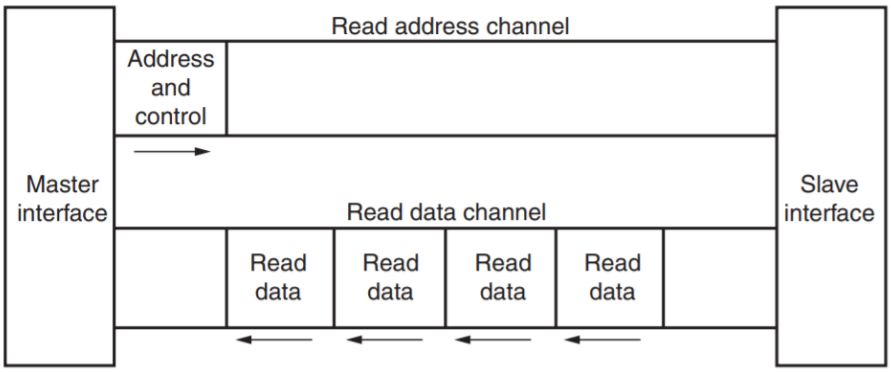

Figure 4.16: AXI4 Read Channels

Figure 4.17 refer to write channels. Together with address and data, a write response channel is also present. It give information on write operation status, the written data can be fully supported by the master system, not present or consisting on a values not supported by the specifications.
In general, writing and reading operation are similar: the data flow starts with the master block that determines the address and which function to perform. After that, if the slave block is ready, the master performs the default function.

[1] Figure 4.16 and 4.17 are taken from Vivado AXI Reference Guide (UG1037)

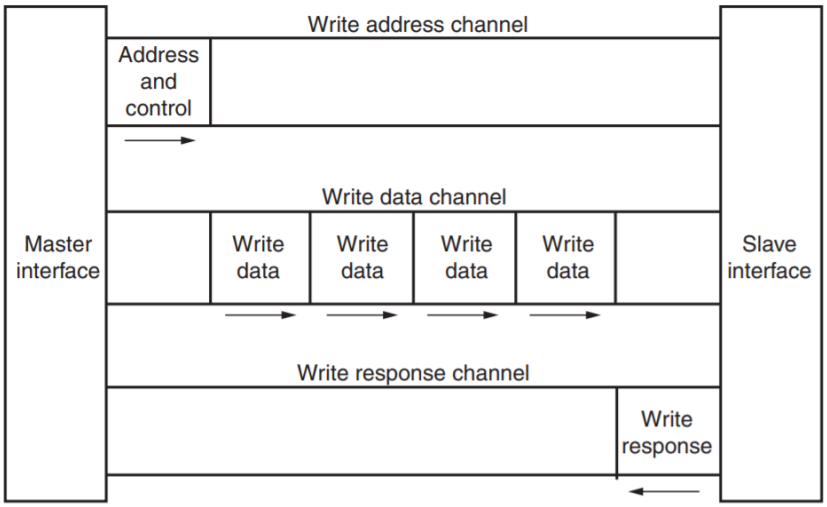

Figure 4.17: AXI4 Write Channels

AXI4-Lite interface is the protocol is going to be used to fatherly develop this thesis work.
To adapt the designed convolutional core to AXI4 protocol interface, it has been necessary to create on Vivado and AXI4 peripheral. This gives the possibility to select the number of interfaces, and registers associated to the peripheral, according to the specific needs. Referring to this thesis applications, a single interface peripheral has always been selected, with a number of register variable from 4 to 7. Vivado tools automatically creates two VHDL files,constituting a sort of AXI4 wrapper for the designed system. Figure 4.18 is a example of AXI4 wrapping creation applied to the previously discussed Convolution Core.

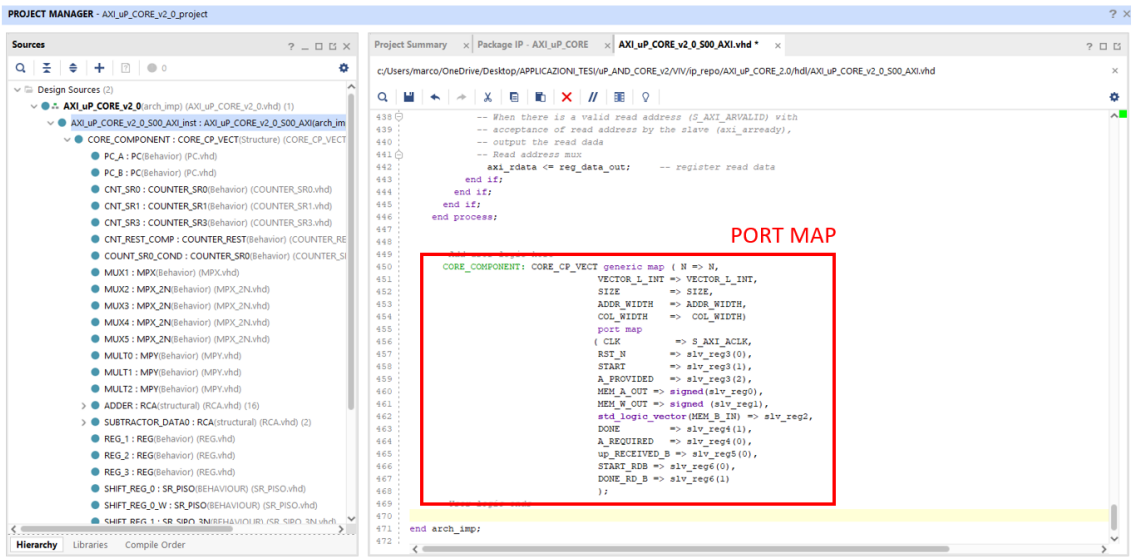

Figure 4.18: AXI4 IP Creation

As it is possible to see, the Core becomes the component for the AXI4 IP automatically created by the tool. As a component, a specific port map is needed for the top level VHDL file of the whole Core (called CORE_CP_VECT in the example), interfacing its input and output with the desired number of registers (*slv_reg* in the figure).

CHAPTER 5

Experimental Results

In this chapter, methods used to carry out some physical tests are going to be presented, also describing the whole structures under test and their actual implementations.
In section 5.1 will be presented the testing environment together with some preliminary results, useful to appreciate the potentialities of the proposed distributed RAM system. In following section, 5.2, test results will be discussed for three different hardware patterns, to finally compare them in terms of performance and FPGA resource utilisation.

5.1 Testing Enviroment

Picture 5.1 shows the actual assets used in lab during the experiments execution and for results extractions. This figure holds the whole essential instrumentation needed to test the designed projects. It consists of PYNQ-Z2 board and three software tools (i.e Xilinx Vivavdo, Xilinx Vitis and Tera Term). For the sake of clarity, a brief description of these tools and their utilisation for the current experiences will be given.
As previously said, **PYNQ_Z2 board** will be programmed so that the VHDL designed systems run on FPGA, properly using also block memory resources. Moreover, these logic components will communicates together with ARM processor, whose main task is to send and receive data to/from the FPGA implemented system, also providing some control signals.
Several testing steps are performed on Xilinx **Vivado** tool:

1. Wrapping the designed system, so that an AXI compatible IP is obtained;
2. Creating a block design that includes both ARM Processor and the AXI IP;
3. Running synthesis and implementation of the VHDL described hardware. During synthesis the RTL-described design is transformed into a gate-level representation. Implementation refers to whole steps performed to place and route the netlist onto device resources, considering the logical, physical, and timing constraints of the design. After the implementation, an insight on time information

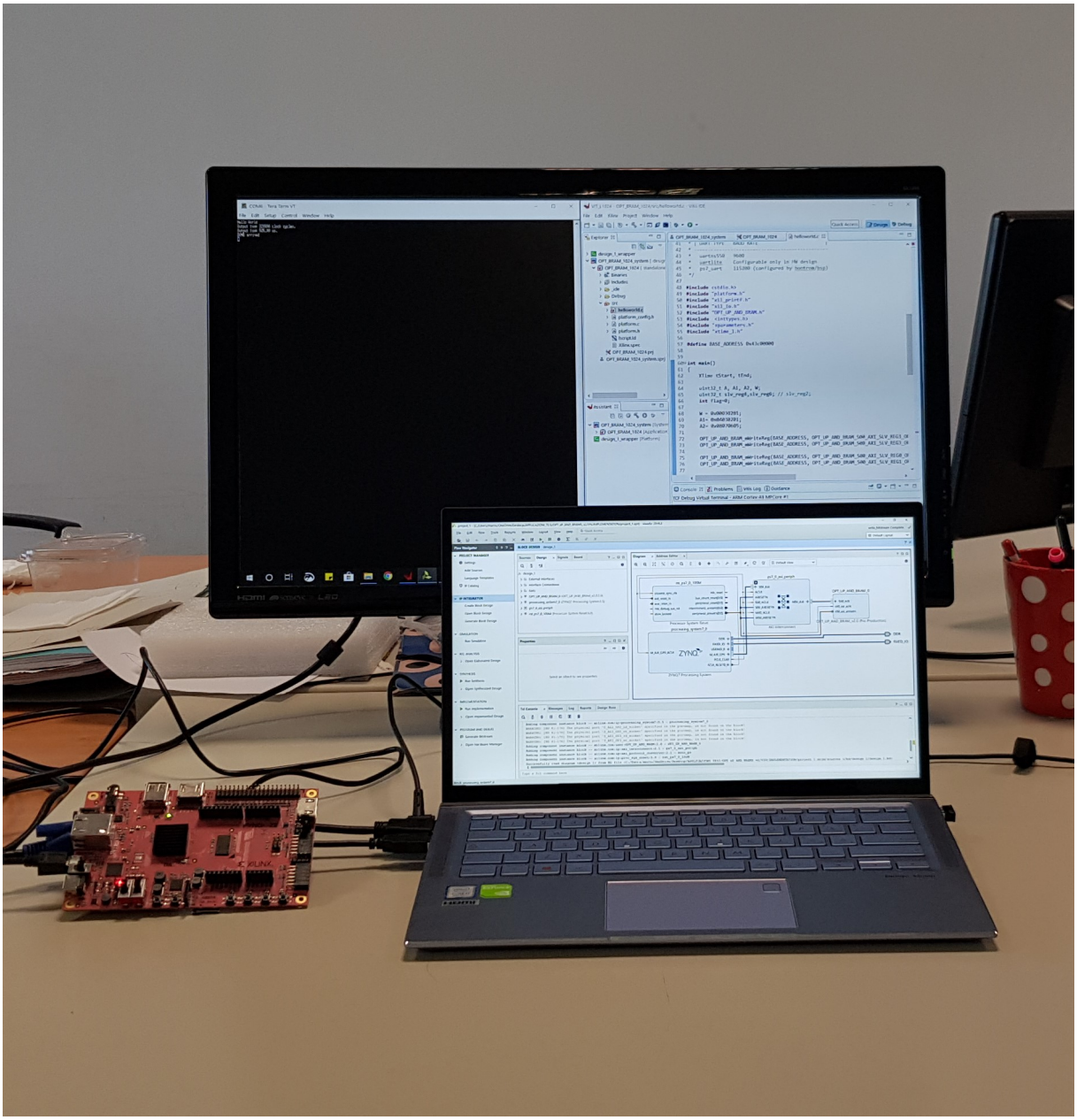

Figure 5.1: In Lab Tools

(setup, hold and pulse width time) is be reported, as well as resource utilisation info;

4. Generating Bitstream. This operation creates a file containing the programming information for the FPGA. Programming a Xilinx FPGA means loading this bitstream file into the FPGA;

5. Exporting the hardware, so that the software elaborate a XSA file for Vitis platform. This files contain all the needed info related to the exported designs, integrating processor and HDL described hardware.

Xilinx **Vitis** is a software platform offering graphical tool used (in this thesis work) to compile, analyse, debug and built a C code. This code has to be changed test by test and contains all the lines needed to write and read register related to AXI4 interface. *Xtime_l* primitive is used to extract timing information needed for performance analysis. This method allows the access to a processor global counter, updating with half the frequency of the processor and giving info expressed in terms of number of cycles. Some mathematical expression is then used to print a time result expressed in μs.
The performance analysis results are printed on screen, using **Tera Term** terminal emulator program. The proper boud rate has to be selected for a correct behaviour, in our case it has been set to 115200.

5.1.1 Preliminary Results

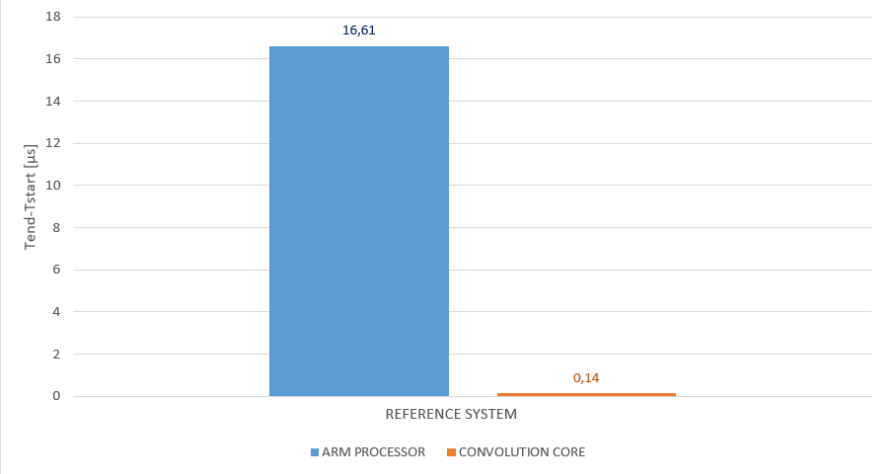

Figure 5.2: Preliminary Result Presentation

A preliminary test has been performed, before implementing more complex designs. With this experiment it has to be understood what is the time spent by two different systems to perform a single convolution product operation. The two selected systems

are the solo convolution core and the solo ARM processor. Result are expressed in terms of T_{end}-T_{start} time, where T_{end} is the time instant when the convolution product is completed and T_{start} when it is started. These results are presented in figure 5.2. They show the big advantage of using a special purpose core implemented in hardware with respect to the general purpose processor present on board. The convolution core is about 114 times faster than than the processor.
This kind of result is actually expected and constitutes a good starting point for the next tests, since it is the first time the developed core can proves its potentialities.

5.2 Tests Results

Three different systems have been analysed in terms of performance and resource utilisation. The first system consists of the convolution core interfacing directly to the board processor. This test has been introduced as a yardstick, according to with it's possible to evaluate the behaviour of this thesis innovation consisting on distributed RAM-based architecture. This system, in fact, is the main character for the second test that has been carried out. The third and final test is performed on a structure which is a modified version of the second one.

5.2.1 TEST 1: Convolutional Core And Processor

The system under test is the one described in figure 5.3.

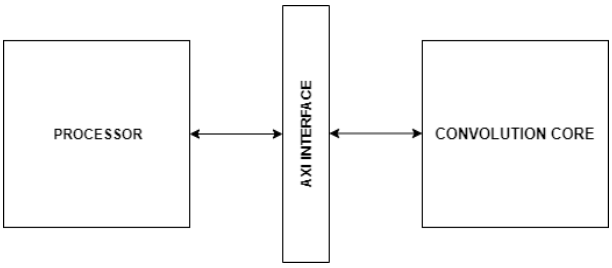

Figure 5.3: Test 1 Proposed System

Having no memories is a deeply affecting factor for the convolution core: its control unit has to be modified so that the system is able to understand when a new input data is necessary, requesting them to the processor while the execution is suspended. Similarly, the core stops again when a new output data has been produced and has to be read by the processor, before it is overwritten by a new convolution product result. Overall, a sort of handshake system has been introduced in input and output to properly manage core starts and stops. The input handshake-like system is graphically presented in figure 5.4.

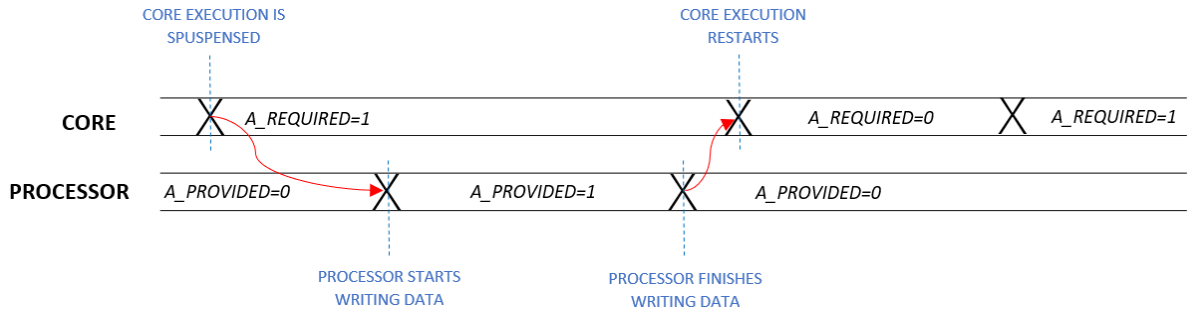

Figure 5.4: Handshake-Like System For Input Data

It is based on two signals, *A_REQUIRED* and *A_PROVIDED*, the first is driven by the core, the latter by the processor. When the cores arrives in a state where a new A is required, then it asserts *A_REQUIRED* signal, suspending its execution. When this happen, the processor sends a new A data, asserting *A_PROVIDED*. When *A_PROVIDED* is asserted, the core execution will be resumed. *A_PROVIDED* has to be 0 before a new A can be requested by the IP. Exactly the same system has been introduced for W vector, dealing with *W_PROVIDED* and *W_REQUIRED* signals. This is a safeguard introduced to avoid that an old A or W data is read by the core. The output handshake used a similar idea but with *START_RD_B* and *μP_RECEIVED_B* signals driven like in figure 5.5.
A, W and B have been introduced in chapter 5 and refer to feature vector, weight and feature map, respectively.

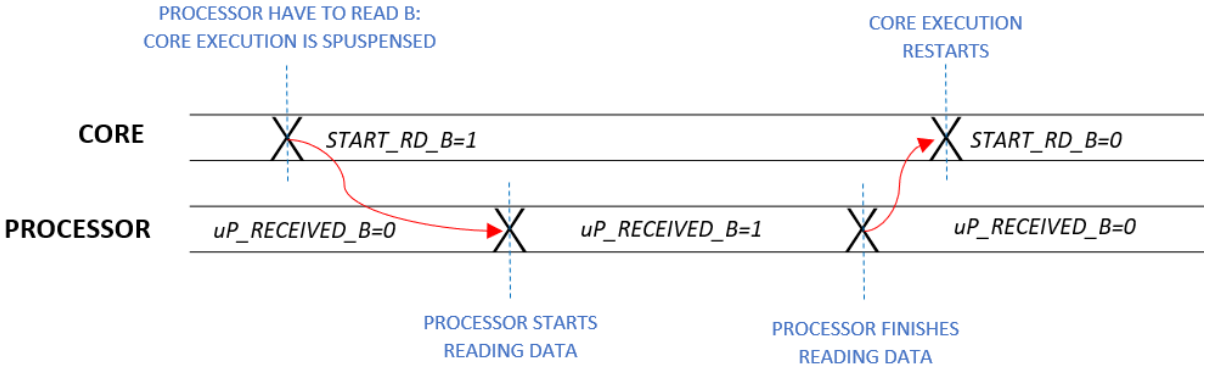

Figure 5.5: Handshake-Like System For Output Data

The studied system device implementation on Vivado is the one in figure 5.6. It can be seen how the logic mapped on FPGA is placed with respect to the processor and how they are connected together.

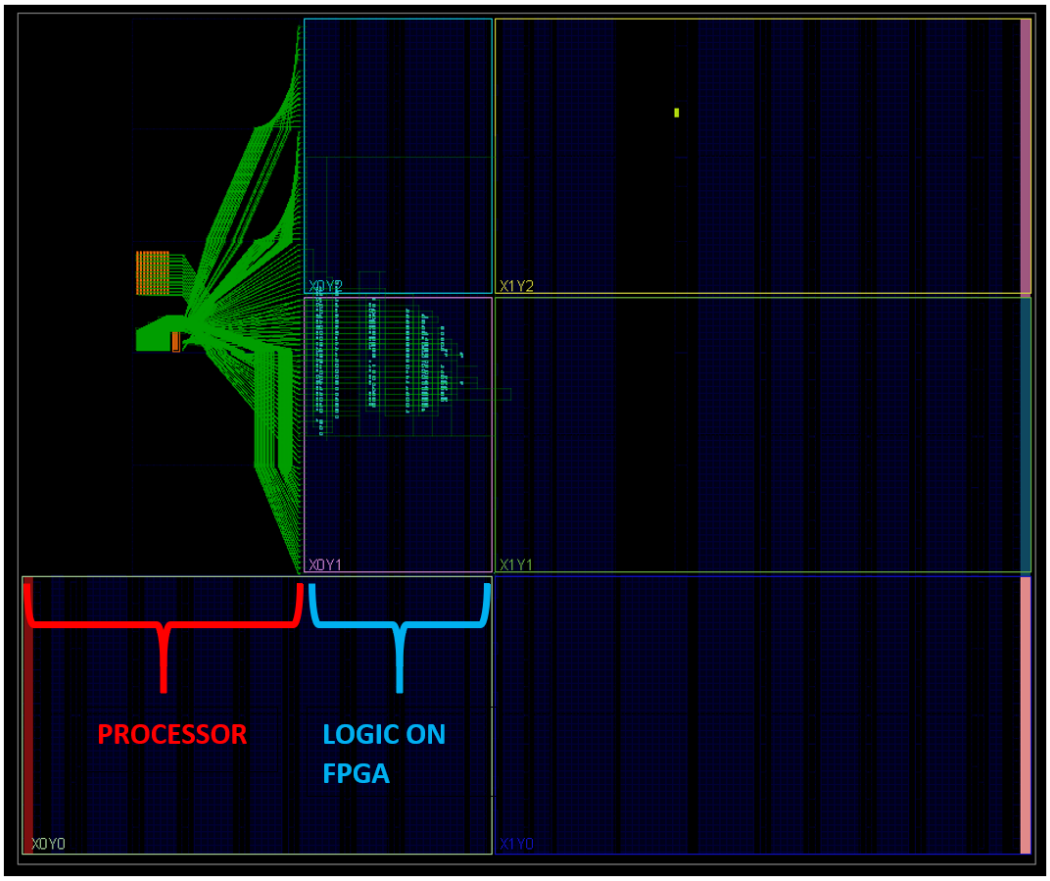

Figure 5.6: Test 1: In Device Implementation

Table 5.1 summarise the performance analysis conducted on four different sizing (L) for the feature vector: 128, 256, 512 and 1024. The weight vector size have been maintained fixed to three. For every vector, data parallelism is 8 bit. Performance analysis consists on evaluating the time difference T_{end}-T_{start}, so that an information is given on the elapsed time between *START* and *DONE* signals assertion.

	TEST 1 PERFORMANCE [μs]
L=128	95,74
L=256	188,20
L=512	374,98
L=1024	745,07

Table 5.1: Test 1: Performance Analysis

Resource utilisation is given in table 5.2.

	TEST 1 RESOURCES
PIPs	9599
LUT (LOGIC)	379
FLIP FLOP	293
BRAM36	0

Table 5.2: Test 1: Used Resources

5.2.2 TEST 2: Distributed RAM-Based System

Previous system has been modified adding three block RAM, according to the reasoning discussed in the chapter 4. The qualitative block diagram of the structure under test is the following:

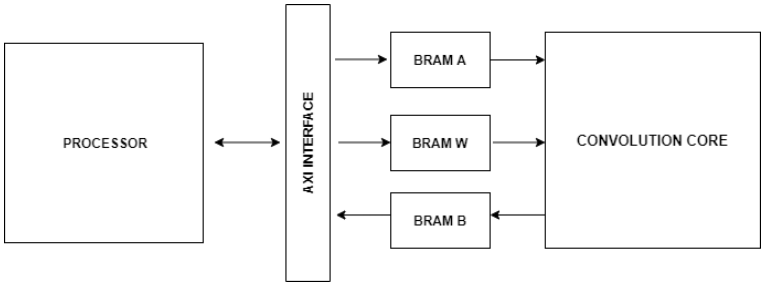

Figure 5.7: Test 2 Proposed System

Its actual implementation on Vivado is presented in figure 5.8, where it is possible to see that, with respect figure 5.6, now FPGA logic and processor have to interface with the three BRAM36. Vivado tool optimises these components placing, so that used block RAMs are the closest possible to the logic on FPGA and the processor.
This test has been performed so that performance analysis can be evaluated considering three different moments for the system operation:

- Processors writes feature vector (A) and weight (W) in the corresponding memories, in time T_{in};
- Convolution Core reads A and W memories to perform its operation on vectors, whose size (L) has to be defined. During this phase, memory B containing output data is written. These operations duration is T_{comp};
- T_{out} gives idea on time needed by the processor to read the whole B memory, previously filled up.

These measures have been made possible by the introduction of a more complex control system, consisting by three control units, one for every time interval to be estimated. They are reported in table 5.3.

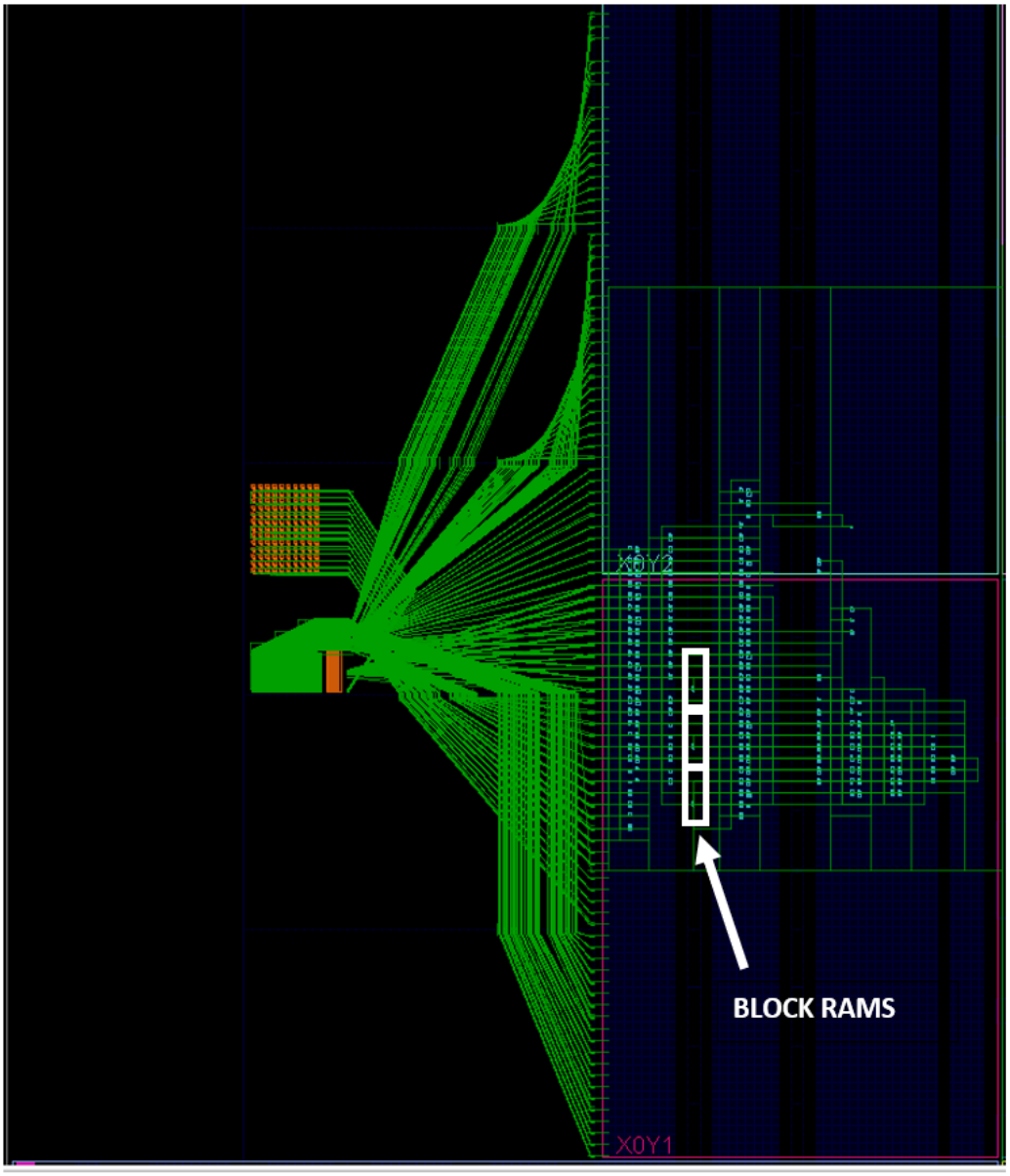

Figure 5.8: Test 2: In Device Implementation

	Tin [μs]	**Tcomp** [μs]	**Tout** [μs]	**Ttot** [μs]
L=128	35,61	6,1	37,29	79
L=256	66,98	12,51	74,41	153,93
L=512	129,96	25,33	148,64	303,93
L=1024	255	51,26	297,13	603,39

Table 5.3: Test 2: Performance Analysis

T_{tot} represents the time difference T_{end}-T_{start} and it is the data on which comparison with the previous architecture can be done. Resource data are presented in table 5.4, as well.

	TEST 2 RESOURCES
PIPs	12271
LUT (LOGIC)	403
FLIP FLOP	284
BRAM36	3

Table 5.4: Test 2: Used Resources

5.2.3 TEST 3: An Optimisation For Distributed RAM-Based System

The test is section 5.2.2 has been introduced with the purpose of performing some specific timing analysis but improvements are still possible. Several optimisation can be introduced to further boost its performance. The one in figure 5.9 is the proposed optimisation for the current technology.

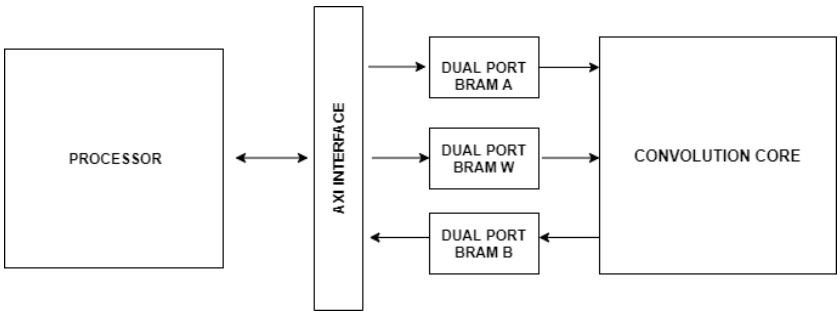

Figure 5.9: Test 3 Proposed System

Overall, the optimisation consists in managing the system in a way so that, when the processor starts writing the A memory, the core starts processing data as soon as possible. Under these conditions, the core has not to wait for the processors finishing to fill input data memories before starting the execution. To do so, some modifications have to be introduced:

- Single port rams, used in the previous configurations, have been replaced with double port rams, so that they can be accessed in read and write modes separately. Together with new memories, additional program counters for separated reading and writing operations are added too;
- System control has been modified as well. Some new signals have been introduced so that the core starts its execution without waiting for A memory to be completely full.

There is no image referring to the in-device implementation since it is expected to be exactly equal to that in figure 5.8, already posted. The same time analysis as for the previous case has been proposed for this architecture. Performance results are collected in table 5.5, resource needs in table 5.6.

	TEST 3 PERFORMANCE [μs]
L=128	66,92
L=256	129,59
L=512	255,08
L=1024	505,98

Table 5.5: Test 3: Performance Analysis

	TEST 3 RESOURCES
PIPs	12254
LUT (LOGIC)	400
FLIP FLOP	290
BRAM36	3

Table 5.6: Test 3: Used Resources

5.2.4 Comparisons

Timing analysis, summed up in graph 5.10, shows that the experimental system proposed in this thesis leads to an effective advantage in terms of performance. With respect to the solo core and processor test (blue branch of the graph), the BRAM-based architectures clearly outperform it:

- The second proposed test, orange branch of the graph, introduce an average speed-up of 1,21 times;
- With the third proposed test, grey branch of the graph, performance increases of 1,43 times.

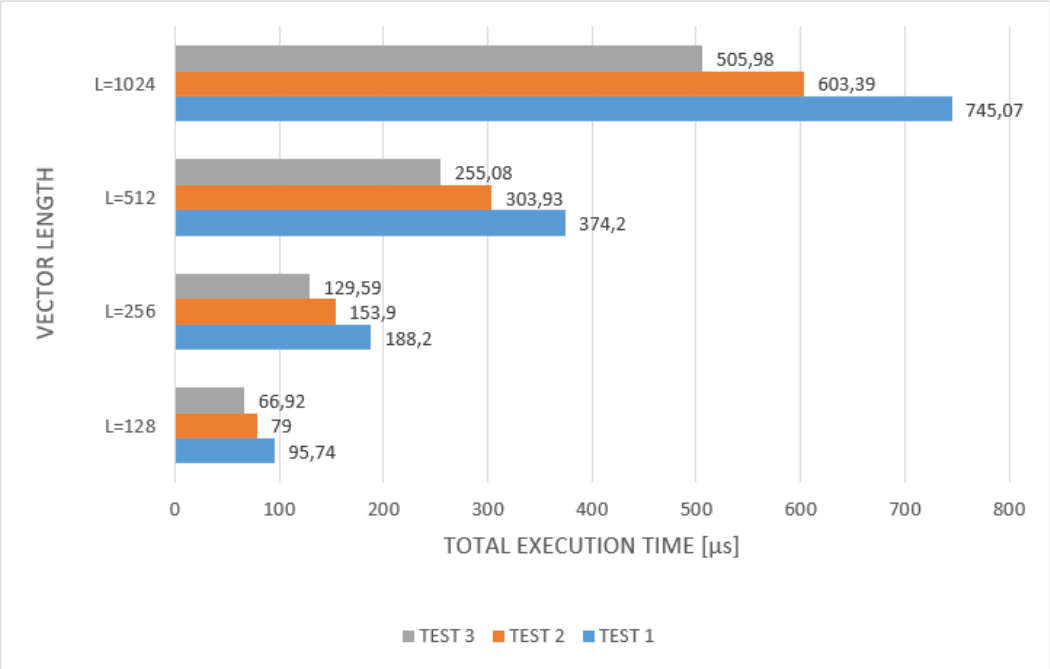

Figure 5.10: Tested Systems Performance Comparison

This is an important evidence of the potentialities for this thesis proposed works. Moreover, it has to be noticed that the resource utilisation is quietly the same for every tested systems, with the only exception of PIPs.
PIPs resource needs are compared in figure 5.13. It's possible to see that BRAM based systems requires much more interconnection points than the simple core and processor architecture. This is probably due to the number of additional interconnections to be added for block RAM routing and implementation in those systems.

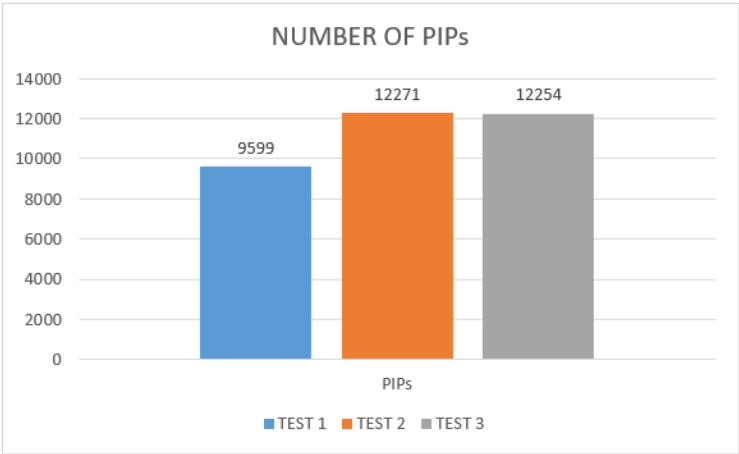

Figure 5.11: Tested System PIP Utilisation Comparison

LUT logic numbers for the three systems are comparable (see figure 5.12). Again, non

BRAM-bases architecture requires smaller logic since it is simpler in term of structure and control.

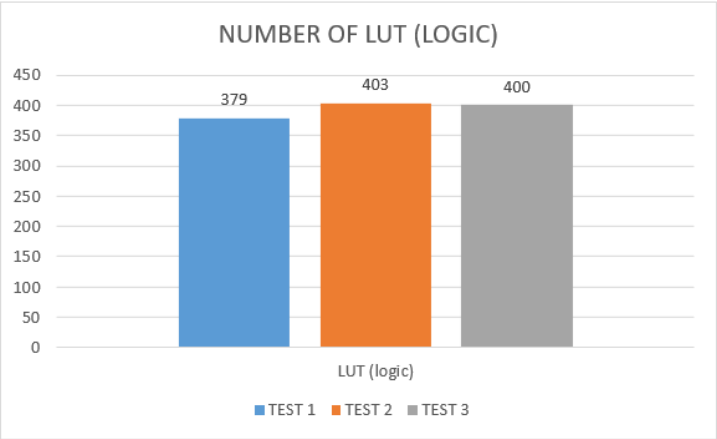

Figure 5.12: Tested System LUT Utilisation Comparison

Actually the flip flop utilisation, whose comparison is presented in figure 5.12, shows an interesting result: flip flop number is decreased in BRAM-based structure, this is probably related to the the fact that in TEST 1 system it was necessary the introduction of and additional 32 bit register, needed to temporarily store B.

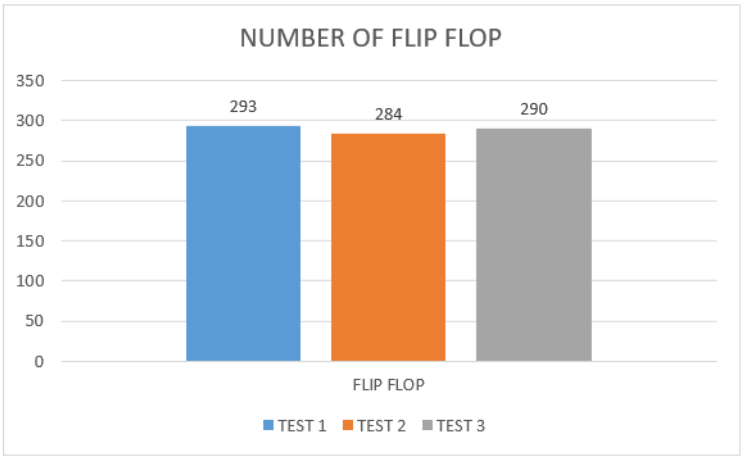

Figure 5.13: Tested System FLIP FLOP Utilisation Comparison

CHAPTER 6

Conclusion

This these work has proposed a distributed RAM-based core design. The automatic area optimisation integrated in Vivado tools have made the design easier in terms of resource utilisation, but still lots of work have been done to obtain a good behaving convolution core. Overall, the designed system have proved to be very promising, topping and exceeding the performances of a more traditional FPGA based implementation. Even if this work has been developed for FPGA, these results can be extended to every structure adopting the same design philosophy. The proposed design on FPGA is due to the lab availability.

6.1 Future works

With this thesis work, the first steps for this technology utilisation have be moved. Actually, a lot of researches has to be carried out from now on. In the next pages two different path for future work development are proposed:

- Further testing the current technology and slightly modify its structure for the sake of speed of execution;
- Realise new distributed RAM-based architectures trying to differentiate their applications.

6.1.1 Existing Architecture Improvements And Tests

The design developed in this thesis work could be tested in terms of **fault injection**, trying to see how it behave under this point of view with respect to the more standard FPGA based architectures. Having reliable hardware components is fundamental in every field of application, in autonomous driving context is fundamental.
Measurements on **power dissipation** and **temperature** could be conducted as well, to test the proposed structures in terms of energy efficiency.

Structural changes on the proposed hardware can be identified as well. Figure 6.1 shows the proposed changes.

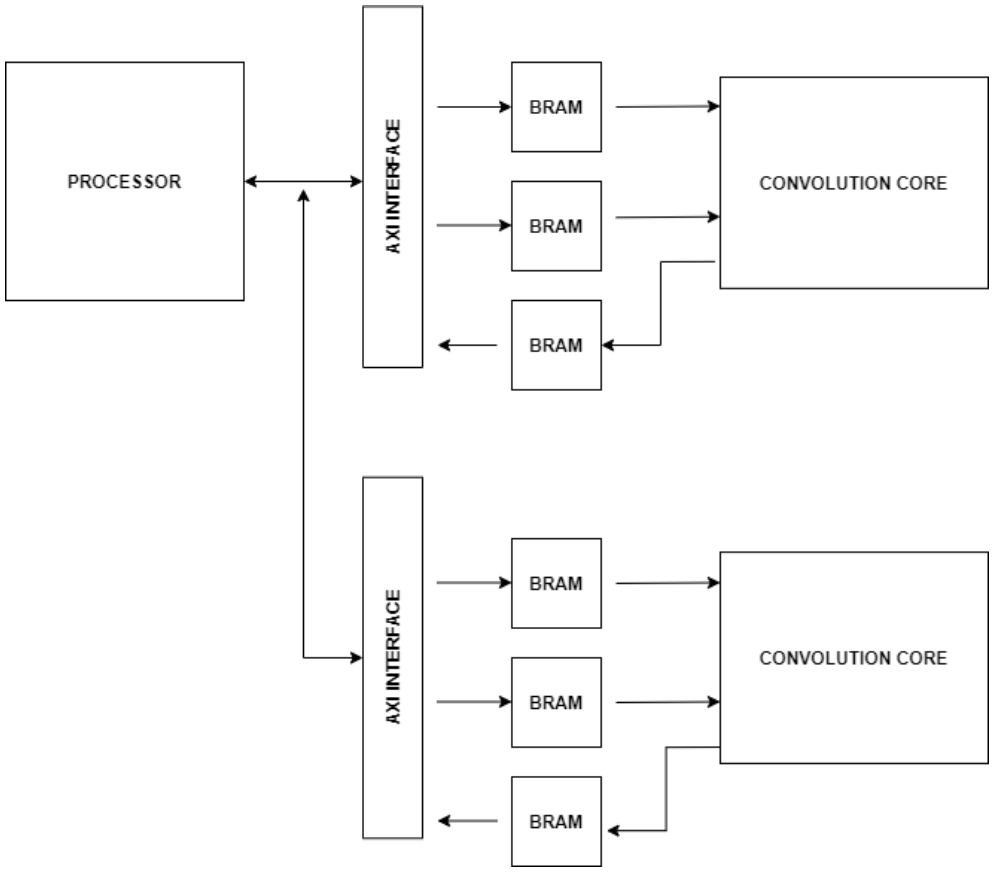

Figure 6.1: Interleaved Structure For Convolution Product

In this structure core and BRAMs system is duplicated in a way so that the processor can refers to two different structure contemporarly, writing and reading data in a interleaved way. This structure can further boost the system performance avoiding any kind of time waste. Of course, this approach introduce the necessity to use two times more resources, this is the reason why its application has to be properly evaluated.

6.1.2 New Architectures Realisation

As regard new architecture realisation, one proposed approach consists on substituting the embedded ARM processor with a **RISCV** structure designed for FPGA. This could be an interesting choice for the sake of integration: processor, convolution core and BRAM could be packaged in the narrowest space possible. Figure 6.2 shows the proposed modification.
In general, having the possibility to use a FPGA implemented processor will traduce in a better area managing so, a reduced data transfer time for data from processor

to memory (and vice-versa): luckily AXI4 protocol can be substituted with a more efficient communication system.

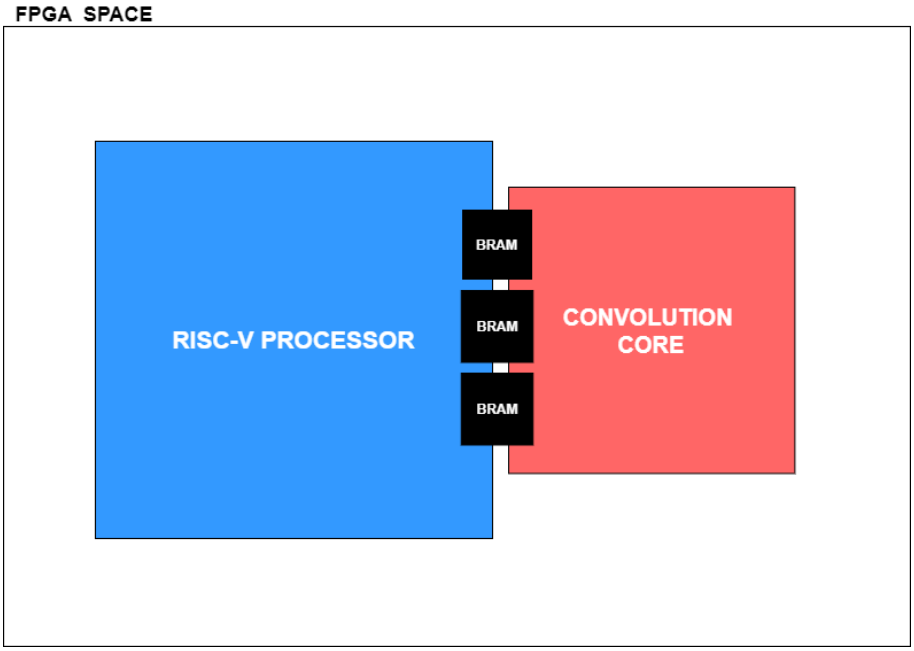

Figure 6.2: RISCV-based Computational RAM

Moreover, new application can be investigated as well. In autonomous driving field could be interesting testing this FPGA-based computational RAM approach for **convolution products on matrices**, **pooling** and **filtering** algorithms, in a way so that a complete set of applications is created.
Based on the obtained results, it can be expected that the approach proposed in this thesis best suits with all those algorithm that works on sequential data accessing, just as for the tested application.

References

[1] M. Xue and C. Zhu, *"A Study and Application on Machine Learning of Artificial Intellligence"*, 2009 International Joint Conference on Artificial Intelligence, Hainan Island, (2009)

[2] Abdallah Moujahid, Manolo Dulva Hina, Assia Soukane, Andrea Ortalda *"Machine Learning Techniques in ADAS: A Review"*, IEEE International Conference on Advances in Computing and Communication Engineering (2018).

[3] Tang Lung Cheung, Kari Okamoto, Frank Maker III, Xin Liu, Venkatesh Akella *"Markov Decision Process (MDP) Framework for Optimizing Software on Mobile Phones"*, EMSOFT '09: Proceedings of the seventh ACM international conference on Embedded software (2009).

[4] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton *"ImageNet Classification with Deep Convolutional Neural Networks"*, Advances in Neural Information Processing Systems 25 (2012)

[5] Takafumi Okuyama, Tad Gonsalves, Jaychand Upadhay *"Autonomous Driving System based on Deep Q Learnig "*, IEEE International Conference on Intelligent Autonomous Systems (2018)

[6] *"A Beginner's Guide to Convolutional Neural Networks (CNNs)"* URL: https://pathmind.com/wiki/convolutional-network/

[7] Shrivakshan, G.T. Chandrasekar, Chandramouli *"A Comparison of various Edge Detection Techniques used in Image Processing"*, International Journal of Computer Science Issues. 9. 269-276 (2012).

[8] *"A Gentle Introduction to Pooling Layers for Convolutional Neural Networks"* URL: https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/

[9] Ivan Vido, Milena Milošević, Ivana Škorić, Marijan Herceg, Dominik Mitrović *"Automotive Vision Grabber: FPGA design, cameras and data transfer over PCIe"*, IEEE Zooming Innovation in Consumer Technologies Conference (2019)

[10] D. G. Elliott, M. Stumm, W. M. Snelgrove, C. Cojocaru and R. Mckenzie, *"Computational RAM: implementing processors in memory"*, in IEEE Design Test of Computers, vol. 16, no. 1, pp. 32-41 (Jan.-March 1999).

[11] M. Sayed and W. Badawy, *"A new class of computational RAM architectures for real-time MPEG-4 applications"*, The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications, 2003. Proceedings., Calgary, Alberta, Canada, pp. 328-332 (2003).

[12] A. Mochizuki, N. Yube and T. Hanyu, *"Design of a computational nonvolatile RAM for a greedy energy-efficient VLSI processor"*, IECON 2015 - 41st Annual Conference of the IEEE Industrial Electronics Society, Yokohama, pp. 003283-003288, (2015).

[13] Du Nguyen, Hoang Anh Yu, Jintao Abu Lebdeh, Muath Taouil, M. Hamdioui, Said, *"A computation-in-memory accelerator based on resistive devices"*, 19-32. 10.1145/3357526.3357554 (2019).

[14] H. A. Du Nguyen, L. Xie, M. Taouil, R. Nane, S. Hamdioui and K. Bertels, *"On the Implementation of Computation-in-Memory Parallel Adder"*, in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 8, pp. 2206-2219, (Aug. 2017).