POLITECNICO DI TORINO

Facoltà di Ingegneria dell'Informazione
Corso di Laurea Magistrale in Electronic Engineering

Tesi di Laurea Magistrale

Implementation of efficient low power Hardware accelerators for applications of Deep Learning Neural Networks

Relatori:
Prof. Guido Masera
Prof. Maurizio Martina

Candidato:
Erik Anzalone

Aprile 2019

Acknowledgments

E' proprio così, finalmente sono giunto alla fine del mio percorso universitario. Pertanto scrivere queste frasi di ringraziamento mi sembra il giusto modo per concludere la mia tesi. È stato un percorso fatto di un vero è proprio "Deep Learning" sia accademico che personale. Ho imparato tanto, soprattutto il giusto approccio da seguire per affrontare i problemi che, ahimè, ho dovuto spesso sfidare e superare per raggiungere il mio obiettivo. Pertanto, mi sento di spendere alcune parole di ringraziamento nei confronti di tutte le persone che mi hanno accompagnato in questo mio percorso sia accademico che umano.
Prima di tutto, il primo grazie va ai miei relatori, i professori Guido Masera e Maurizio Martina per avermi sempre guidato e ascoltato durante questo mio lavoro di tesi.
Un grazie immenso va alla mia splendida famiglia, i miei genitori e le mie sorelle per avermi sempre sostenuto con i loro saggi consigli, nonostante la lontananza, e per avermi permesso di realizzare tutti i miei sogni.
Un grazie speciale va a te "Grilletto" Kri, di essermi stata sempre accanto e di non avermi fatto mai pesare l'esserti lontano; sei riuscita sempre ad accorciare le distanze che ci separavano, provando sempre a spronarmi a dare il meglio di me. Inoltre a te va il merito della "English revision" della mia tesi.
Ringrazio soprattutto i miei nonni, Pino e Tota perché continuate sempre ad insegnarmi il significato profondo della vita.
Grazie anche a Nicola, mio caro amico e compagno di avventura per l'esperienza Torinese. Presto sarà anche il suo momento.
Vorrei ancora ringraziare tutti i ragazzi del VLSI e i colleghi che mi hanno permesso di alternare i momenti di concentrazione e impegno all'interno del laboratorio a quelli di svago.
Per ultimi ma non meno importanti, i miei amici: Ciccio, Peppe, Giulio e Salvatore con cui condivido tutto. Superare questo scoglio è stato merito anche della loro presenza.
E per concludere, bisogna essere anche un po' onesti, grazie anche un po' a me per averci sempre creduto da quando tutto è cominciato ad oggi. Adesso si chiude un ciclo ma "il meglio deve ancora venire".

Summary

Artificial intelligence and the world of the Machine and Deep learning are in the center of the most frequent researches in engineering and even other fields. The analysis of "Big data" and the new automatic platforms (also referred to IoT) are more and more of interest. In particular, the world of the Artificial and Computer Vision bases its applications on this aspect very much.
The main source of studies is that of Neural Networks, created to identify the relations that link the elements of a set of data, by using a similar process to the human nervous system. They have many fields of application (Automotive, Biomedical and so on). So, they can re-elaborate and even modify the inputs to extract a lot of useful information to recognize an object in an image, recognize numbers or words written by hand, and so on...
In this work, the Deep Convolutional Neural Network will be analyzed, and overall a detailed analysis of the convolution operation will be done. This operation allows to perform matrix-matrix multiplication in a very efficient way. It is a way to filter the input and to extract useful information. For this reason, these networks can provide a classification for a set of images in order to recognize the images belonging to a specific class. However, these networks have to compute many convolutional operations based on their depth. The proposed work focuses on the implementation of an Hardware accelerator referred to a single layer of the Network, able to perform convolution operation of the RGB inputs. The goal is trying to increase its speed of execution and to reduce its power consumption. About that, the operation of multiplication has been analyzed in detail, by trying to optimize it in a low power way. The idea is to understand when this operation can be skipped to improve the power consumption (since the multiplication is the most onerous operation of the convolution) without worsening the throughput. So, some techniques have been applied to reach it, some of which have required an approximation of the results.
Then, the implemented architectures have been simulated and synthetized.
Finally, an hardware validation phase has been necessary to verify that the applied optimizations were suitable to an already trained and tested network. About that, a software model has been implemented and its results have been scheduled.

Table of contents

Chapter 1

Introduction

1.1 The world of the Machine and Deep learning

Nowadays people often hear about *Artificial Intelligence*, but also about *Machine-learning* and *Deep-learning*, terms sometimes used improperly as synonyms of the first. The term "Artificial Intelligence" (AI or IA) was first used in the 1950s and involves all those computational machines capable of performing tasks that are characteristic of human intelligence. These concern about planning, language comprehension, objects and sounds recognition, learning and solving problems [1]. But at the same time, the world of the *Machine-learning* and *Deep-learning* is constantly in evolution. Indeed these two techniques are simply some ways to achieve *Artificial Intelligence*. But there is a particular difference between two mechanisms.
Machine learning (ML) is a kind of AI subgroup that focuses on the ability of the machines to receive a specific set of data and to learn by themselves, by modifying algorithms as they receive some information about what they are processing. Therefore it is a way to "educate" an algorithm so that it can learn from various environmental situations. Education, or even better training, involves the use of enormous amounts of data and an efficient algorithm in order to adapt and improve the machine according to the situations that occur.
On the other hand, Deep learning (DL) is one of the ML approaches (based on the representation of feature learning of Machine learning theory) that refers to the structure of the brain and precisely to the interconnection of the various neurons. The "deep" in Deep learning depends on the many layers that are inserted into the deep learning models, which are typically neural networks [2]. For this reason, it exploits computational advances and training techniques to learn complex models by means of an enormous amount of data.

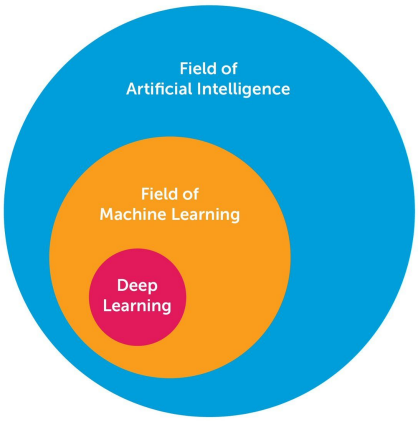

Figure 1.1. Different fields of Artificial Intelligence [3]

Both ML and DL base their applications on artificial vision, also called *computer vision*, that is the ability of a computational system to recognize objects digitally acquired by image sensors. The most recent studies base on the research of an efficient way to extract useful information from huge datasets that often contain billion of images or audio files. The algorithm used in these particular systems can recognize certain objects, differentiating them between different classes such as animals, things and people; at the same time learning from situations, or having a memory of what has been done to effectively use it in the next artificial acquisitions.

1.2 Importance of CNNs and Hardware accelerators

The most commonly used DL algorithm is the Convolutional Neural Network (to which reference will be made from now on, with the acronym CNN) that bases on Neural Networks specialized in data processing that have a grid or matrix structure such as black and white images (two-dimensional grid where the single value represents the intensity of the pixel in grey scale) or, as in the case of this thesis, color images where the grid is three-dimensional (typically RGB image). These particular Neural Networks use a mathematic linear operation called *convolution*. It is considered a state of art technique for object recognition in images and sound, and it is necessary in applications such as video surveillance, mobile robot vision, image search in data centers, and more [4] [5] [6] [7]. Nowadays there is a huge amount of devices able to capture pictures and videos; so the potential for CNNs

have enormously increased. For this reason, there could be a lots of potential interesting applications in the DL world.
Therefore CNNs are excellent algorithms in terms of accuracy and their computational structure is highly parallelizable, which, when exploited, can greatly increase performance. Indeed *General-Purpose Central Processing Units* (GPCPUs) are not optimized when computing such networks, because they are designed for mainly serial computations, and cannot exploit their parallel structure.
In the contrary *Graphic processor Units* (GPUs), *Field-Programmable Gate Arrays* (FPGAs) and *Application-Specific Integrated Circuits* (ASICs) are hardware architectures that can be designed to strongly exploit parallelism, and are surely better than CPUs in parallel applications.
However, GPUs perform very well on parallel applications, but their drawback is related on power consumption. Since lots of data centers consider power being the mainly financial cost and all mobile devices such as computers and mobile phones operate on limited power budget [8], GPUs could not used for several applications.
So, FPGA and ASIC accelerators have become surely the most popular for CNN applications because they are suitable for applications that require lower power and high performance.
For this reason many researchers are focusing on development on FPGA and ASIC accelerators in order to achieve excellent performances (speeding up the computation) and to optimize their power consumption at the same time. Indeed convolution operation requires many multiplications and sums that increase based on the complexity of the structure and size of the Neural Network. A similar system causes a certain consumption. What is more, the accelerators often require a large number of memory accesses to save and read partial results to obtain the correct result of the algorithm.
But, as it has been discussed before, parallelized approach allows to compensate this drawback, because the major part of the CNNs patterns are similar, so the entire structure can be pipelined to improve it in terms of throughput and frequency.
However, after the memory accesses, the multipliers executing represent the greatest source of energy expenditure and for this reason, some techniques can be applied to improve this aspect. The goal will be to optimize the number of multipliers that must actually be performed, because in the convolution there could be unnecessary products based on specific conditions.
All in all, in this thesis it will discuss on how a specific architecture can be improved, by reducing the memory accesses and the number of multipliers performed, for reasons that will be explained in the next chapters.

1.3 Proposed work and contributions

The aim of this thesis is to design and analyze a hardware accelerator which performs the convolution operation in order to be suitable in each layer of a CNN. What is more, the next goal is to apply some techniques to modify the accelerator in order to achieve significant low power results. These techniques would allow to minimize the number of accesses to the external DRAMs to fetch and to save data and reduce the number of multipliers, by skipping the unnecessary ones, without causing penalties in terms of speed and errors in terms of accuracy.
The main aim of this thesis focuses of the convolution operation and for this reason first, many researches have been investigated to study the Neural Networks theory (with a biological and physical analysis), the analytical model of this operation and how it can be represented in terms of hardware architecture. What is more, some techniques such as *Zero Skipping* [9], *Predict Multiplier Skipping* [10] already studied in the past and a new possible technique, that exploits *Equal Consecutive Weights Skipping*, have been elaborated and merge to understand how an efficient low power hybrid architecture could be created. Subsequently, synthesis logic has been executed to compare the different architectures in terms of speed, area and power. Finally, the hardware has been validated by using an already trained and tested Neural Network with a specific dataset to verify which architectures are better than others between different layers.
The following part of this thesis work is organized in other 6 Chapters whose topics are reported as follows.
In Chapter 2, the Neural Networks theory will be presented briefly and an introduction to the mathematical model of the Convolutional Neural Networks will be given. Thus the thesis enters into the heart of the proposed model, trying to explain in detail the functioning of CNNs.
In Chapter 3, a focus on the possible ways to create an hardware accelerator will be described, by starting from a detailed analysis of the 2D convolution showed in a previous work based on the rescheduled Data Flow Diagram [11]. Then the different architectures developed in VHDL will be described focusing in all the internal blocks.
In Chapter 4, the next step will be to simulate the hardware accelerators to verify their correct behavior. So a comparison of the results by applying different kernel filters is done by means of a *Matlab* script, in order to analyze the reduction of the number of executed multipliers among the various architectures.
In Chapter 5, the logic synthesis phase will be analyzed, showing how to obtain information about area, speed and power. Then the different architectures will be classified and compared between them and with similar architectures.
In Chapter 6, there is the functional validation of the Hardware accelerators where the different architectures will be applied to an already trained and tested Neural

Network model [12] in order to decide which architectures are better than others in the specific layers.
Finally Chapter 7 draws the conclusions of this thesis work, also including outlined considerations and suggestions about future works.

Chapter 2

Background on Neural Networks and Convolutional Neural Networks

2.1 Neural Networks

The concept of Deep/Depth learning is often associated simply to "Deep Neural network", in reference to the many levels involved. But what is a Neural Network? The first model of Neural Network was born in the 80s but they had a relatively short period of popularity. After they re-emerged in 2010 with the improvement of computational resources and the beginning of the DL world.
A Neural Network bases on the concept of neuron that is the basic element of nervous system and also the most important because it can be considered the primary calculation unit of the human intelligence. The human nervous system is made up of approximately 86 billion neurons connected to each other by a number of synapses (particular structures that connect neurons with other biological cells) of the order of 10^{14}-10^{15}. The diagram below shows a drawing of a biological neuron (on the left) and a common mathematical model associated to the connection among neurons (on the right).

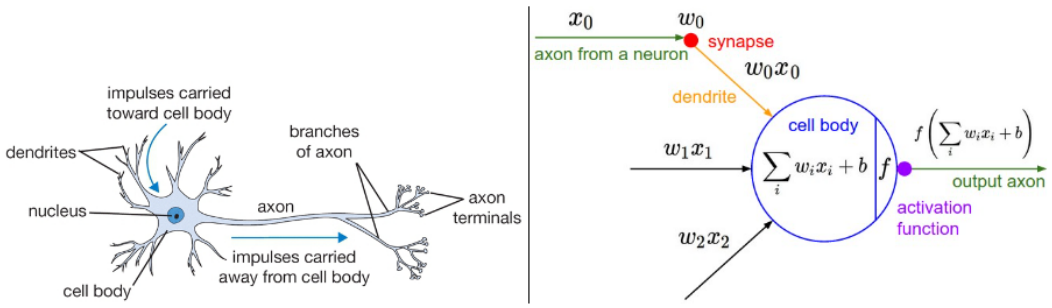

Figure 2.1. Drawing of a biological neuron (left) and a common mathematical model associated to the connection between neurons (right) [13]

Each neuron receives as input the signal from its dendrites (the minor fibers that branch out from the neuron) and, once processed, produces an output signal along its one axon. It branches off itself and connects the output to the next neurons through the synapse.
This biological activity can be represented by a mathematic model where signals that move along the axons (e.g. $\mathbf{x}_o$) interact with dendrites of neurons that are connected through synapses (e.g. $\mathbf{w}_o$). This interaction is a product (e.g. $\mathbf{w}_o\mathbf{x}_o$). The idea is that the synaptic strengths (the weights $\mathbf{w}$) can be learned and can control, in reference to the sign, the intensity and the influence of one neuron on another. In the basic model, the dendrites carry the signal to the cell body where they are summed. If the final result exceeds a certain threshold, the neuron send a spike through its axon. The precise timing of the spikes is not relevant, and it is assumed that the information is contained in the sum of the spikes. This sum is modelled as an **activation function** that typically is the **sigmoid function** ($\sigma(x) = \frac{1}{1+e^{-x}}$) but other times it is the **hyperbolic tangent** ($tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$) [13].
So each neuron executes a vector product among its inputs and its set of weights, adds a distortion term and finally applies a non-linear activation function (in the contrary case the Neural network would be a generalized linear model). The reason these functions are used is that their non-linear property increases the expressiveness of the network.
Neural Networks are structures that contain sets of neurons connected in an acyclic graph (cycles are not allowed). In particular, the outputs of some neurons can become inputs to other next neurons. For this reason, Neural Network models are often organized into different layers of neurons. Generally, the most used layer type is the **fully-connected** layer where there is a fully connection between neurons of two adjacent layers; instead there is not a sharing of connections between neurons within the same layer.

The two following pictures show two example Neural Network topologies made up from fully-connected layers:

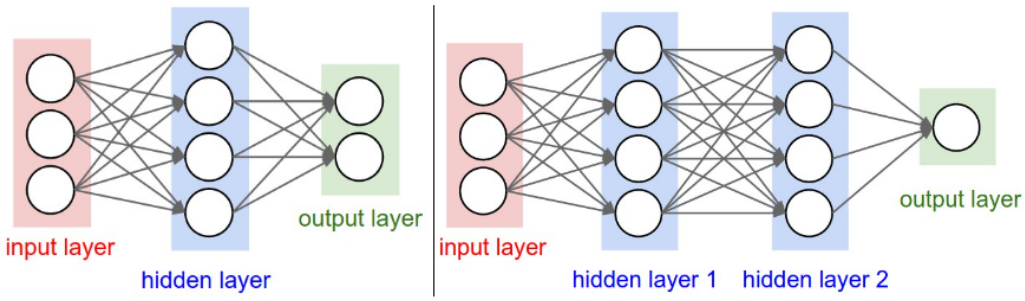

Figure 2.2. Fully connected layers Neural Network topologies [13]

As it can note in the images, the two Neural Networks are structured in a certain number of layers, each containing a set of neurons. The first layer is named *the input layer*, and the last layer is named *the output layer*. Intermediate layers are named *the hidden layers*. The input layer contains the initial input and uses it to compute the activation function for each of its neurons. Its outputs are carry out to the first hidden layer, and propagates until it reaches the output layer, where the final result is provided. The depth of the network depends on how many hidden layers there are. More hidden layers belong to network, more the network is deep.
People use the number of neurons, or more commonly the number of parameters to measure the size of neural networks. Referring to the two networks shown in the above picture, and if the inputs are excluded:

- The left network has $4 + 2 = 6$ neurons, $[3 \text{ x } 4] + [4 \text{ x } 2] = 20$ weights (because each neuron receives the results of the previous layer) and $4 + 2 = 6$ biases, for a total of 26 learnable parameters;
- The right network has $4 + 4 + 1 = 9$ neurons, $[3 \text{ x } 4] + [4 \text{ x } 4] + [4 \text{ x } 1] = 12 + 16 + 4 = 32$ weights and $4 + 4 + 1 = 9$ biases, for a total of 41 learnable parameters.

The most modern Convolutional Networks provide about 100 million parameters and are usually contain approximately 10-20 layers. However, more parameters are shared, the number of effective connections is considerably greater [13].
What is more, Neural Networks can evidently be considered as probabilistic classifiers. Indeed, once all the features of the network have been established, such as topology, number of neurons, connections, and so on, the weights of the connections have to be chosen in order to build a classifier, by using a training set: this operation

is called **neural network training**. This technique allows to update the weights based on the mistakes that network makes from time to time. So it can recognize different classes by being provided a specific set of training data, e.g. a set of cars and a set of non-cars. The training phase is necessary because the network can then learn to decide if a picture contains a specific object or not. For this reason, it will calculate the probability that the data belongs to a certain *class* based on the input data.
However neural network training is not a topic of this thesis, but the testing is the main one. But, first of all a detailed excursus about the Convolutional Neural Networks has to be done.

2.2 Convolutional Neural Networks

The Convolutional Neural Networks are extensions of the *Neural Networks* models, where object recognition in images or speech recognition are the main applications on which they work. As mentioned earlier, their name depends on the fact that in at least one of its layers a convolution is performed.

2.2.1 Inadequacy of the fully connected structure

As seen in the previous paragraph, the traditional Neural Networks transform their input, that is a single vector, through a series of hidden layers, where each neuron is connected to each neuron of both the previous layer and the next one (*fully connected*). The behavior of each neuron in the same layer is completely independent because they do not share connections with closer neurons.
In CIFAR-10 [14] there are small size images $32 \times 32 \times 3$ (32 *height*, 32 *width*, 3 *color channels*), a single neuron connected with this system would cause $32 \times 32 \times 3 = 3072$ weights, that is a very large amount but still manageable. However, the problems would occur if the dimensions of the images become important: for images that have 256 pixels for side, there would be a load of $256 \times 256 \times 3 = 196608$ weights for single neuron. If the network contains many hidden layers the number of parameters would become too huge, so the *fully-connected* architecture would be too expensive for this context and would quickly lead to overfitting [15].
The CNNs exploit the fact that the inputs consist of images and the architecture can be created in a more sensible way. In particular, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth. (depth refers to the third dimension of an activation volume). As next paragraphs will explain, the neurons in a layer will only be connected to a small part of the previous layer, instead of a completely fully-connected architecture. But the architecture fully-connected allows to obtain the final result, in the output layer. For instance, CIFAR-10, that

shows activation inputs with a volume $32 \times 32 \times 3$, would obtain an output layer of dimensions $1 \times 1 \times 10$, because the end of a CNN architecture reduces the full image into a single vector of class scores, arranged along the third dimension (depth).

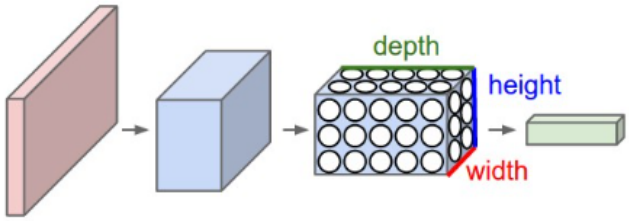

Figure 2.3. Arrangement of neurons in three dimensions (width, height, depth) [15]

2.2.2 Architecture of a CNN

The CNN model shows some different types of layers, in addition to the standard Neural Network layers: a *convolution layer* (that is often followed from a *ReLU* layer) and a *subsampling/pooling layer*. The presence of these new layers allows to handle particularly the local 2D structure of images, where there is an high correlation between pixels close to each other. Local correlation allows to extract and combine some features of images (e.g. points, edges, corners) to define specific features (e.g. wheel, windscreen, headlights) in order to recognize a specific object (e.g. a car). Following Figure shows a full network:

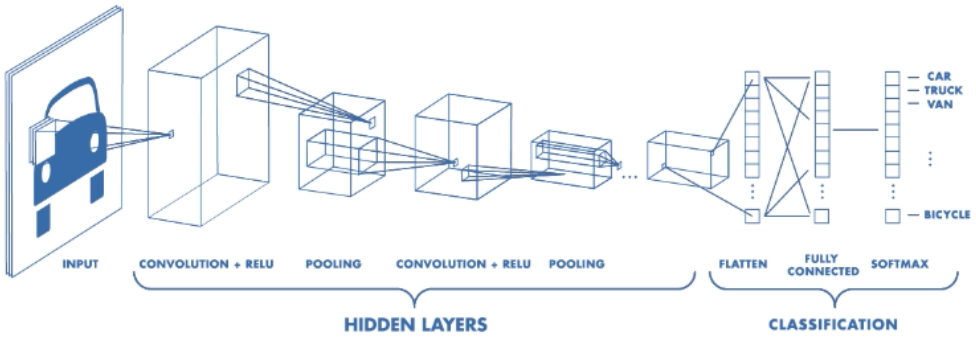

Figure 2.4. Architecture of a CNN [16]

As it can be seen, the *Hidden layers* perform a series of operations to detect the features. In the contrary, the classification part contains the fully connected layers that classify these extracted features. They will assign a probability for the object on the image in order to obtain a good prediction for the next images.
So the simplest CNN is able to perform four main operations, shown in the list below, that are also used to perform the most complex networks:

- Convolution operation
- ReLU
- Subsampling/pooling
- Fully Connected Layer

In the following paragraphs these operations will be described and analyzed to understand better the functionality of a CNN.

Convolution Operation

The *convolution operation* defines the *convolution layer* and is the core of a CNN. In discrete problems, the convolution operation is nothing more than the sum of the **Hadamard product** elements [17] between a set of parameters (which is called a **filter**) and a portion of the input that has the same size.
To give a detailed description of this operation when input are images, it is necessary to talk about image processing. As it has been anticipated before, a generic image is a matrix of pixel values. If there is a color image each pixel is associated to three different channels Red, Green and Blue (RGB); in case of grayscale images, there is a single channel where each pixel can be represented. This operation is a product between two matrices where the first is the image and it is called *Input feature map* (iFMAP) and the second is the filter (also called kernel), necessary to extract the features of image. The figure below helps to understand better this operation by showing an example that concerns about a two-dimensional input with a kernel 2×2.

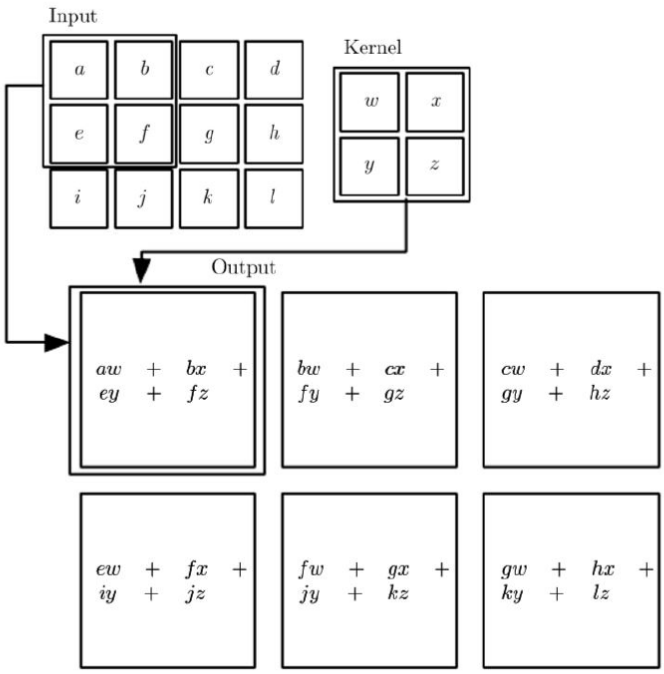

Figure 2.5. Convolution Operation in a two-dimensional case with filter 2×2 [16]

A 2×2 kernel is multiplied element by element for a portion of the input of the same size. The output of this operation is the sum between these products. The convolution operation is repeated by moving the filter along the entire surface of the input, both in height and in width. Its result is called *Output features map* (oFMAP) which is the first hidden layer of the network.
On the other hand, oFMAP is made up of neurons connected locally to the input by means of the parameters of the filter that generated them. Moreover there is the same process when there is a three-dimensional input, but the filter is extended in depth at the same way of the input, while maintaining the same spatial dimensions (width and height).
The following example shows a convolution operation of two 3×3 kernels (W0 and W1) applied at a three-dimensional input ($7 \times 7 \times 3$).

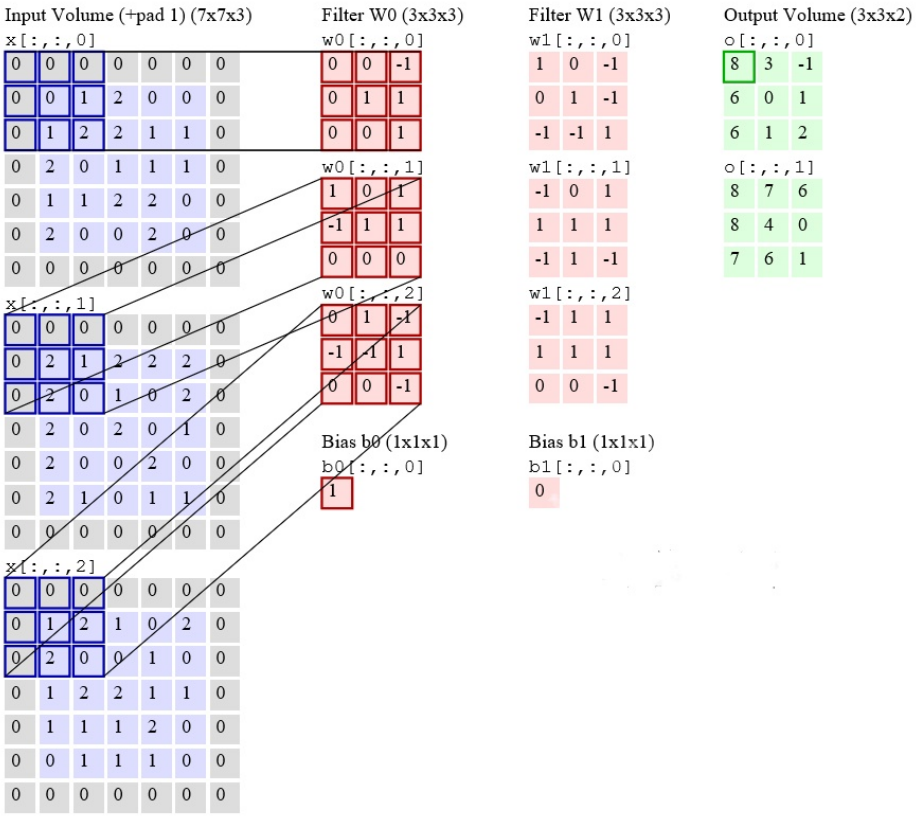

Figure 2.6. Convolution Operation in a three-dimensional case with two filters 3×3 [15]

For representative reasons, the volumes have been divided in two-dimensional matrices, but the depth of the filter is the same of the input one, because it operates for the entire depth while it is moving on the entire surface. For this reason there are two different approaches between the three dimensions: height and width have an interaction with the filter that is called *sparse*, while the depth maintains a *fully connected* approach.
Therefore, the values of the Output Volumes will depend on the different products between pixels and values of the filters that are performed and then added, but also b that is the bias associated at each specific operation that is always added at each result of the convolution.
So in a two-dimensional case, if $\mathbf{X}$ is the input image, $\mathbf{W}$ is the kernel matrix, $X * W$ is the convolution operation, the resulting feature map is defined as:

$$o_{ij} = \sum_{t=1}^{n} \sum_{s=1}^{n} x_{i+t,j+s} w_{ts} + b \tag{2.1}$$

Where x_{ij} is a value of the input matrix, w_{ts} is a value in the $n \times n$ kernel matrix, b is the bias and o_{ij} is a value of the resulting feature map matrix.
Now one question could be ask: how many neurons are there in the output volume and how are they arranged? There are three hyperparameters that control the size of the output volume: the *depth*, *stride* and *zero-padding*.

- **Depth**: it corresponds to the number of filters N_F that compose the layer, that look for different features in the input.
- **Stride**: is a very important parameter because indicates the sliding size and in particular the number of pixels for which the filter can be translated for each movement. Indeed *stride* is a system for regulating the output dimensions. If *stride* is one so the filter moves one pixel at a time, and for this reason each portion of the input is examined. Higher values move the filter by using greater jumps. Generally, the greater the stride, the smaller the output size will be. In the example above, the stride is equal to 2.
- **Zero-padding**: is a system that allows to a border of zeroes to the input volume in order to control the output sizes and to avoid inconsistencies during operations. This parameter indicates the width of this border, and it is often used to have the same dimensions between input and output.

So, width (O_W) and height(O_H) of the oFMAP can be computed how function of the relative dimension of the input (I_W or I_H), the relative dimension of the filter (F_W or F_H that are almost always equal), the *stride* (S) that is applied to the movement of the filters, and the amount of *zero-padding* (P) that is used for the borders:

$$O_W = \frac{(I_W - F_W + 2P)}{S} + 1 \tag{2.2}$$

$$O_H = \frac{(I_H - F_H + 2P)}{S} + 1 \tag{2.3}$$

Instead the depth is equal to N_F used inside layer. So if the input is $W_I \times H_I \times D_I$ and the filter has the same dimensions in height an width, the convolutional layer will perform an output $W_O \times H_O \times D_O$ where:

$$W_O = \frac{(W_I - F + 2P)}{S} + 1 \tag{2.4}$$

$$H_O = \frac{(H_I - F + 2P)}{S} + 1 \tag{2.5}$$

$$D_O = N_F \tag{2.6}$$

The choice of the stride and zero-padding has to be made in order that 2.2 and 2.3 return an integer. For instance, Krizhevsky architecture [12] (that will be described after) receives images with size $[227 \times 227 \times 3]$ and uses 96 filters $[11 \times 11]$ in the first convolutional layer, stride 4 and no zero-padding. For this reason the output of the first layer will have a size $[55 \times 55 \times 96]$, so each neuron of this volume is connected to a single region with size $[11 \times 11 \times 3]$; all 96 neurons, that belong to the same depth column, are connected at the same region $[11 \times 11 \times 3]$, but with a different set of weights.

ReLU

Until now the *sigmoid function* and the *hyperbolic tangent* have been presented as the possible choices for the *activation function*, but these solutions have been progressively set aside due to some practical problems, even if *hyperbolic tangent* is still used by some researchers [13].
The *Rectified Linear Unit* (ReLU) has become very popular in the last few years thanks to the increased performance that offers in providing a non-linearity. It computes the function $f(x) = max(0,x)$. So, each negative result of the convolution operation is simply thresholded at zero. One of the main advantage of the ReLU is its simplicity, because compared to tanh/sigmoid neurons that depend on expensive operations, the it simply consists of approximating to zero the negative elements of a matrix of activations. Moreover, it is a fast way to train the network, without losing much accuracy.

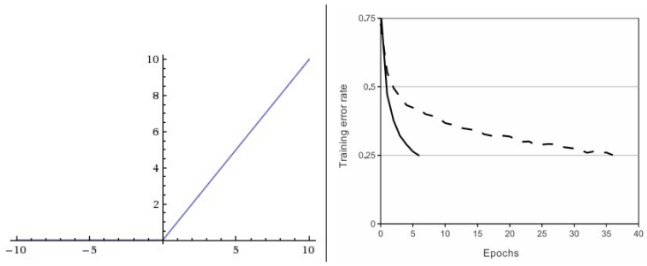

Figure 2.7. ReLU activation function and difference of velocity with hyperbolic tangent [13] [12]

It could have some problems during training due its fragility that can cause its

"death". *Leaky* ReLU represents an improvement of the ReLU because inserts a small negative slope (of 0.01, or so) in its null region but this aspect is not treated in this thesis.

Subsampling/pooling layer

When a specific feature of the image has been detected, its exact position loses its importance. For instance, the distance between two lighthouses of two cars depends on the type of the car and so it can be different. For this reason, the sensitivity of the network to the relative placement of features could be decreased, by reducing the accuracy of the all feature maps. So the feature map can be subsampled into $S \times S$ windows (submatrices) by means of **subsampling** operation, and then a **pooling** operation is performed on each respective matrix.
The pooling operations, which are used for CNNs, are deterministic without using weights. There are two types of pooling operations:

- **Max-pooling** (that is the most used) extracts the higher value of the analyzed window.
- **Average-pooling** obtains the average value between all the elements of the analyzed window.

Even in this case, the computation of the final output volume depends on both $W \times H \times D$ and the two request hyperparameters, stride (S) and dimensions of the window.
The following picture shows an example of *max-pooling*:

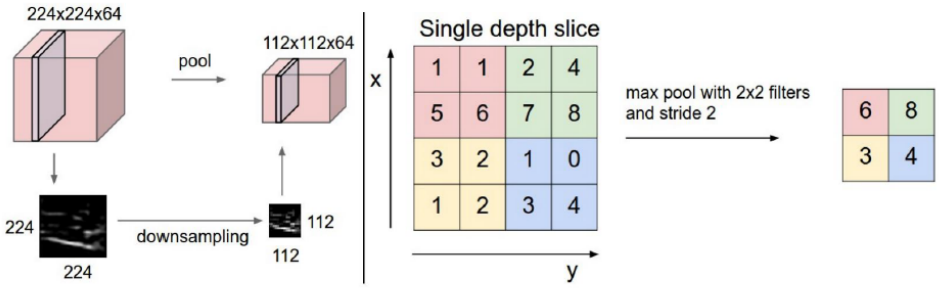

Figure 2.8. Example of max-pooling [15]

As it can be seen, the filter operates independently on each feature map, so the depth of the volume doesn't change. Instead, width and height are resized by a

factor $F = 2$ and $S = 2$, because each max is taken over 4 numbers. So 75% of the activations are discarded and the computational load is reduced for the next layers.

Fully-Connected Layer

The *Fully-Connected Layer* (FC) is exactly equal to any hidden layer that composes the traditional Neural Networks and operates on the output of the previous layer. These layers complete the structure of network and their main function is to execute a sort of grouping of information detected in the previous layers. So they associate to these information a number which will be useful to next computations necessary to obtain the final classification.
The idea behind this is that the network learns filters which activate themselves when specific types of features are detected (e.g corners, lines or blocks of color), that are called *low level features*; moreover they can still activate themselves when complex combinations are detected (high level features).

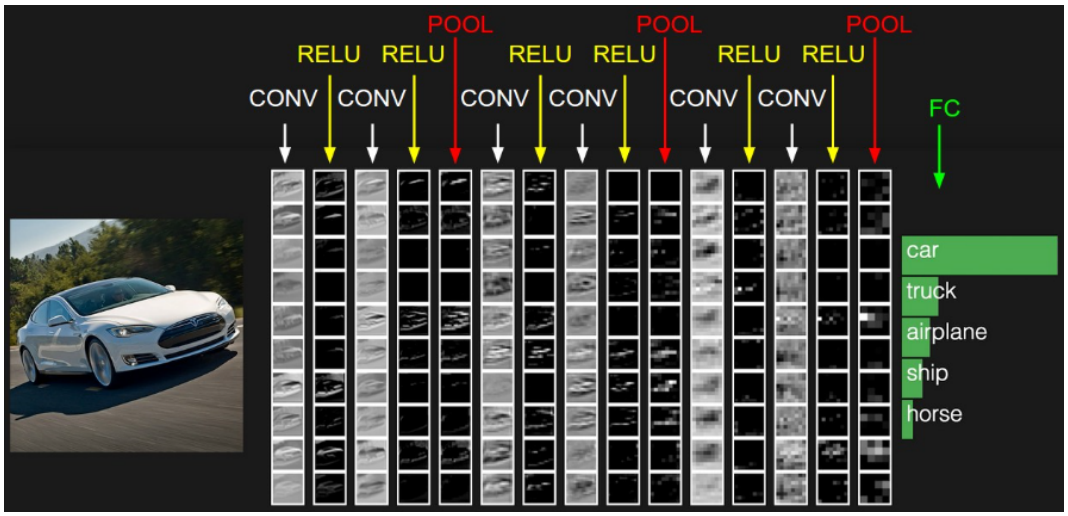

Figure 2.9. Different steps of a CNN architecture [15]

The figure above shows the different steps of a CNN architecture before that the FC guesses the object of the image. The different layers extract some information from the image and send them to next layers. In this case, there is always a ReLU after a convolutional layer, while the pooling layer is inserted after each two groups (CONV and ReLU layers). The final stage of the FC layer stores a statistic where the sorted top 5 scores are visualized and printed, after that machine has been tested more times. The architecture shown here is a tiny VGG Net [15].

2.3 Convolution Neural Network in Hardware

As it has been said before, today, CNNs can be employed in a wide range of applications, ranging from autonomous driving to industrial and medical applications. In this applications, indeed, CNNs are now able to emulate human performance and even overcome them. Therefore CNNs allow to obtain excellent results in term of accuracy even if this last one implies an high computational complexity. All CNN networks are extremely exacting from a computational point of view and require billion of calculations. For instance, the largest CNNs (from the VGG neural networks software models) [15] [18], when have to process an input image, require more than 30 billion computations. This could be a problem because an huge number of calculations often could reduce the use of CNNs in embedded/edge devices.
What is more, all CNNs also need memory, since their network parameters must be stored into megabytes of memory space. For instance, AlexNet[12] and VGG-16 CNN[18] have respectively about 63 million and more than 138 million network parameters which, if represented on 16-bit fixed-point, require respectively 126 and more than 276 Mbytes of memory for their allocation only. In addition, significant memory resources are required from intermediate feature maps, that can also have very large dimensions. For instance, the VGG-16 CNN requires about 6 Mbytes to store the largest input feature map [18]. Thus, since VGG-16 CNN extends on 21 layers, the required memory increases during processing of the input image. It is certainly less than the network parameters, but it may not fit to many embedded applications.
So the complexity of calculation and the large required memory can have a significant impact on both velocity of the network and its power consumption. For this reason, design and implementation of CNNs in hardware have to respect all these aspects in order to be preferred to General Purpose systems and GPUs on mobile applications and low power systems. Even if the complexity of the computation causes some drawbacks, the future of CNNs appears to be prosperous. Specialized hardware architectures for processing CNNs must be developed, by exploit their width, their depth, their ability to process great amount of input data, and their use to define the most complex classification by showing higher speeds, without exceeding low-power restrictions.

2.3.1 Hardware Acceleration of CNNs

During the last decade, there have been several proposed hardware architectures of CNNs that focused their design on the concept of *Acceleration*. The high computational complexity of CNNs, as well as their considerable memory requirements, causes an high demand for methods to obtain a faster computation and more efficiency both from the point of view of the resources used and the power consumption. One of these methods is the use of dedicated hardware accelerators.
To increase the speed of the architecture, the clock period must be optimized by reducing the critical path as much as possible without worsening latency too much (because these applications require a fast promptness of response). There are many techniques to reduce the critical path, but they must be applied without increasing power consumption, since an efficient hardware accelerator also must be optimized energetically. Therefore, the main objective of researchers is to reduce energy expenditure while respecting real-time constraints on speed and accuracy.
So, taking into consideration what has been said above, this work of thesis focuses on other aspects to create a good Hardware Accelerator of CNNs:

- First, convolution is a particular operation where there is an high repetitiveness of the data. Indeed, when a specific window of computation is executed, some data have to be reused in another window of computation. Moreover, also the weights have to be reused in the same channel. For this reason, CNN architecture can be pipelined and parallelized.

- The second aspect concerns about memory. Although the parallelization and pipeline can resolve the problem of velocity, a enormous amount of power consumption depends on off-chip and on-chip memory accesses. So, energy efficiency can be maximized by reducing redundant memory accesses. For instance, the use of registers and on chip memories can be a good way to store input and filter data. Thus, they can be reused without accessing to off-chip memory that contain them at the start. However, off-chip and on-chip memory accesses can also be reduced by studying a specific dataflow in order to avoid too accesses to off-chip memory and to use on-chip memories only for partial results and no for inputs.

- Then, parallelism of the data is another important aspect. Therefore, the appropriate parallelism must be chosen according to the type of application of the network. This choice could have many advantages on speed and energy efficiency. Indeed, the speed and cost in terms of energy of some operations, such as multipliers and sums, depend on parallelism of the input data. Also the cost of memory depends on parallelism of the data. For this reason, the choice of the parallelism can be done based on hardware components and it can have

some differences into the same architecture. For instance, the parallelism of the internal data to input memory can be different with respect to internal one to output memory; the multipliers can have input data with lower parallelism than sums because they are more onerous operation; and so on. Obviously the choice also depends on the required accuracy (the greater the parallelism, the more precise the data will be).

- Finally, some techniques can be used to reduce the number of multipliers of the convolution operation, by skipping redundant ones. Therefore, based on inputs or filters some multipliers of the convolution operation are unnecessary or negligible and for this reason, it can be skipped. For instance, a possible technique used to understand which multipliers can be skipped is *Zero-skipping* (where the multiplication can be skipped when at least one between a specific activation of the input matrix and the respective weight which have to be multiplied, is zero). Also the prediction of the result of multiplier can help in choosing if which specific multiplier is useful or no (it can be skipped if its result is less then a threshold fixed, which would be negligible for the final calculation of the oFMAP). Moreover, other techniques can be designed to skip a multiplication by analyzing activations and filters. Obviously, even in this case, the best technique is which that allows to have the best accuracy and the most efficient power consumption (more techniques can be merged to obtain the best results). However, the possible techniques to reduce the number of multipliers will be discussed in detail in the next chapter.

Some of these aspects also concern about subsampling/pooling operation but this thesis will focus on the detailed analysis of a hardware accelerator for the convolution operations which occupy more than 95% of CNN operations.
The general scheme of the Hardware accelerator proposed in this thesis is illustrated in the following figure:

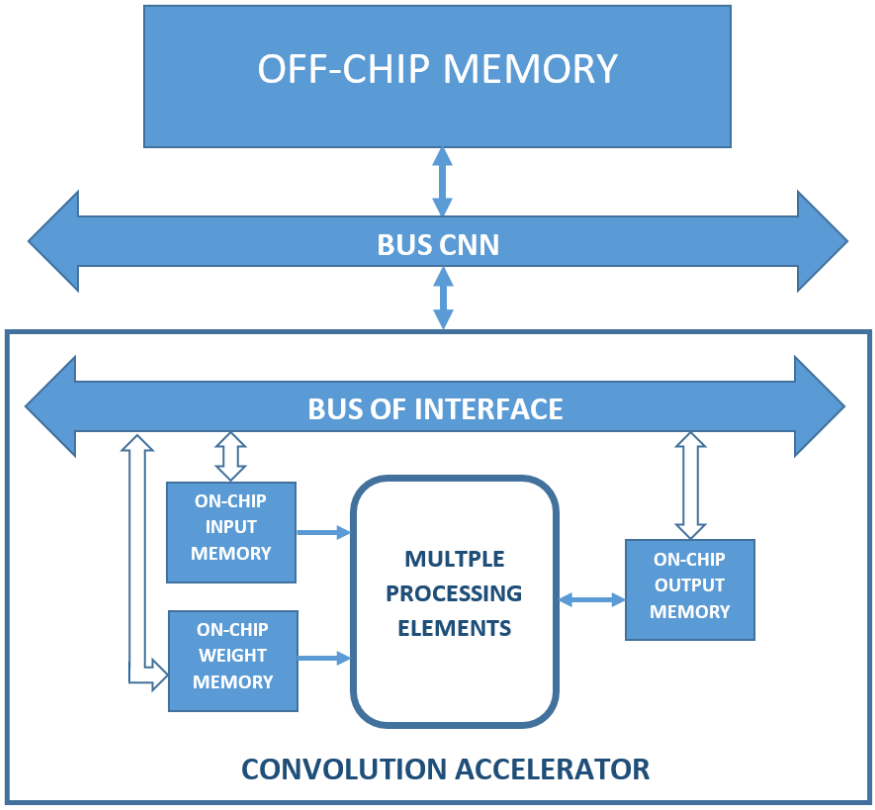

Figure 2.10. Hardware Accelerator Architecture

Externally to accelerator there is the *off-chip memory* that contains input features and filter weights which it sends to convolution accelerator. This contains:

- *Multiple Processing elements* (PEs) working in parallel by executing the convolution operation;
- Some *on-chip memories* to store inputs, weights and the partial and output results;
- A system of bus that allows to obtain a good interface between *Convolution Accelerator* and *off-chip memory*, and with other layers such as ReLU and subsampling/pooling by means of the external bus *BUS CNN*. Moreover, it also allows to have an internal communication between various components of the hardware accelerator.

The internal structure of each component will be described in detail in the next chapter. Of course, this structure can have different implementations and it can be

designed from scratch by keeping in consideration each aspect of design described before. The project discussed in this thesis starts from a general architecture such as the described one above, and it reworks past works [9] [10][11] to create a new optimized hardware accelerator.

2.3.2 Implementation choice

As it has been introduced in the previous chapter, the best method to implement an hardware accelerator for CNN applications are FPGA and ASIC (discarding CPU and GPU). The FPGA (*Field Programmable Gate Array*) is a semiconductor device, made up by a matrix of configurable blocks, connected via programmable interconnection. The FPGA is configured by using an *Hardware Description Language* (HDL). Because it is very reprogrammable, a FPGA offers an high flexibility on the possible hardware solutions that can be designed even after manufacturing. What is more, in order to get excellent potential performances, FPGAs need to be described at the low hardware level and not at the application level.
So FPGAs are different with respect *Application Specific Integrated Circuits* (ASICs), which are integrated circuits (ICs) manufactured for a specific use (they are not flexible). Reprogrammable hardware for CNN applications in the form of ASICs and FPGAs has been proposed to exploit the parallelism of the convolution operation and to achieve efficient performances. ASICs structures were preferred in the past to implement hardware accelerators for their high performances in making faster architectures (better than FPGAs) even if they cause a *non-recurring cost* (NRE). But nowadays FPGAs are more used because they represent a proper trade-off between flexibility of the applications, cost, performances and power consumption.

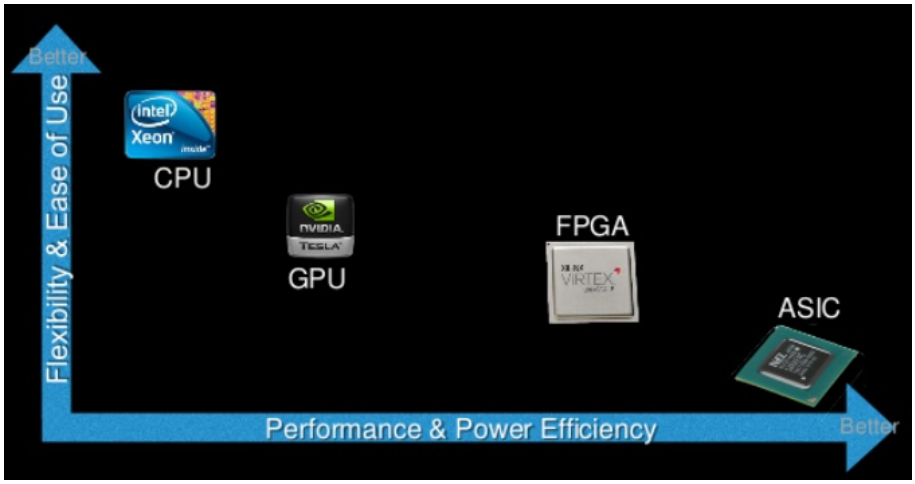

Figure 2.11. CPU vs GPU vs FPGA vs ASIC [19]

This thesis compares various hardware accelerators implemented through an ASIC structure since convolution operation is designed without many degrees of flexibility. What is more, it is the best solution for the logic synthesis to report each information about performances and energy efficiency regardless of the cost. The project will be described in VHDL and Verilog.

2.4 Previous work on CNNs and others hardware accelerators

Their accuracy and performance are the main reasons that make CNNs the most used systems for image processing and objects recognition; so their applicative fields increase. The most important projects about CNNs involve specific databases used to verify their ability in recognizing objects. For example, **ImageNet** is a large visual database that contains about 14 million images and it is one of the most used one to train and test a CNN (it will be the same database to which the project of this thesis refers). A part of this dataset is shown in the following picture, where there is a large choice about cars:

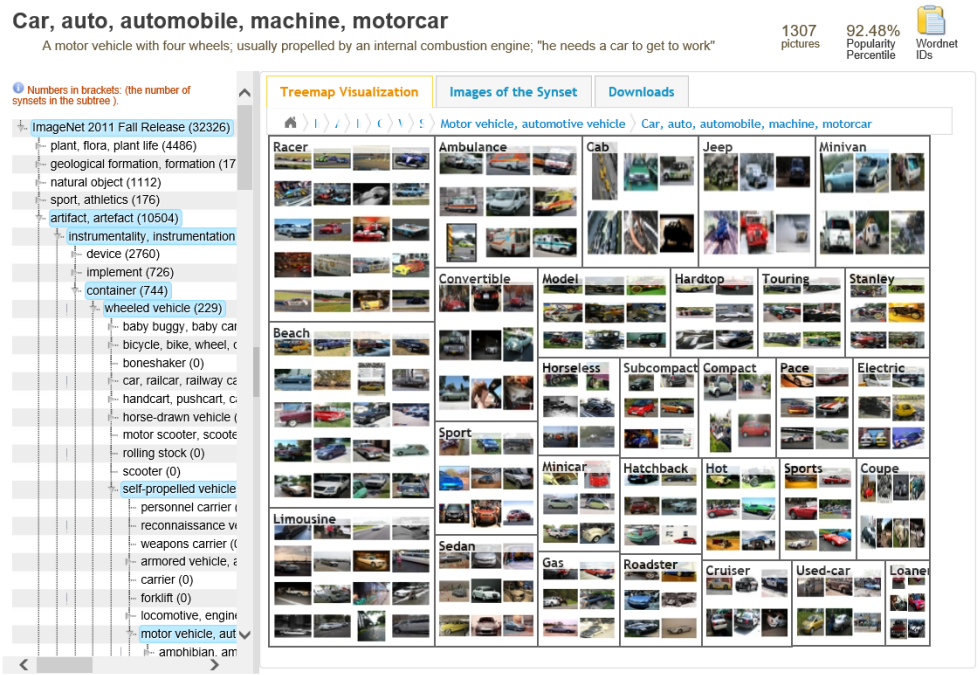

Figure 2.12. ImageNet Dataset [20]

What is more, there is also an ImageNet project that is a large visual database

designed for the use in visual objects recognition and in the software researches. Every year, it runs a global software contest, the **ImageNet Large Scale Visual Recognition Challenge** (ILSVRC), where there is a competition between software programs to correctly classify and to detect specific objects and even scenes and landscapes. Here the CNN architectures of ILSVRC top competitors will be described:

- **LeNet-5**[21], a 7-level convolutional network by LeCun et al in 1998, was used to classify digits and applied by several banks to recognize hand-written numbers (in 32x32 pixel greyscale input images). It was a network that achieved good results in those years but it had the problem of the low resolution of images;
- In 2012, **AlexNet**[12], developed by Krizhevsky from the University of Toronto, won the challenge against prior competitors by reducing the top-5 error from 26% to 15.3%. The top-5 error indicated the percentage of times that the correct prediction was not between the first 5 choices of the classification. It had a very similar architecture to LeNet-5 but it was deeper;
- **ZFNet**[22] won the challenge one year later, by achieving a top 5-error rate of 14.8%;
- The winner of the ILSVRC 2014 competition was **GoogleNet**[23] by Google. It reduced the top 5-error rate to 6.67%. A result very close to human performances. Moreover, the network had less parameters than AlexNet (from 60 to 4 millions);
- The second place of the same year was wan by **VGGNet**[18], an uniform architecture made up by 16 convolutional layers with many more parameters than AlexNet (138 millions). Its top 5-error rate was 7.3%;
- Kaiming He with his Residual Neural Network (**ResNet**)[24] introduced an innovative architecture thanks to "skip connections" and the batch normalization for features. His results were excellent because the top 5-error rate was 3.57%, by training 152 layers but with lower complexity than VGGNet.

The following table schedules these results:

Year	CNN	Developed by	Place	Top-5 error rate	No. of parameters
1998	LeNet(8)	Yann LeCun et al			60 thousand
2012	AlexNet(7)	Alex Krizhevsky, Geoffrey Hinton, Ilya Sutskever	1st	15.3%	60 million
2013	ZFNet()	Matthew Zeiler and Rob Fergus	1st	14.8%	
2014	GoogLeNet(19)	Google	1st	6.67%	4 million
2014	VGG Net(16)	Simonyan, Zisserman	2nd	7.3%	138 million
2015	ResNet(152)	Kaiming He	1st	3.6%	

Figure 2.13. Summary table of ILSVRC

Subsequent works focused mainly on the search for efficient architectures to implement hardware accelerators. For instance, in 2017 Yu-Hsin Chen proposed an hardware accelerator *Eyeriss* [25], where the energy efficiency was optimized including off-chip memory into the architecture. This work implemented multiple PEs of the architecture in matrix form, where the particular dataflow exploited was called *Row Stationary* (RS). This approach allowed that the rows of the filters were associated to PEs horizontally while the rows of iFMAP values were associated to PEs diagonally. Instead, the rows of partial sum of the convolution window were accumulated across PEs vertically, and there was a system to reuse and accumulate data within a PE in order to reduce accesses to off-chip memory.
The following figure shows this approach by applying a convolution between 5×5 iFMAP and 3×3 weight filter:

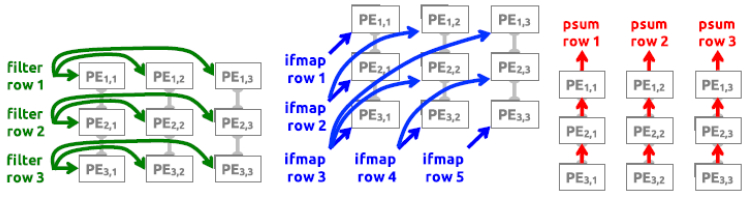

Figure 2.14. Dataflow in PE matrix in Eyerris [25]

Other works such as Desoli's one [26] and Kim's one [9] starts from *Eyerris* to obtain ultra low power optimization. Then there are other works, such as Huan's one [10] where parallel multipliers compute all products necessary to an output at the same clock cycle and there is their aggregation by means of an adder tree.
Instead, the proposed work in this thesis exploits a rescheduled dataflow to obtain further improvements. It will use a new approach different with respect Row Stationary by associating a different weight to each PE while the same activation is sent to each of ones. So there is a reduction of the data movement, by achieving excellent results in terms of speed and power. The next chapter will be about this, while the penultimate chapter will describe AlexNet, that is the used network to validate the work in question.

Chapter 3

The proposed work

In this chapter, the developed architectures for hardware accelerators will be presented. First, there will be a detailed analysis about the Data Flow Diagram of the convolution operation and how it can be rescheduled to obtain an efficient reduction of the memory accesses.
After, the whole implemented architecture will be described, by focusing on each block, its state machine (Control Unit) and the interaction between them. Then the starting architecture will be modified, by applying different techniques to improve it in the power consumption.

3.1 Rescheduled Data Flow Diagram

When an algorithm has to be studied in each its aspect, drawing its Data Flow Diagram (DFD) is the best way to start its analysis. Indeed, it provides a graphic overview of the system without going into detail too much, but that allows to know data dependencies, their real movement and how various operative blocks are connected.
For sake of simplicity, the DFD of the 2D convolution, represented in the following figure, will be analyzed.

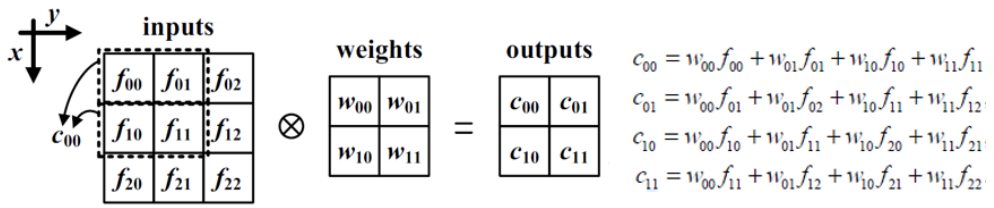

Figure 3.1. 2D Convolution [11]

The picture refers to a convolution between a 3×3 iFMAP and a 2×2 weight filter. As it can be seen, many input features are reused multiple times in the 2D convolution. For instance, f_{11} is necessary to compute c_{00}, c_{01}, c_{10} and c_{11}. So it is loaded four times. For this reason, in a 2D convolution the number of reuses is generally proportional to K^2, where K is the number of the filter weights, by causing a big impact on memory-accesses.
Now a dependence graph of 2D convolution is illustrated in the following figure:

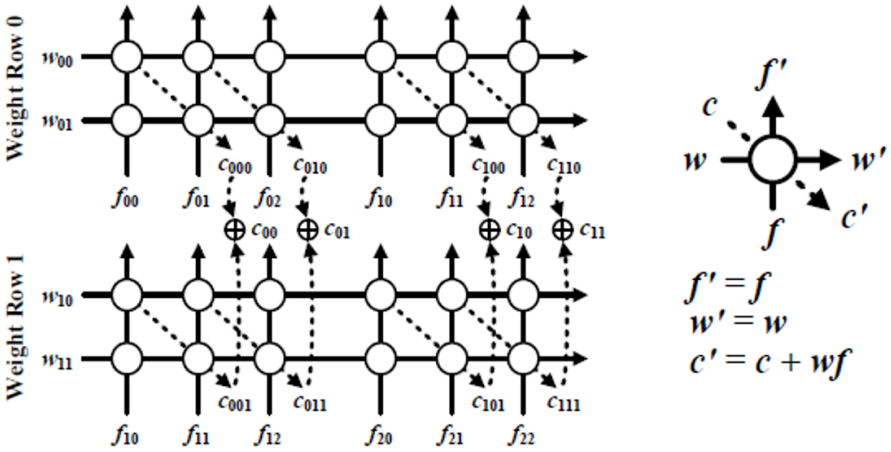

Figure 3.2. 2D Convolution DFD [11]

The circle indicates a multiply-and-accumulate (MAC), that represents the core of the convolution operation in hardware. It executes the product between a feature input and a weight and adds the result to a partial sum. The DFD is divided into two parts associated differently to two weight rows. Instead, only f_{10}, f_{11} and f_{12} are common to the upper and lower parts of the graph because they have to be multiplied for each weight. What is more, the intermediate partial sums are computed separately. So the final output feature is obtained by adding together the partial sums computed at the same column of the graph.

However, the portion of the input features that is common to the two separate graphs is loaded redundantly and it increases the number of the memory accesses. Previous approaches [25][26][10] focus on this kind of DFD. Although they were successful from the point of view of speed, their drawback consisted of high energy consumption due to the redundant memory accesses. Since the energy consumed for memory accesses is greater than that due to computational units, a rescheduled dataflow becomes necessary to improve energy efficiency of the algorithm, by minimizing unnecessary memory accesses.
Among the various rescheduling architectures proposed, that Jihyuck Jo'one [11] seemed to be the most interesting. The following figure shows Jo's rescheduled DFD of the convolution 2D:

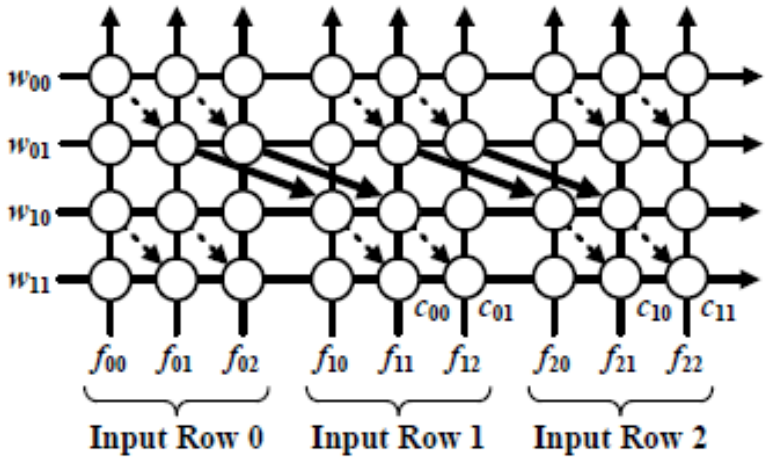

Figure 3.3. Rescheduled Data Flow Diagram [11].

As it can be seen, input features of the iFMAP are loaded just once. The partial sums obtained in the first and in the second filter rows are not added immediately, but they are accumulated waiting for the execution of the next input row. This accumulation that produces an incremental process is shown by black tick diagonal arrows in the picture. However, the time which the accumulated value must wait before being added depends on the size of the iFMAP; so this accumulation cannot be obtained through registers, but on-chip memories are needed. The use of the latter increases on-chip memory accesses but at the same time it reduces off-chip memory accesses because input features are loaded in the architecture once. This choice is surely a good solution because working with on-chip memories is better than working with off-chip memories. Next paragraphs show better this aspect.

3.2 Different architectures of accelerator based on rescheduled DFD

After the detailed analysis about rescheduled DFD, now some types of architecture can be presented to understand as rescheduled DFD can be applied to them. Indeed, previous researches analyzed different architectures for an hardware accelerator to apply in a CNN. There are three possible architectural implementations in fetching and transferring input and weight data from memories to PEs:

- The first is called **Broadcast**, where data are fetched one by one and they are sent to multiple PEs every clock cycle. Therefore, in the same clock cycle each PE receive the same data. Even if an Input Memory is often only required, some registers can be used to separate the data flow between memory and PEs;
- The second scheme is called **Forwarding**, where data are fetched one by one from memory but each PE receives the same data in different clock cycles, by transferring and delaying them through registers;
- The last scheme is called **Stay**, where data, once loaded, are kept fix in a PE for the entire convolution. This is a good way to reduce the number of memory accesses, because data are reused by the same PE. It is like there is a register that keeps the same data coming into the PE.

The following figure shows these architectural choices:

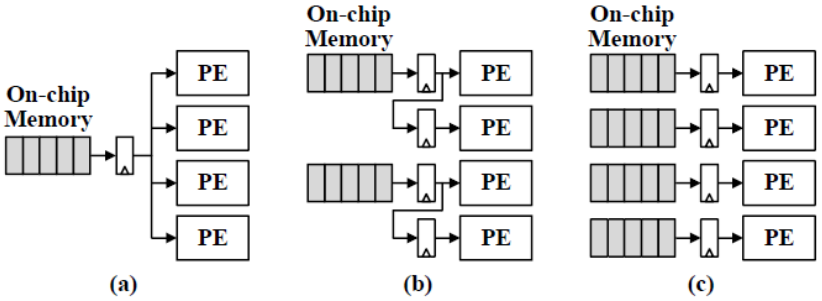

Figure 3.4. Transferring input and weight data. (a) Broadcast, (b) Forwarding, (c) Stay [11].

What is more, there are also three architectural methods to classify the process of partial-sum accumulation:

- The first method is **Aggregation**, where all partial-sum data are added at the same time by means of an adder tree. Nowadays this system is the least used due to its not efficiency in a parallel structure although it requires less memory resources. Moreover, it is very difficult to support when input sizes are very large;

- The second method is **Migration**, where partial sum is transferred to a neighboring PE or the same PE that generated it;

- The last method is **Sedimentation** where each partial sum is stored into an on-chip memory associated to each PE.

These architectural choices are shown in the following figure:

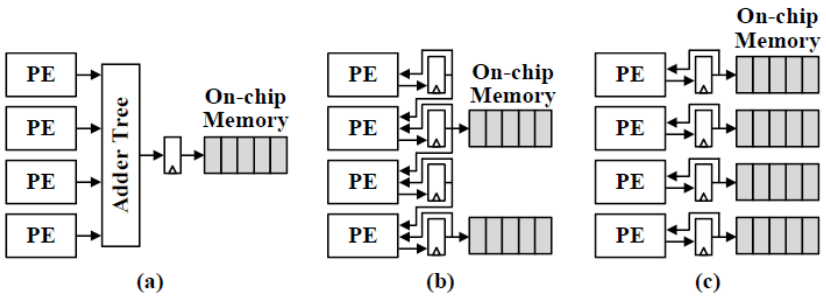

Figure 3.5. Output accumulation schemes. (a) Aggregation, (b) Migration, (c) Sedimentation [11].

As it can be seen, these choices involve the use of on-chip memories and registers, even if in different amounts and ways. Different convolution accelerators can be realized by choosing different loading and storing schemes and then adapting them to a rescheduled dataflow.
Among all possible configurations, the most ones investigated by Jihyuck Jo [11] are: BSM, BFS, FSM and FFS. The **BSM** (Broadcast Input, Stay Weight and Migration Output) and **FSM** (Forwarding Input, Stay Weight and Migration Output) are the two most efficient architectures. They are different only in input loading. In BSM inputs are loaded through a *Broadcast* configuration while FSM is characterized by input loading with a *Forwarding* configuration.
The **BFS** (Broadcast Input Forwarding Weight and Stay Output) and **FFS** (Forwarding Input, Forwarding Weight and Stay Output) are less used due to their lower efficiency in terms of power and area. Moreover, they can also cause higher normalized energy consumption than previous hardware accelerators [25][26] based on the different sizes of iFMAP and filter.

Fig. 3.6 and Fig. 3.7 show the normalized energy consumption of the previous architectures and the four different proposed ones, relative to two models, where both have input size 64×64 and filter size respectively 5×5 and 3×3.

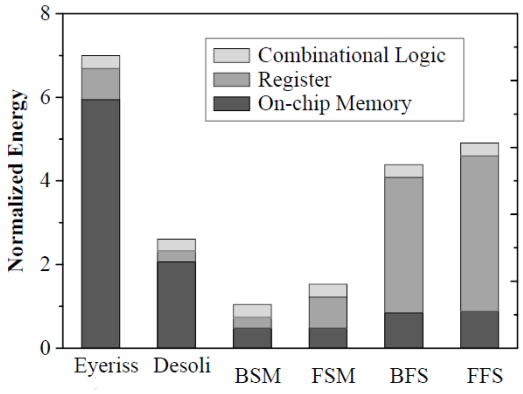

Figure 3.6. Normalized energy consumption for Input size 64×64 and Filter size 5×5 [11].

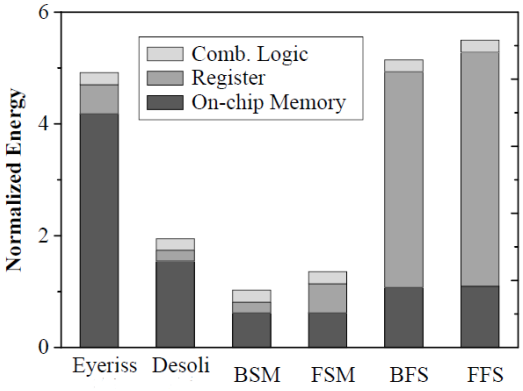

Figure 3.7. Normalized energy consumption for Input size 64×64 and Filter size 3×3 [11].

As it can be noted, both cases show as BFS and FFS, even if reduce energy due to on-chip memory, are worse with respect Desoli's accelerator, and in the case with Filter size 3×3 consume more than *Eyeriss*. Instead, BSM and FSM are the most efficient architectures in terms of energy. Even if the BSM architecture is the best approach for what concerns energy consumption, the final choice of the proposed work is FSM architecture because its approach results easily modifiable with further low power techniques. So the *Forwarding* approach, applied to iFMAP inputs loading, allows a delayed parallelization of all the computations (it is a form of parallelization worse than the BSM, but still effective). The idea is to load an input at a time and to sent it to all PEs with a certain delay given by the registers. This system also makes sure that input is fetched from the input memory once only. Instead for the weights, since the kernel does not change during a convolutional operation, the *Stay* approach is an optimum choice, because weights are loaded just once and remains inside architecture for the entire operation.
Finally *Migration* is the chosen approach for the output process, since it allows to use internal registers to transfer partial sum between PEs, by reducing memory operations. It is a good solution for what concerns the energy efficiency, since working with the registers is less energy expensive than working with the memories. The next sections will show how this architectural choice is applied to create the proposed accelerators.

3.3 Architecture Design

This section will show the suggested architecture for a hardware accelerator of a Convolutional Neural Network used for recognizing image of the dataset ImageNet. The architecture is related to a single convolutional layer and it will be presented through the following approach.
First of all, the *off-chip* elements have to be presented:

- There are one **Activations Memory** and one **Weights Memory** which contain *activations* and *weights* respectively (in other works a single memory is used to store both).
- An other element *off-chip* is **Output Memory** that is necessary to store the final result product by architecture.
- Then, there is a control block **Controller**, that handles off-chip memory accesses.

What is more, the *on-chip* elements that create the architecture are:

- The **3 channels Datapath** that is designed by means of a FSM approach.
- A **Control Unit** that manages the functioning of the internal elements of the Datapath.

Each element has been described in VHDL with a behavioral approach but only the Datapath results such as a *top entity* (structural approach).
Before describing various components, the general block diagram of the proposed architecture and even of the optimized ones have to be analyzed. It is shown in the following figure:

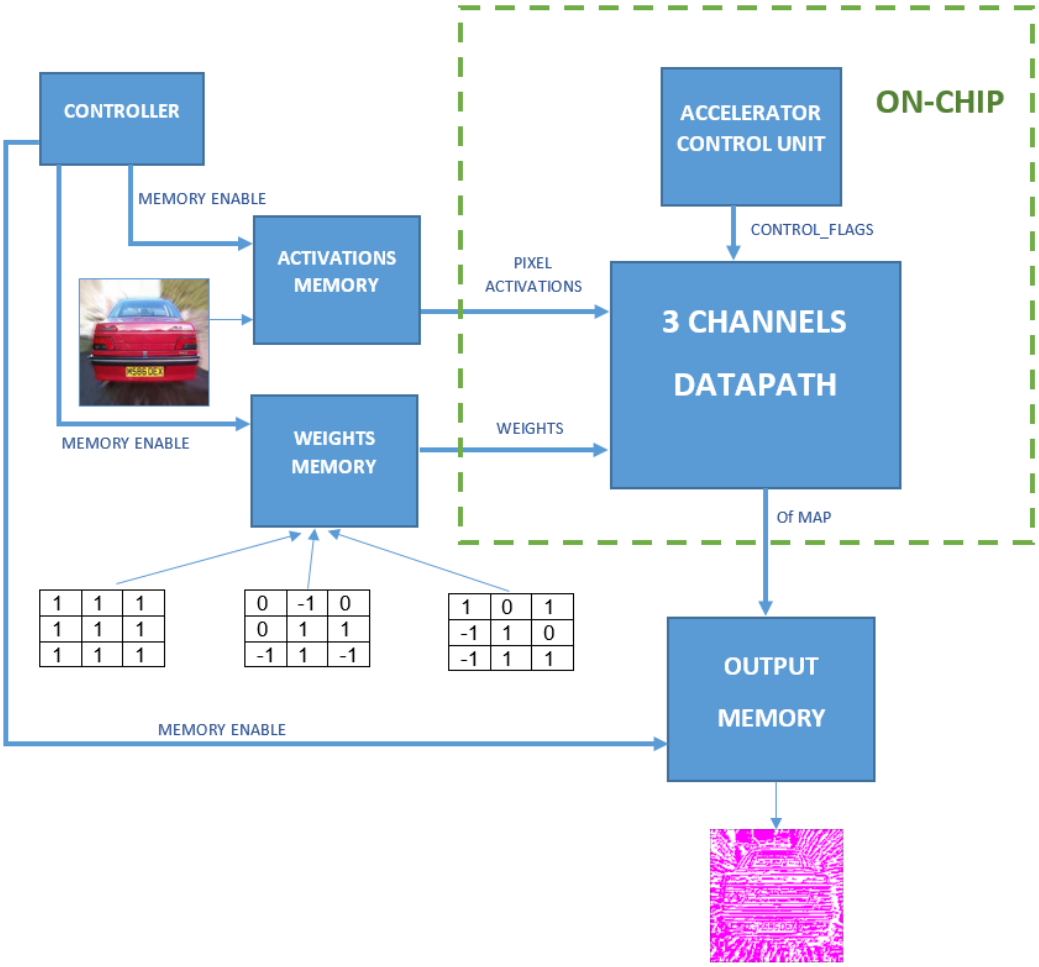

Figure 3.8. Block Diagram of the proposed accelerator

As it can be seen, the proposed accelerator is divided in two parts: off-chip and on-chip. The input RGB image is loaded into Activations Memory while Weights Memory receives kernels useful to perform convolution. Both memories can be written and read thanks to a enable signal generated by a specific controller. So, the Datapath receives pixel activations and weights respectively by Activations Memory and Weights Memory and associates them to respective channels. At the same time, the Control Unit handles the reception of these data and controls the processing of the Datapath by means of *control-flags*. After a certain time, Datapath generates the final oFMAP and it is stored into Output Memory, enabled by external controller.
In the next paragraphs the off-chip components used will be described but also the

on-chip ones associated currently to the starting proposed accelerator.
However the descriptions and the figures will be simplified to avoid complex discussions and explanations. There will rather be a focus on the important ideas behind the design and optimizations chosen.

3.3.1 Off-chip architecture

The off-chip elements are common to all proposed accelerators. They are described in the following paragraphs.

Activations Memory

The Activations Memory is a particular kind of memory where the input image is loaded. Its matrix structure allows to read some txt files that contain all the pixels of the RGB image represented in hexadecimal. In this thesis 128×128 image have been analyzed. As it is mentioned in other works [25] [9], this memory is usually a DRAM and it can be described in order to send data to on-chip Datapath with a specific order. For instance, both accelerators shows the matrix structure of the Datapath. But in *Eyerris* each PE receives activations of the same row, while in *Zena* each PE has to compute the result of a specific window of the convolution operation; so a zig zag scan reading of the memory is necessary to obtain that. In this thesis the design choice consists of reading activations column by column serially so that each input is sent to Datapath only once (it is a very important choice to implement Forwarding input configuration)
The memory is described in order to read a word of 27 bits every *clock cycle* because the pixels of each channel are quantized into 9 bits. For this reason, in the memory a word contains the pixels of the same matrix position of all channels.
So this memory has 16384 rows and 27 columns and an *Output enable* signal is used to start memory reading.
The following image shows an example of this memory structure, where pixels of the different RGB channels form the word:

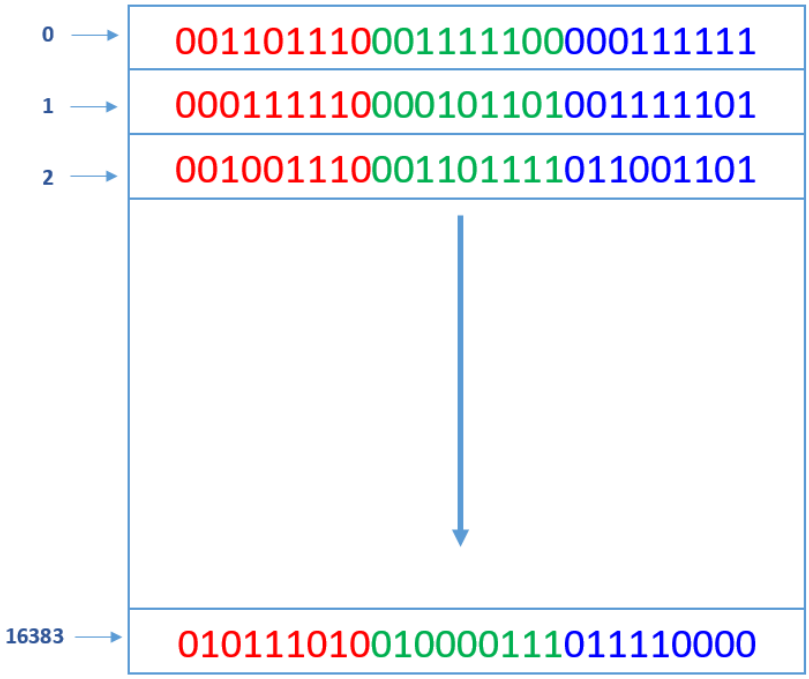

Figure 3.9. Activations Memory srtucture

Weights Memory

The Weights Memory has a similar functioning and structure of the Activations Memory. Also, it reads some text files that contain the kernel weights represented in hexadecimal. In this thesis 3×3 kernels are analyzed, so a smaller memory needs. Since a *Stay* configuration is chosen for loading weights into Datapath, this memory has to read all weights at the same *clock cycle*. So this memory is like a *Register File* composed by 9 registers and its 9 outputs are directly connected to Datapath of the architecture. Even in this case, each word is composed by three pixels associated to different channels. It is on 18 bits because the weights used have a small range of values and they can be represented on signed vector of 6 bits. Generally the kernels of the most tested Neural Networks have easily representable values on 6 bits. Reading of the memory depends on a control signal generated by the *off-chip controller*.
The following image shows the Weights Memory structure:

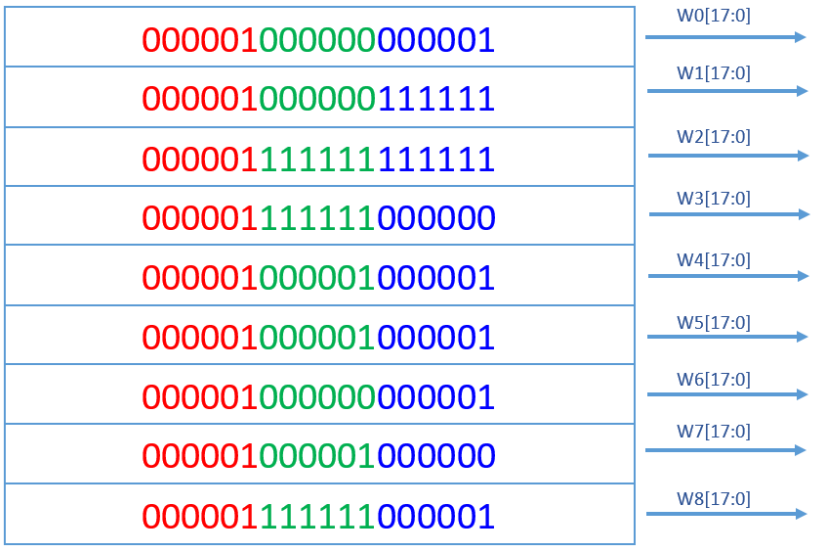

Figure 3.10. Weights Memory srtucture

Output Memory

The Output Memory contains the oFMAPs of the layer. In this thesis, the oFMAP computed is 126×126 since there are a 128×128 input, a 3×3 kernel and a stride equal to one, without zero-padding. Moreover, since the analyzed layer has to handle 3 channels, this memory can be designed such as a three-banks memory. Each bank has 15876 addresses and their words are on 16 bits that is the internal parallelism chosen for the architecture. Both reading and writing are controlled by Off-chip controller but instead to previous memory, the final results are written in a txt file.
When all values stored have been written into this file, a particular signal, named `End_Sim` is activated, to indicate that writing is finished. This signal is very important during simulation and during synthesis to extract switching activity of the architecture and after to evaluate the real power consumption.
The following image shows the Output Memory structure:

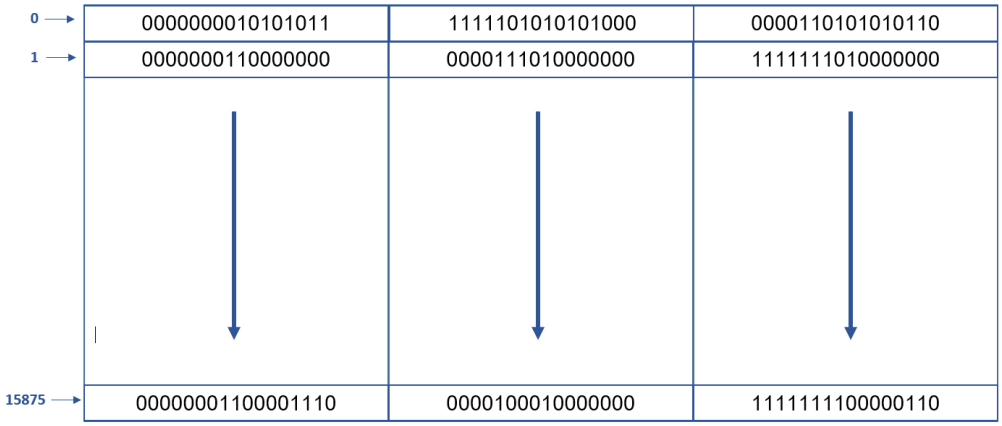

Figure 3.11. Output Memory srtucture

Controller

The *off-chip controller* is a state machine which handles the functioning of the off-chip architecture. It has three states: Idle, Loading inputs/writing outputs and finally Managing output, as it can be seen in Figure 3.12. In Idle the architecture is reset and all control flags are deactivated. The second state is achieved by a `START` signal. First, it involves loading activations and weights by respectively Activations Memory and Weights Memory. So a signal `Output_Enable` is activated to read by Activations Memory for sequential clock cycles, while the signal that control weights loading is activated only in one clock cycle and then it is deactivated. But, while the activations are read, at a specific clock cycle the first output, that is generated, is written into oFMAP memory by means of a control generated in the same state, and so on. The last state handles oFMAP reading of the final results. It is achieved when an internal counter counted 16402 Clock cycles from the beginning (when the `START` signal is received). So `START` signal is deactivated and the machine returns in idle. However, when `RESET` is active, the machine always returns in idle whatever the state it is in.
What is more, when there is the passage from Idle to Loading inputs/writing outputs the signal `START` is also sent to the *Accelerator Controller* internal to the chip. In the following image, the state machine is shown.

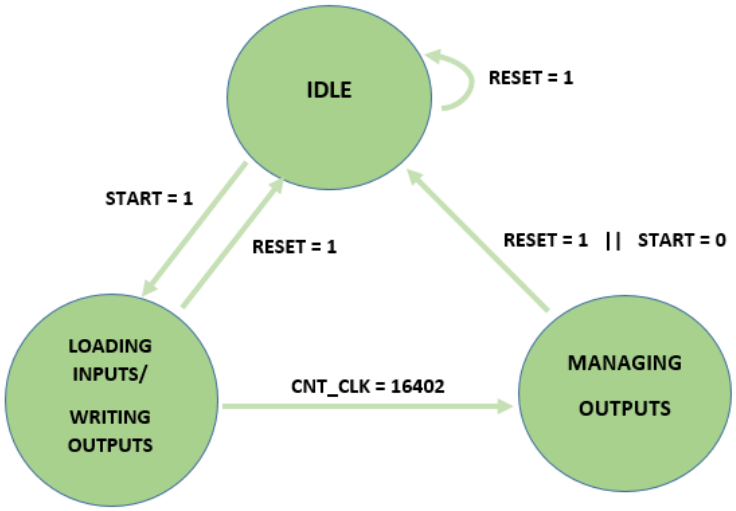

Figure 3.12. Off-chip State Machine

3.3.2 Datapath

The heart of the Hardware accelerator is into the Datapath where the entire convolution of the layer is computed. The goal is exploiting the parallelism of its structure in order to manage more channels in parallel. So the proposed accelerator is applied to handle 3 channels and to analyze RGB images even if the analysis is done on single channel.
By following the FSM architecture chosen, previously described, the Datapath is composed by three different elements: Processing Elements (PE), internal registers and On-chip memories. The rescheduled dataflow allows to delay of a certain number of clock cycles before performing useful multiplication, by using registers that store activations data. The number of registers inserted between the inputs of the Datapath and various PEs depend on size of kernel. For instance in the proposed work, input data are pipelined every time by means of 2 registers before reaching each PE, because 3×3 kernels are used. Moreover 9 PEs are necessary to perform the whole convolution. The idea is to divide all PEs in three groups, each of which manages the sum of the products of the column analyzed. A very important aspect concerns how weights are associated to each PE. In [11] is shown how the filter columns are distributed in opposite order with respect BSM, because FSM approach is more ALAP than BSM one. The common aspect is that the *Stay* approach allows to load them once. But in every PEs group of FSM approach, the first PE receives the last weight of each filter column, while the last one receives the first weight.

Thus, in every PEs group, the useful multiplications are executed when each PE receives the right activation which characterizes a column-by-column product between the convolution window and the filter one.
However other registers are necessary to store the partial result of each PE and to transfer it to neighboring PE of the same group. So the *Migration* method is exploited by delaying the partial sum every time for one clock cycle. Thanks to it, a *Multiply Accumulator* (MAC) is created. Indeed when the second and the third PE of the group receive the right activation to compute the useful column-by-column product, they receive the right partial sum at the same time.
Finally two On-chip memories need to store the partial result of each group and send it to the first PE of the next group to emulate a MAC even in this case. So they are connected at the end of each group and they are also separated from the last PE of the group by means of a register. On the other hand, the last group allows to obtain oFMAPs that are stored in a register and then directly in the Output Memory.
To better understand the data flow of the Datapath, the 3 groups of PE must be analyzed separately, through some schemes. The scheme in Fig.3.13 shows an example of the dataflow in the first group of PEs. As it can be seen, each row indicates the flow associated to each PE of the group. The oblique arrows indicate the registers used in the architecture. In particular, the red ones represent the registers that store partial sum of each PE and take it to neighboring PE. Instead the blue ones indicate the registers that store the final result of the group before it is stored into the first on-chip memory.

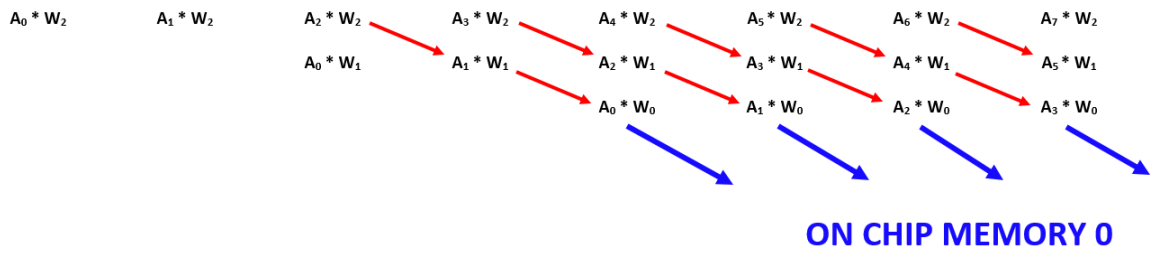

Figure 3.13. Example of the dataflow of 1st group of PE

The next schemes in Fig.3.14 and 3.15 show some examples of the dataflow associated to the other 2 groups of PEs. The vertical arrows indicate that data are read by on-chip memory and added to the product referred to the same window of convolution. The other arrows have the same function than the ones shown in the previous scheme.
Therefore, the last scheme shows how the oFMAPs are stored into a register connected to the last PE of the third group and then stored directly into the Output

Memory.

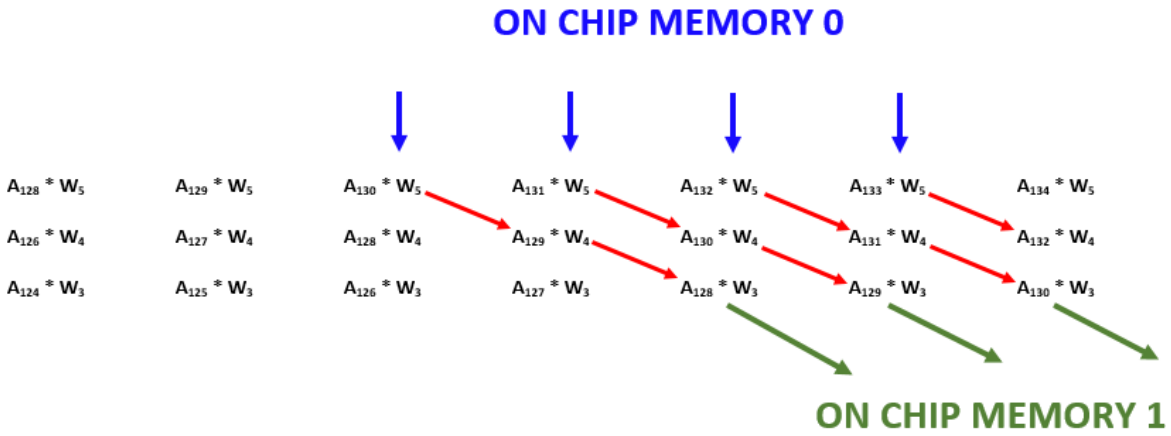

Figure 3.14. Example of the dataflow of 2nd group of PE

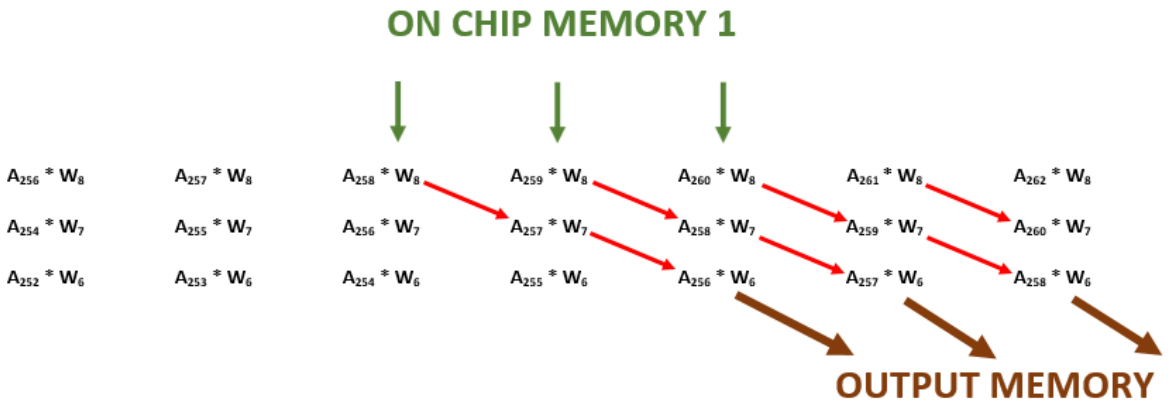

Figure 3.15. Example of the dataflow of 3rd group of PE

FSM approach consumes more cycles to set up the activation data, but the resulting throughput is the same as that of BSM. This approach is an original way to implement a convolution by using few resources. As previously mentioned, this choice is necessary to apply the low power techniques which will be explained in the next sections.
The following image shows the block diagram of the Datapath:

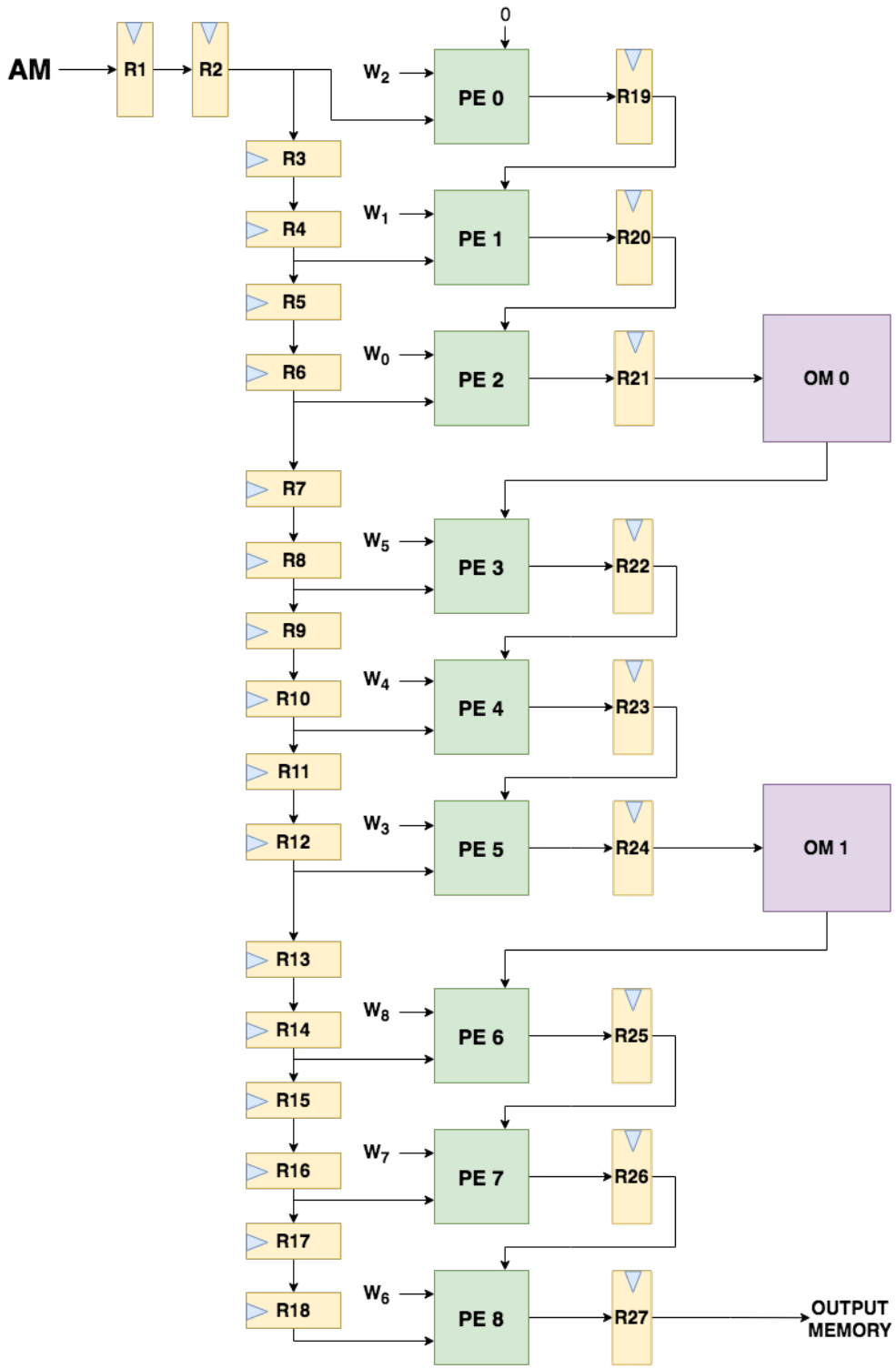

Figure 3.16. Block Diagram of the Datapath

As it can be seen, there are two groups of registers. The first group contains 9-bit registers that are used to forward the activations to each PE. Instead, the second one contains 16-bit registers since the partial sums are represented on 16 bits. Moreover, as said before, 9 PEs are divided into three groups. Finally, there are 2 On-chip Memories to store the partial result of the first two groups of PEs.
Now the PE and the OM are described in detail.

Processing Element

The PE is the most important element of the Datapath, because it allows to compute the product activation-weight and to add the result to the partial sum of the previous PE. So it contains a multiplication block followed by an addition block, that is named Multiply and Accumulate (MAC). In the most used software Neural Networks, activations and weights have a Floating-point representation, but in the proposed work a fixed-point representation has been preferred to reduce the complexity. However a signed multiplication 10×6 (and not 9×6) is executed to have a major precision. This choice is also motivated by the fact that, due to a *Batch Normalization*, often used in the first layer of networks, negative inputs can be generated; so a further extension of sign inside the multiplier is necessary.
So the product result is represented on 16 bits and it is an input of the adder. It adds the partial product to the partial sum of the previous PE of the same group or to the partial result of the previous group, in the case of PE3 and PE6. Instead, only PE0 computes a sum between its partial product and zero. For this reason, the internal parallelism of the architecture is 16 bits. This choice allows to reduce the power consumption.
The following picture shows the PE architecture:

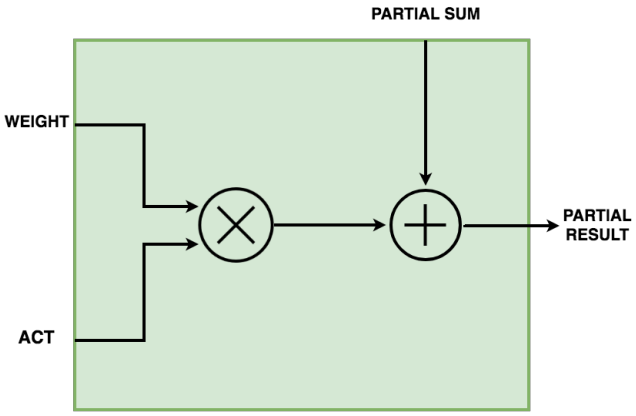

Figure 3.17. Block Diagram of the PE

On-chip Memory

The On-chip Memory is necessary to store the partial results obtained during the computation of each convolution window. As it has been said before, in the Datapath there are two On-chip Memories associated to the first two groups of PEs. Thanks to them, FSM approach is executed to achieve the final and the correct result. Its structure is very simple. It is similar to a Dual-port FIFO of 128 locations of 16 bits. Indeed, the first data stored is the first data that is read. As it will be seen in the next section, in some cases this memory has to be written and read at the same clock cycle. So it has to be a Dual-port Memory. Its size depends on the iFMAP that in this work is 128 × 128 pixels. Generally the size of the iFMAP impacts in a linear way on the dimensions of the On-chip memory. As Figure 3.19 shows, this memory, in addition to `CLK` and `RESET`, has the following control signals: `Write_Enable` (WE) and `Output_Enable` (OE). They handle respectively writing and reading of the memory and they can be active at the same time since the memory is Dual-Port. These signals arrive from the *Accelerator Controller*.
Below, the structure and the block diagram of the On-chip Memory are shown:

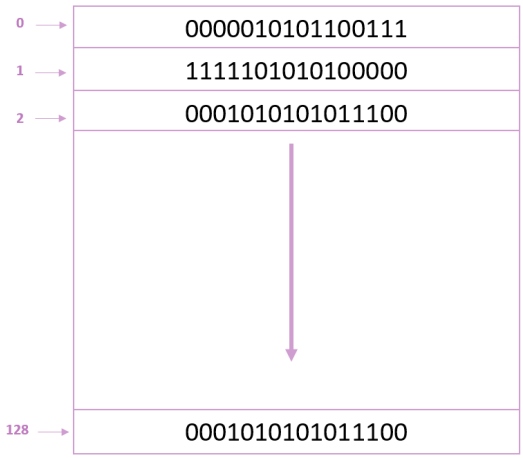

Figure 3.18. On-chip Memory structure

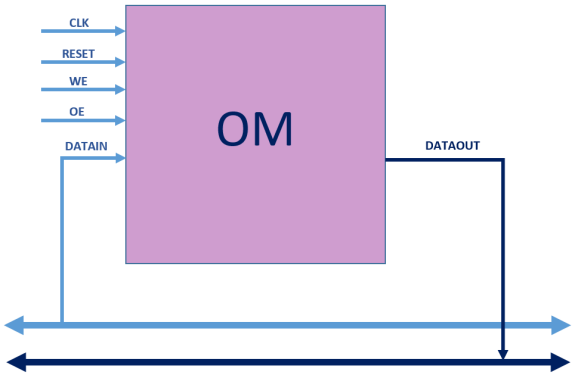

Figure 3.19. Block Diagram of the On-Chip Memory

3.3.3 Accelerator Controller

The Accelerator Controller is the part of the Hardware Accelerator that allows to handle the working of the internal elements of the Datapath and its dataflow. It is the state machine that works in parallel to the Off-chip Controller. Instead of the this latter, the Accelerator Controller is composed by two states: Idle and Processing.
Idle has the same features seen in the Off-chip Controller, where the machine waits that `START` signal is active, and where all internal registers and memories are reset

(internal values are zero). When START = 1, the Machine goes to Processing where dataflow is processed. The following image shows the on-chip state machine:

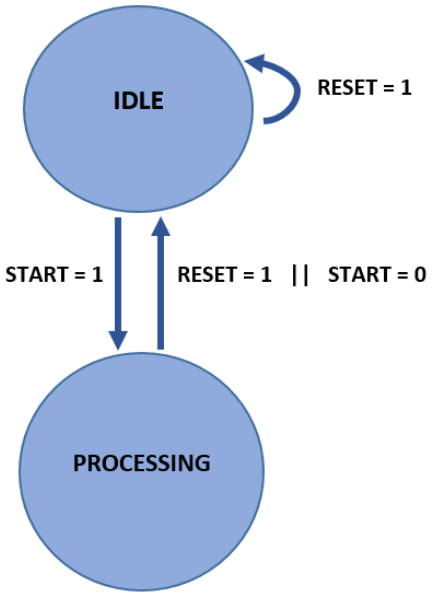

Figure 3.20. State Machine of the Accelerator Controller

In the Processing state, the internal registers are enabled to store new values and WE and OE are activated and deactivated based on some delays to handle writing and reading of On-chip Memories in order to limit the number of accesses inside them. Indeed, for some clock cycles, On-chip Memories have not be written and read. For instance, some products are not useful to obtain the final result since they do not belong to the convolution windows; so the partial result depending on them must not be stored. Moreover, when each column of the iFMAP is sent to the Datapath, 126 partial results have to be stored inside the On-chip Memory. So WE has to be active for at least 126 consecutive clock cycles, then to be deactivated for the next 2 clock cycles and after to be reactivated again for the next 126 cycles and so on. The same system is also used for reading, when data stored inside On-chip Memories need. About that, Dual-port Memories can be written and read at the same time, so for most of the time the two WE and OE signals are active together. All enable signals are deactivated when On-chip Memories have not be used anymore.
Finally, the whole Control Unit is synchronized by CLK and RESET. When RESET = 1 the Machine goes to Idle whatever the state it is in.
What is more, this controller is also used in the next proposed architectures, with some changes in terms of delays.

3.4 Low Power Architectural techniques

As highlighted in [11], On-chip Memories Accesses are the most energetically onerous but the rescheduled dataflow chosen allows to have an optimized architecture that reduces the number of accesses. The second component that causes major power consumption is the PE and precisely the Multiplication internal to it. Indeed, Multiplication is a very onerous operation both in term of time and power and for this reason the architecture, previously described, could be optimized furtherly. First, the previous architecture executes all possible operations to perform the convolution operation. A possible optimization is to avoid unnecessary operations which are often performed the same. For instance, activations and weights can be analyzed to understand which operations can be skipped and to create low power architectures. These choices can increase latency and throughput, but they are drawbacks that can be easily compensated. However, the worsening would be minimal, and therefore negligible in the face of low power improvements.
This thesis resumes low power techniques that have already been used in past works [9] [10], even if they have been applied in array structures; it also shows new techniques to create new hybrid architectures. For this reason, in the next sections, 4 changes applied to the starting architecture are presented: *Zero Skipping Architecture*, *Equal Weights Skipping Architecture*, *Approximation Skipping Architecture* and *Hybrid Equal Weights and Approximation Skipping Architecture*.

3.4.1 Zero Skipping Architecture

The Zero Skip is the first technique that can be applied to starting architecture to decrease its power consumption. It takes inspiration by ZeNA [9], even if it is applied with an original way. It allows to skip ineffectual operations caused by both zero weights and zero activations to reduce energy consumption of convolutional layers. When in a convolution window there are multiplications between zeros and no-zeros, these operations are unnecessary and can be skipped by obtaining the correct result the same. Indeed, by analyzing the common Neural Networks both filters and activations can be zero. Especially the last ones, since the ReLUs are often many, are often null in the subsequent layers of the Network. For this reason, a new block is necessary inside the architecture. Its name is *Zero Input Recognizer* (ZIR). It checks if at least one of the inputs to be multiplied is zero. In the positive case, the PE, that has to perform that operation, does not execute the multiplication but only the sum with the partial sum, as said before. What is more, also PE must be modified, because the multiplication has to be controlled.
The following image shows the Block Diagram of Zero Skipping Architecture:

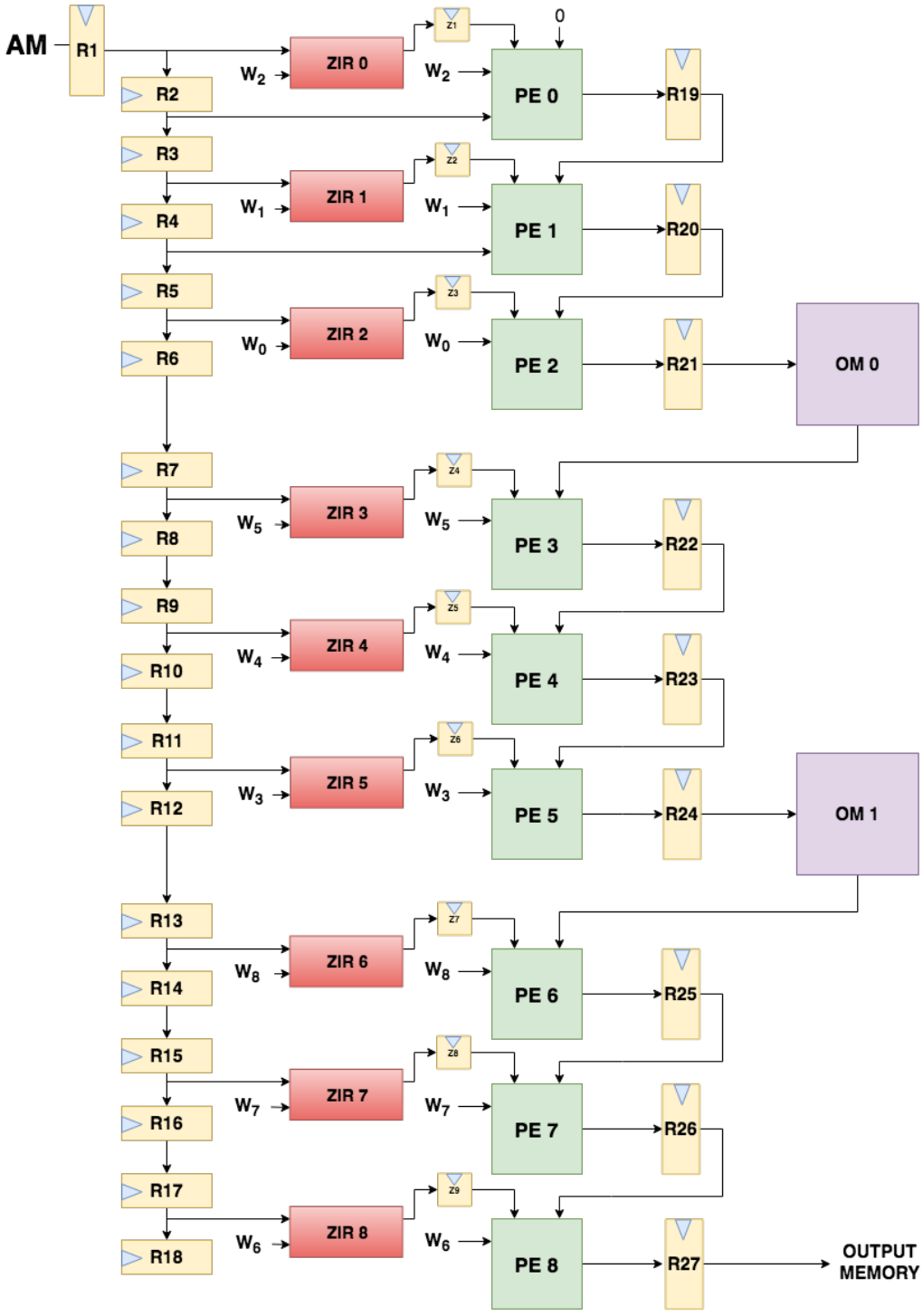

Figure 3.21. Block Diagram of Zero Skipping Architecture

As it can be seen, 9 ZIRs are inserted to recognize if the weights or activations are zero. Its generated flag is pipelined by means of a 1-bit register before to reach the respective PE. So 9 1-bit registers are furtherly necessary. However, except for the changes of PEs, described below, the architecture remains the same.
Now the block of ZIR and the new PE are described in detail.

Zero Input Recognizer

The Zero Input Recognizer (ZIR) receives activation and weight as inputs and analyzes them. There are two equal units that set their output to zero when the input is zero. The final "and" port sets to zero a `Zero Flag` if at least one between activation and weight is zero. This system provides a prevision of the null result of a multiplication. This block is used to avoid a unnecessary multiplication. For this reason, in the architecture it is inserted before each PE to analyze its inputs before performing multiplication.
The following image shows the Block Diagram of ZIR:

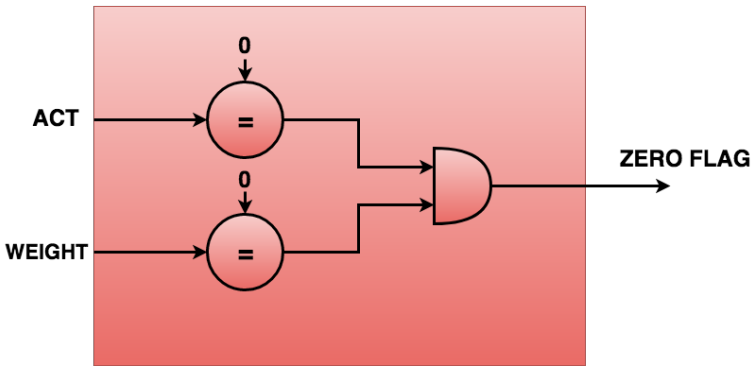

Figure 3.22. Block Diagram of ZIR

Modified Processing Element

In the Zero Skipping Architecture and in the next proposed architectures, the PE is modified because it must allow to skip the multiplication when the `Zero Flag`, generated by ZIR, is 0. The new PE has 4 inputs and an output. Activations and Weights are stored inside a register before arriving at the multiplier. These registers are enabled by the `Zero Flag` only if both activation and weight are not zero (`Zero Flag` is 1); otherwise the new inputs are not loaded (`Zero Flag` is 0). Since the multiplier, which is made up of combinational logic, consumes only when its inputs

change, the use of these registers is a system to skip the operation when it is unnecessary. So a multiplexer needs to select to 0 the result of the multiplier when it is not executed. This multiplexer is selected by `Zero Flag` that is delayed by means of a 1-bit register. This PE has a critical path slightly slower but it is a drawback that can be accepted. The following picture shows the Block Diagram of modified PE where the red arrows indicate the path of the `Zero Flag`, that plays the role of `Enable` of registers and selects the multiplexer.

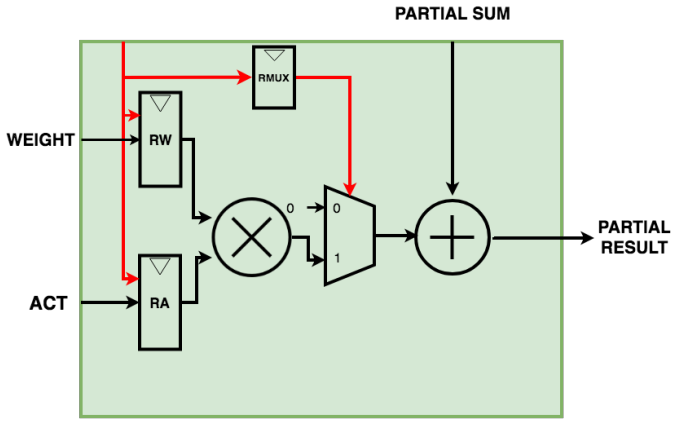

Figure 3.23. Block Diagram of modified PE

3.4.2 Equal Weights Skipping Architecture

The second low power proposed technique consists of an analysis of the weights. In a convolution window there is a sum of products between the activations and weights. For instance, in this thesis each convolution window is 3×3 (it depends on kernel) and each group of PEs must compute 3 multiplications and 3 sums. When two consecutive weights are equal a multiplication can be skipped because it can be replaced by a sum, that is less onerous. For instance, in the following image there is an example where in a window of convolution two consecutive weights are equal:

$W_2 = W_1$

W_0	W_3	W_6
W_1	W_4	W_7
W_2	W_5	W_8

A_0	A_{128}	A_{256}	A_{384}	A_{512}
A_1	A_{129}	A_{257}	A_{385}	A_{513}
A_2	A_{130}	A_{258}	A_{386}	A_{514}
A_3	A_{131}	A_{259}	A_{387}	A_{515}
A_4	A_{132}	A_{260}	A_{388}	A_{516}
A_5	A_{133}	A_{261}	A_{389}	A_{517}
A_6	A_{134}	A_{262}	A_{390}	A_{518}

Figure 3.24. Convolution Window with $W_2 = W_1$

Generally, the highlighted column of the window of convolution is managed in this way:

$$PartialResult = A_2W_2 + A_1W_1 + A_0W_0 \tag{3.1}$$

But with $W_2 = W_1$ it can be described in this way:

$$PartialResult = (A_2 + A_1)W_2 + A_0W_0 \tag{3.2}$$

This expression obtains the same result but an unnecessary multiplication is skipped. As it can be seen, this mathematical simplification can be executed independently by the value of W_0. But, for instance another simplification can be applied when $W_1 = W_0$ and $W_2 \neq W_1$. In this case the equation can be described:

$$PartialResult = A_2W_2 + (A_1 + A_0)W_1 \tag{3.3}$$

So this technique can be applied by modifying the starting architecture. A new block *Equal Weights Recognizer* (EWR) has to be inserted to verify that two consecutive weights in each group of PEs are equal. At the same time this block has to execute the sum between the two consecutive activations to create a possible activation input of the PE. The correct input of each PE is generated by this block depending on the comparison between weights.
Some combinational logic is necessary to enable the multiplication of each PE, with the same mechanism said before. The first PE of each group is always enabled; the others are controlled by the comparison between weights. The *Stay* approach for weights is a good system to avoid that this system is onerous in term of power, because weights are loaded in the Datapath once and the energy consumption of their comparison is irrelevant.
The following image shows the Block Diagram of Equal Weights Skipping Architecture:

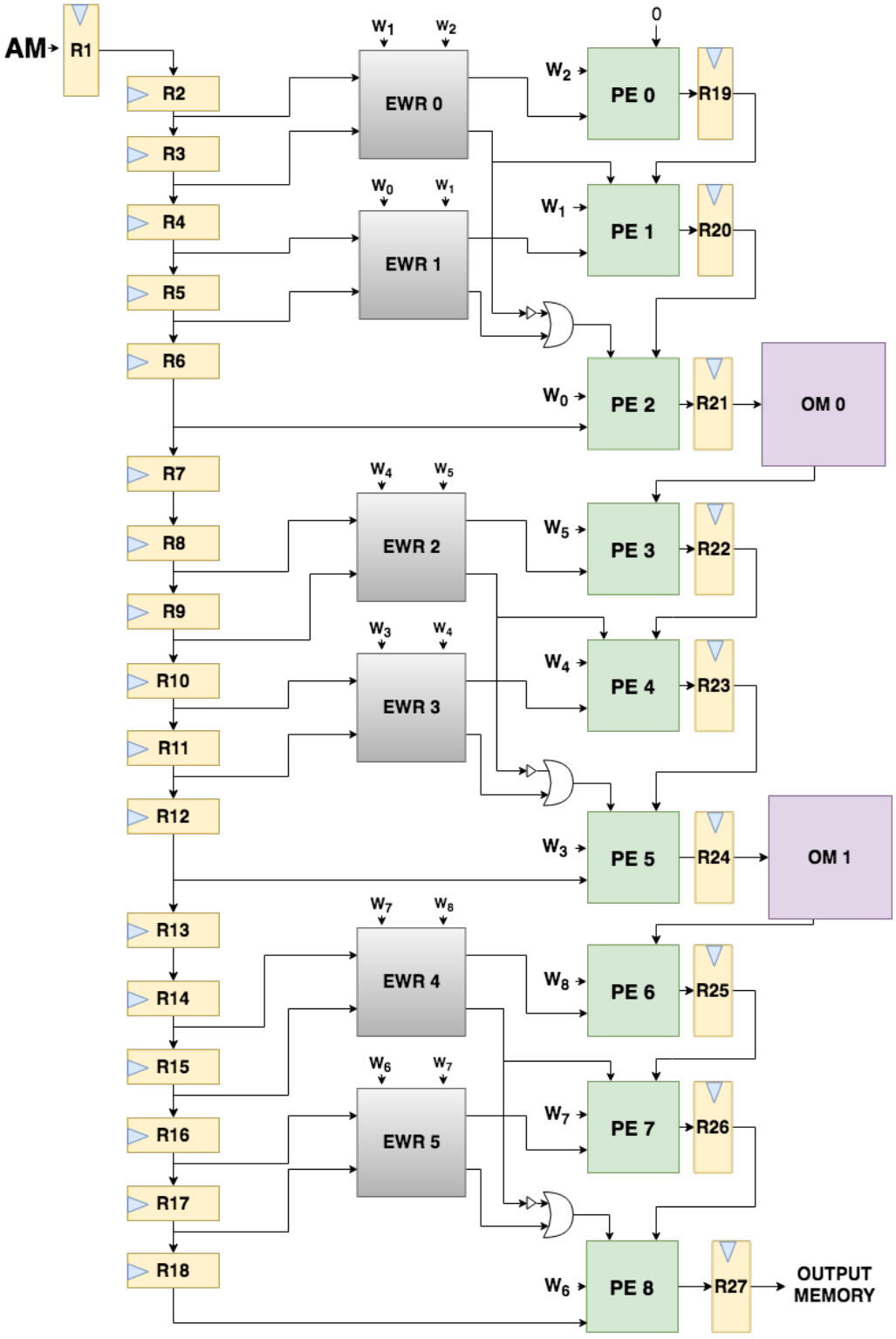

Figure 3.25. Block Diagram of Equal Weights Skipping Architecture

As it can be seen, 6 EWRs are inserted to verify if two consecutive weights are equal in each group of PEs. They generate a Flag that by means of a combinational logic controls the last two PEs of the group. So 4 "or" and 4 "not" ports are necessary to handle all possible cases. However, most PEs are similar to ones of the Zero Skipping Architecture, except for the sign extension of the activation which is executed only in PE2 ,PE5 and PE8. Then the architecture remains the same of the starting one. Now the block of EWR is described in detail.

Equal Weights Recognizer

The Equal Weights Recognizer (EWR) receives two consecutive activations and two consecutive weights as inputs and generates two outputs (`Zero Flag` and `Act`). The `Zero Flag` is generated by an equal unit that sets its output to zero when the two consecutive weights are equal. At the same time an adder computes the sum between the two consecutive activations. A multiplexer, controlled by the generated `Zero Flag`, selects the current activation or the sum between it and the previous activation loaded in the Datapath. The output of the multiplexer is on 10 bits in order it can handle the possible overflow of the sum. So the comparison between the weights decides what to send to PE. However, this block is used to avoid an unnecessary multiplication, even if an extra sum is performed. It is an irrelevant aspect if the filter contains many equal weights and many multiplications are skipped, otherwise the computational weight can make this architecture heavier than the starting one without providing the desired results. What is more, as for ZIR, it is inserted in the architecture before each PE to analyze its inputs before performing multiplication.
The following image shows the Block Diagram of EWR:

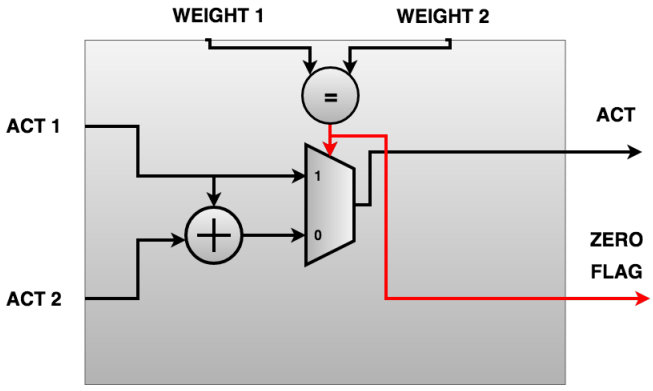

Figure 3.26. Block Diagram of EWR

3.4.3 Approximation Skipping Architecture

The third low power proposed technique consists of an approximation of the convolution operation. In a convolution window some products could be irrelevant to obtain the final result, so they could be skipped. Some previous works, such as [10], have already presented a system to skip a multiplication that produces a too low result. So this thesis starts from Huan's work and applies an approximation to the starting architecture, previously presented. The idea is to understand when a product activation-weight can be negligible.
A rule that can be followed is that of *Leading Zeros*. Indeed when a product is computed, the number of *Leading Zeros* of the result is almost always the sum between the number of Leading Zeros of the operands. So a result could be discarded if its number of the *Leading Zeros* is less than a fixed threshold. Two examples of a 10×6 multiplication are shown in the following figures:

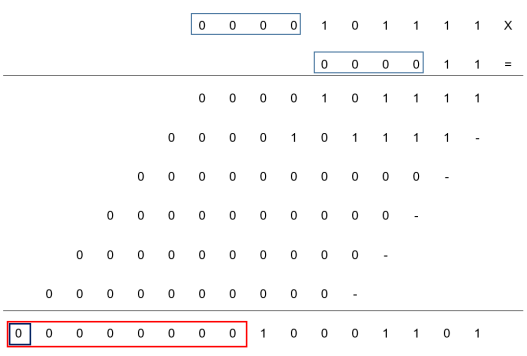

Figure 3.27. Case 1. Sign bit of the result is included in the total leading zeros of the operands

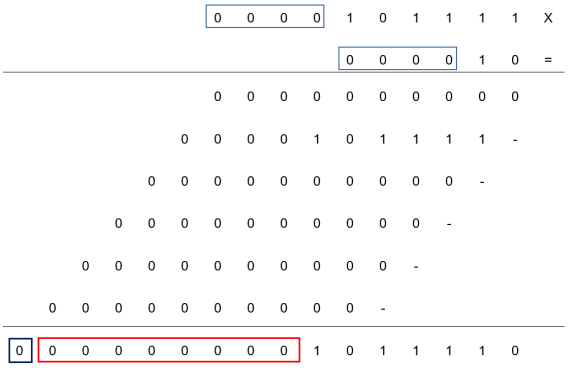

Figure 3.28. Case 2. Sign bit of the result is not included in the total leading zeros of the operands

As it can be seen, the two operands have 4 leading zeros (including the sign bit). The images shows two cases: the right image shows that the result has 8 leading zeros (the sum of the leading zeros of the operands), and the position of the sign bit is determined by the total leading zeros of the operands; on the left, instead, the obtained result has 9 leading zeros, so the position of the sign bit is determined by the total leading zeros of operands + 1.
The proposed architecture exploits the second case for the approximation to apply. Indeed, since the approximations must be limited to maintain the correct functioning of the Network, the types of multiplication that could be skipped are as the ones of the second case, where the sign bit is external to the total number of leading zeros. So in this work, the fixed threshold is always compared to the total number of leading zeros, by excluding the sign bit of the result.
The idea is to send a threshold to the Datapath, besides weights and activations,

that indicates the maximum number of leading zeros that each partial product can have (by excluding its sign bit). So one further block, named *Leading Zeros Recognizer* (LZR) is needed to count the leading zeros of operands and to compare their sum with given threshold. If this sum is larger or equal than the threshold the multiplication will can be skipped (approximated to 0), otherwise the multiplication will be computed. This system is an improvement of the Zero Skipping Architecture, because the multiplications that can be skipped are both those where at least one operand is zero and those where the partial product is a low value.
What is more, this system also can be applied to partial product negative, because the rule of leading zeros can be still applied to its absolute value. Therefore if at least one of operands, or both are negative, LZR can count the leading zeros of its 2-complement. Thus, this system allows to skip all the multiplications that product a result with a range nearest to zero (negative and positive). However, it can not involve all the intervals. In fact, the given threshold could indicate multiple quantized integer values. For example, a threshold of 14 indicates 0, 1, -1; a threshold equal to 13 indicates 0,1,2,3, -1, -2, -3 and so on. So the various ranges of approximation depend on the powers of 2. Then:

- Threshold = 14 indicates a range [-1,1];
- Threshold = 13 indicates a range [-3,3];
- Threshold = 12 indicates a range [-7,7];
- And so on...

This aspect makes little flexible this architecture, if a major precision is required. But it is a good way to obtain a further low power optimization. The following image shows the Block Diagram of the Approximation Skipping Architecture:

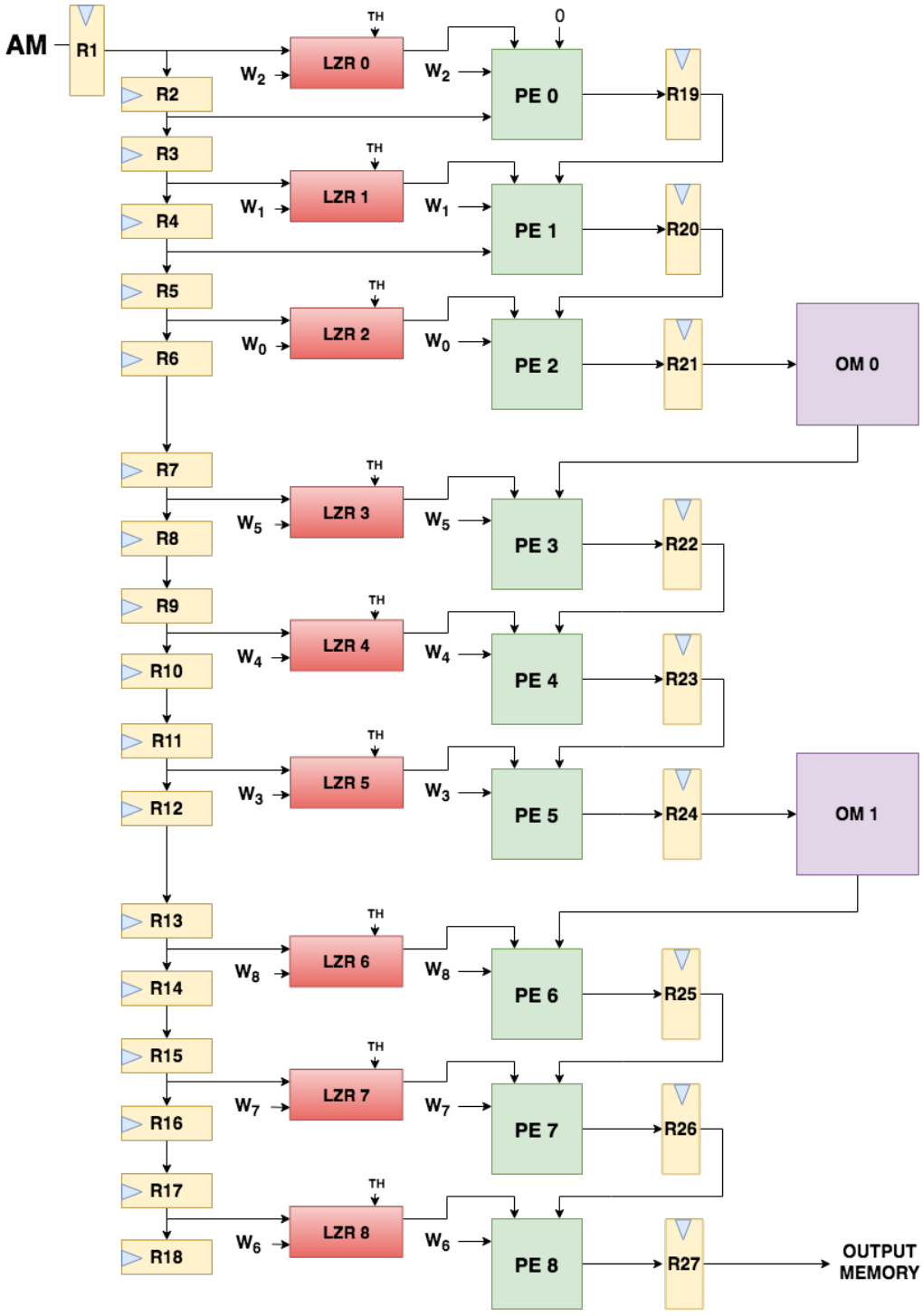

Figure 3.29. Block Diagram of Approximation Skipping Architecture

As it can be seen, 9 LZRs are inserted (3 for group) to provide a prevision of each partial product result. They generate a Flag that controls each PE of the group. The threshold is equal for each LZR, and it is loaded in the Datapath with a *Stay* approach like for the weights. So Weights Memory could have an extra register to store also thresholds, since they are represented on 6-bits such as the weights. What is more, even if this architecture is similar to the Zero Skipping's one, no 1-bit register is necessary to pipeline the `Zero Flag`. It is due to the presence of a register in the main path inside the LZR block. However, except for the changes of PEs, described before, the architecture remains the same of the starting one.
Now the LZR block is described:

Leading Zeros Recognizer

The Leading Zeros Recognizer (LZR) receives the two operands of the mutliplication that must be forecasted (activation and weight) and the threshold of leading zeros as inputs and generates a `Zero Flag`. This block computes the 2-complement of the activation and of the weight; then it sends them or the not complemented input to a *Leading Zeros Counter*, based on their sign (the sign of the activation or of its 2-complement is also extended). These counters have been implemented with a carry-lookahead Leading Zero Counting, proposed by Giorgos [27], which is used for the normalization of the floating point operations. In this thesis, it is used to obtain a fast low power prediction and approximation of the multiplication. The following image shows the adopted logic circuit to count leading zeros of a 16-bit input:

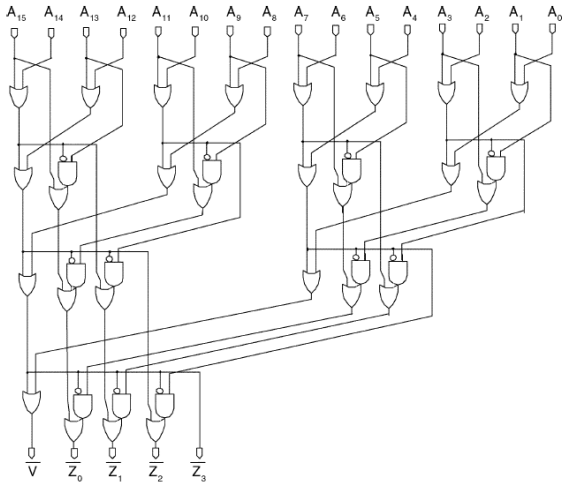

Figure 3.30. Schematic of the leading zero counter [27]

This counter adds dummy zeros to create a 16 bit data (6 and 10 zeros for activations and weights respectively) and obtains the number of leading zeros on 4-bits. The flag V is active when the input is 0, but it does not propagate itself in the architecture. This combinational structure could generate some glitches, so one register is necessary at the end of the structure. The leading zeros of the operands are added; the result is on 6-bits (to avoid overflow) and it is compared to threshold. The `Zero Flag` is generated by the comparator that sets its output to zero when the total leading zeros are larger or equal than threshold; otherwise it is set to 1. So the comparison between total leading zeros and threshold decides what to send to PE. However, this block is used to avoid an unnecessary multiplication, if an approximation has to be applied. What is more, as for ZIR and EWR, it is inserted in the architecture before each PE to analyze its inputs before performing multiplication. The following image shows the Block Diagram of LZR:

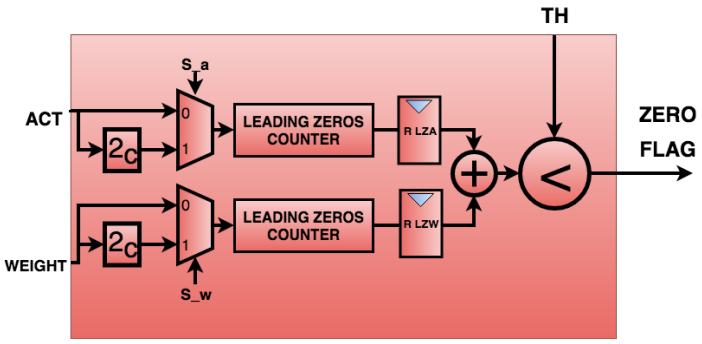

Figure 3.31. Block Diagram of LZR

3.4.4 Hybrid Equal Weights and Approximation Skipping Architecture

The last low power proposed architecture is an hybrid architecture between Equal Weights Skipping Architecture and Approximation Skipping Architecture. It links the advantages of both architectures in order to skip more multiplications. In this case, the combinatorial logic is modified thanks to the addition of 6 "and" ports to handle the flag generated by the 5 EWRs and the 9 LZRs. What is more, more registers need to synchronize the data arriving to PEs and LZR0, LZR1, LZR3, LZR4, LZR6 and LZR7 receive the activation inputs on 10 bits.
The following image shows the Block Diagram of this Hybrid Skipping Architecture:

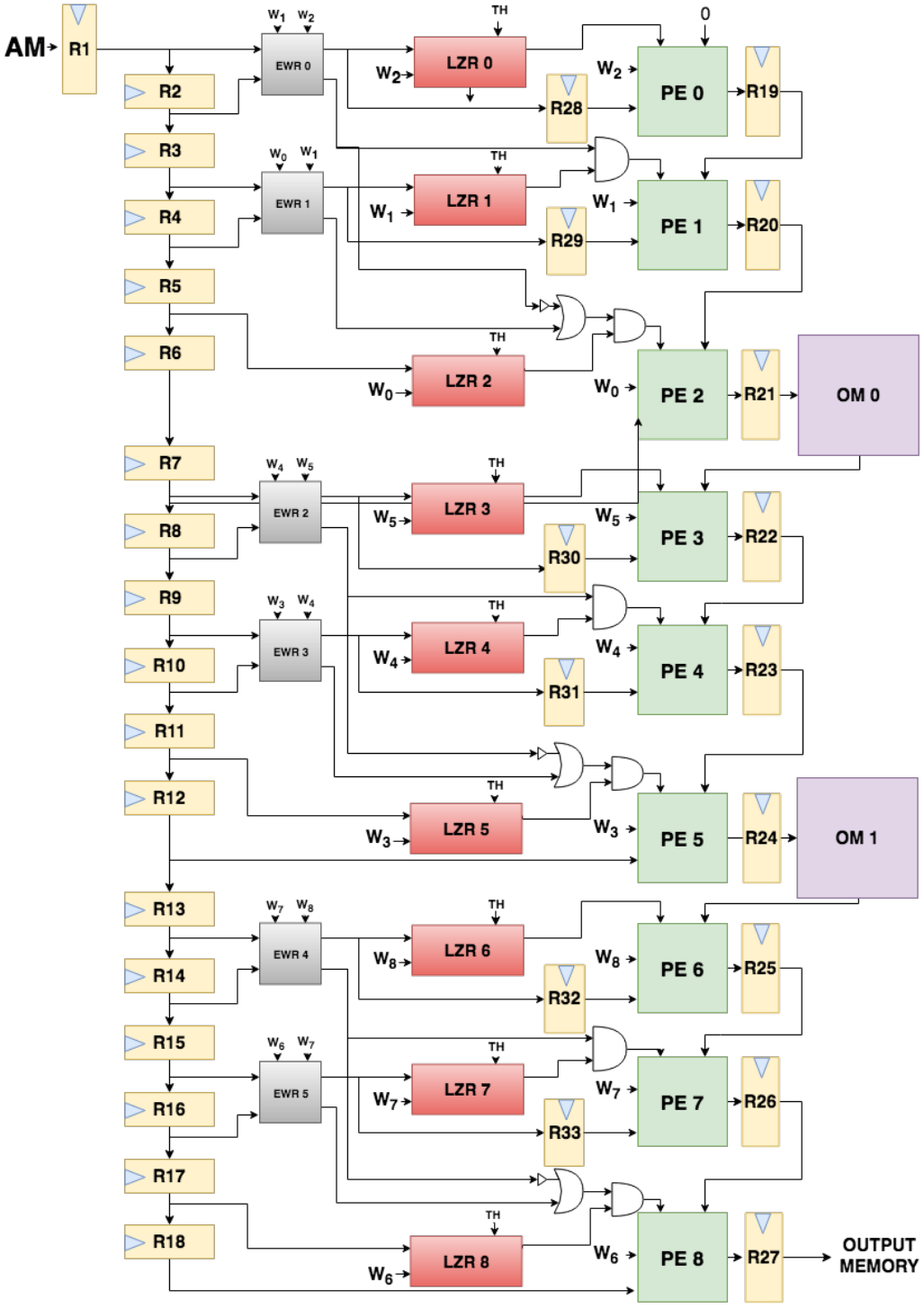

Figure 3.32. Block Diagram of the Datapath

Chapter 4

Simulations

The simulation phase is one of the most important during a design implementation since it allows to analyze the design flow step by step. The Hardware simulator that will be used in this thesis is *ModelSim* of the Mentor Graphics. In the following sections the simulations of the proposed architectures will briefly analyze by showing the main feature of their timing diagram.
What is more, the various testbenches have been written in Verilog while the entire architectures are written in VHDL. Thanks to *ModelSim*, VHDL and Verilog blocks can be mixed in a single design.
Finally, a *Matlab* script, necessary to generate inputs (as txt files by ImageNet dataset) to send to architecture, will be described in order that its simulations are reported, comparing the results with an exact *Matlab* model. So the behavior of the accelerator will be verified.

4.1 Modelsim Simulations

The *Modelsim* Simulation has been done by creating two further VHDL blocks: **Simulation** and **CLK Generator**. The first manages the start of the simulation by sending the `START` signal to each block. Then, when the simulation is ended, it receives an `END_SIM` signal generated by the Output Memory (this signal will result necessary in the computing of the switching activity for the power consumption, described in the next chapter). So it sets to 1 an `END` signal that is sent to CLK Generator. This latter always generates a `CLK` signal until the `END` signal arrives. Moreover, a `RESET` signal is generated to initialize the entire architecture. The following image shows the Block Diagram of the simulation chosen:

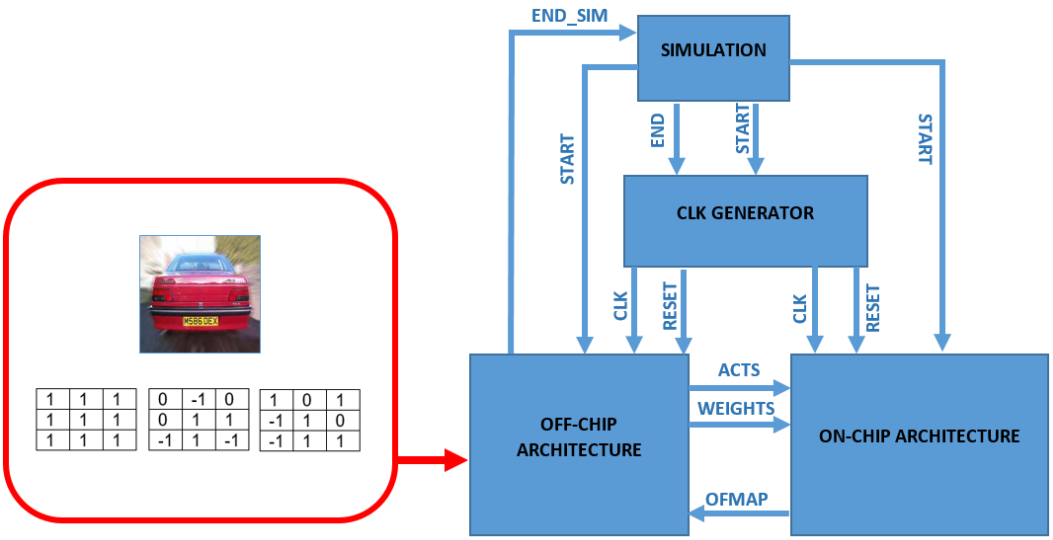

Figure 4.1. Block Diagram of the simulation

The inputs are sent to architecture by means of the txt files generated by *Matlab*. Now the various architectures can be simulated and their timing diagrams can be analyzed.

4.1.1 Starting Architecture

The Starting Architecture has a latency of 16404 clock cycles. It mainly depends on the size of the input feature map, that is represented on 128x128 = 16384 pixels. So from the last input, inserted into architecture, pass 20 extra clock cycles. They represent the internal latency of the architecture due to the forwarding registers, the partial sum accumulated inside registers and the partial result accumulated inside the on-chip memories.
What is more, the three channels work in parallel and their result are computed in the same time. In the following timing diagrams the entire simulation and a close look of the start and of the end of the simulation are shown:

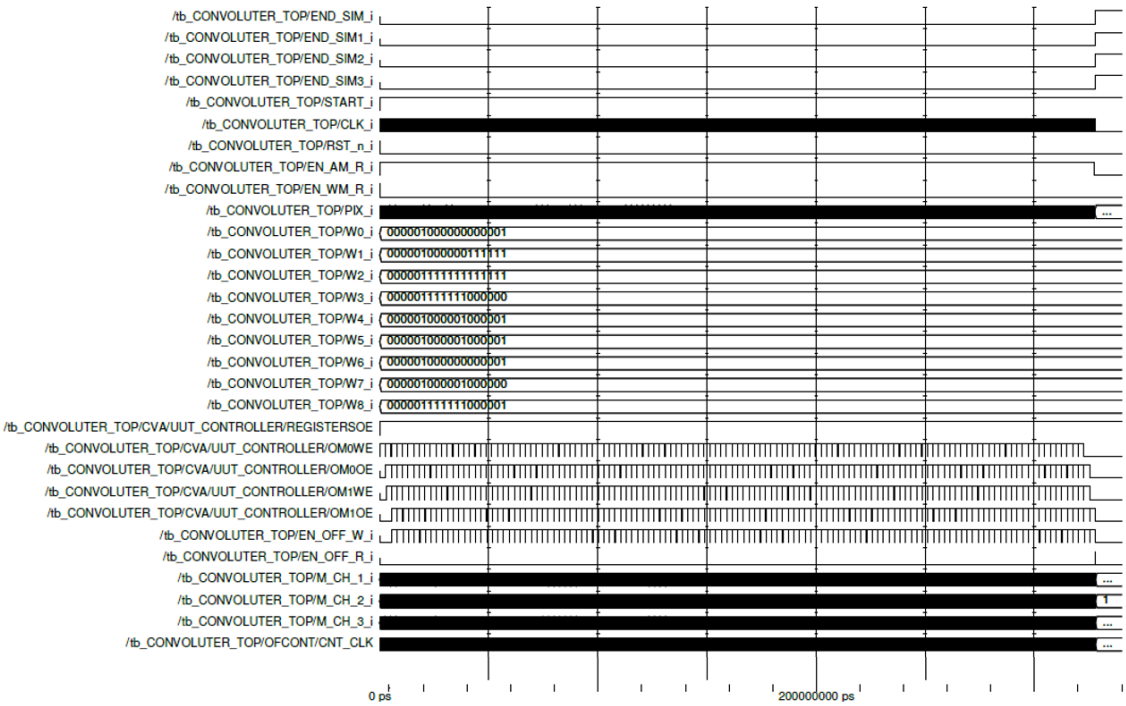

Figure 4.2. Full Simulation of the Starting Architecture.

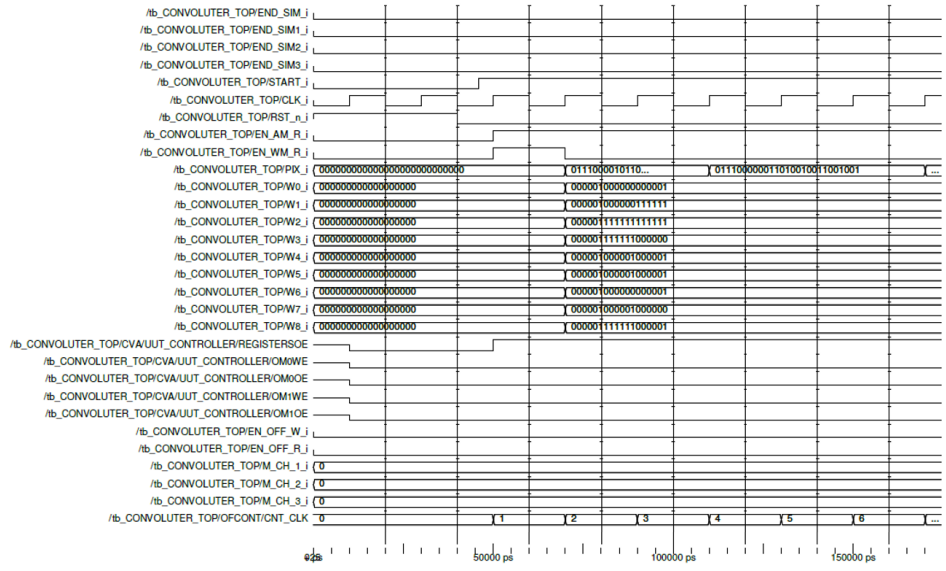

Figure 4.3. End of the Simulation.

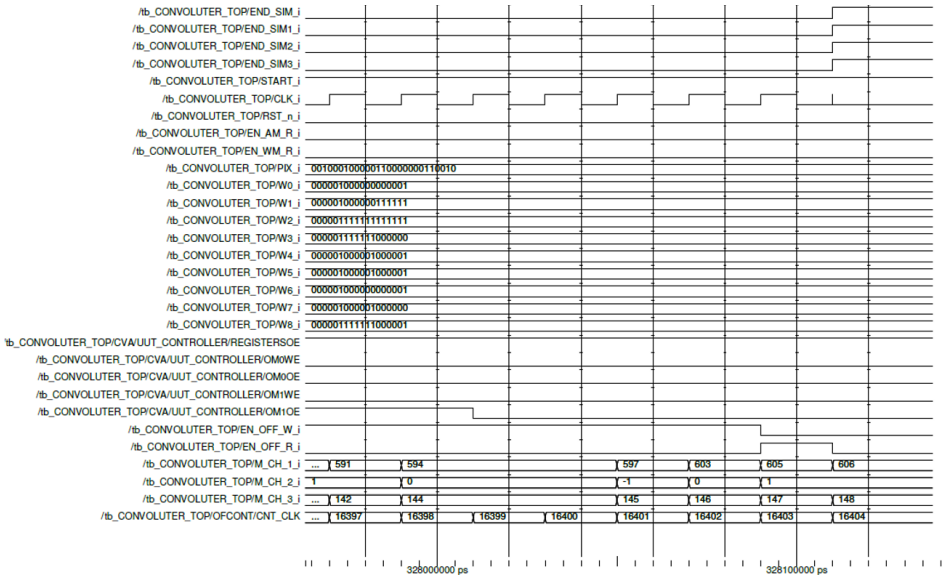

Figure 4.4. End of the Simulation.

As it can be seen, the *control flags* generated by the Accelerator Controller create the correct internal timing of the architecture. The initial part of the simulation shows the activation of the `RESET`, necessary to initialize the machine. The `START` signal is active when the `RESET` goes down, at the next rising edge of the clock the loading of the inputs inside the architecture starts. At the end of the simulation the writing of the results into txt files is executed and the `CLK` is blocked; after that, a script of *Matlab* will be able to read the results and to provide information about all the image processing.

4.1.2 Low Power Architectures

The other architectures described in this thesis show a worse latency (16405 clock cycles) since an additional register is inserted in the main path of the architecture. Indeed, the *control flags* generated by Accelerator Controller are delayed with respect the previous architecture.
In the following timing diagram the a close look made at the end of the simulation of the Approximation Skipping Architecture is shown:

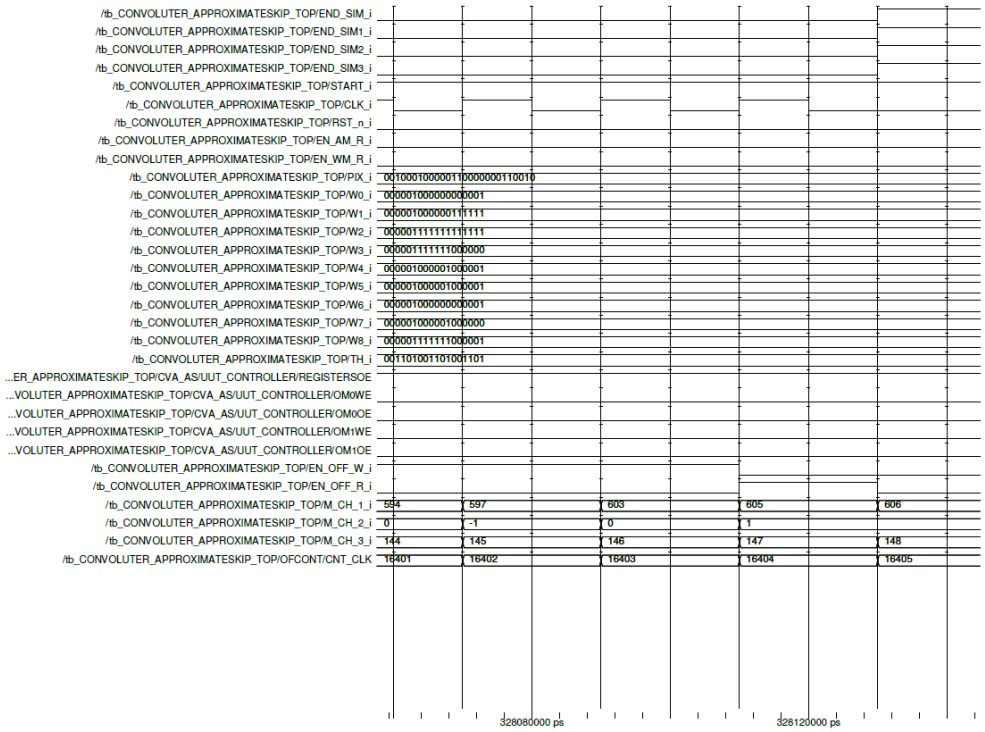

Figure 4.5. End of the Simulation of the Approximation Skipping Architecture.

4.2 Matlab Processing

An important part of this thesis has been the *Matlab* processing of the images. In the Chapter 6, *Matlab* has been used to create AlexNet model to validate Hardware implementation, while in this section the *Matlab* has been used to generate an exact software model of the various implemented architectures in order to verify the results that the ASIC designs generate.
This script can be used to analyze any images but in this work the processed images belong to the ImageNet dataset.
The image processing follows some steps. First, a script allows to read an image of the dataset; then it allows to resized the image in 128×128 pixels; so, the resolution of the image has been reduced, but this step is necessary to adapt the input to the proposed architecture. This choice allows to have low resolution but at the same time smaller On-chip memories inside the Datapath. The following images show the loss of resolution to pass by 227×300 to 128×128 pixels:

Figure 4.6. (a) The initial image with maximum resolution 227 × 300. (b) The input for the architecture with low resolution 128 × 128.

Then three txt files that contain the hexadecimal values of the pixels for each RGB channel have been generated. The following images show the different inputs sent to each channel with grayscale representation:

Figure 4.7. (a) Grayscale image of Red channel; (b) Grayscale image of Green channel; (c) Grayscale image of Blue channel.

Thus, these txt files and the txt files of the weights are sent to architecture in order to compute the various oFMAPs.
Another script defines the exact model of the various architectures. It applies two types of kernels to the three channels and computes the various oFMAPs. The next figures show the kernels applied in the exact model and in the ASIC architecture:

1	1	1
1	1	1
1	1	1

0	-1	0
0	1	1
-1	1	-1

1	0	1
-1	1	0
-1	1	1

(a)

0	1	0
1	-4	1
0	1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

1	-2	1
-2	4	-2
1	-2	1

(b)

Figure 4.8. (a) Kernel 1 applied to the RGB image; (b) Kernel 2 applied to the RGB image.

In the next step, various oFMAPs are computed and compared with the results of the ASIC structures, by verifying that results are the same. What is more, in the Approximation Skipping Architecture and in the Hybrid Equal Weights Approximation Architecture more ranges of the approximation are used.
Now the obtained oFMAPs are shown in the following images and the reduction of the number of the multipliers is reported in order to verify the different impact of each architecture.

4.2.1 Results of architectures without approximation

The oFMAPs are computed and linked into a single image by means of Matlab (an array concatenation is executed). The following images shows them computed with different kernels:

Figure 4.9. Linked oFMAPs with Kernel 1

Figure 4.10. Linked oFMAPs with Kernel 2

The tables below show the number of multipliers executed:

Number of Multipliers computed by no-approximated architectures with Kernel 1

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Starting	**428652**	142884	142884	142884
Zero Skipping	**(18.5%) 349272**	(0.0%) 142884	(33.3%) 95256	(22.2%) 111132
Equal Weigths Skipping	**(25.9%) 317520**	(33.3%) 95256	(22.2%) 111132	(22.2%) 111132

Number of Multipliers computed by no-approximated architectures with Kernel 2

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Starting	**428652**	142884	142884	142884
Zero Skipping	**(14.8%) 365148**	(0%) 79380	(0.0%) 142884	(0.0%) 142884
Equal Weigths Skipping	**(7.4%) 396900**	(0.0%) 142884	(22.2%) 111132	(0.0%) 142884

As it can be seen, the impact of two filters is different. Indeed, the low power techniques allow to obtain some advantages based on the image and the kernel which are used. For instance, the Kernel 1 allows to reduce the number of multipliers more than Kernel 2 for both architectures. Moreover, the use of Kernel 1 makes the Equal Weights Skipping Architecture better than Zero Skipping one; on the other hand the use of Kernel 2 makes the Zero Skipping Architecture better than Equal Weights Skipping one.

4.2.2 Results of architectures with approximation

For the architectures with approximation, various thresholds have been applied to involve more ranges of approximation.

Range [-1,1] with Threshold = 14

Figure 4.11. Linked oFMAPs of Approximation Skipping Architecture with Kernel 1 and range of approximation [-1:1]

Figure 4.12. Linked oFMAPs of Approximation Skipping Architecture with Kernel 2 and range of approximation [-1:1]

Figure 4.13. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 1 and range of approximation [-1:1]

Figure 4.14. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 2 and range of approximation [-1:1]

Number of Multipliers computed by approximated architectures with Kernel 1

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(18.5%) 349265**	(0.0%)142884	(33.3%) 95256	(22.2%) 111125
Hybrid EW and APP Skipping	**(40.7%) 254013**	(33.3%) 95256	(44.4%) 79380	(44.4%) 79377

Number of Multipliers computed by approximated architectures with Kernel 2

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(14.8%) 365144**	(44.4%)79380	(0.0%) 142884	(0.0%) 142880
Hybrid EW and APP Skipping	**(22.2%) 333392**	(44.4%)79380	(22.2%) 111132	(0.0%) 142880

Range [-3,3] with Threshold = 13

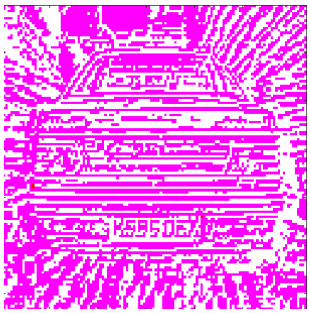

Figure 4.15. Linked oFMAPs of Approximation Skipping Architecture with Kernel 1 and range of approximation [-3:3]

Figure 4.16. Linked oFMAPs of Approximation Skipping Architecture with Kernel 2 and range of approximation [-3:3]

Figure 4.17. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 1 and range of approximation [-3:3]

Figure 4.18. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 2 and range of approximation [-3:3]

Number of Multipliers computed by approximated architectures with Kernel 1

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(18.5%) 349230**	(0.0%)142875	(33.3%) 95244	(22.2%) 111111
Hybrid EW and APP Skipping	**(40.7%) 253996**	(33.3%) 95253	(44.4%) 79372	(44.4%) 79371

Number of Multipliers computed by approximated architectures with Kernel 2

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(14.8%) 365112**	(44.4%)79376	(0.01%) 142868	(0.01%) 142868
Hybrid EW and APP Skipping	**(22.2%) 333368**	(44.4%)79376	(22.2%) 111124	(0.01%) 142868

Range [-7,7] with Threshold = 12

Figure 4.19. Linked oFMAPs of Approximation Skipping Architecture with Kernel 1 and range of approximation [-7:7]

Figure 4.20. Linked oFMAPs of Approximation Skipping Architecture with Kernel 2 and range of approximation [-7:7]

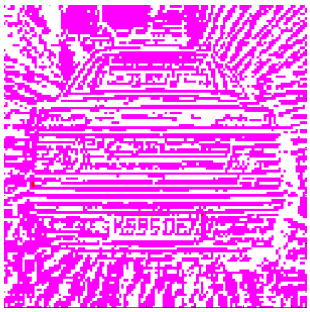

Figure 4.21. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 1 and range of approximation [-7:7]

Figure 4.22. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 2 and range of approximation [-7:7]

Number of Multipliers computed by approximated architectures with Kernel 1

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(18.6%) 349093**	(0.0%)142875	(33.4%) 95142	(22.3%) 111076
Hybrid EW and APP Skipping	**(40.7%) 253996**	(33.3%) 95253	(44.4%) 79304	(44.4%) 79354

Number of Multipliers computed by approximated architectures with Kernel 2

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(14.9%) 364950**	(44.4%)79376	(0.1%) 142732	(0.03%) 142839
Hybrid EW and APP Skipping	**(22.3%) 333271**	(44.4%)79376	(22.3%) 111056	(0.03%) 142839

Range [-15,15] with Threshold = 11

Figure 4.23. Linked oFMAPs of Approximation Skipping Architecture with Kernel 1 and range of approximation [-15:15]

Figure 4.24. Linked oFMAPs of Approximation Skipping Architecture with Kernel 2 and range of approximation [-15:15]

Figure 4.25. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 1 and range of approximation [-15:15]

Figure 4.26. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 2 and range of approximation [-15:15]

Number of Multipliers computed by approximated architectures with Kernel 1

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(19.3%) 346001**	(0.1%)142776	(35.3%) 92394	(22.4%) 110831
Hybrid EW and APP Skipping	**(41.2%) 251926**	(33.4%) 95220	(45.8%) 77459	(44.5%) 79247

Number of Multipliers computed by approximated architectures with Kernel 2

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(15.8%) 361076**	(44.5%)79331	(2.7%) 139068	(0.14%) 142677
Hybrid EW and APP Skipping	**(22.7%) 331206**	(44.5%)79331	(23.6%) 109198	(0.14%) 142677

Range [-31,31] with Threshold = 10

Figure 4.27. Linked oFMAPs of Approximation Skipping Architecture with Kernel 1 and range of approximation [-31:31]

Figure 4.28. Linked oFMAPs of Approximation Skipping Architecture with Kernel 2 and range of approximation [-31:31]

Figure 4.29. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 1 and range of approximation [-31:31]

Figure 4.30. Linked oFMAPs of Hybrid Equal Weights and Approximation Skipping Architecture with Kernel 2 and range of approximation [-31:31]

Number of Multipliers computed by approximated architectures with Kernel 1

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(24.3%) 324293**	(0.8%)141705	(45.7%) 77532	(26.5%) 105056
Hybrid EW and APP Skipping	**(44.3%) 238743**	(33.6%) 94848	(53.0%) 67143	(46.3%) 76752

Number of Multipliers computed by approximated architectures with Kernel 2

Architectures	Total Multipliers	Multipliers Channel 1	Multipliers Channel 2	Multipliers Channel 3
Approximation Skipping	**(21.3%) 337337**	(44.8%)78855	(16.5%) 119250	(2.55%) 139232
Hybrid EW and APP Skipping	**(26.1%) 316559**	(44.8%)78855	(31.1%) 98472	(2.55%) 139232

As it can be seen, when the threshold increases, the number of skipped multipliers increases too. But the errors also increase, so the thresholds can not be chosen above a certain value. However, when the thresholds are high, the skipped multipliers do not change the oFMAPs too much, because the concatenated results seem almost identical to the oFMAPs obtained without approximation. Instead, lower thresholds cause many errors in the oFMAPs, even if they exploit a more low power architecture. Then, these architectures allow to have the major reduction of computed multipliers than the no-approximated architectures.
In the Chapter 6, the choice of thresholds is analyzed based on the correct functioning of already tested Neural Networks, such as AlexNet.

-

Chapter 5

Synthesis

Synthesis is a necessary task to provide information about area, speed and power of the proposed Hardware accelerators. It is composed by many phases. For this reason, many commands have been studied to manage the synthesis phase in the correct way. The software used to do that is *Synopsys Design Compiler* by using a particular library that is **UMC 65 nm** which is provided by *Politecnico of Turin*. In the next sections, the basic flow of Synopsis will be described and the netlist of each proposed architecture will be generated, by comparing their results in term of area, speed and power.

5.1 Design Compiler Flow

The Design Compiler is a tool that exploits the Synopsys synthesis. It allows to perform the logic synthesis by optimizing HDL designs into technology-dependent, gate-level designs. What is more, it supports various hierarchical designs to represent really the hardware description that HDL design provides. So the logic synthesis is very important because it is a necessary step to optimize the hardware design (both combinatorial and sequential cells) by providing information about speed, area, and power.
The logical synthesis process follows a specific flow in order to generate the netlist that will be simulated after to obtain power information. The first analysis that allows to start the logical synthesis is shown in the following scheme (in this thesis not all the commands shown have been used):

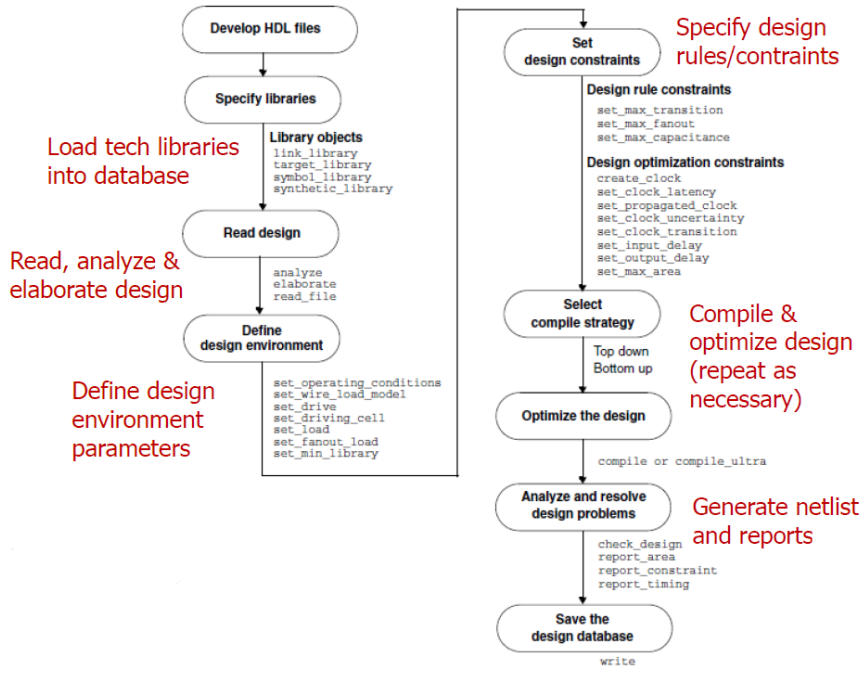

Figure 5.1. Synthesis stages and commands [28]

This thesis uses the following main tasks: reading VHDL source files and defining of the technology library (umc 65 nm), applying constraints (only on the clock), compiling and optimizing the design, save the results and generate the netlist.
After, the next step will be the power analysis that is useful to compare the power consumption of each architecture by giving them the same input. To do that, an estimation of the switching activity of each netlist will be done by using both *Modelsim* and *Synopsis Design Compiler*.
Now the results obtained by synthesis will be reported, by analyzing before speed and area. The power will be analyzed subsequently.

5.2 Logic Synthesis

The Design Compiler flow has been followed step by step by using some scripts containing the commands to use. Because Design Compiler can not synthesize the memories directly (unless the use of a *memory generator*), a *Top Entity* of each architecture has been created without incorporating on-chip memories. This choice not only simplifies the synthesis but it allows to obtain true results as any synthesis of the memories, without *memory generator*, would not provide correct information.

So the VHDL project has been modified and the VHDL source files have been read and analyzed to elaborate the synthesized architectures.
The following pictures show the schematic of the Top Entity and the Datapath of the Hardware accelerator of the starting architecture (it has more input and output to communicate with the on-chip memories that are not included):

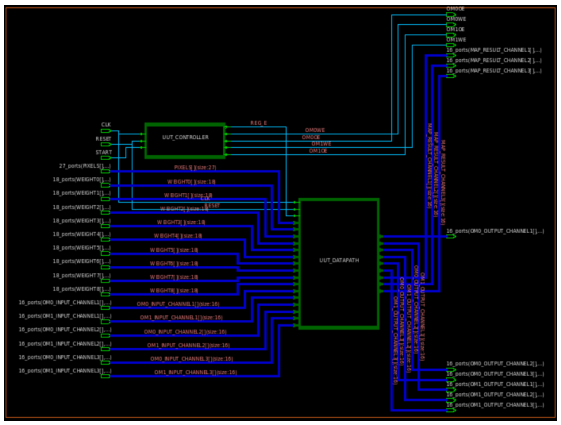

Figure 5.2. Schematic of the Top Entity of the starting architecture of the Hardware Accelerator.

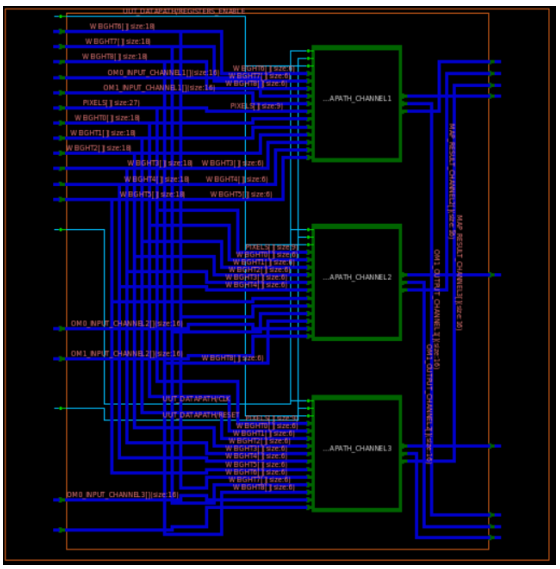

Figure 5.3. Schematic of the Datapath of the starting architecture of the Hardware Accelerator.

In the Design Compiler there is the possibility to apply constraints to the design. This thesis shows how only some constraints of the clock have been applied and the load of each output has been set by choosing the input capacitance of a buffer available in the UMC 65 nm technology. Then, the compilation allows to optimize the entire architectures. However the clock period that is given to each design can be modified in order to find the maximum one that does not violate the constraints. The following table shows the maximum clock, the respective frequency and the area of each architecture:

Architecture	T_{clock}[ns]	F_{max}[MHz]	Area[μm^2]
Starting	1.45	689.65	40948.20
Zero Skipping	1.53	653.59	41575.76
Equal Weights Skipping	1.53	653.59	42182.64
Approximation Skipping	1.53	653.59	47296.08
Hybrid EW and APP Skipping	1.53	653.59	51039.68

As it can be noted, the modified architectures have a lower maximum frequency than the starting one; even if the difference is very subtle. At the same time, the area have increased because more registers and more logical elements are used. However, the Hybrid architecture has the largest area since it is the most complex architecture.
The next step is to generate the netlists of each architecture. A netlist is a description of the connectivity of an electronic circuit. In its simplest form, a netlist consists of a list of the electronic components in a circuit and a list of the nodes they are connected to. [29]
By following a specific flow of *Design Compiler*, each netlist has been generated. What is more, also a *sdf* file has been generated. *Sdf* file contains the delay value of each timing arc corresponding to each cell in the netlist.[30] These delay values are necessary to obtain the correct real-time beahavior of the netlist. This file has been generated based on the library of technology that has been used.
So the netlist of each architecture has been simulated by means of *Modelsim* to verify its correct behavior. This step is always mandatory since even if the architectures directly simulated in *Modelsim* seem to work correctly, after the logic synthesis they could have undefined nets. The following timing diagram shows the simulation of the netlist of the starting architecture (only it has been reported but all produced netlists have been simulated):

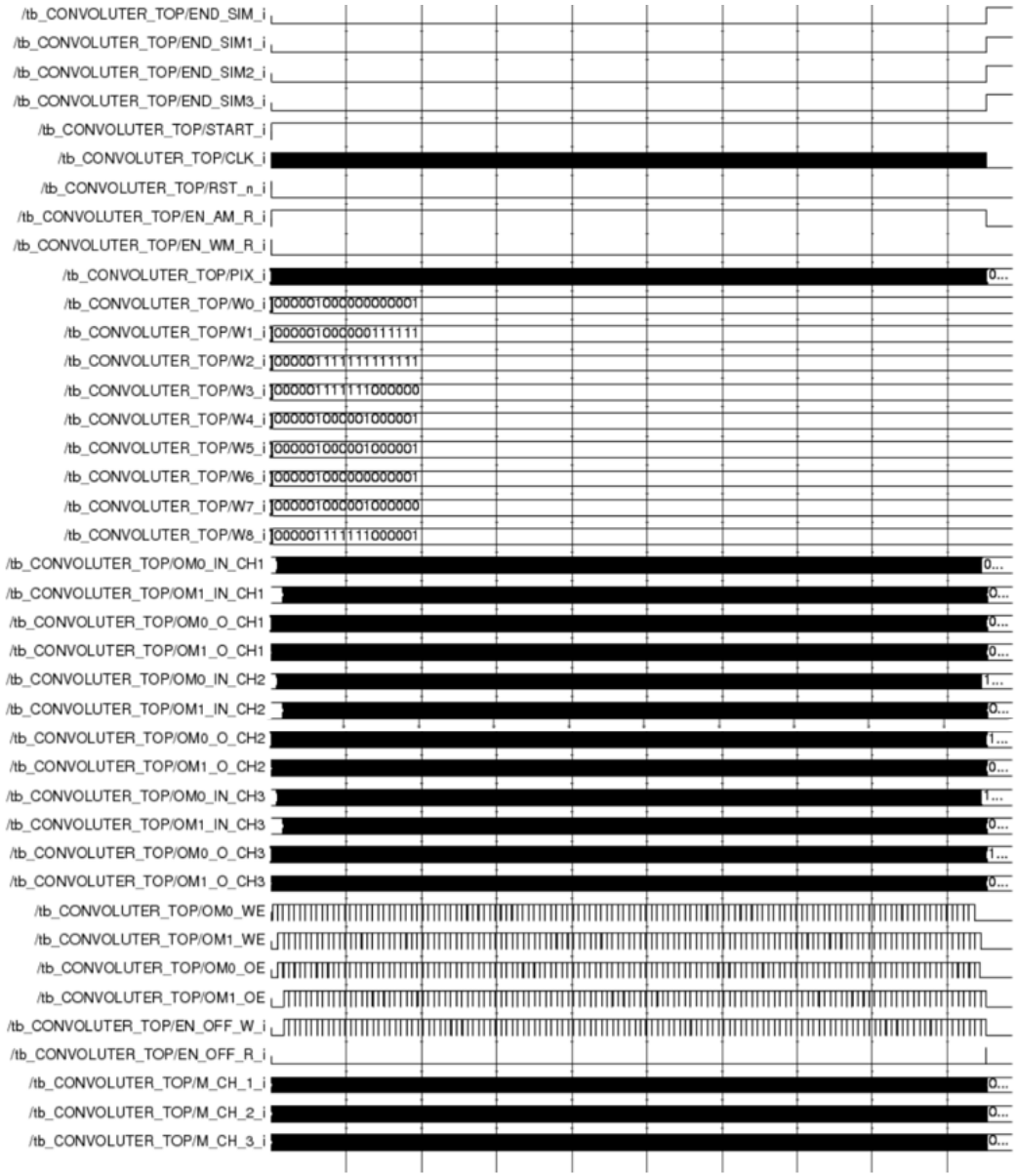

Figure 5.4. Timing Diagram that shows the correct behaviour of the Starting architecture

5.3 Power estimation

The final step of the synthesis phase was the estimation of power consumption by using the switching activity model. In this phase working with both Synopsys and *Modelsim* is necessary. The switching activity is the measurement of changes of

signal values in a determined clock cycle (indeed it can change from 0 to 1 or 1 to 0). It is also essential to measuring power in digital circuits.
After creating a *saif* file where Synopsys extrapolates the technological libraries, the Verilog testbench has been modified in order to manage some statements to get the switching activity of each design. Its information have been written in another *saif* file at the end of the simulation. The following scheme shows this approach:

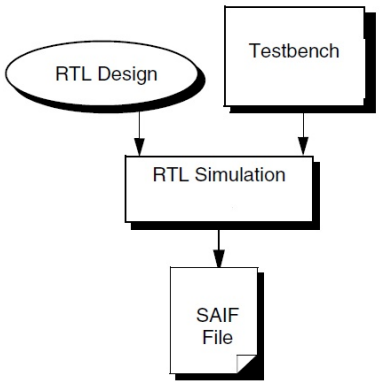

Figure 5.5. Dynamic power-frequency graph in different architectures. [31]

To do that, a `End_Sim` signal, generated from the Output Memory, has been used. It has been necessary to indicate that the simulation could be stopped. Indeed the end of the Simulation generates the `END` signal that blocks the generation of the clock. This step is necessary to obtain a correct estimation of the switching activity. Indeed Verilog testbench has been modified in order to define the window of the simulation where the power simulation had to be performed. The following table shows the average values of power estimation of the each architecture at the maximum operating frequency, obtained by testing more inputs.

Architecture	Dynamic Power $[mW]$	Leakage Power$[\mu W]$
Starting	21.01	3.92
Zero Skipping	20.18	3.77
Equal Weights Skipping	20.51	3.86
Approximation Skipping [-31:31]	19.29	3.96
Hybrid EW and APP Skipping [-31:31]	20.07	4.36

Table 5.1. Power consumption of different architectures

As it can be seen, the low power techniques applied to the Starting Architecture allows to reduce the switching activity and so the power consumption due to it. The

increase of the number of skipped multipliers allows to reduce the Dynamic power. The different areas do not influence the power even if they are the cause of its low reduction. Moreover, the Leakage Power also depends on the type of architecture. More complex architectures have higher leakage.
The Approximation Skipping architecture produces the most reduced power consumption; for this reason is the best low power technique that can be applied. Particular is the case of the Equal Weights Skipping Architecture that although it has lower switching activity it provides more consumption than the Approximation Skipping one due to its larger area and its internal cells.
However, the Dynamic Power has been computed also at lower frequency and the results have been represented by means of the following graph:

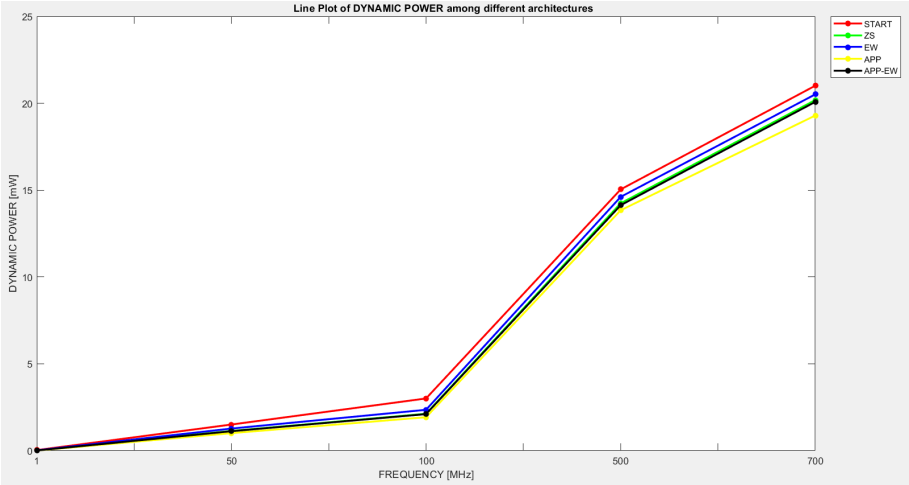

Figure 5.6. Dynamic power-frequency graph in different architectures

As it can be noted, the Approximation Skipped Architecture still provides the best results even at lower frequencies. But there is a reduction of the overall power consumption for each architecture. This reduction can not be linear due to the Leakage power. The choice of working at lower frequency can be correct since the most recent video-processing camera (that need in any possible application) have a throughput of 1 thousand frames per second (FPS). So there is no need to push too much on the frequency. This aspect is very important because it allows to decrease not only the power but even the area.
However, since all results obtained do not depend on on-chip memories, the real consumption have a further contribution due to them, that is equal for all architectures.

5.4 Comparison with other works

Finally, the information given in the previous sections, can be useful to do a comparison between the proposed work and other previous Hardware accelerators. The structural differences of *Eyerris* [25] and *ZeNA* [9] can not be compared due to difference in terms of technology, speed and area, since they represented an entire network.
An useful comparison can be done with Huan's work [10] because both analyze a single layer even if they have some structural differences. Moreover, a useful comparison with more complex platforms (CPU, GPU, and so on) can be done.

Accelerators	Technology node $[nm]$	Bid Width	CLK Frequency $[MHz]$	Power $[mW]$
Starting (Huan)	65	16	500	59
Starting (**This work**)	65	16	689.65	**21**
Zero Skipping (Huan)	65	16	500	38
Zero Skipping (**This work**)	65	16	653.39	**20**
Approximation Skipping (Huan)	65	16	500	31
Approximation Skipping (**This work**)	65	16	653.59	**19**
CPU Core-i7 5930k [32]	22	not given	3500	73000
GPU GeFore Titan X [32]	28	not given	1075	159000
mGPU Tegra K1 [32]	28	not given	852	5100

Table 5.2. Comparison with existing platforms

The proposed work provides better results than the Huan'one even if there is no the contribution of the on-chip memories in the analysis. But the number the multipliers in the architectures are different (9 vs 256); the maximum frequency is higher in the proposed work thanks to the *Migration* output accumulation used with respect the adder tree of Huan's work. However, both ASIC works allows to decrease the power consumption with respect CPU for PC, GPU and mobile GPU even if the speed is necessarily reduced.

Chapter 6

Functional validation

The functional validation phase allows to verify how the hardware optimizations, which have been done, can be applied to an already trained and tested neural network to obtain a validation of the choice of the hardware. This concept should mainly concern architectures where an approximation has been applied because only in this case the result of each layer would be influenced by the approximation interval. In fact, this step is necessary, because without hardware validation there could be a problem. For example, an image may be recognized by means of low-power no-approximated architecture, but may not be recognized after some low-power optimizations from the same architecture. Thus low power optimizations may not be suitable for the specific network or even for other networks.
However, validation can help to choose the right threshold to approximate the calculations without altering the operations of the network; but it can also help to understand which architectures can be better than the other ones depending on the level in which they are to be applied. This last choice may depend on the analysis of the number of skipped multipliers at each layer. It allows to understand the right combination of architectures within the network without stressing the machine where it is not necessary.
To obtain a good validation, first an already trained and tested Neural Network has been analyzed. In this thesis, **AlexNet** has been chosen [12] thanks to its simplicity and its small number of layers. In the following section the AlexNet will be presented, while the validation process will be described in the next sections.

6.1 AlexNet

As it has been said in the Chapter 2, the **AlexNet**[12], developed by Krizhevsky from the University of Toronto, won the ILSVRC in 2012 by obtaining better results than the previous networks. The following image shows its structure:

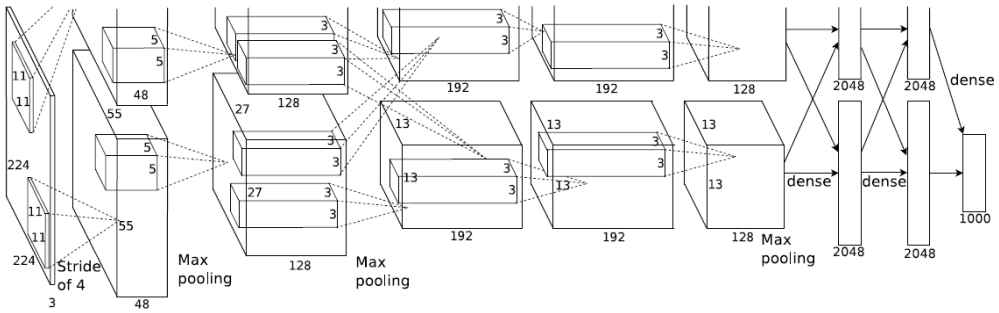

Figure 6.1. AlexNet network design [12]

The network contains 8 layers. The first five are convolutional and they represent the main computational power of the entire networks. On the other hand, the remaining three layers are fully-connected. The 5 convolutional layers are divided so:

- In the first layer, the convolution is computed by applying to a $224 \times 224 \times 3$ input 96 kernels of size $11 \times 11 \times 3$ with a stride of 4 pixels;
- The second convolutional layer receives as input the output of the previous layer and filters it with 256 kernels of size $5 \times 5 \times 48$ with a stride of 1;
- The third layer computes the convolution between the output of the previous layer and 384 kernels of size $3 \times 3 \times 256$ with a stride of 1;
- The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$ and it applies them to the output of the previous layer with a stride of 1;
- Finally the fifth layer computes the convolution between the output of the fourth layer and 256 kernels of size $3 \times 3 \times 192$ with a stride of 1.

What is more, a particular response-normalization, said *Batch-normalization* [12], is applied to the first two layers. On the other hand, Max-pooling layers are applied to the layers where Batch-Normalization is applied and in the fifth; instead the ReLU is applied to the output of every convolutional and fully-connected layer. In the second, fourth and fifth layer the kernels are connected only to those kernel maps in the previous layer which reside on the same GPU; instead in the third layer the kernels are applied to all kernel maps in the previous layer.
The fully-connected layers have 4096 neurons each. Therefore, the output of the last fully-connected layer depends on 1000-way softmax which produces a classification and distribution over the 1000 class labels.
The next section will show how a Matlab model of the AlexNet, already trained and tested with ImageNet Dataset, has been modified to recreate the hardware optimizations in software in order to proceed with the validation.

6.2 AlexNet Software Model

An already trained and tested Matlab model has been analyzed to understand the structure and to extract the weights and the biases used. So the DAG network model (a particular system to implement Neural Networks in Matlab) has been divided in more parts in order to replace only the convolutional layer with the hardware modifies implemented in software, and reuse the DAG structure for the other operations (Normalization, ReLU and Max pooling).
The following scheme shows the DAG network structure of the entire AlexNet Software Model:

ANALYSIS RESULT

	NAME	TYPE	ACTIVATIONS	LEARNABLES
1	data 227x227x3 images with 'zerocenter' normalization	Image Input	227×227×3	-
2	conv1 96 11x11x3 convolutions with stride [4 4] and padding [0 0 0 0]	Convolution	55×55×96	Weights 11×11×3×96 Bias 1×1×96
3	relu1 ReLU	ReLU	55×55×96	-
4	norm1 cross channel normalization with 5 channels per element	Cross Channel Nor...	55×55×96	-
5	pool1 3x3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	27×27×96	-
6	conv2 256 5x5x48 convolutions with stride [1 1] and padding [2 2 2 2]	Convolution	27×27×256	Weights 5×5×48×256 Bias 1×1×256
7	relu2 ReLU	ReLU	27×27×256	-
8	norm2 cross channel normalization with 5 channels per element	Cross Channel Nor...	27×27×256	-
9	pool2 3x3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	13×13×256	-
10	conv3 384 3x3x256 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	13×13×384	Weights 3×3×256×38 Bias 1×1×384
11	relu3 ReLU	ReLU	13×13×384	-
12	conv4 384 3x3x192 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	13×13×384	Weights 3×3×192×38 Bias 1×1×384
13	relu4 ReLU	ReLU	13×13×384	-
14	conv5 256 3x3x192 convolutions with stride [1 1] and padding [1 1 1 1]	Convolution	13×13×256	Weights 3×3×192×25 Bias 1×1×256
15	relu5 ReLU	ReLU	13×13×256	-
16	pool5 3x3 max pooling with stride [2 2] and padding [0 0 0 0]	Max Pooling	6×6×256	-
17	fc6 4096 fully connected layer	Fully Connected	1×1×4096	Weights 4096×9216 Bias 4096×1
18	relu6 ReLU	ReLU	1×1×4096	-
19	drop6 50% dropout	Dropout	1×1×4096	-
20	fc7 4096 fully connected layer	Fully Connected	1×1×4096	Weights 4096×4096 Bias 4096×1
21	relu7 ReLU	ReLU	1×1×4096	-
22	drop7 50% dropout	Dropout	1×1×4096	-
23	fc8 1000 fully connected layer	Fully Connected	1×1×1000	Weights 1000×4096 Bias 1000×1
24	prob softmax	Softmax	1×1×1000	-
25	output crossentropyex with 'tench' and 999 other classes	Classification Output	-	-

Figure 6.2. DAG network structure of the AlexNet

As it can be seen, all layers and their parameters are scheduled. The layers that have been modified are *conv1*, *conv2*, *conv3*, *conv4*, *conv5*; they are replaced with all proposed architectures implemented in software.
First, the standard operation of convolution has been implemented by means of "*for*" cycles. The following code implements the convolution operation in the third layer:

```
%%STANDARD CONVOLUTION

 for l=1:N34
   h=0;
   for i=1:K345
      k=0;
      for j=1:K345
          for g = 1:D3
              for s= 1:C345
                  for t=1:C345

                      O_P_3(i,j,l) = O_P_3(i,j,l) + CONV_LAYER3(h+s,k+t,g)*FF3(s,t,g,l);
                      MULTIPLICATIONS_LAYER3(1,1,g) = MULTIPLICATIONS_LAYER3(1,1,g) + 1;
                  end
              end
          end
          k = k+1;
          O_3(i,j,l) =  O_P_3(i,j,l) + BIAS3(1,1,l);
       end
       h = h+1;
   end
 end
```

Figure 6.3. Matlab code that implements the standard convolution in layer 3

In the code `N34 = 384` (number of kernels), `K345 = 13` (input size), `D3 = 256` (number of channels) and `C345 = 3` (filter size). The input `CONV_LAYER3` is convoluted with the kernel `FF3` to obtain the oFMAP `O_3` after the sum with the bias.
From this structure, the other software architectures can be elaborated. The Zero Skipping model is described in the following code, where the check of the inputs is done and the multiplications are skipped when at least one input is zero :

```
%%ZERO SKIP CONVOLUTION

for l=1:N34
    h=0;
    for i=1:K345
        k=0;
        for j=1:K345
            for g = 1:D3
                for s= 1:C345
                    for t=1:C345
                        if ( CONV_LAYER3_ZS(h+s,k+t,g) ~= 0) && ( FF3(s,t,g,l) ~= 0)
                            O_P_3_ZS(i,j,l) = O_P_3_ZS(i,j,l) + CONV_LAYER3_ZS(h+s,k+t,g)*FF3(s,t,g,l);
                            MULTIPLICATIONS_LAYER3_ZS(1,1,g) = MULTIPLICATIONS_LAYER3_ZS(1,1,g) + 1;
                        end
                    end
                end
            end
            k = k+1;
            O_3_ZS(i,j,l) =  O_P_3_ZS(i,j,l) + BIAS3(1,1,l);
        end
        h = h+1;
    end
end
```

Figure 6.4. Matlab code that implements the Zero Skip convolution in layer 3

The Equal Weights Skipping model of the third layer is described in the following code:

```
if t > 1
    if ((FF3(t,s,g,l) - FF3(t-1,s,g,l)) < EQ)&&((FF3(t,s,g,l) - FF3(t-1,s,g,l)) > -EQ)
        if    flag_LAYER3(1,g,l) == 0
              Partial3 = (CONV_LAYER3_EW(h+t,k+s,g)+CONV_LAYER3_EW(h+t-1,k+s,g))*FF3(t,s,g,l);
              MULTIPLICATIONS_LAYER3_EW(1,1,g) = MULTIPLICATIONS_LAYER3_EW(1,1,g) + 1;
              flag_LAYER3(1,g,l) = 1;
        else
              Partial3 = 0;
              flag_LAYER3(1,g,l) = 0;
        end
    else
        if flag_LAYER3(1,g,l) == 0
           Partial3 = CONV_LAYER3_EW(h+t,k+s,g)*FF3(t,s,g,l);
           MULTIPLICATIONS_LAYER3_EW(1,1,g) = MULTIPLICATIONS_LAYER3_EW(1,1,g) + 1;
        else
           Partial3 = 0;
           flag_LAYER3(1,g,l) = 0;
        end
    end
else
        if flag_LAYER3(1,g,l) == 0
           Partial3 = CONV_LAYER3_EW(h+t,k+s,g)*FF3(t,s,g,l);
           MULTIPLICATIONS_LAYER3_EW(1,1,g) = MULTIPLICATIONS_LAYER3_EW(1,1,g) + 1;
        else
           Partial3 = 0;
           flag_LAYER3(1,g,l) = 0;
        end
 end
 O_P_3_EW(i,j,l) = O_P_3_EW(i,j,l) + Partial3;
```

Figure 6.5. Matlab code that implements the Equal Weights Skip convolution in layer 3

The code focuses only on the internal part of the for cycles. In the Equal Weights Skipping model the check of the weights is done. Since the AlexNet networks often have consecutive filters, which differ very little (about 0.00001), a system has been adopted to consider them the same (since in hardware, the quantization of the values would make them practically equal ones). So a flag is used to manage which multiplications to skip, and the value `EQ`, used to check, is 0,00001.
The Approximation Skipping model of the third layer is described with the following code:

```
%%APPROXIMATE CONVOLUTION

for l=1:N34
    h=0;
    for i=1:K345
        k=0;
        for j=1:K345
            for g = 1:D3
                for s= 1:C345
                    for t=1:C345
                        Partial_3 = CONV_LAYER3_APP(h+s,k+t,g)*FF3(s,t,g,l);
                        if ((Partial_3 >= (TH3*-1)) && (Partial_3 <= TH3))
                            Partial_3 = 0;
                        else
                            MULTIPLICATIONS_LAYER3_APP(1,1,g) = MULTIPLICATIONS_LAYER3_APP(1,1,g) + 1;
                        end
                            O_P_3_APP(i,j,l) = O_P_3_APP(i,j,l) + Partial_3;
                    end
                end
            end
            k = k+1;
            O_3_APP(i,j,l) =  O_P_3_APP(i,j,l) + BIAS3(1,1,l);
        end
        h = h+1;
    end
end
```

Figure 6.6. Matlab code that implements the Approximate Skip convolution in layer 3

The described system analyzes the range given by the comparison with `TH3`.
Finally the Hybrid model of Equal Weights and Approximation Skipping is written to involve the previous 2 models. Even in this case the following code focuses only on the internal part of the for cycles.
The following code shows the Hybrid model of the third layer:

```
if t > 1
      if ((FF3(t,s,g,l) - FF3(t-1,s,g,l)) < EQ)&&((FF3(t,s,g,l) - FF3(t-1,s,g,l)) > -EQ)
          if    flag_LAYER3_APP(1,g,l) == 0
                Partial3 = (CONV_LAYER3_APP_EW(h+t,k+s,g)+CONV_LAYER3_APP_EW(h+t-1,k+s,g))*FF3(t,s,g,l);
                if ((Partial3 >= TH3*(-1)) && (Partial3 <= TH3))
                    Partial3 = 0;
                else
                    MULTIPLICATIONS_LAYER3_APP_EW(1,1,g) = MULTIPLICATIONS_LAYER3_APP_EW(1,1,g) + 1;
                end
                flag_LAYER3_APP(1,g,l) = 1;
          else
                Partial3 = 0;
                flag_LAYER3_APP(1,g,l) = 0;
          end
     else
          if flag_LAYER3_APP(1,g,l) == 0
             Partial3 = CONV_LAYER3_APP_EW(h+t,k+s,g)*FF3(t,s,g,l);
             if ((Partial3 >= TH3*(-1)) && (Partial3 <= TH3))
                 Partial3 = 0;
             else
                 MULTIPLICATIONS_LAYER3_APP_EW(1,1,g) = MULTIPLICATIONS_LAYER3_APP_EW(1,1,g) + 1;
             end
          else
             Partial3 = 0;
             flag_LAYER3_APP(1,g,l) = 0;
          end
      end
else
      if flag_LAYER3_APP(1,g,l) == 0
         Partial3 = CONV_LAYER3_APP_EW(h+t,k+s,g)*FF3(t,s,g,l);
         if ((Partial3 >= TH3*(-1)) && (Partial3 <= TH3))
             Partial3 = 0;
         else
             MULTIPLICATIONS_LAYER3_APP_EW(1,1,g) = MULTIPLICATIONS_LAYER3_APP_EW(1,1,g) + 1;
         end
      else
          Partial3 = 0;
          flag_LAYER3_APP(1,g,l) = 0;
      end
end
O_P_3_APP_EW(i,j,l) = O_P_3_APP_EW(i,j,l) + Partial3;
```

Figure 6.7. Matlab code that implements the Equal Weights and Approximate Skip convolution in layer 3

Thus, these models have been written for each layer, by following the AlexNet structure previously described. What is more, a system that counts how many multipliers are executed allows to track the efficiency of each algorithm. In the next section the validation phase will be described and the obtained results of the simulations will be shown.

6.3 Validation phase

To validate hardware accelerators proposed, this thesis uses a dataset of the ImageNet, already trained and tested in the AlexNet model. So the various models will be compared among the various layers to classify which is better than other ones.
Now, as it has been done in the Chapter 4, the validation phase will be divided in two parts: an analysis of the application of architectures without approximation and that where the approximations are applied.

6.3.1 Applications on AlexNet without approximation

The first analysis is done on the Zero Skipping Architecture. The following image shows how this architecture influences the number of multipliers in the entire network, by using the specific dataset:

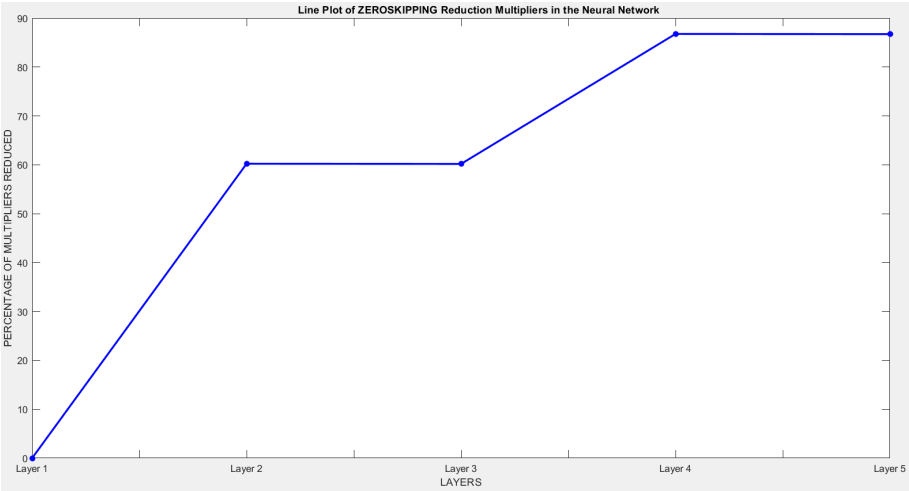

Figure 6.8. Reduction of number of multipliers among various layer with Zero Skipping Architecture

As it can be seen, the number of multipliers executed in the standard architecture is reduced in the Zero Skipping Architecture. The most important reduction is in the Layer 4 and Layer 5 where there is a reduction of about the 85%. In the first layer of the AlexNet, ZeroPadding is not applied such as in the other layers, so no multiplier can be skipped. Instead, Layer 2 and Layer 3 show good results with a reduction of the 60% (ReLU and ZeroPadding increase the number of zeros in the deeper layers).
Then, another analysis concerns about the Equal Weights Skipping Architecture.

The following image shows how this architecture influences the number of multipliers in the entire network, by using the specific dataset:

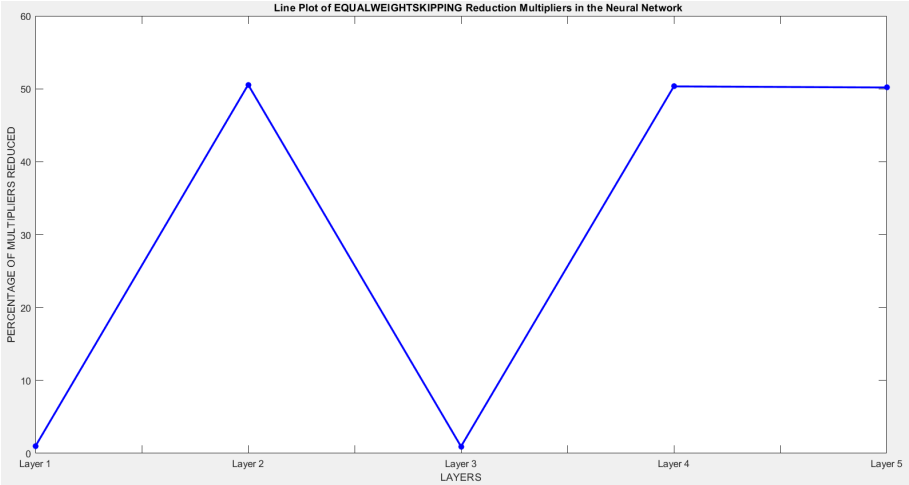

Figure 6.9. Reduction of number of multipliers among various layers with Equal Weights Skipping Architecture

As it can be seen, in the AlexNet this architecture is less efficient than the previous one, because only Layer 2, Layer 4 and Layer 5 show a significant reduction of the number of multipliers; but it is lower than the Zero Skipping architecture (about 50 percent in each named layer). This aspect is very important, because it shows how the efficiency of an architecture depends on the type of the Network where it is applied (an other network could report inverse results).

6.3.2 Applications on AlexNet with approximation

The architectures with approximation have been applied in order to involve a large range of approximation $[-2 : 2]$, by starting by a low threshold that approximates the results of multiplications inside the range $[-2^{-5}:2^{-5}]$. The idea is to verify which is the minimum range of thresholds where the dataset is still recognized by the network in order to discard the thresholds that are not suitable. The application of the thresholds inside the network can be done, in different ways. The idea is to measure the impact of approximation on the accuracy of the network and the possible reduction of the number of executed multipliers, not only with various threshold values, but also with uniform and non-uniform thresholds for different layers. A possible method is to use a mono-threshold, common to each layer in order to verify the behavior of each layer depending on the same threshold. Otherwise, different

thresholds can be applied to different layers in order to understand which layers can accept the highest errors, by obtaining the correct operating of the network.

Mono-threshold method

First, the mono-threshold method can be applied to the approximate architectures. A *Matlab* simulation has been done, by providing to architectures 64 different thresholds in order to modify the boundary of the approximation range from 2^{-5} to 2. The following graph shows how the probability of the dataset recognizing changes based on the application of the thresholds in the both architectures.

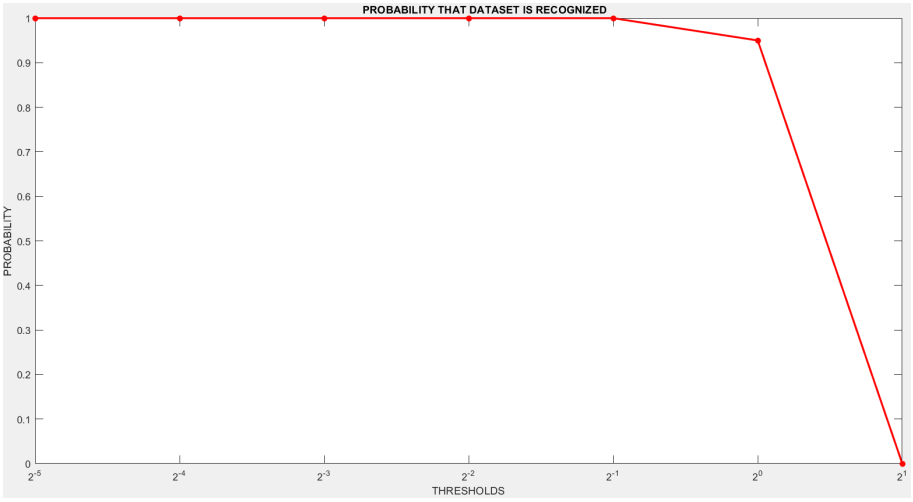

Figure 6.10. Probability that Dataset is recognized by the Network when threshold changes

As it can be seen, the dataset is recognized by using a wide range of boundaries of approximation ($[2^{-5}{:}2^{-1}]$). What is more, the range $[2^{-1}{:}2^{0}]$ still provides acceptable results while admitting some mistakes that reduces slightly the probability. But, the range $[2^{0}{:}2]$ can not accepted due to a high reduction of the probability of recognizing. So 2^{-1} is the maximum threshold that can be accepted to always obtain the recognizing of the dataset.
Then the reduction of the number of multipliers can be explored. The following graph shows it, by using the Approximation Skipping Architecture:

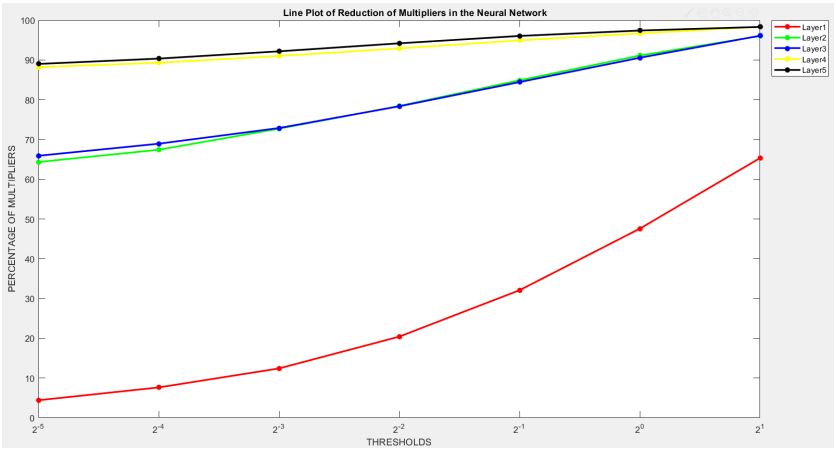

Figure 6.11. Reduction of number of multipliers among various layers with Approximation Skipping Architecture

As it can be seen, there are some improvements with respect the best result obtained in the applications without approximation. If the analysis focuses only on the threshold 2^{-1}, the improvement is clear overall in the first layer where there is a passage of reduction from about 0% to more than 30%. In the Layer 2 and Layer 3 there is an improvement of the 20% while in the last two layers the improvement is of the 10/15%. This aspect shows how in the depth of the Network there are many zeros with respect the first layers, so the major number of skipped multipliers depends principally on their presence.
The following graph shows the errors due to the approximation applied:

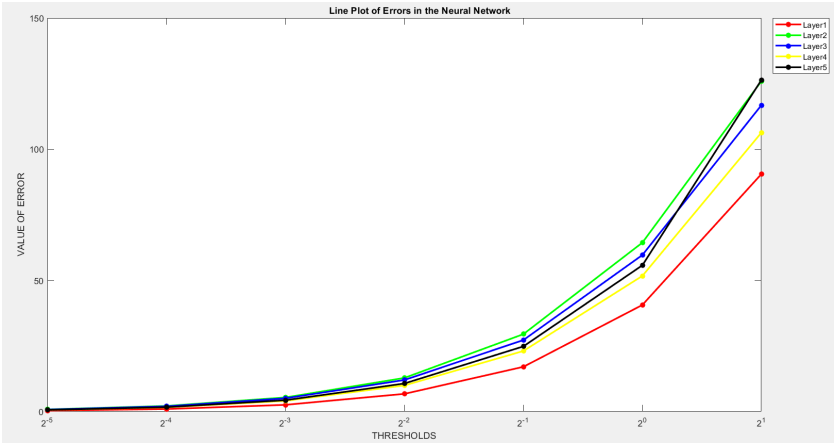

Figure 6.12. Errors due to approximations

This graph shows how the errors increase as the threshold increases. Indeed, the highest errors are done in the critical region of the range, and they are the reason of the mistakes of the network.
However, in the AlexNet the Hybrid solution (Equal Weights and Approximation Skipping Architecture) does not provide some clear improvements with respect the previous architecture, for the same reasons for which the Equal Weights Architecture is not better than Zero Skipping one.
The following graph shows how the results of both approximated architecture are almost similar, overall in the interesting regions.

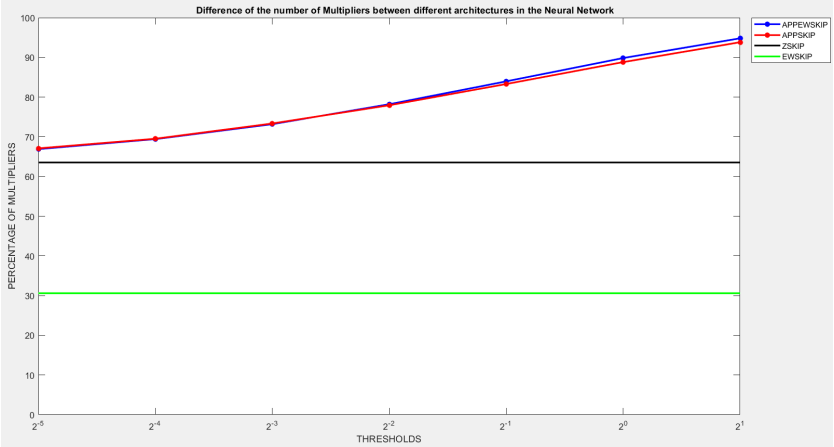

Figure 6.13. Difference between various architectures in the AlexNet

Those shown are average values calculated between all the layers. The values of no-approximated architectures are constant because they do not depend by thresholds. It is clear how the Hybrid solution could be avoided in order to reduce the stress of the network, considering that the results of the approximated applications are very similar. However no-approximated architectures provide worse results than approximated ones. So the approximated approach are surely better in low power terms.

Different threshold method

The application of different threshold does not produce expected results. The simulations done can be useful to understand which layers can accept higher threshold than others ones. The goal of the AlexNet is always trying to bring as much information as possible to the end. For this reason, the management of the thresholds is certainly delicate.

More test have been done:

- First, an uniform threshold has been applied in the first layers while in the others the thresholds have been changed;
- Then, an uniform threshold has been applied in the last layers while in the first ones the thresholds have been changed;
- And so on...

Overall, the results did not improve ones obtained with a mono-threshold method. But these tests have been useful to understand how thresholds could be managed among different layers.
So, the thresholds can not increase in the first layers since the inputs contain the most important information. So the last layers could have thresholds higher, by reducing the thresholds in the others layers. But this approach is not very useful because in the last layers there are many zeros and a Zero Skipping architecture may already be enough. For instance, if the threshold in the first layer defines the range of the approximation $[-2^0{:}2^0]$, the dataset is not still recognized by reducing the thresholds in the other layers. On the other hand, if the last layer has a range of approximation $[-2^0{:}2^0]$, the dataset is still recognized with a reduction of the number of executed multipliers of further 10%. So some solutions can be discarded, while others provide good results.
Thus, even if the last layers can be approximated more than the first layers, the mono-threshold is a simpler approach and it already allows to reduce the number of multipliers significantly.

6.4 Evaluations

Then, the proposed accelerators can be applied to an AlexNet model by obtaining interesting results, by doing some changes (number of PEs, number of on-chip memories and so on). From Validation phase, it is clear that the approximated architectures and specially, the Approximation Skipping Architecture can be applied at the first layer with a threshold that allows to skip all multiplications that produce a result inside the range $[-2^{-1}{:}2^{-1}]$. This condition allows to reduce the number of executed multipliers of more than 30%.
On the other hand, even if the Approximation Skipping Architecture allows to reduce more the amount of executed multipliers, the Zero Skipping Architecture is the best choice to decrease the power consumption without stressing the network, by reducing the cost and maintaining the accuracy. This choice is due to the fact that the major number of skipped multipliers depends on presence of zeros.

The Following graph shows how this choice influences the number of multipliers in the entire network, by using the specific dataset of the ImageNet:

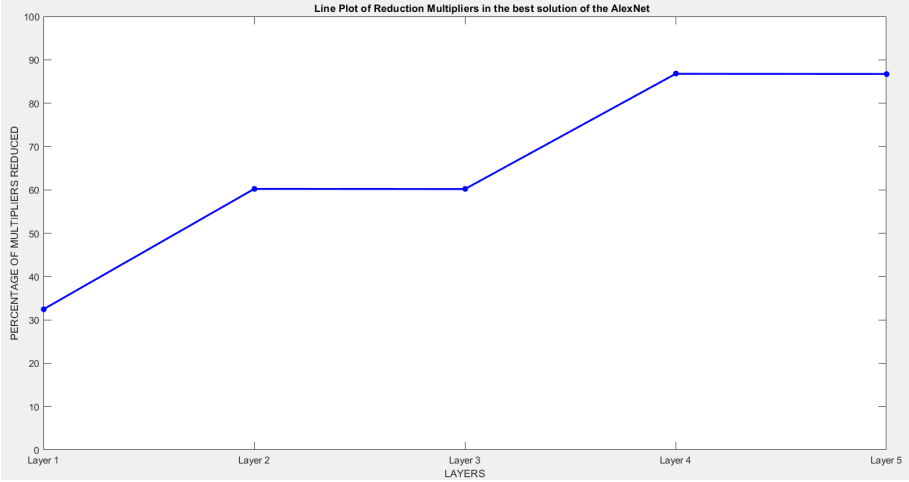

Figure 6.14. Reduction of number of multipliers among various layers with Approximation Skipping Architecture in the first layer where the range of approximation is $[-2^{-1}:2^{-1}]$ and with Zero Skipping Architecture in the others layers

Chapter 7

Conclusions and future works

This chapter completes this thesis work by providing an overall analysis and its possible future works or applications. The core of the proposed work is the design of different hardware accelerator prototypes for a Convolutional Neural Network. The different architectures are referred to the first layer of a CNN but they can be adapted for any layers thanks to some changes. For instance, the size of on-chip memories could be increased based on the input feature map of the layer and also the number of channels could increase if the number of kernels are more than one. However the proposed architectures have been described in detail, by showing the particularities of one compared to another. Then their performances have been demonstrated before with a Logical Synthesis and after by using them to compute a AlexNet inspired software network, in order to classify some images. They have still been compared in order to understand which architecture was the best in terms of power consumption without neglecting the area and above all the speed.
From the previous chapter, it is clear that the Approximation Skipping Architecture is the best to reduce power consumption of an AlexNet network overall in the first layer. The Zero Skipping Architecture provides good results in the other layers thank to sparsity due to ReLU and MaxPooling that generates always more zeros in the deep layers.
But this work could be applied to other Software Network models to verify its impact in terms of reduction of multipliers. For instance, various models of ResNet (*ResNet18* and *ResNet50*) [24] could be used to a further functional validation. Even new datasets could be used. For instance MNIST dataset [33] and CIFAR dataset [14] could be used as a validation set with nearly 80% sparse input data.
What is more, a possible future work could be to apply a similar system in the training phase in order to obtain different weights. This could be a good system to obtain a correct prevision already in the training phase in order to reduce the complexity of the computation in the test phase.

However, the proposed architectures could be still improved in order to skip the multiplication during the computation, not only with a reduction of the power but also with a reduction of the latency. Indeed, if a multiplication has not to be executed, the multiplier of that PE could be used for the next multiplication. This aspect is enough complex since the dataflow should be still modified and a similar architecture could not be able to perform a convolution without wasting more resources and not increasing the critical paths (many multiplexers could be inserted in a possible critical path). So the optimization could be done at the external memory level. New researches are analyzing the concept of the sparse memory in order to send inside the Datapath of an hardware accelerator, directly the useful data to compute the convolution.
Also, future extensions of this work could be to add other types of operation (ReLU, Max Pooling and so on) trying to optimize them in low power terms. Moreover, the internal parallelism could be reduced based on the applications, maintaining a fixed-point precision. So, these architectures could be improved making a trade off between latency and resource utilization. What is more, a possible future work of this thesis could be to implement the various architectures or the best one at level of FPGA in order to synthetize and to verify their performances, by comparing the two different approaches (ASIC and FPGA). [9] and [25] work at FPGA level; for this reason, a precise comparison between this work and them can not be done.
Finally, a similar architecture with all these optimization could improve the IoT devices. Indeed, nowadays the IoT world involves the Machine Learning and Deep learning world very much. For instance, many machine learning methods and algorithms are applied to the data in order to extract higher level information even in IoT purpose [34].

Bibliography

[1] M. K. Alsedrah, *Artificial Intelligence*, American University of the Middle East, Kuwait, 2017, vol. U1.

[2] H. Wehle, "Machine learning, deep learning, and ai: What's the difference?" in *International Conference on Data scientist innovation day*, Bruxelles, Belgium, July 2017.

[3] J. Le, "The 10 deep learning methods ai practitioners need to apply," in *Cracking the data science interview*, 2017.

[4] X. Zhang, J. Zhao, and Y. LeCun, "Character-level convolutional networks for text classification," Courant Institute of Mathematical Sciences, New York University, New York, 2016.

[5] C. Farabet, C. Poulet, and Y. LeCun, "An fpga-based stream processor for embedded real-time vision with convolutional networks," Courant Institute of Mathematical Sciences, New York University, New York, 2009.

[6] J. Jin, A. Dundar, J. Bates, C. Farabet, and E. Culurciello, "Tracking with deep neural networks," Purdue and New York University, 2013.

[7] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," 2015.

[8] M. Duranton, D. Black-Shaffer, K. De Boschere, and J. Maebe, *The Hipeac vision for advanced computing in horizon 2020*, 2013.

[9] D. Kim, J. Ahn, and S. Yoo, "Zena: Zero-aware neural network accelerator," 2018.

[10] Y. Huan, Y. Qin, Y. You, L. Zheng, and Z. Zou, "A low-power accelerator for deep neural networks with enlarged near-zero sparsity," 2017.

[11] J. Jo, S. Kim, and I.-C. Park, "Energy-efficient convolution architecture based on rescheduled dataflow," *IEEE Transactions on Circuits and Systems I: Regular Papers*, no. 99, pp. 1–12, 2018.

[12] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[13] A. Karpathy, "Stanford university cs231n: Convolutional neural networks for

visual recognition," vol. 1, 2018.
[14] [Online]. Available: https://www.cs.toronto.edu/ kriz/cifar.html
[15] A. Karpathy, "Stanford university cs231n: Convolutional neural networks for visual recognition," vol. 2, 2018.
[16] S. Patel and J. Pingel, "Introduction to deep learning: What are convolutional neural networks?" 2017.
[17] E. Million, "The hadamard product," pp. 2–4.
[18] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," Visual Geometry Group, Department of Engineering Science, University of Oxford.
[19] L. Feng, P. Djahani, and O. Gunasekara, "Deep learning, 3d content rendering, and massively parallel, compute intensive workloads in the cloud," 2016.
[20] [Online]. Available: http://www.image-net.org/about-overview
[21] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," pp. 1–46, 1998.
[22] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *European conference on computer vision*. Springer, 2014, pp. 818–833.
[23] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolution," 2014.
[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
[25] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.
[26] G. Desoli, N. Chawla, T. Boesch, S.-p. Singh, E. Guidetti, F. De Ambroggi, T. Majo, P. Zambotti, M. Ayodhyawasi, H. Singh *et al.*, "14.1 a 2.9 tops/w deep convolutional neural network soc in fd-soi 28nm for intelligent embedded systems," in *Solid-State Circuits Conference (ISSCC), 2017 IEEE International*. IEEE, 2017, pp. 238–239.
[27] G. Dimitrakopoulos, K. Galanopoulos, C. Mavrokefalidis, and D. Nikolos, "Low power leading-zero counting and anticipation logic for high-speed floating point units," *IEEE Transactions on very large scale integration*, vol. 16, no. 7, pp. 837–850, 2008.
[28] V. P. Nelson, "Automated synthesis from hdl models design compiler (synopsys) leonardo (mentor graphics)."
[29] [Online]. Available: https://en.wikipedia.org/wiki/Netlist
[30] [Online]. Available: http://www.vlsiip.com/asic dictionary/B/back annotation. html.
[31] *Power Compiler User Guide*, Synopsys, 2010, vol. Version D-2010.03-SP2,.

[32] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," *arXiv:1602.01528*, pp. 245–254.

[33] [Online]. Available: http://yann.lecun.com/exdb/mnist/

[34] M. S. Mahdavinejad, M. Rezvan, M. Barekatain, P. Adibi, P. Barnaghi, and A. P. Sheth, "Machine learning for internet of things data analysis: a survey," *Digital Communications and Networks*, no. 4, pp. 161–175, 2018.