POLITECNICO DI TORINO

Master's Degree in Electronic Engineering

Master's Degree Thesis

Reconfigurable Spiking Neural Network architecture on FPGA

Supervisors

Prof. Guido MASERA

Prof. Maurizio MARTINA

Candidate

Dino SANTORO

April 2020

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recently, we are witnessing to a rapid spread of the NNs in many fields thanks to their ability to reproduce and model nonlinear processes. Mainly driven by recent advances in computer vision, a fast improvement of the ANNs ensures the development of a more efficient artificial intelligence processing, which leads to the reduction of power consumption and latency time.

Nowadays, Deep Neural Networks (DNNs) have reached extremely high values of accuracy, over 90-95 % in complex image classification tasks, even surpassing human capabilities. The DNNs are ANNs with more than one hidden layer and are used to perform complex classification. A large number of neurons and a sufficient depth lead to a higher classification accuracy, compared to a smaller and shallower network on the same task. This high accuracy had a not negligible cost in terms of area and power consumption. Even considering only the DNN inference processing, each neuron executes a multiplication and a sum for each clock cycle, if we do not consider design optimizations, such as a possible pipeline. To clarify, the inference process consists in applying a trained neural network model and using a uses it to infer a result. This continuous flow of data and the huge amount of neurons bring the circuit to consume a huge amount of power that could be managed only by big server. There is, therefore, a need to improve this aspect to allow the implementation of an efficient artificial intelligence at the edge, where the hardware resources are limited. Practically, for example, complex neural networks must be deployed efficiently on cars or smartphones.
To improve the trade-off between speed and power several specialized accelerators have been proposed that work on GPU or more recently the Google TPU. They improve for sure the performance of the computations but highlights also that a radical change in the type of NN is necessary to bridge the gap between the power consumed by the DNN accelerators and the power consumed by the human brain. From this perspective, Spiking Neural Networks (SNNs) represent a significant step forward compared to the standard DNNs. This new type of network

forces us to redefine the hardware and the software in order to adapt them to the new requirements. This innovative network models the exact behaviour of our brain where each neural node receives and eventually transmits spikes that will stimulate the synapses connected and this information will be propagated to the following layer. It is so easy to understand that, in contrast to the DNNs, the SNNs are event-driven so they compute data only when this event, in SNN a spike, arrives. Another big advantage of the SNNs, not easy to catch at first sight is the modification of operations. The SNN, as already mentioned, are very near to represent a real human brain and this is possible thanks to the very detailed neuron models. One drawback of the SNN is that to communicate with the environment it is necessary to have a particular interface because the external world had continued value and do not send pulses. All these aspects will be better described in the next chapter. After an overview of the most recent state-of-the-art about the NNs and accelerators, the following chapter illustrates the reasons and limitations in the use of an FPGA for building an SNN accelerator. There will be described also the main hardware features and the software environment that help to create an efficient accelerator. In the subsequent chapter will be described the architecture developed and the algorithm implemented to execute the main layers type in SNN. This layers are convolutional, pooling and fully-connected, each of them with their tailored algorithms. The last two chapters will show all the testing steps executed to be sure that everything works correctly and in the right moment and also the results obtained with some future development.

Chapter 2

Neural Networks background and accelerators

2.1 Deep Neural Networks

In the current state, it is possible to witness to a huge variety of NNs that are originally developed for DNNs. Starting from 1998 the number of layers and neurons is increased to improve accuracy.

2.1.1 LeNet

This NN [1] was proposed in 1998 but we had to wait for 2010 to implement it. This happened because it required too much computation capability for that period. It is a CNN network with two convolutional layers followed by a pooling layer each. After these steps are present two fully connected layers. This network is also illustrated in figure 2.1

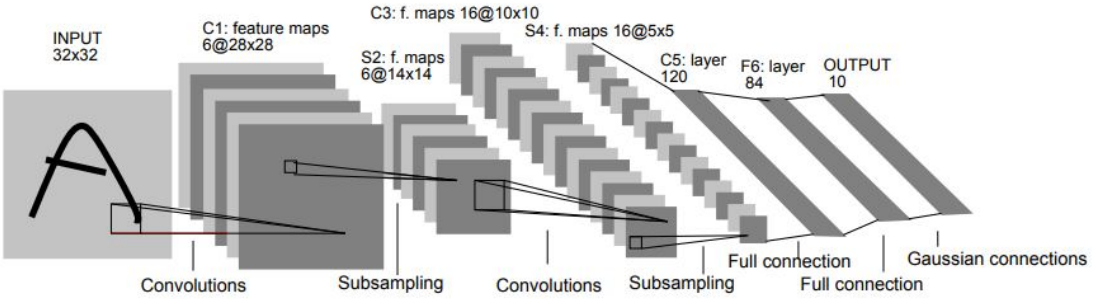

Figure 2.1: LeNet5

2.1.2 AlexNet

This network was proposed in ILSVRC (ImageNet Large Scale Visual Recognition Challenge) [2]. In fact, in 2012 by Alex Krizhevesky and others won the ImageNet challenge for visual object recognition with this architecture shown in figure 2.2. It is composed of only one more layer than the LeNet but the number of neurons increases dramatically. Looking, for example, the number of neurons in the fully connected layers that passing from about one hundred to few thousands. In this network, it is possible to find different innovations. First of all, it uses a ReLU activation function instead to the usual sigmoid or tanh suffer less the vanishing problem so the learning is faster. The ReLU has a drawback, the variables could assume too much higher values. For this reason, a new type of layer, in addition to the most common ones, was introduced. This new layer is called LRN (Local Response Normalization) and its job is to normalize a local and restricted area. Doing this it is possible to obtain a lateral inhibition and a consequent increase of the contrast between the excited neuron and the surrounding ones. All this novelties bring AlexNet to reaches an accuracy of 84.7%.

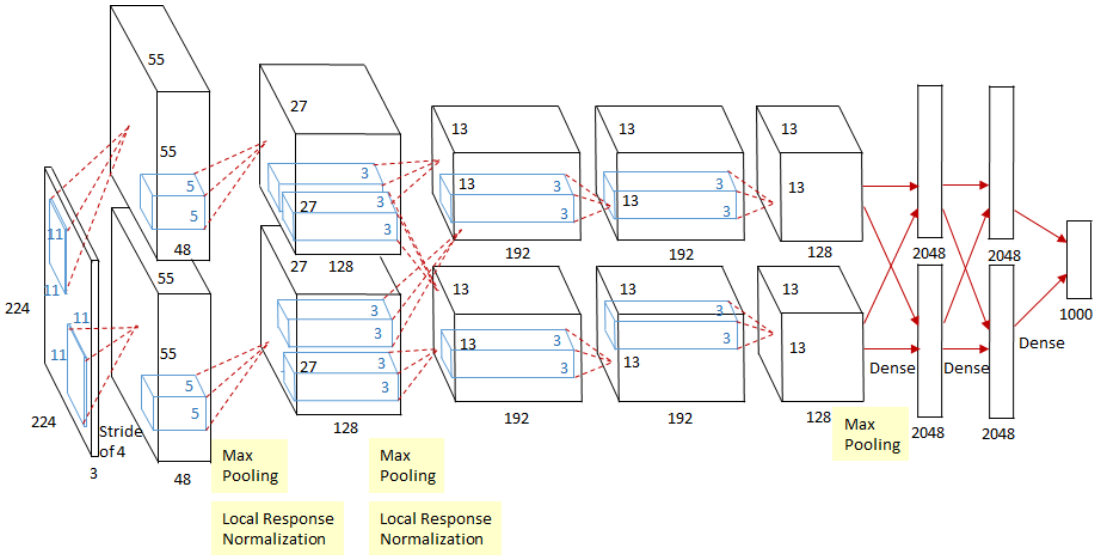

Figure 2.2: AlexNet

2.1.3 VGG

This NN was the runner up of the ILSVRC of 2014 and its acronym means Visual Geometry Group[3]. It has three different versions but all the three present a depth higher than the AlexNet one. The versions have respectively 11, 16 and 19 layers that makes it clear a criticality, that is that to improve accuracy the number of layers must increase. The goal of this NN and also of the subsequent was to obviously improve the accuracy but reduce also the number of parameters speeding about the training phase. The solution found is to fix to 3x3 the dimensions of filters (or kernels). The AlexNet presents different dimensions for the filters in convolutional layers that go from 3x3 to 11x11. VGG instead use only 3x3 filters because each dimension can be substituted by a multiple use of 3x3 filter. The architecture is shown in figure 2.3 and it presents the same pattern replicated more times. This pattern is composed of two or three convolutional layers followed by a max-pooling layer. At the end of this NN are present three fully connected layers and the usual soft-max layer for the classification. The accuracy of this network is equal to 92.3% with an increase of about 10% with respect to AlexNet.

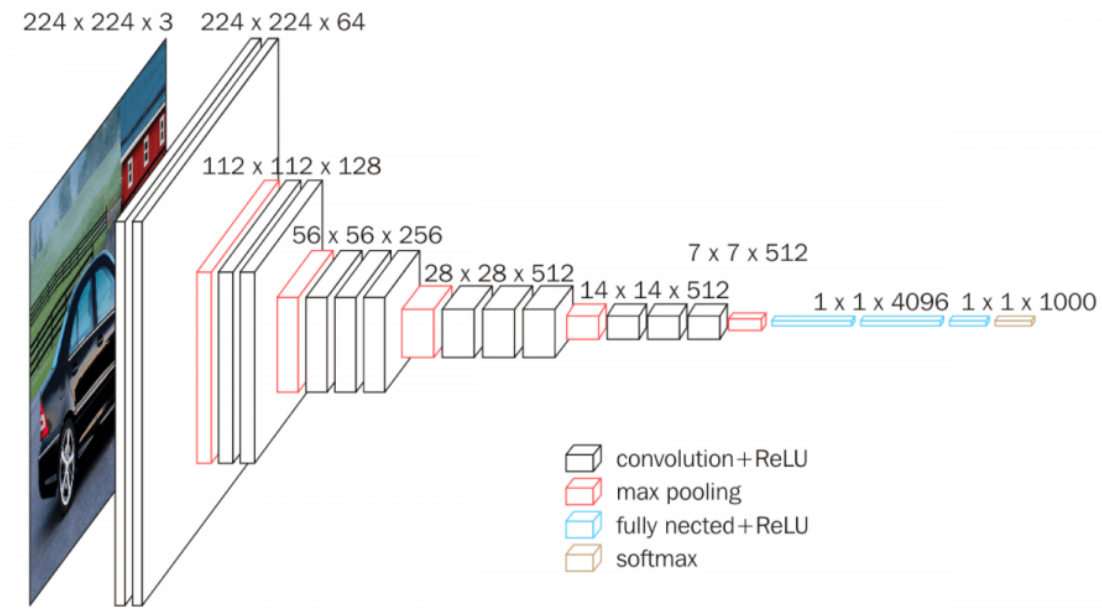

Figure 2.3: VGG16

2.1.4 GoogLeNet

Also, this NN was developed for the ILSVRC of 2014 and it won[4]. This network uses a different approach to reduce the parameters. The different dimensions of filters is important because the features can vary their size so in this NN the dimension of filter are not fixed. It presents different layers called Inception layers, shown in figure 2.4. They are a mix of convolutional layers with different dimension of filter and also convolutional layers. After this parallel computations, all the results are merged thanks to a filter concatenation. GoogLeNet is composed even of twenty-two layers but has half of weights and one-tenth of the Multiply-Accumulate operation with respect to VGG-16.

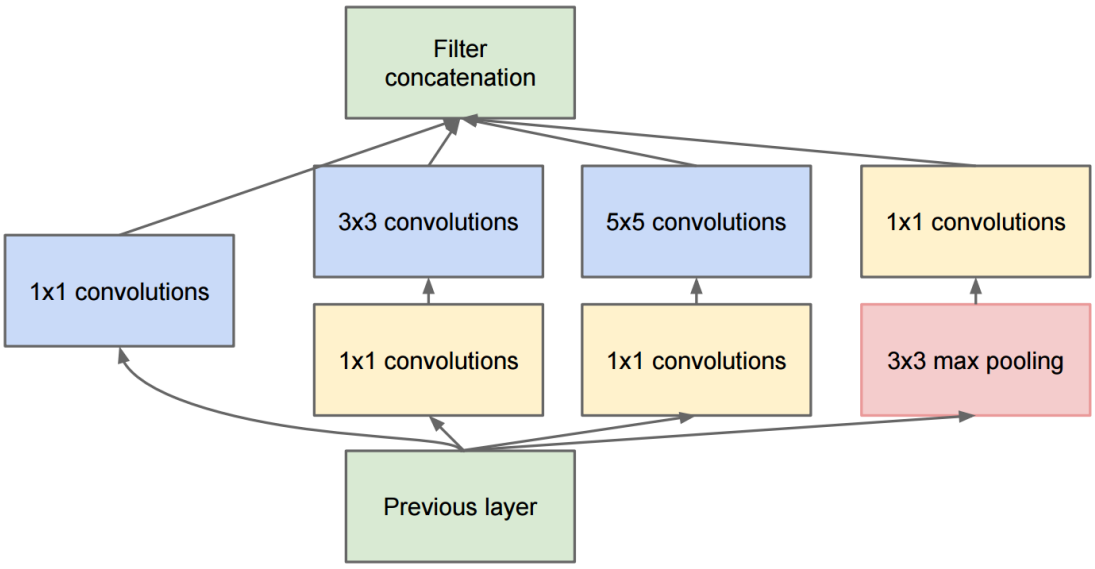

Figure 2.4: Inception layer

2.1.5 ResNet

This NN [5] was proposed in 2015 at the ILSVRC and present different version with numbers of layers that goes from 34 to even 1202 but the most popular is the version with forty-nine convolutional layers and one fully-connected layer called ResNet50. The main feature of this NN that distinguishes it from the previous ones is that are present some shortcuts between layers. These shortcuts are shown in figure 2.5. When in an ultra-deep NN the derivative goes to the initial layers to correct the weights its value become too much little. This implies that the learning of the NN will stop. This is the vanishing gradient problem that was partially solved by ReLU before.

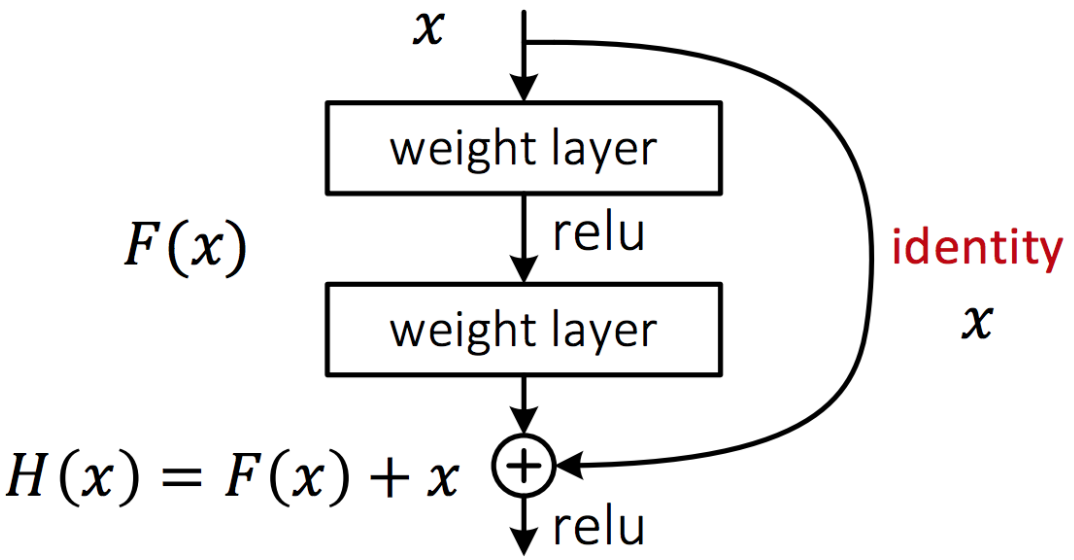

Figure 2.5: Shortcuts in ResNet

2.1.6 Comparison

	LeNet-5 (1998-2010)	AlexNet (2012)	VGG-16 (2014)	GoogLeNet (2014)	ResNet-50 (2015)
Accuracy (Top-5)	-	84.7%	92.3	93.3	94.7
Inpu size	28x28	227x227	224x224	224x224	224x224
Number of Conv Layers	2	5	16	21	49
Filter Size	5	3,5,11	3	1,3,5,7	1,3,7
Number of Feature Maps	6,16	3-256	3-512	3-1024	3-1024
Number of FC layers	2	3	3	1	1
Total Weights	431k	61M	138M	7M	25.5M
Total MAC operations	2.3M	724M	15.5B	1.43B	3.9B

Table 2.1: Comparison Neural Network

For table 2.1 is possible to understand several things describe before. First of all, the accuracy that had an huge improvement in 2010-2012 years and a saturation around the 92-95% for the basic version of these networks. All these NN are used for image classification except LeNet and this could be understood looking the input size that are 224x224. A clear data is the increase of the layers especially of type convolutional. It is possible to notice that the weights and MAC operations are more with respect to AlexNet but this is due to the increasing of accuracy. Because the fixed filter size reduce the weights with equal accuracy. The difference between GoogLeNet and VGG-16 is in terms of parameters and operation is wide in favour of the first one.

2.2 DNN accelerators

2.2.1 DianNao

The DianNao represents the first accelerator of a series developed during the years. It is composed of the following components:

- Neural functional unit (NFU). It performs the MAC operations and nonlinear function.
- Three different buffers that are used respectively for input neurons, output neurons, and weights.
- A control processor to manage the operations and the transfer of data.

The following architectures of the series can reduce power or execute multiple ML algorithms in parallel.

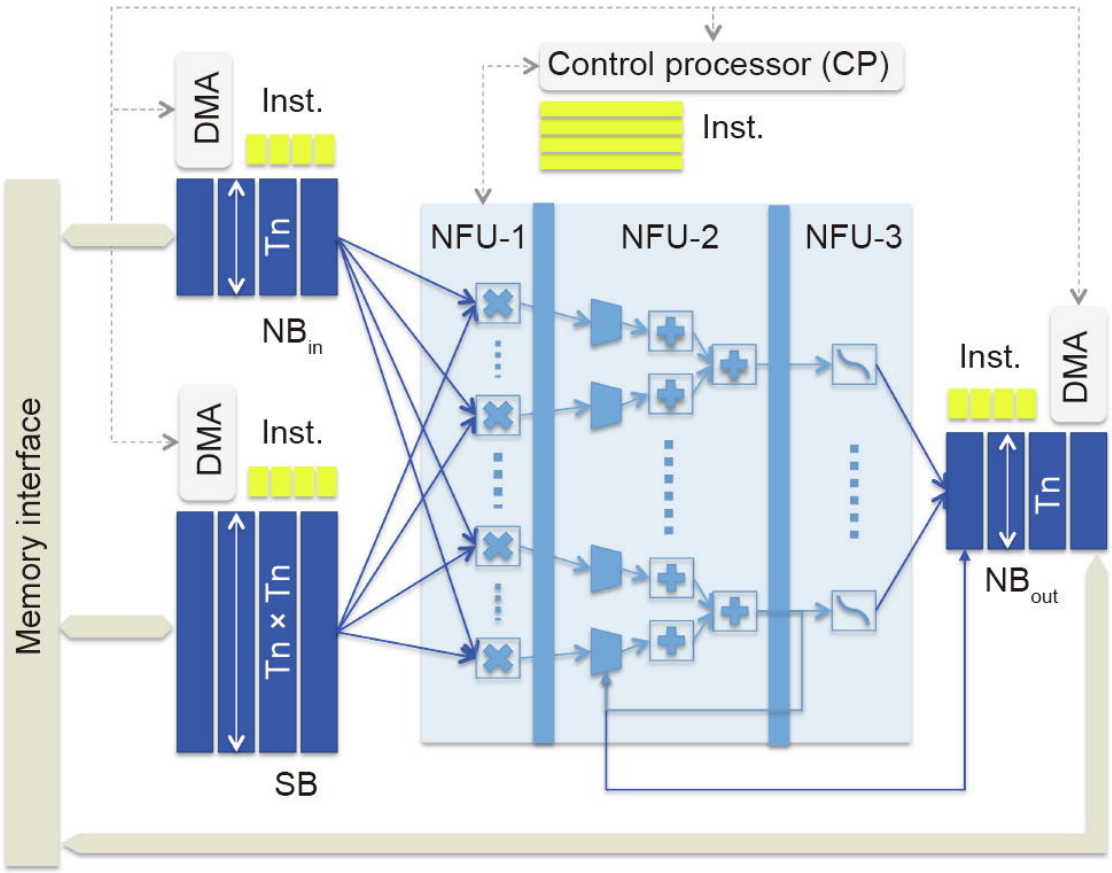

Figure 2.6: DianNiao Architecture

2.2.2 TPU

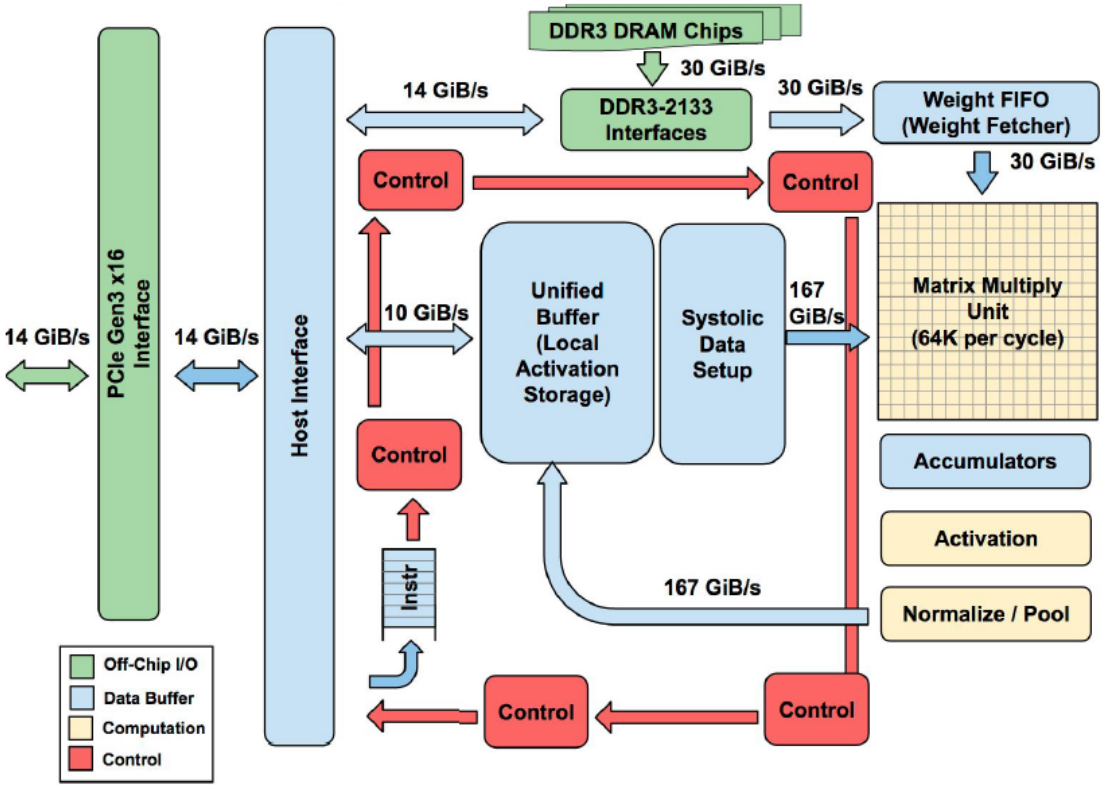

Figure 2.7: TPU Architecture

The first version of TPU was released by Google in 2017 to execute the inference. In next evolution of it are introduced systolic array and unit able to process vectors (useful for CapsNet).

2.3 Spiking Neural Networks: the third generation of NNs

The SNNs represent an important and maybe irreversible breakthrough in the field of Machine Learning. They are developed to achieve two different goals.
First, to reduce the power consumption in the NNs. In fact, the increase in the number of layers and the consequent increase of operations in NNs have increased the power used to execute them. The power has two big collateral effects.

- More power means more energy necessary from the power supply, therefore the NN could not be a simple supplied by a power-cell. This means that no portable device could use an optimal NN.

- More power means more heat so it is necessary to remove it to not destroy the integrated circuit or anyway slowdown it. Often, for this operation, it is not enough a simple heat sink but is fundamental to have a controlled temperature, like for the server.

The second goal is to understand better the natural neural network in order to cure several mental disease or understand the emotions creating, therefore, "robot with feelings".
The SNNs can mimic the human brain adding the concept of time to operations. In our brain, the propagation of a signal (spike) happens only when the membrane potential reaches a specific value (threshold). An SNN saves a lot of power with respect to a DNN for two main reasons. The first one is that the SNN is event-driven. This means that all the operations are triggered by a spike coming from the external or a previous layer. This reduce dramatically the number of operations.
The second reason of power reduction is due to the different type of operation. In a DNN the output of a neurons is calculated as described in equation 2.1

$$output = \sum_{\#neurons} input_i * weight_i + bias \tag{2.1}$$

In an SNN, the inputs are codified as binary spikes so can assume value equal to "1" or to "0". This implies that the multiplication, the most costly operation in the previous equation, is not necessary. This characteristic acts in all the aspects of an integrated circuit. The removal of multiplication reduces the complexity and consequently clock period and the area but reduce also the dynamic power used.
In summary, it possible to say that in SNN are executed only additions and the number of times that these additions are executed is much less with respect to DNN.

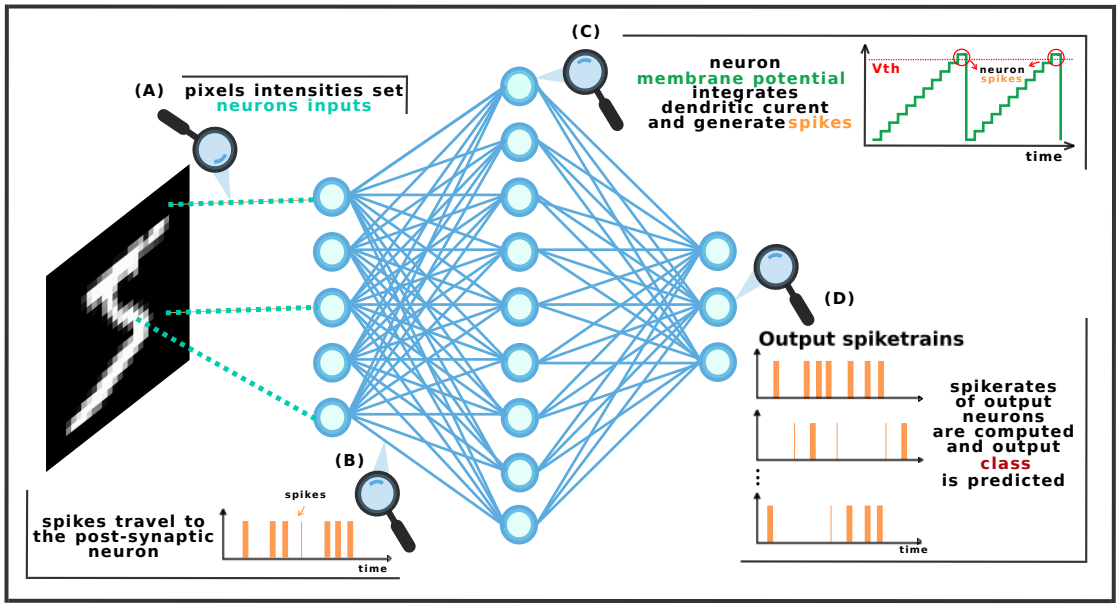

Figure 2.8: SNN

It is possible to see in figure 2.8 that the procedure in an SNN could be divided into four phases.

1. Each pixel intensity is used to set the input neurons. Details in 2.3.2. Each input neuron represents a specific pixel.

2. The spikes travel to the post-synaptic neurons propagating the information through the network.

3. Each neuron has a membrane potential that integrates during the time, using a specific equation, the dendritic current. Detail of equations are described in 2.3.1. When this membrane reaches the threshold potential a spike is generated by the neuron. This spike will be propagated like in the previous phase.

4. At the end, the output class is predicted looking the spike rates of the output neurons. The logic winner-takes-it-all is applied so the winner class is the one with high spike rate.

2.3.1 Neurons

To mimic in the best way the human brain several kinds of research are done also to model the neuron itself. The most common models are three.

- The first one is the Izhikevich model [6]. It is a very detailed equation, to compute the voltage of neurons and if a spike is emitted, and it is used if the goal is to obtain a high value of accuracy of the SNN.
- The second one is the Integrate and Fire model. This one uses a very simple equation 2.2 to evaluate the state and the emission of a spike.

  $$V(t) = V_r + \frac{1}{C}\int_{to}^{t} I(s)ds \tag{2.2}$$

  The output is calculated doing a sum between all value connected to emitting neurons. This one is used in case of high speed SNN without high accuracy necessity.
- The third one is the Leaky-Integrate and Fire and is a natural trade-off between the two previous models. It is easy to implement like the second one but implement also a subtraction that models the losses, or better the leaking of the neurons.
  This last one will be implemented as the default model in the accelerator but, with an easy modification, the accelerator can use the second model.

2.3.2 Coding of signals

In SNN the signal are codified in a spike train but exist several types of coding. Usually, the input signal will be converted into a series of impulse depending on the intensity of the signal (audio or video). It is necessary, therefore, to define a time-period or time-window. The latter is defined in terms of numbers of step time. Increasing the number of step time in the time window the conversion is more accurate. This accuracy has like the collateral effect the slowdown of the network drawback. Each new step time added means that, to obtain the result from the network, it is necessary to execute the whole network one time more. This conversion process of the input can be avoided if instead to use a standard camera, the image acquisition is done by a DVS camera. In a DVS camera, or event camera, each pixel reports changes in brightness independently from the others.

2.4 Accelerators

Several accelerators have been developed to implement SNNs with dimensions. Some of the most popular are listed and described below

2.4.1 SpiNNaker

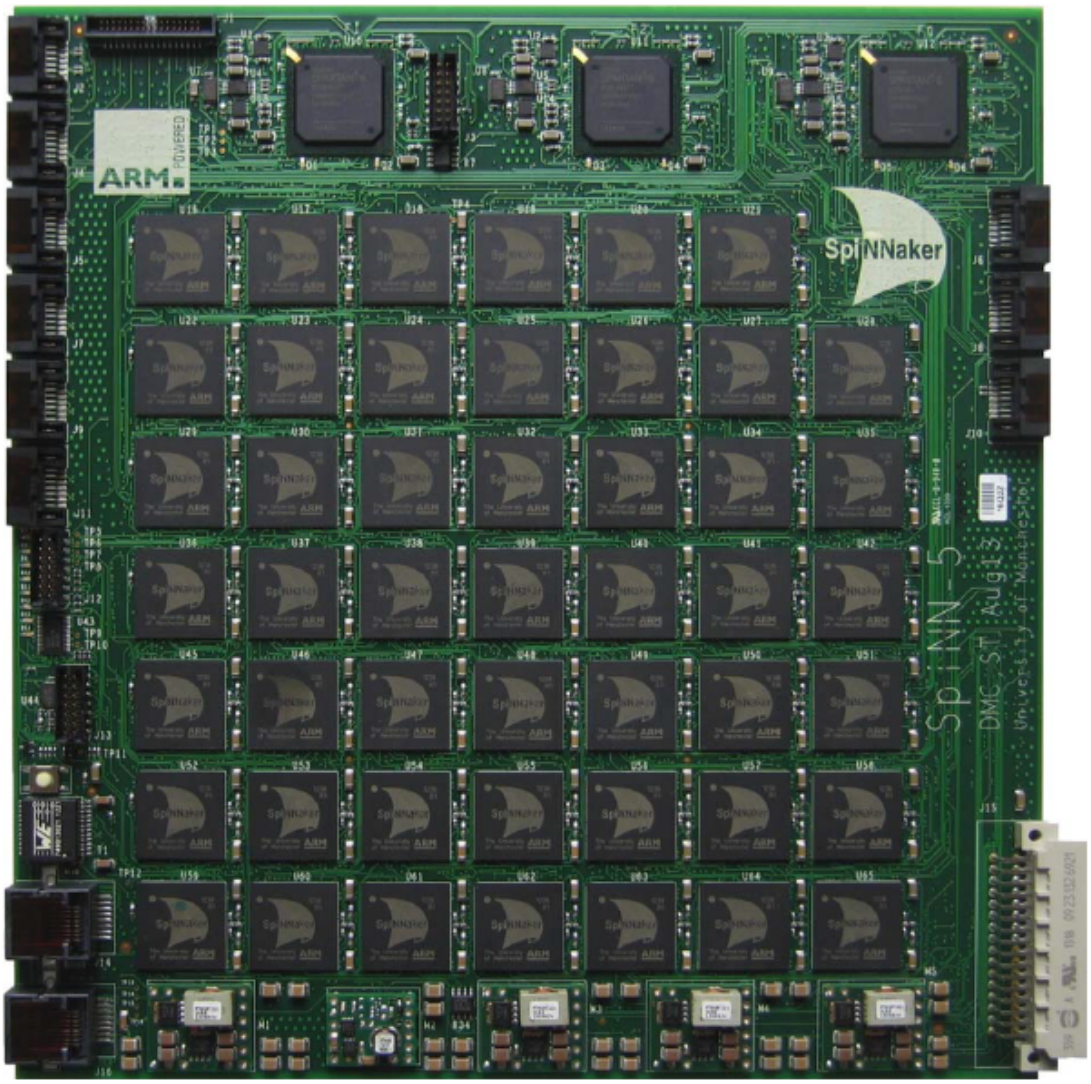

Figure 2.9: SpiNNaker PCB

It can manage up to a million of neurons. The innovative aspect of this accelerator is the efficient communication. It is optimized to send a large number of message with "reduced" bit parallelism. Each node communicate with the others with messages of 72 bits. It can handle different type of neuron models, including the ones cited before in 2.3.1. It does not use a global memory but instead, the cores ARM-9 memory can access to small local memory. The circuit board shown in figure 2.9 incorporates 48 SpiNNaker packages with a total of 864 ARM968 processor.

2.4.2 TrueNorth

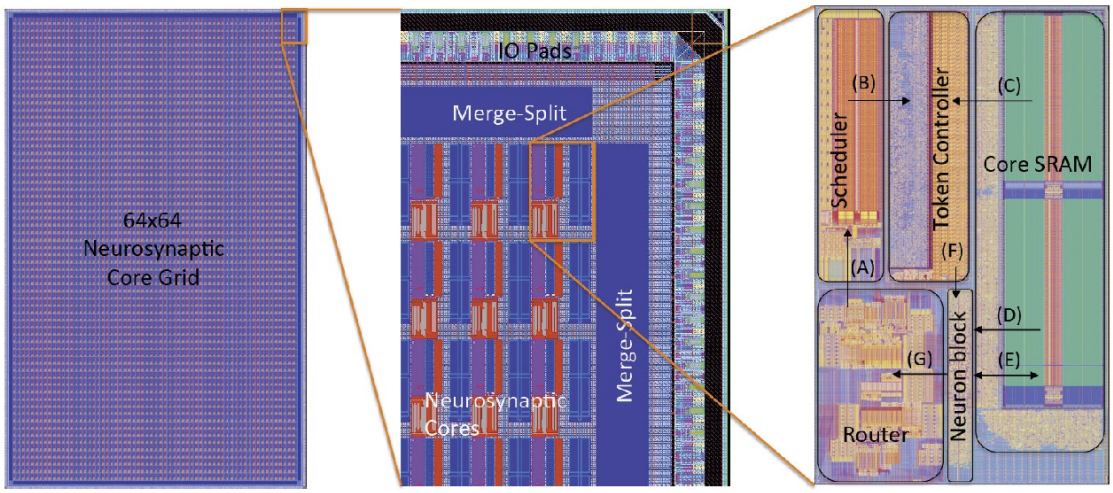

Figure 2.10: TrueNorth layout

The layout of the TrueNorth is shown in figure 2.10. It is composed of a grid 64x64 cores each one presents a scheduler, an SRAM, a router, a block of neurons and a control. It can simulation a million of neurons exactly like SpiNNaker and 256 million synapses. It consumes 100 mW for classification tasks. It has a parallelism equal to 4096 cores and actuate time-division multiplexing in order to reduce area. It handles only LIF neurons. Data parallelism are 9-bit signed weights for neurons. The time window can vary in a range that goes from 1 to 15 time-steps.

2.4.3 Loihi

It was developed from Intel in 2018 using a 14 nm technology. It supports about 130 thousands of neurons and 130 million synapses. It consents the adaptation of the threshold. This accelerators supports the variable resolution for synapses going from 1 to 9 bits. In figure 2.11 is reported the microarchitecture of the cores.

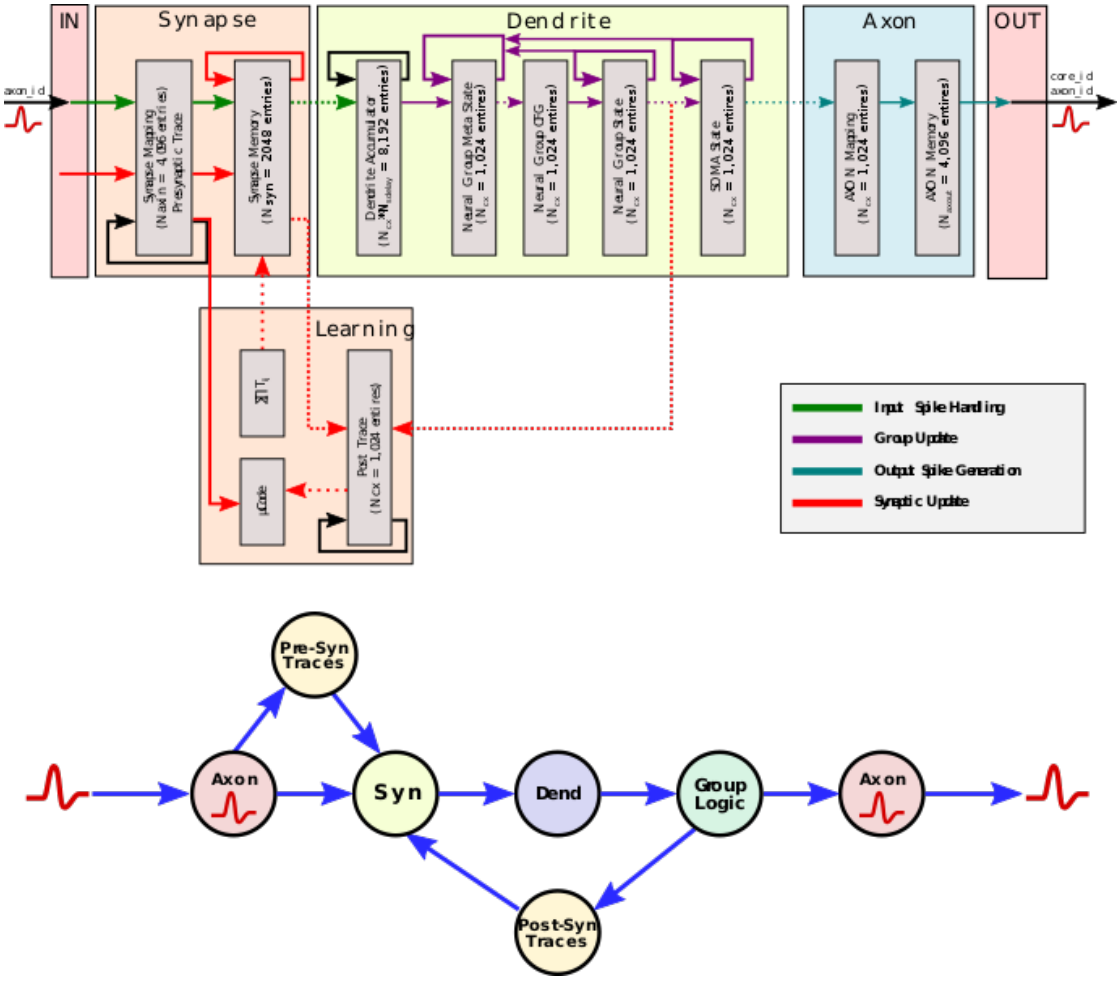

Figure 2.11: Loihi Microarchitecture

2.4.4 Comparison

	SpiNNaker	TrueNorth	Loihi
Max Number of Neurons	1B	1M	130k
Number of Synapses	80M	256M	130M
Weights resolution	1-40	9	1-9
Others Features	It consumes few nJ in nominal condition. Can execute real-time simulation with 72-bit messages	Consume 26 pJ for each operation. Can be used only for inference because the support of learning is absent.	In nominal condition, it consumes 23.6 pJ and can learn using different types of on-line learning.

Table 2.2: Comparison SNN accelerators

From table 2.2 is possible to notice the main characteristics of these three accelerators. For example, the max number of neurons that goes from 130k of the Loihi and could reaches 1 billion in SpiNNaker one. Also the number of synapses is over around the one hundred millions. The reported number of synapses in SpiNNaker referenced the example reported in original paper. Another advantage of SpiNNaker is the high resolution that could be implemented, up to 40 bits, compared to the 9 bits that the other two can support. In terms of power all the accelerators have low values but, at least in nominal case reported in paper, the SpiNNaker is the best one.

Chapter 3

ZEDBoard: The chosen FPGA

3.1 Overall illustration of ZEDBoard

The FPGA is a configurable integrated circuit that represents a perfect trade-off between the efficiency of hardware and the flexibility of software. It is composed of a set of virtual logic gates configurable directly by the user. The chosen board is the ZEDBoard that has on it a System-on-Chip Zynq™-7000 AP SoC XC7Z020-CLG484-1 which include Dual-core ARM Cortex-A9 MPCore™ with CoreSight™ and XC7Z020 Xilinx SRAM-Based FPGA and a DDR Memory with a capability of 512 MB. These three parameters were instrumental in the choice of the board in addition to the high number and type of the peripheral like switches, HDMI connectors and so on. The main characteristics of the FPGA that ensures a big and fast accelerator are listed below.

- Programmable Logic Cells = 85000
  These are necessary to implement any kind of combinatorial logic function, with mixing of 6 input Look-Up Tables (53200), and sequential block, with Flip-Flops (106400).

- Block RAM = 140
  They are necessary to create a local cache memories big enough to avoid the continuous transfer of data to and from the DDR Memory. The total amount of the internal memory that could be used is equal to 4.9 MB because each Bram has a capacity of 36 kb.

- DSP Slice = 220
  These blocks are optimized to execute addition, subtraction, multiplication

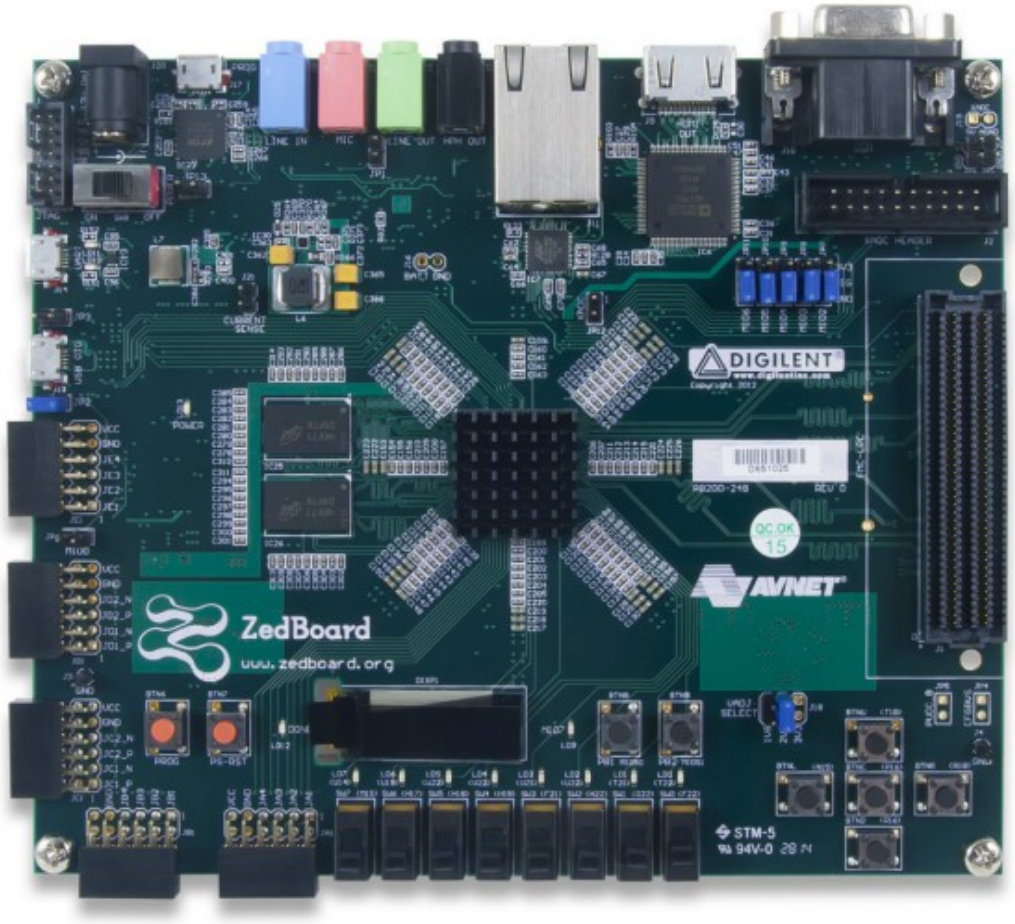

Figure 3.1: ZEDBoard

or a mix of these operations in a fast way and reducing power consumption without using LUTs.

- I/O Blocks= 200 These represent the number of input or output pin with control components (like tristate, diodes,..)to guarantee compatibility between inside and outside and avoid the high current value go inside the FPGA and potentially destroying it.

This values will influence the design of the accelerator and the future configurations that will be developed.

3.2 Why use FPGA and don't create an ASIC?

Application Specific Integrated Circuit is created to solve a specific computation with less power consumption and/or reaching the maximum speed. This solution is often the best but after having produced it nobody could make any change and for this reason, an FPGA is used. The goal of this thesis is to define an architecture that the user could modify in some parameters related to the specific case leaving everything else intact. The FPGA gives to the future user a high degree of flexibility but with good efficiency in terms of timing and power consumption and especially throughput compared to a CPU due to the sequential behaviour of the last one.

3.3 Vivado

Xilinx provides also a software to make easy the realization and integration of the accelerator in FPGA. The name of the program is Vivado and the version used is the 2018.3. The interface of the program is shown in figure 3.2.

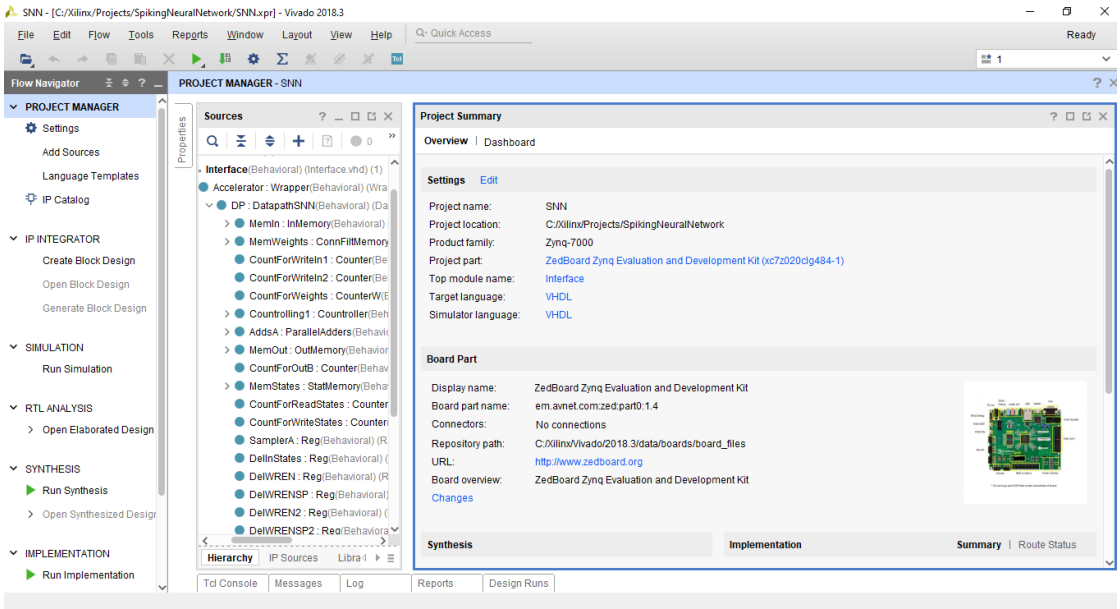

Figure 3.2: Vivado inteface

This software is useful in several aspects of the design. It provides in addition to the text editor, to write hardware code, also an IP Catalog shown in figure 3.3, where the user can select IPs already optimized by Xilinx.

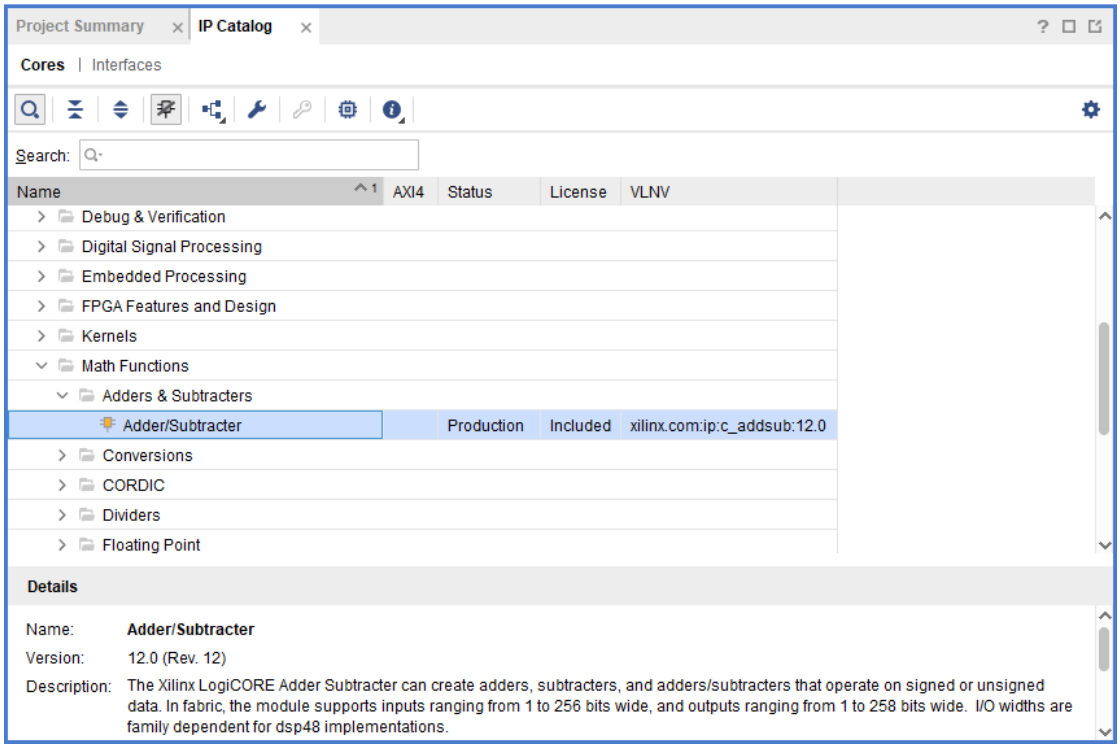

Figure 3.3: IP Catalog

IP catalog contain several useful IP for any need. It is possible to select between complex arithmetic units (multiplier, divider, ...) or communication devices like for ethernet or other IEEE standards. For example, in my architecture will be used adder and memories with all the needed controls. Adder and one memory can be seen in figure 3.4 and figure 3.5. From the graphic interface of IP the user can set the desired configuration in terms of data parallelism and controls like the bypass signal or the internal pipelining, or for memory the initialization. These IPs are easy to integrate with the code written by the user.

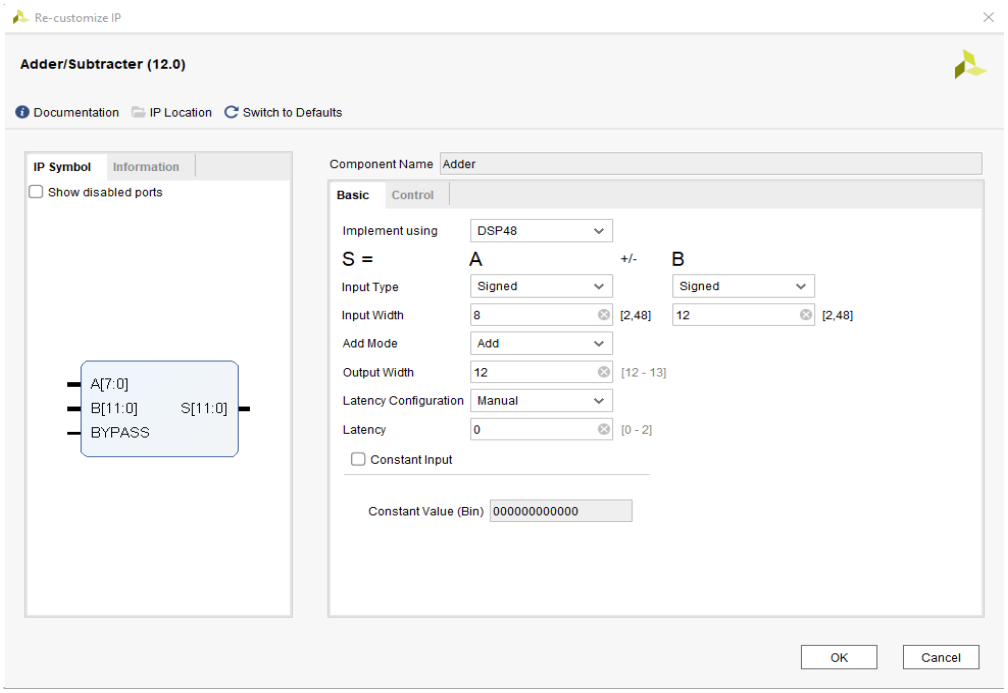

Figure 3.4: Adder IP

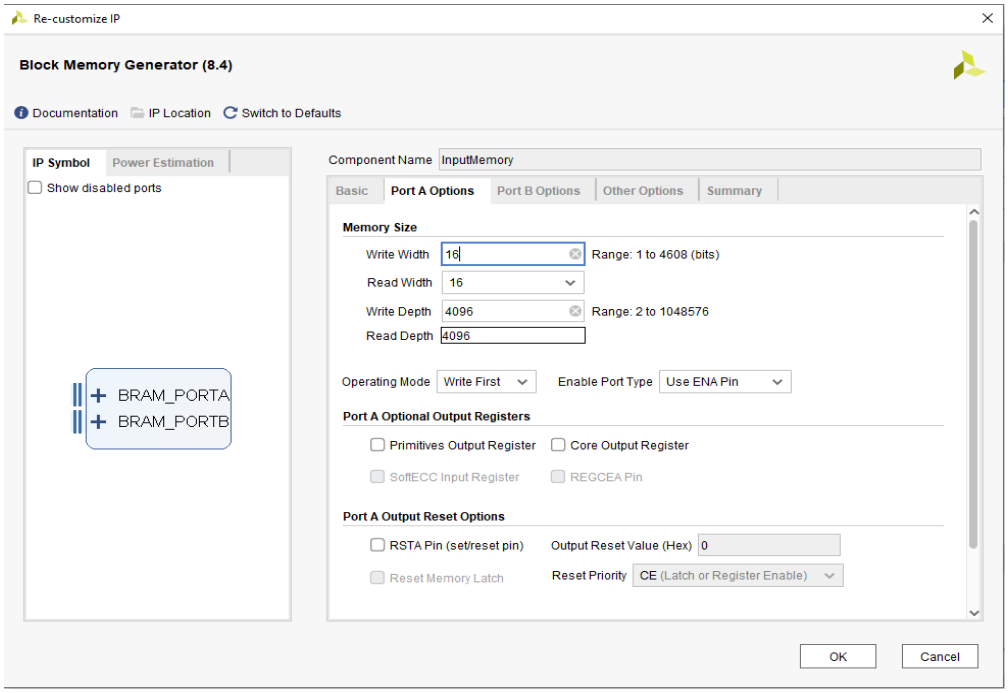

Figure 3.5: Memory IP

In the figure 3.2 it is possible to see on the left side all section necessary to execute the digital design flow. It has, therefore, the sections to execute the simulation, the synthesis, the implementation and the one to generate the bitstream. Vivado has its own simulator, that will be used in this thesis, in addition to the possibility to use the common ones like ModelSim. To force synthesis and implementation can be created file .xdc where all the constraints are listed about timing, area, power and even I/O.

Chapter 4

Design of the SNN accelerator architecture

The goal of this thesis is to design an efficient architecture for a hardware accelerator able to compute all the necessary operations in an SNN inference. The developed circuit can elaborate on how spikes will propagate through the SNN analyzing the input spikes coming from the external like a specific camera and the current state of neurons and compute also the new states of neurons. This behaviour must be ensured regardless of the type of layers that could be, in an SNN like in a DNN, convolutional, pooling or fully-connected. Another important factor that must be taken into account is the generalization ability of the hardware accelerators. Different SNNs require different sizes of the filters and its number can vary from the usual a power of 2. This accelerator supports potentially any number of filter and any dimension of filter. Anyway, it is suggested to choose a quantity of filter equal to a multiple of parallelism to avoid to waste space in memory.
In this chapter will be shown how the algorithm has been developed considering all the constraints, making it as much generic as possible, and how it is optimized knowing the features and the limitations of the chosen FPGA.

4.1 Algorithm

4.1.1 Convolutional layer

How it works A convolutional layer is used to extract some specific features from an image, like edges or shapes. It takes its name from the mathematical operation called convolution. In this layer, it is necessary to define a set of learnable filters. After the training of these filters, the next step is the inference process where the convolution is executed between the input area and one of the filters. From this result is possible to understand if the wanted feature is present or not in that portion of the input image.

To obtain the next state and the possible spike event of every single neuron in the convolutional layer, all the inputs that are contained in a specific area must be analyzed to add specific weights with the current state. This operation requires an accumulation of value until this area is over. The area is defined by filter dimensions and the input channels. Pseudocode of convolutional layer is reported in figure 4.1

```
for g = 1:G                                                    % Loop on rows of input
    for h = 1:H{                                               % Loop on columns of input
        for i = 1:I{                                           % Loop on filter series
            for j = 1:J;                                       % Loop on rows of filters
                for k = 1:K;                                   % Loop on columns of filters
                    for l = 1:L;                               % Loop on input channels
                        NewState = OldState + Weight_jk * Spike_jkl;
            Store;
            Change series;
        }
        Apply Horizontal Stride;
    }
    Apply Vertical Stride;
}
```

Figure 4.1: Convolutional Pseudocode

The flowchart instead is reported in figure 4.2 explain every single step to execute a whole convolutional layer, or a partial part of it, independently of the dimension of the filter and the image. To understand better all the passages have been created also a sequence of images the show graphically the main steps. These images, for the sake of simplicity but also completeness, consider the following parameters.

- Image Dimensions= 10x10
- Input Channels= 7
- Filter Dimensions = 2x2
- Number of filters = 6
- Parallelism = 4
- Stride(Horizontal and vertical) = 1

In all the images is shown the selected input data, the selected filter values, the partial sum and the storing in output when it happens.

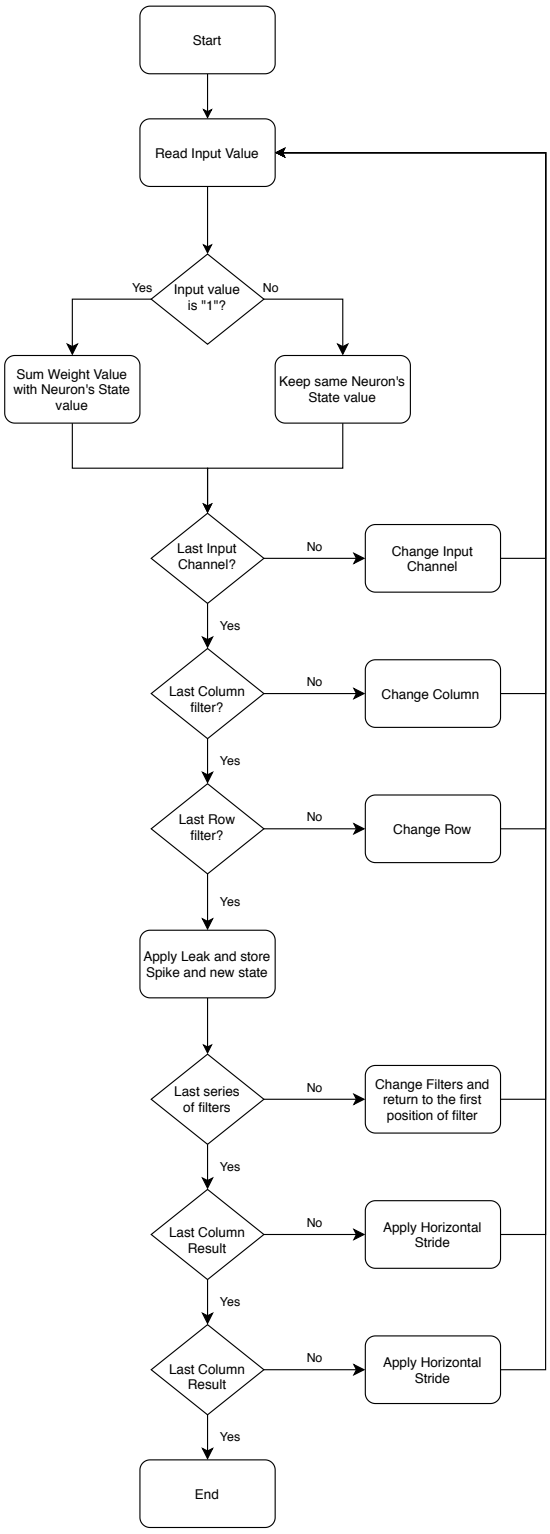

Figure 4.2: Convolution Algorithm

This first couple of images in figure 4.3 show how the input value selected change from one input channel to another because the last input channel has not yet been reached. It is possible to see that the four filters selected (four is due to the parallelism chosen) will keep the same position, that in this case is (0,0) and the movement is along the z-axis of the input and imply that the four partial sum will be incremented by the value of related filter if the new data selected is "1" otherwise will keep the previous value. If the image analyzed has only one channel this step will not exist and the algorithm moves to the next directly.

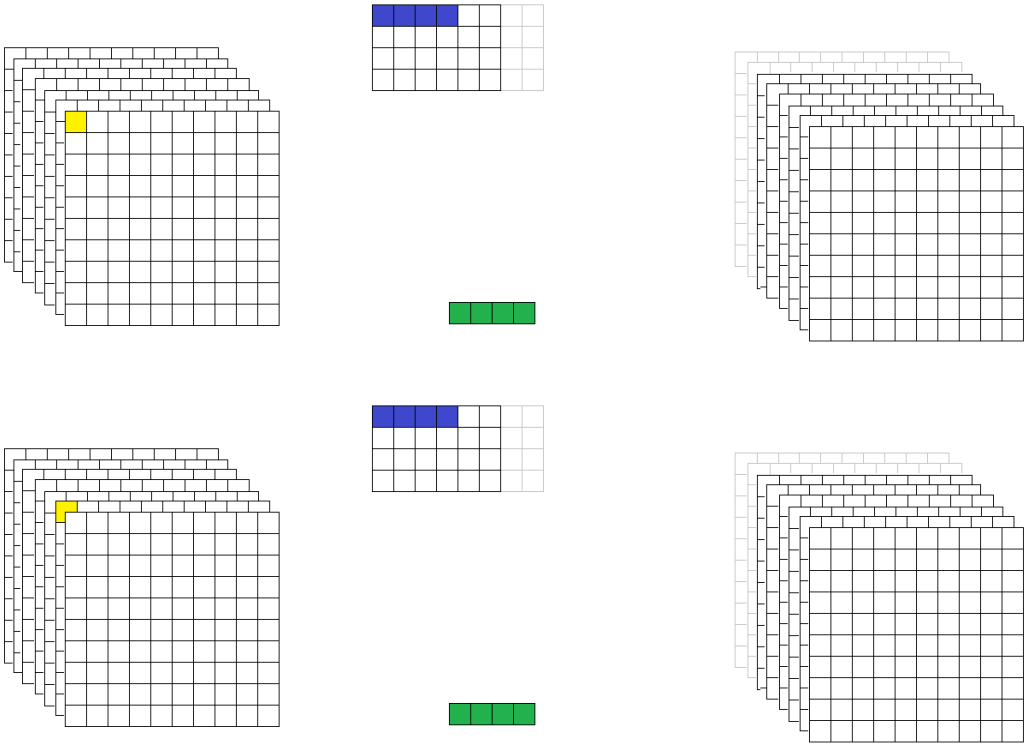

Figure 4.3: Changing Input Channel

This second couple of images in figure 4.4 shows the case of last Input channel selection and the passage to another column. It's possible to see that not only the input data will change but also the position of the filter moving to the position (0,1). This position of filer will remain fixed until all the input channel will be scanned. The four partial sums will continue to increment by the value of related filter if the new data selected is "1" otherwise will keep the previous value.

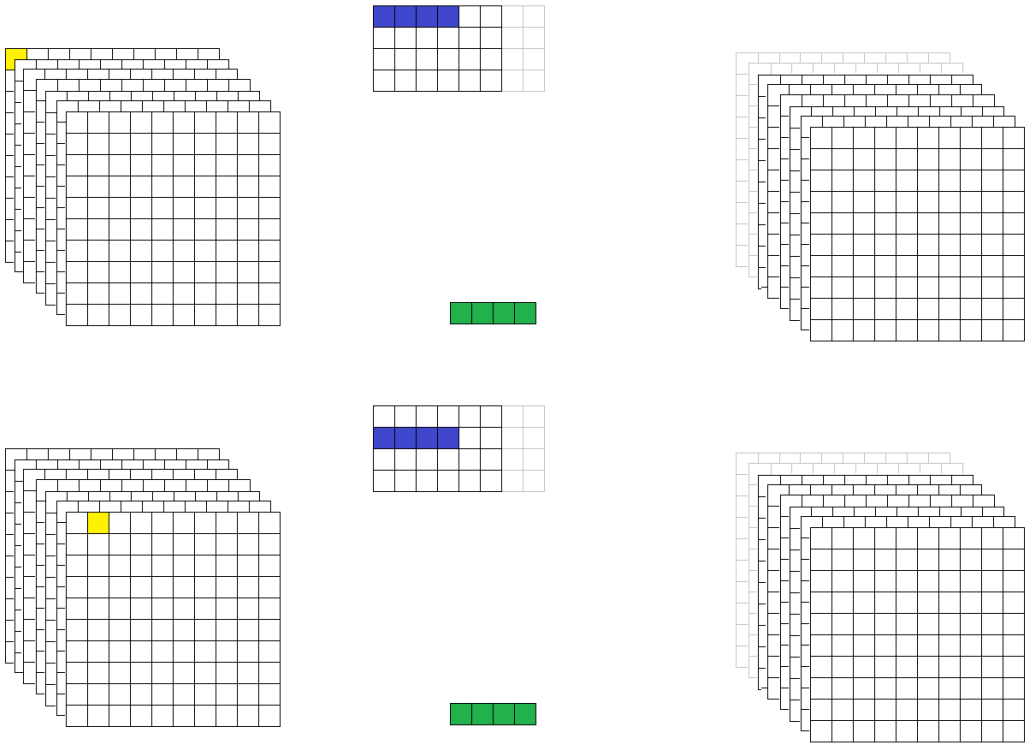

Figure 4.4: Changing Column

This third couple of images in figure 4.5, instead, shows the case when the selected data is in the last column filter and also in the last input channel so, consequently, the next position is in the other row resetting the value of column and channel. Like in the previous case also the position of filters will change moving from (0,1) to (1,0). The four partial sums will continue to change or not their value, following the previous rules, based on input values.

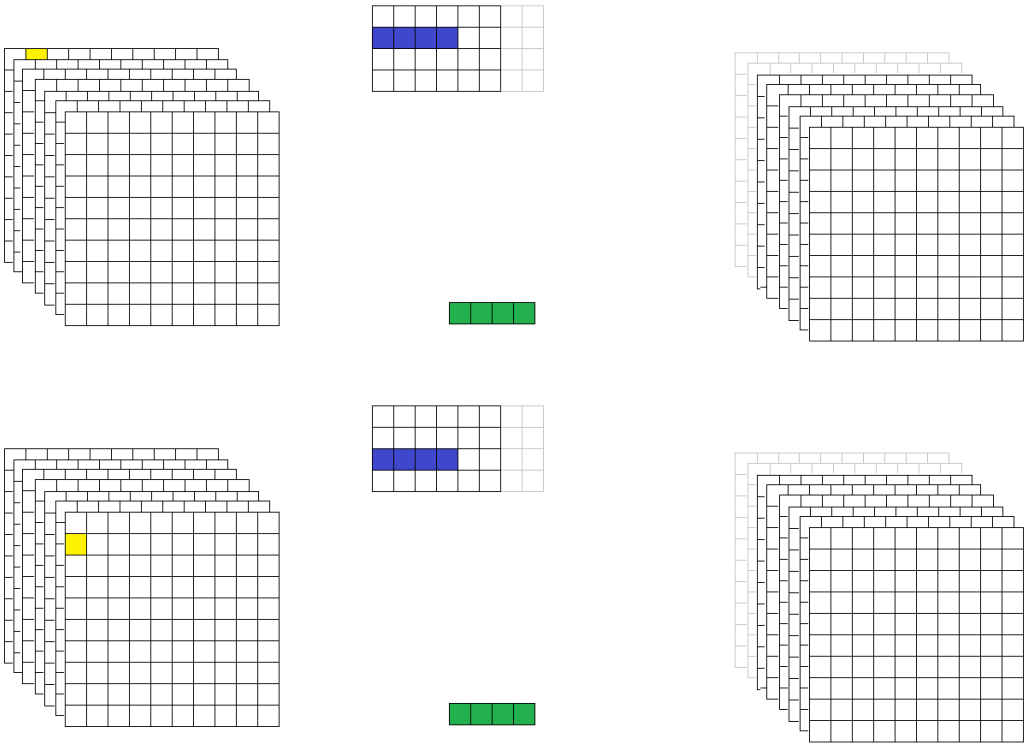

Figure 4.5: Changing Row

This fourth couple of images, in figure 4.6 and figure 4.7, shows the last part of elaboration to obtain the single neuron's state and the emission or not of the spike (in this example we analyze four neurons simultaneously due to the parallelisms). In the first of two images (fig. 4.6), we could see that it is reached the last value of input the contribute to the value of first P neurons, where P is the parallelism, so the partial sums could be sent to output or stored in memory. This would be correct if the model used does not take into account the leaking voltage and the partial sum must be modified to include also this contribution. This is shown in the second figure (fig.4.7) where it is selected only the leak and partial sums and neither the input nor the filter.

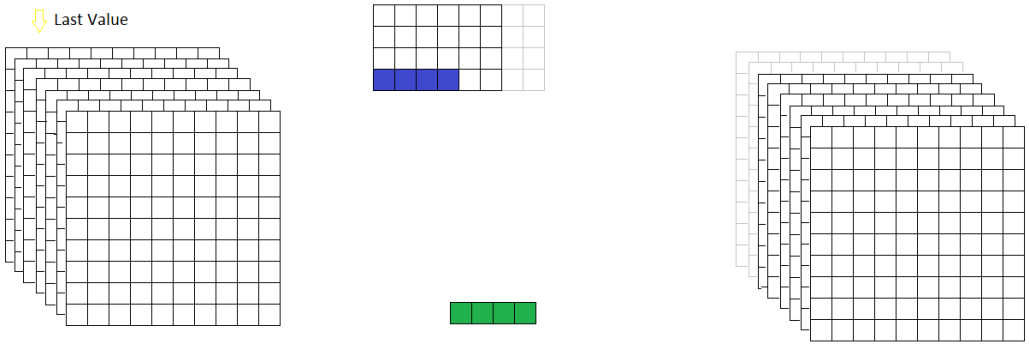

Figure 4.6: End of filter

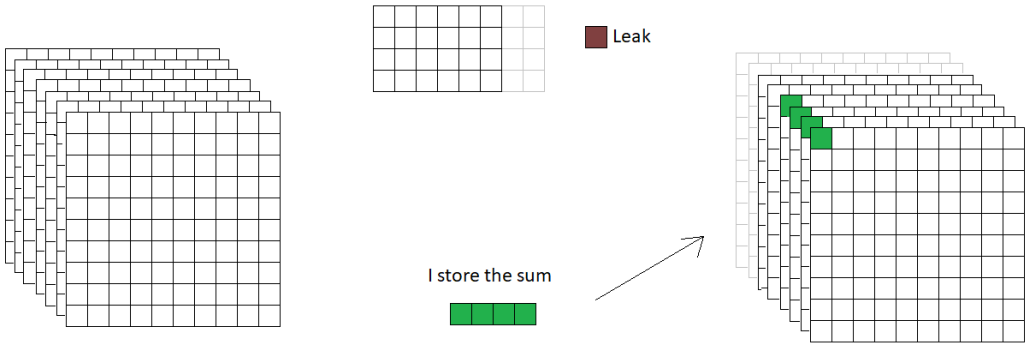

Figure 4.7: Leak and store

Before to store the value of the state must be applied the activation function to understand if a neuron emits a spike or not. The activation function used in this thesis is called ReLU that stands for Rectified Linear Unit. It is the most used one in ANN has several advantages.

- The derivative of the ReLU is one. This implies that during the training is not necessary extra time so ReLU is more faster than the other activation functions.
- ReLU is not bounded. This implies that it does not suffer from the vanishing gradient problem.

From the graph below it is possible to see the transfer function. On the x-axis is present the value computed of the state for the neuron and on the y-axis can be found the stored value and if a spike is emitted. When the computed state is negative the stored value will be zero and no spike will be emitted. When the computed value instead is positive but does not overcome the threshold the stored state will be equal to the input and also in this situation no emission of a spike. In the third and last case, the input exceeds the threshold so it means that a spike is emitted and the new state is equal to zero.

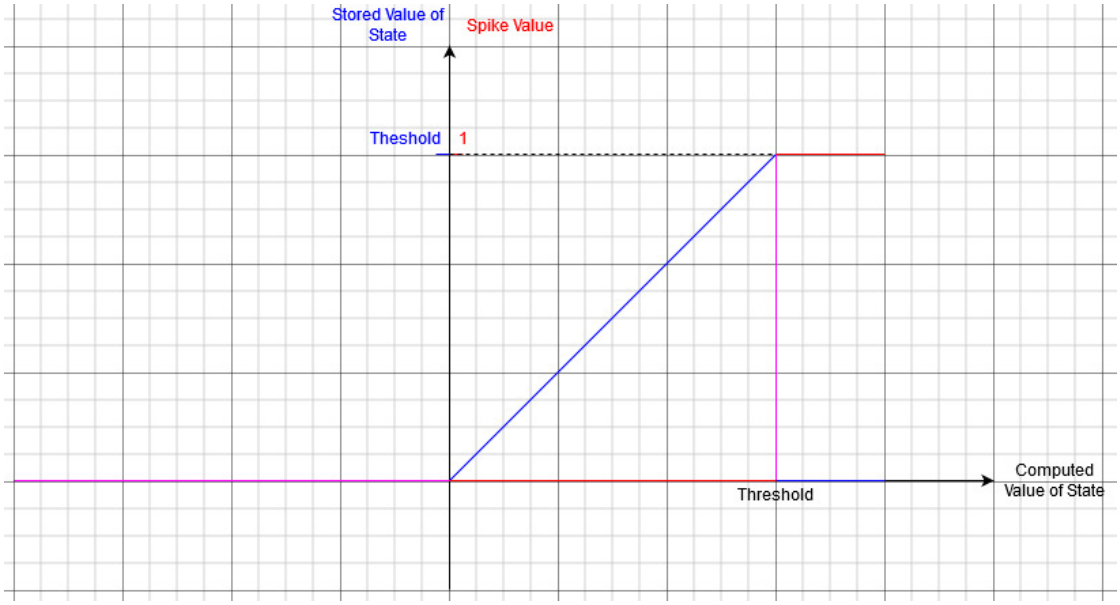

Figure 4.8: Transfer Function ReLU

After the storage, the algorithm must go on and the next movement depends on the series of filter that was selected before this step. At this point, in the example, we computed the first four filters so the other two must be executed now. The algorithm, in this case, will restart from the first position where the previous filter where located but changing the filters selected. This step is shown in figure 4.9. It is possible to see that are present four remaining filters, and not two as one could expect but paying special attention only two of them had the black borders. The colour difference is due to a different type of filters, the black border one is a real value of a real filter that acts in this layer the grey border one is a "ghost" value of a "ghost" filter.

Ghost Filters

To avoid restriction for the user in the number of filters in a single layer could happen that if the number of the filters is not a multiple of the parallelism, like in our case, the remaining filters are less than the parallelism chosen so it is necessary to add these "ghost" filters. The "ghost" filters do not act to extract some features but are all zero. Their contribution in terms of timing is null because are in parallel with real filters. Also, the contribution in terms of dynamic power is quite null because they don't change any bit also when the input bit is "1", indeed, the partial sum will sum with a zero. It is present the static contribution but can not be avoided on FPGA. In terms of area, contrary to what one might think, their contribution is also zero because the width of the memory is fixed so that space is already present on-chip. Another big advantage of this "ghost" filters is that do not increase the transfer timing because the ghost values (all zeros) are on the same word of the values of real filter but instead reduce the control. To avoid incorrect output writes is not necessary extra control circuitry because the previous values stored in that position of memory, related to a previous layer, are overwritten with the zeros of ghost values.
The image in figure 4.9 shows, how explained previously, that the first value of input selected is the same but the filter selection switch to the other four. In this adder is not in accumulation mode, because are the first sums, but the partial results will be the sum between the current states value of the neurons and filter weights if the input value is equal to "1" otherwise the current state will be kept.
Once that the algorithm changes the selected filters, the rest moves as seen before which is first moving along the z-axis changing input channels than change column and row in the right moment arriving at the storage of value. Obviously during the execution of successive steps until the store, the adder will be in accumulation mode between partial sums and weights/leak.

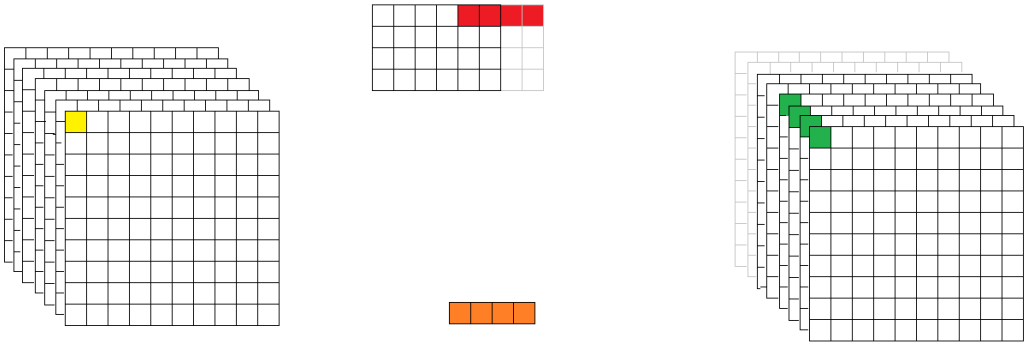

Figure 4.9: Change Series of Filters

Once that all filters have been executed and the last values are stored. The first series of filters can finally move along the x-axis by the horizontal stride. In the next figure 4.10 is shown this passage where the selected filters are first four and the input value selected is not the same before but moved by (0,Sh,0), where Sh is the horizontal stride. In this example Sh=1 so the first input value covered by the filter is in the figure is (0,1,0).

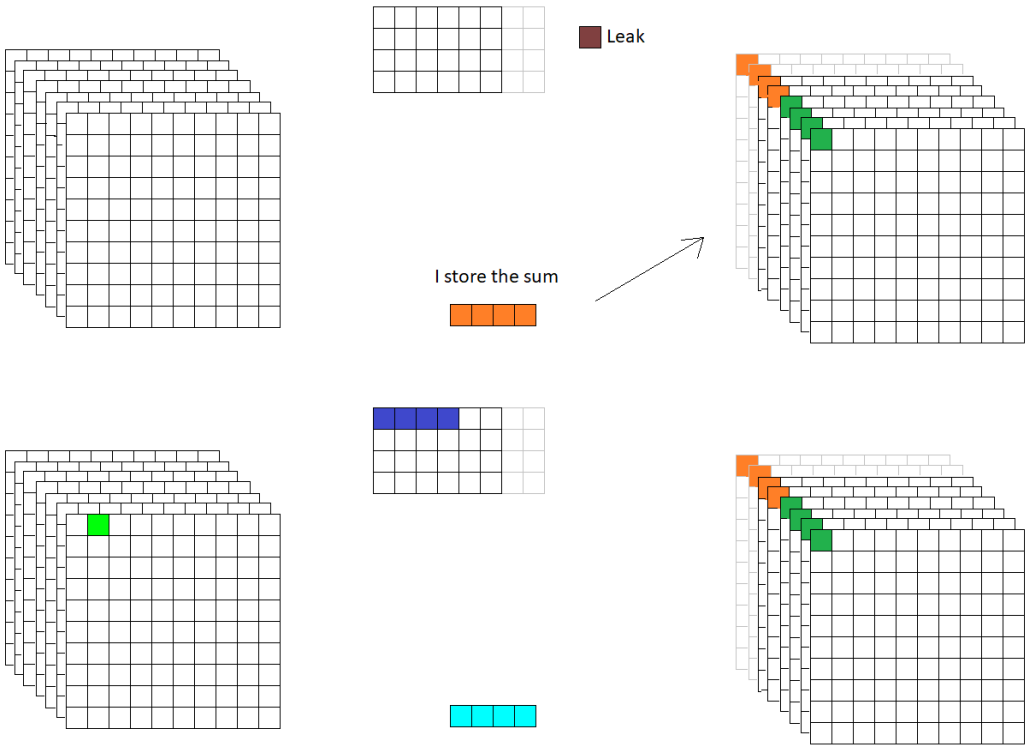

Figure 4.10: Right Shift of filter position

It is not hard to imagine the steps that follow the right shift of filter position on the input image passing through all steps described before. It is not hard also to imagine that the algorithm will move to the end of the image's columns ending the first row of results and at that moment the algorithm will change the row of the input. This action is illustrated in the following images in figure 4.11.

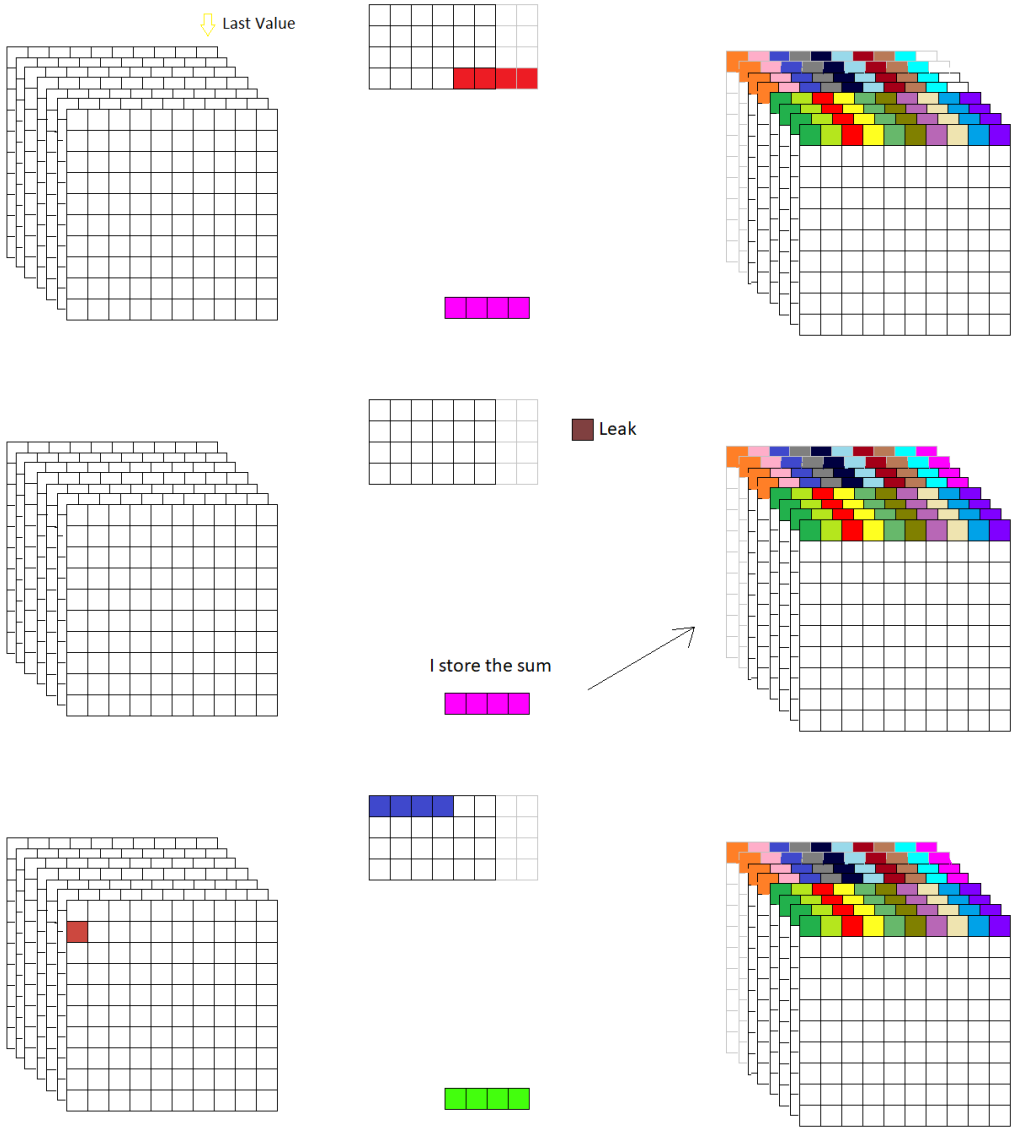

Figure 4.11: Down Shift of filter position

The algorithm will proceed, executing these steps at the right moment until the end of the store of the last values.

All these passages, done in this way, provides a lot of advantages that are listed below

- First of all using this procedure the memory word is written only one time storing the correct state and spike value. In this way is avoided the storage of any partial sum and move to another word for other neurons forcing then the update of every time the neurons.
  This reduces the commutations inside the memory that as is a huge load so consequently the dynamic power consumption.

- Another advantage not negligible is that once that a neuron is stored in output memory could be sent to the external environment. These output values can be potentially used by another accelerator if present and free increasing the speed of execution.

- Last advantage is that in this way it's preserved the spatial location of data through the layer. The input and output memory have the same order of data. Data with the same xy position but from different input channels are stored one after the other. Then the next row contains data of the next column position. This will be more clear in the figure below where is illustrated the positioning in the case of an image 3x3 with eight input channels and parallelism equal to four. The numbers written in the 4.12 represent the position in memory, not the effective value.
  Keep the spatial location equals allow to do not change the value of jump address for the input memory when layer change. The input address jumps for example when it is reached the last column filter. Looking, for example, the case draws below if the filter is 2x2 after the position 15 located in the fourth row the next value is in position 24 located in the seventh row and not in the fifth. This case described above represent the situation of the last column filter and last input channel seen in figure 4.5

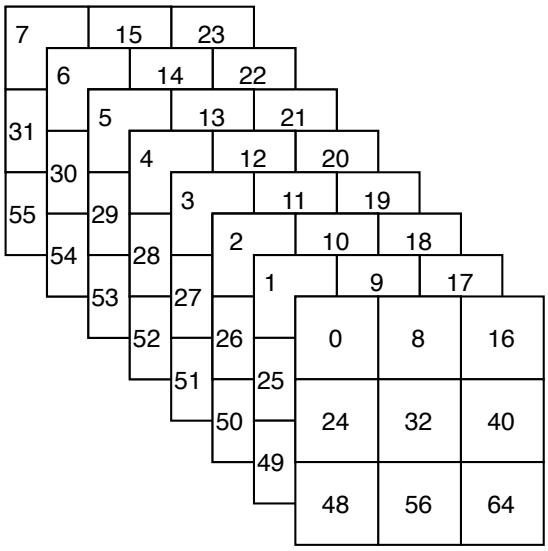

Figure 4.12: Positions of image

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15
16	17	18	19
20	21	22	23
24	25	26	27
28	29	30	31
32	33	34	35
36	37	38	39
40	41	42	43
44	45	46	47
48	49	50	51
52	53	54	55
56	57	58	59
60	61	62	63
64	65	46	67
68	69	70	71

Figure 4.13: Positions in memory

4.1.2 Pooling layer

How it works The pooling layer is used to reduce the complexity of network. After a convolutional layer, the number of neurons change reach unmanageable values so a pool layer group a fixed number of inputs in a single output value. This operation can be done in two different ways. The first one is the average pooling where the result is the average of input values. The second one is the max pooling where the result is the max of input values. Both types generate a loss of information but ensure a substantial reduction of neurons. The pooling layer, equal to convolutional one, uses a moving filter that select inputs but acts on each channel singularly.

In this layer is present only one filter done by the same constant value for each position, so the to exploit the parallelism that hardware accelerator could offer, data extracted from input memory are equal to the parallelism. This operation is done to improve the speed and don't waste static power of the unused adders. Besides, in the case of pooling layer, data from different channels will not be mixed but must remain separate. Therefore, to keep the same advantages of the previous type, it is necessary to change the order of some loops. Pseudocode of pooling layer is reported in figure 4.14

```
for g = 1:G                                                          % Loop on rows of input
    for h = 1:H                                                      % Loop on columns of input
        for i = 1:I {                                                % Loop on input channels series
            for j = 1:J;                                             % Loop on rows of filters
                for k = 1:K;                                         % Loop on columns of filters
                        NewState = OldState + Weight (costant) * Spike_jk;
                        Store;
            Change series;
        }
        Apply Horizontal Stride;
    }
    Apply Vertical Stride;
}
```

Figure 4.14: Pooling Pseudocode

Several aspects described before can be understood better from this sequence of images. Firstly, the number of input values that is equal to the parallelism and the filter values equal to a constant (the specific values will be defined better in the architecture section 4.2). The second difference is the moving of the input data selected. It is possible to notice that first moving are that ones that had the goal to end as soon as possible the area covered by pooling filter and not go through the z-axis ad happen in the previous type of layer. The condition for the sums instead remains equal so if the newly selected input data is "1" the constant will add to state otherwise the value computed before will be kept.

Also for this case to understand better all the passages have been created also a sequence of images the show graphically the main steps highlighting data selected for the sums. These images, for the sake of simplicity but also completeness, consider the following parameters.

- Image Dimensions= 10x10
- Input Channels= 7
- Filter Dimensions = 2x2
- Number of filters = 1
- Parallelism = 4
- Stride(Horizontal and vertical) = 2

The only difference in parameters is the value of stride equal to 2 to avoid overlapping in addition to the already cited numbers of filter that in pooling is always one. In all the images is shown the selected input data, the selected filter values, the partial sum and the storing in output when it happens.

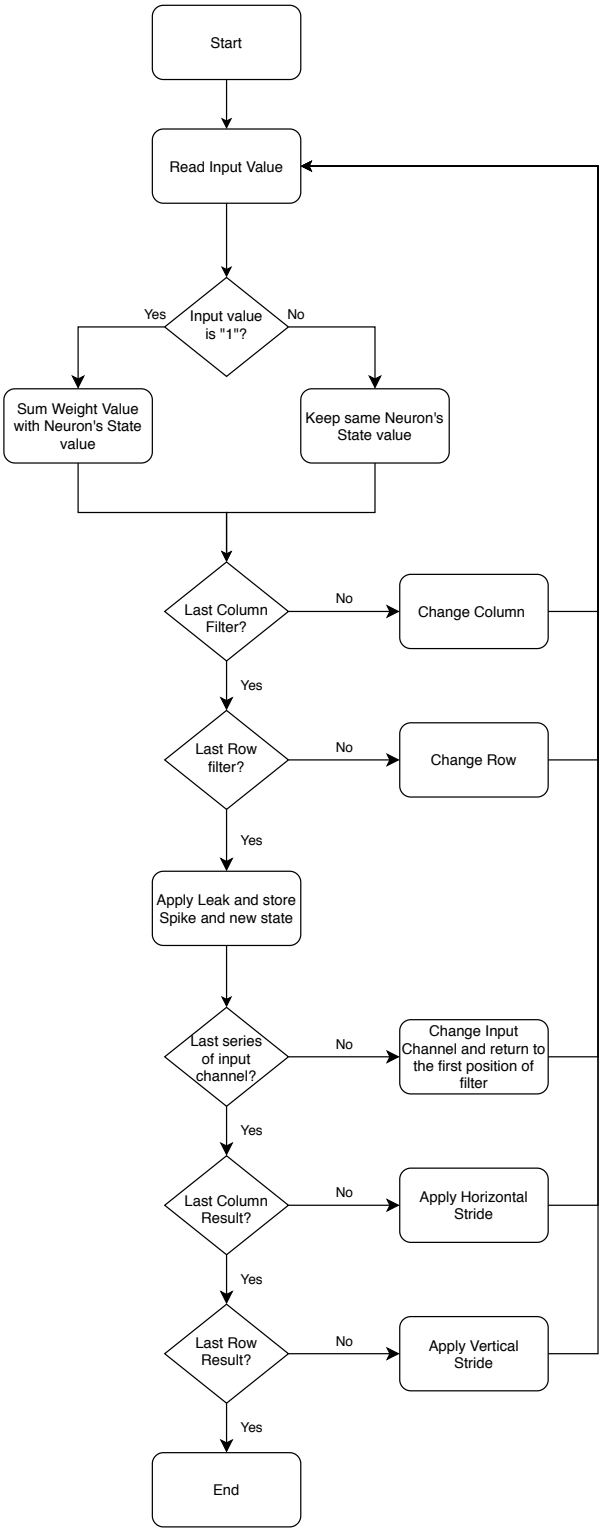

Figure 4.15: Pooling Algorithm

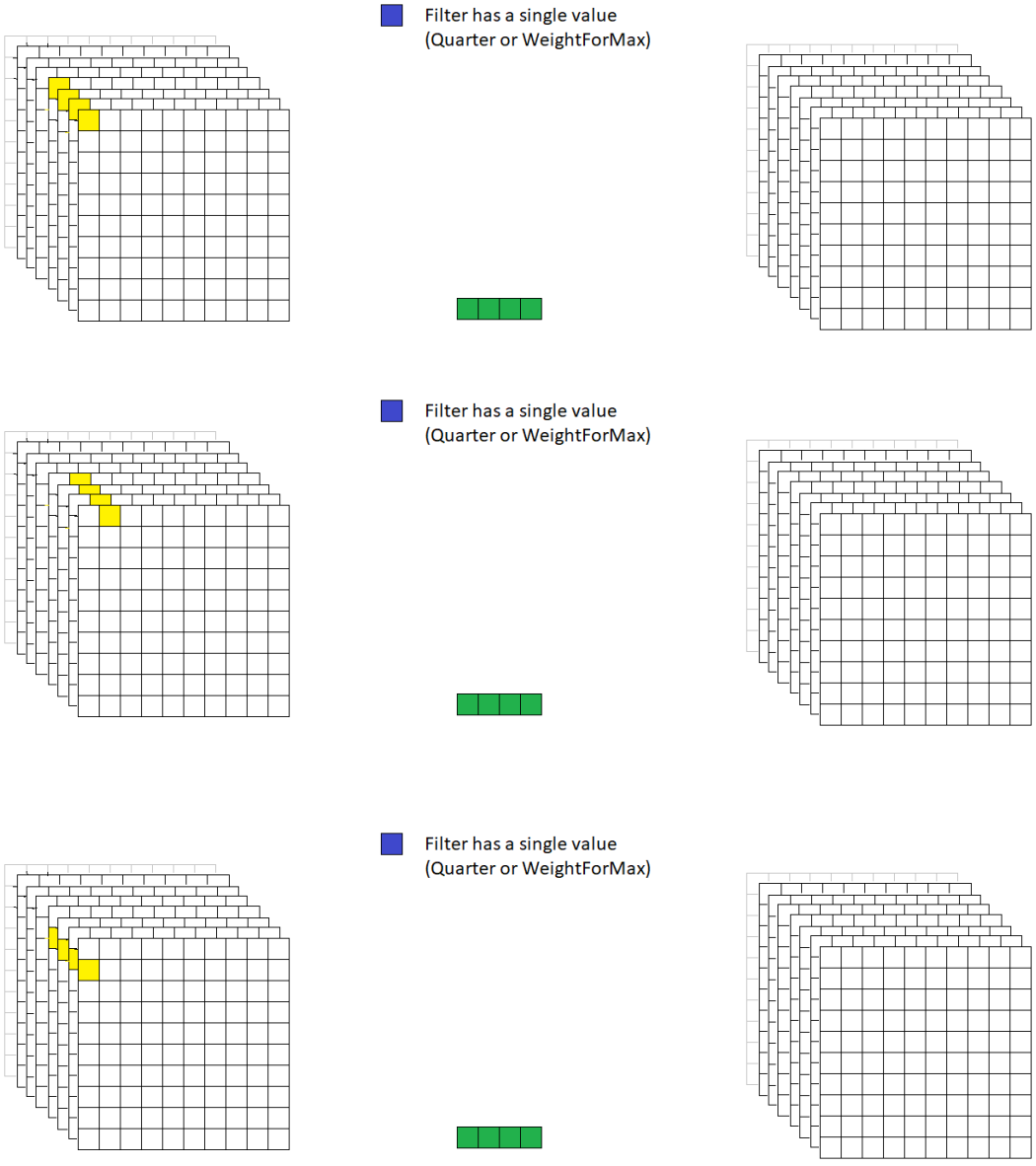

Figure 4.16: Changing Input Channel

Then in this following figure is possible to see the phase of application of leaking to neurons and the storing in output memory. This is equal to what had already been said for the previous type.

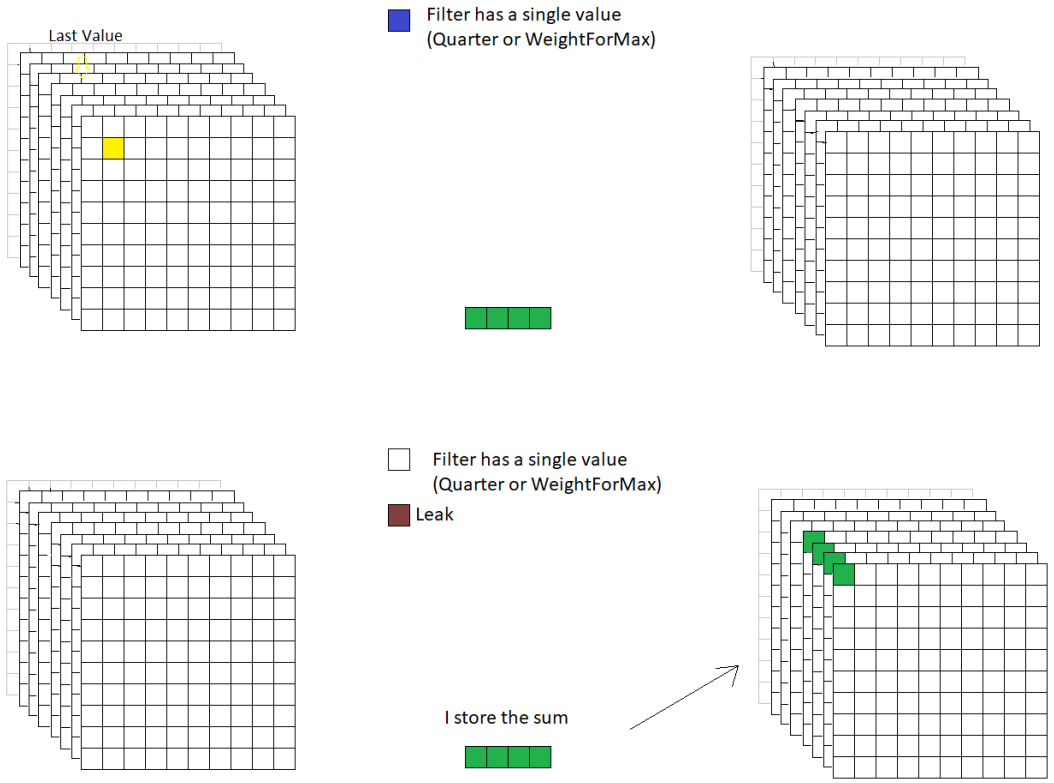

Figure 4.17: Add of Leak and storing

The next step is to change the input channel series and do the pool operation with these new data. This is illustrated the next figure where the selection shift along z-axis taking the values of the remaining three channels. Actually, the channels selected are four but the last one in the figure is grey and it represents the ghost channel.

Ghost channel

The ghost channels are present when the input channels of the pooling layer are not multiple of parallelism. This happens naturally when the number of filters of the convolution layer just before is not a multiple of parallelism, like in the example used for this section.

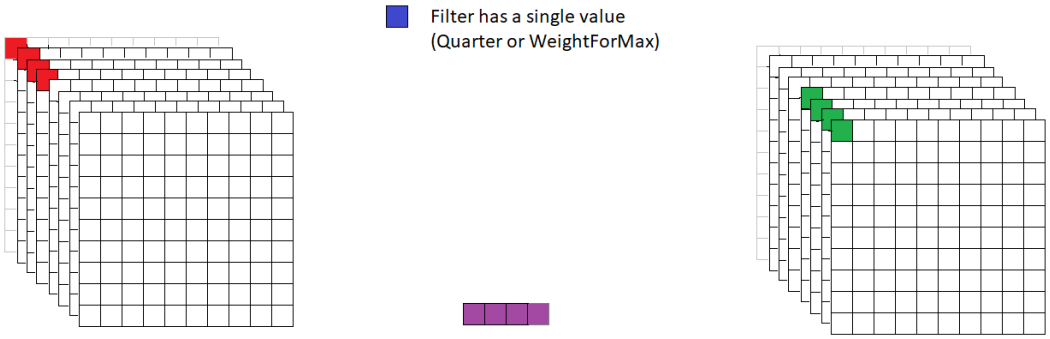

Figure 4.18: Change Input Channels

When all the input channels were used to compute first pooling filter and the related values stored it is possible to change the position of filter over the input image. Consequently, the starting point of the filter is located (0,Sh,0), where Sh is the horizontal stride, respect to the (0,0,0) used for the initial position of the filter. This passage is shown in figure 4.19 where, after the storing of the last channel, the position change applying the stride along the x-axis.
Once stored the last column of the results it is possible to change the row restarting from the first column and first input channel series. The new row selected depends on the imposed vertical stride by the user. In this example Sv is equal to two.

From this point, all these passages will be repeated sequentially in the correct order when the related condition is verified.

All these transitions are done in this order ensure the same advantage obtained before for convolutional layer.

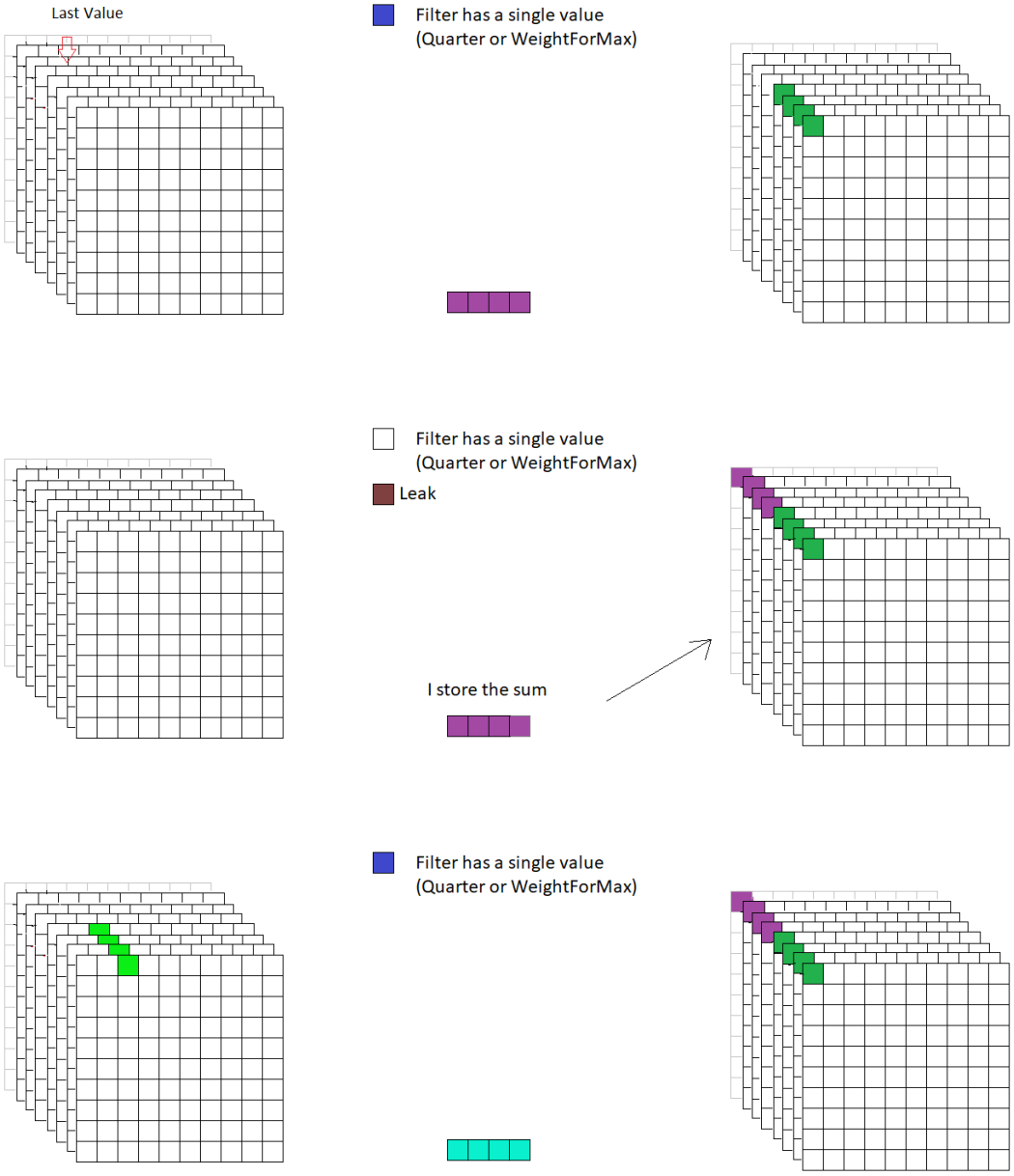

Figure 4.19: Application of horizontal stride

4.1.3 Fully-connected layer

This layer differently from the two previously described it is not used extract feature or reduce the complexity but it's used for classification. This difference falls into the possible algorithm procedures. The output neurons are not computed thanks to moving filter but considering all the input neurons and the connection weights to the single output neuron. The possibilities are two and both had been analyzed in detail before to choose the final form.

Comparison between possible algorithm

Initial idea The initial idea is to scan all the input neurons and when one emits a spike the single output neuron taken into account is updated to the new value obtained by the sum of previous state and the weight connected otherwise the previous value is kept. The scan will proceed until the end of input neurons and restart again where the same input neurons will send impulses through the layer changing the state value of the next output neuron. This is shown in figure 4.20.

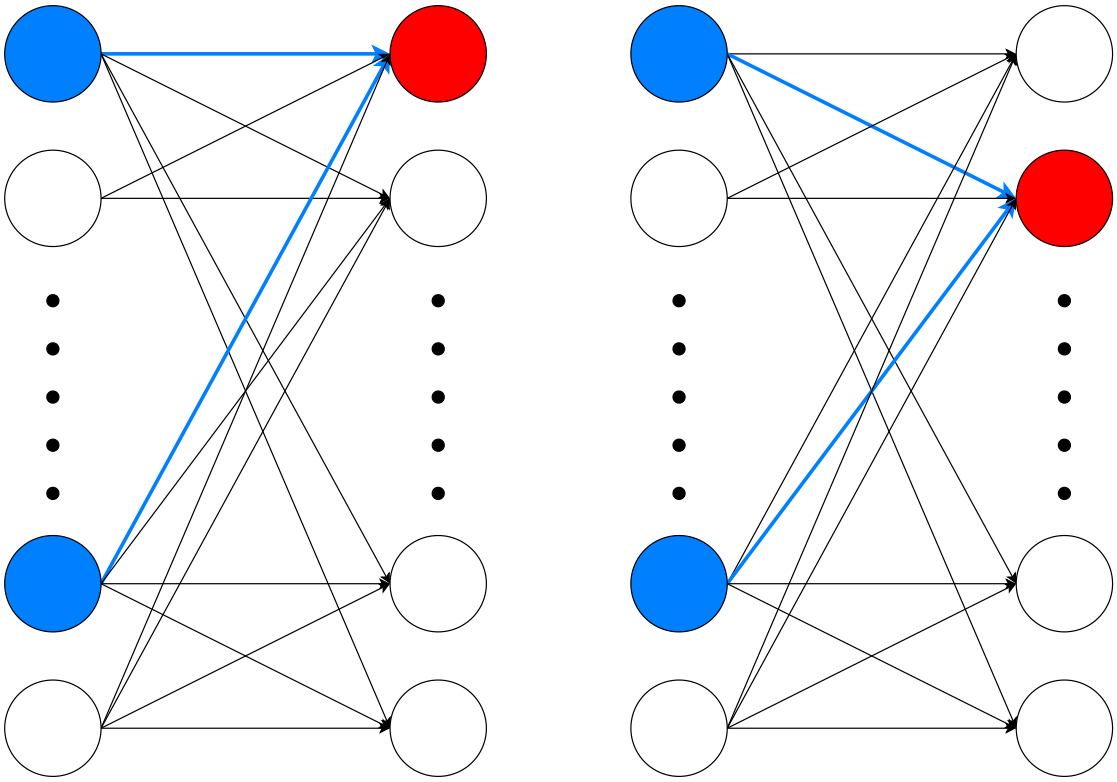

Figure 4.20: Initial idea for Fully Connected Algorithm

The light blue circle represent an input neuron that emit a spike. The light blue

arrows represent the active synapses and the red circle represent the output neuron considered at that moment. starting with the left side it is possible to understand that during the scan of input neurons only the light blue synapses will contribute to the change the state of the specific output neuron so those contributions must be extracted from memory and added to the state of neuron connected. Obviously, also the leak value must be included as usual and then the data can be stored. The right side indicates the next step where the analysis is done for the second output neuron with the same procedure.

Alternative idea The alternative is to see the layer from the opposite side. So instead to scan all the inputs for each output neurons it's possible to update all the output neurons controlling the single input neurons. This option is shown in figure 4.21.

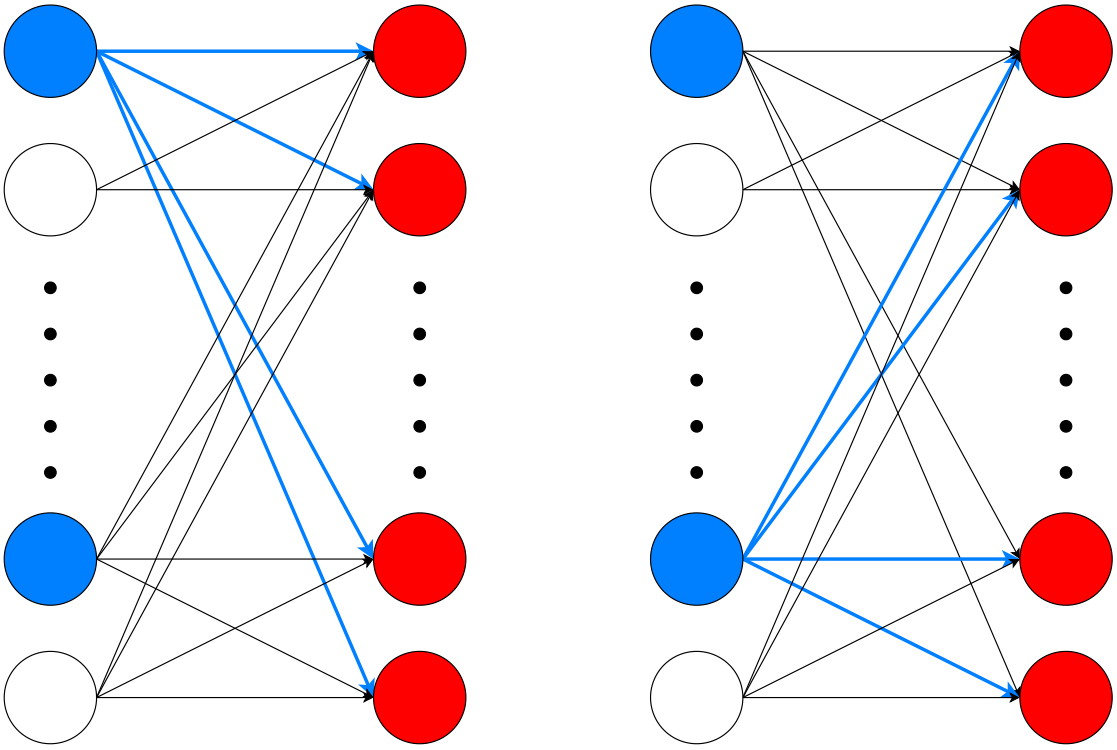

Figure 4.21: Fully Connected Algorithm

The legend in figure 4.21 is the same as before so the light blue circle represent an input neuron that emits a spike. The light blue arrows represent the active synapses and the red circle represent the output neuron considered at that moment. The left side shows the computation done by the first spike emitting neuron where all the synapses connected to it will change the states of all output neurons. After

the check of many inactive neurons, it is possible to find another light blue neuron as shown on the right side. Just like before all the synapses with their own weight will to influence the output neurons changing the partial state. In this second the case the leak value is applied at the end of the scan of all the inputs.

Comparison To a first approximation, in terms of speed, the two algorithms are the same but they present relevant differences in terms of power consumption and control complexity. The first idea executes as many times as output neurons all the input neurons but write only one time the value of output state so we have a large power consumption on the counter that provides the input address but a minimum impact on the power in the output memory. The second idea instead stores all the partial results of output neurons so an increase of commutations in memory and a decrease of them in counter for input. The second big difference is the weights control and its influence is not negligible. In fact, the position of spike emitting neurons is not known in advance so the weights that will be used are spread all over the memory. This situation is more problematic in the case of the initial idea because every single weight must be addressed singularly. On the contrary, using the alternative all the weights that goes to one input neurons to all the output neurons could be placed one after the other and this is a relevant advantage because when an input neuron emits the spike all the value of weights connected can be extracted from the memory with a burst operation that is faster than an individual addressing. So in light of this information, it has been decided to adopt the second formulation trying to speed up the execution exploiting the burst operation.
Pseudocode of fully-connected layer is therefore reported in figure 4.22

```
for g = 1:G                                                    % Loop on input neurons
    if (Empty Row=1)                                           % Check if row does not have any spike
        g++;
    else
        for h = 1:H                                            % Loop on memory row
            if (Column Value=1)                                % Check if neuron emits a spike
                h++;
            else
                for i = 1:I                                    % Loop on weights connected
                    NewState = OldState + Weight_ghi;
                    Store;
```

Figure 4.22: Fully-connected Pseudocode

In the next figure is shown the flowchart illustrating the selected algorithm. It contains also the optimization element that are cited just above. In fact, the input data are stored in a memory with a defined parallelism so the row of the memory contains several neurons. This is an advantage in the case of adjacent neurons are inactive. To exploit this, in the algorithm has been included a control when an entire row contain only inactive neurons it is possible to move the next row and check this new series of neurons. This allows to reduce time in the scan of the input neurons and also to avoid useless commutations. When, on the contrary, the row contains at least one spike the scan of the selected row will start to find all the "1" that, indicating a spike, will cause the updating of all output neurons. As already said after the scan of all rows the leak value is applied with a full scan of the output neurons.

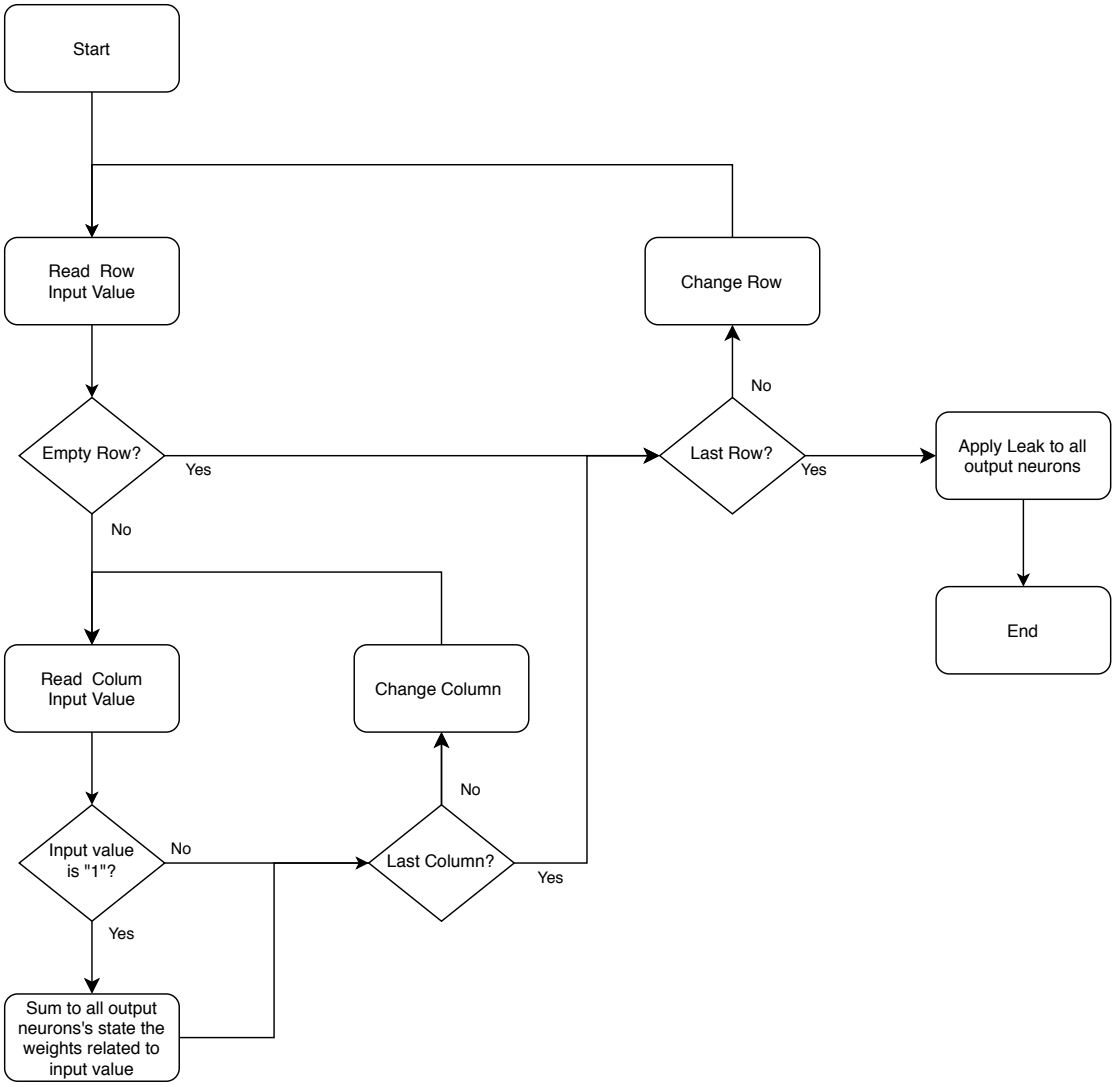

Figure 4.23: Fully Connected Algorithm

4.2 Architecture

Before to develop a unified architecture has been realized architecture singularly to better understand all the potential timing problems and how to manage precision bits. All three layers type present a similar macro architecture but with some details that could cause troubles. In figure 4.24 is illustrated a simplified version of the architecture. In this image miss the counters to address the Weights, States and Output memory. They are omitted because they work as usual counters controlled by a Finite State Machine and the architecture is more clear.

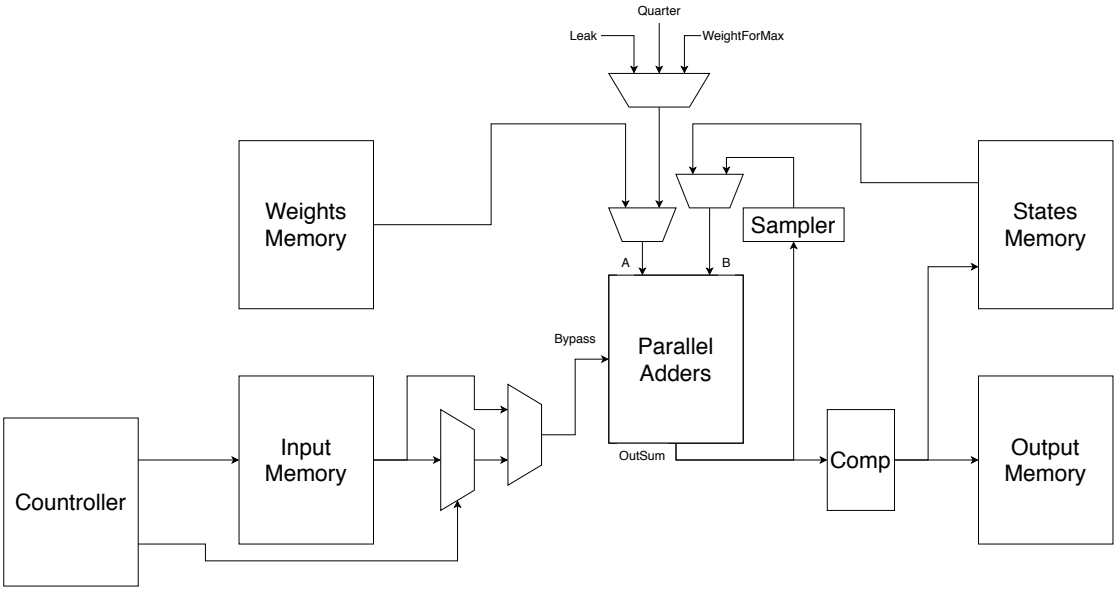

Figure 4.24: Simplified Architecture

Before to describe in details all the block represented I would list all parameters were used in the hardware description. These parameters are not fixed in general but are a personal choice to obtain a good trade-off between speed and power consumption and accuracy on the target FPGA.

- Precision Bit For States= 12
- Precision Bit For Weights= 8
- Parallelism = 16

4.2.1 Input Memory

The Input memory contain all the spikes from the input neurons. It has word length equal to the parallelism because each neuron needs only one bit to keep the information of emitted or not spike. The number of rows of input memory instead depend on many neurons the user want to execute in a full cycle of the accelerator.

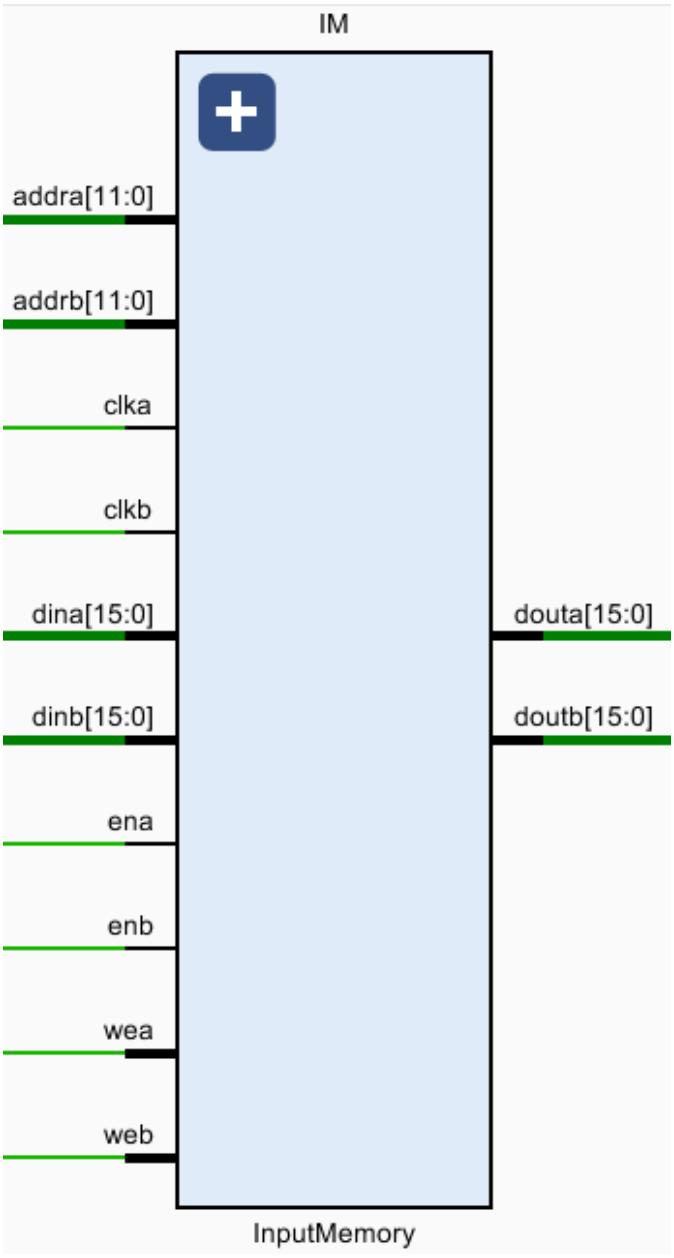

Figure 4.25: Input Memory

4.2.2 Weights Memory

The weights memory contain all the weights associated to a synapses independently that these values are related to a moving filter, like in the case of a convolutional layer, or a pure connection, like in fully connected layer. The word length is equal to one hundred twenty-eight (parallelism multiplied by Precision Bit for weights). The number of row of weights memory instead depend on many weights (direct connection or filters' value) user want to have in a full cycle of the accelerator.

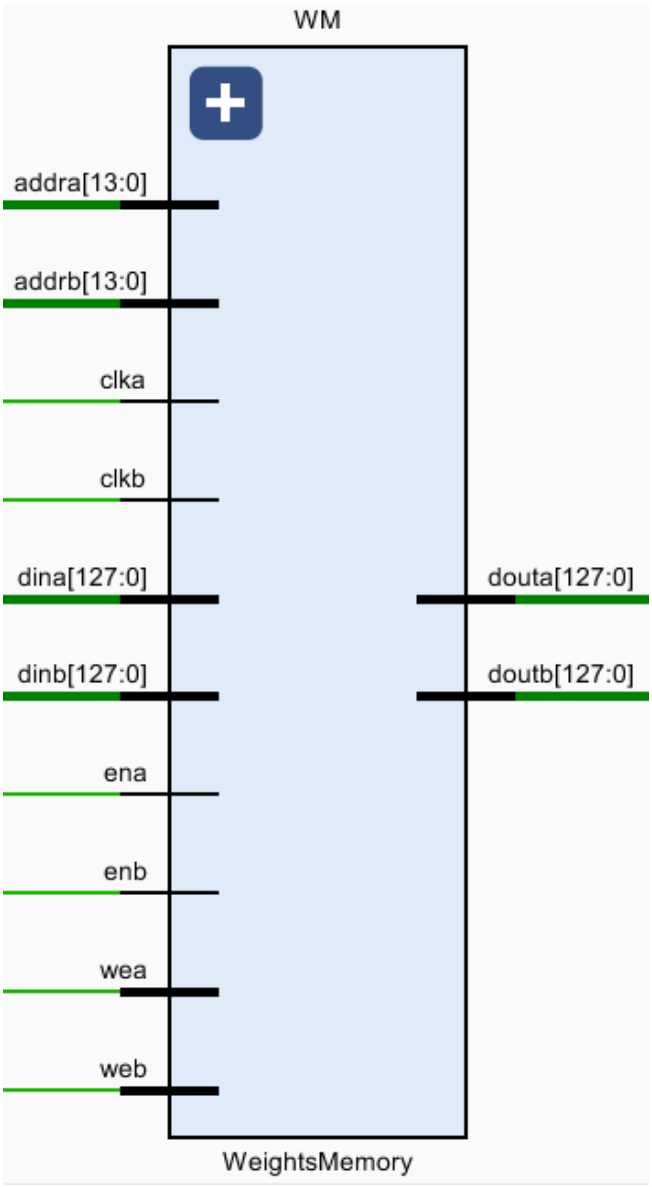

Figure 4.26: Weights Memory

4.2.3 Output Memory

The output memory contains all the spikes computed in the accelerator for each output neurons. For the sake of simplicity, the dimensions of this memory are the same as the input memory.

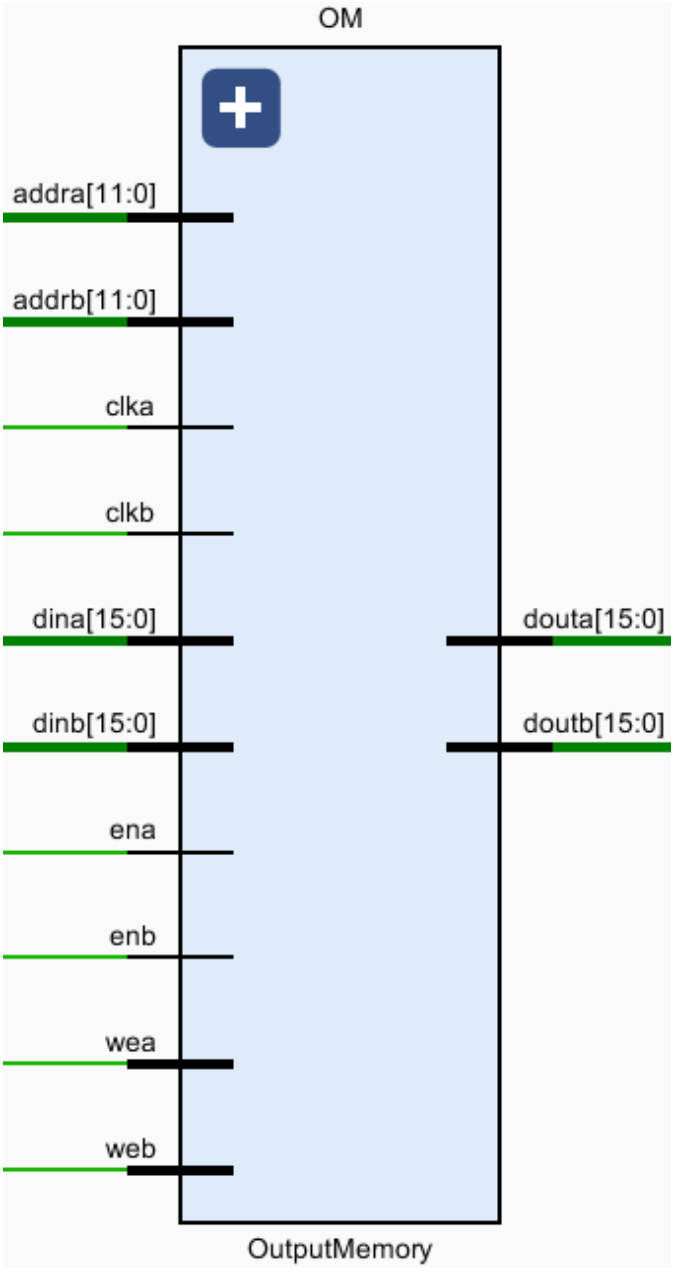

Figure 4.27: Output Memory

4.2.4 States Memory

The states memory contain all the states related to the output neurons. The word length is equal to one hundred ninety-two (parallelism multiplied by Precision Bit for weights). The number of row of weights memory is equal to the max output neurons implementable chosen by the user.

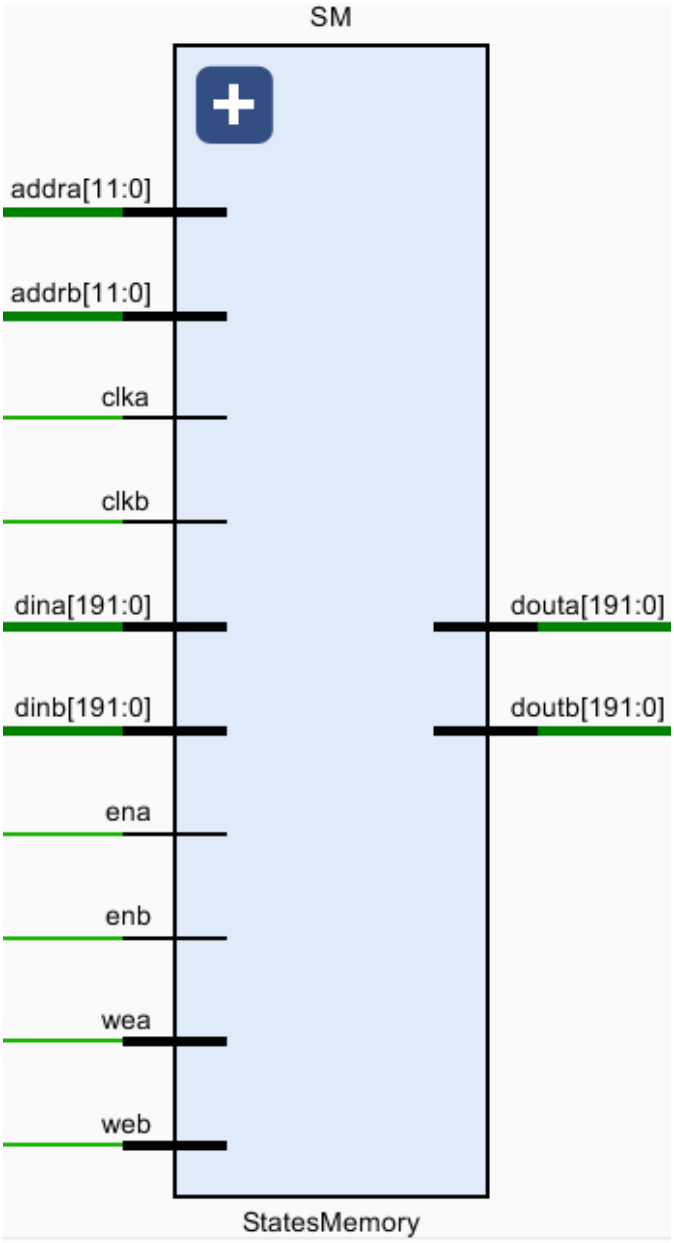

Figure 4.28: States Memory

BRAM partition in the FPGA

The dimensions of these four memories must be set carefully, taking into account several factors.
First of all, the whole amount of the Block RAM available in the FPGA. If the number of declared BRAM exceed the physical BRAM present on the integrated circuit, the accelerator is not implementable.
Secondly, the type of neural network that will be executed on the accelerator. To speed up a convolutional layer is better to increase the number of neurons (pixels) and so to expand the memories that are related to keeping these data. In convolutional layer, therefore, is important to have a reduced weights memory and, on the contrary, bigger memory for input, output and states. This is valid even more so for the pooling layer where the weights' memory could be null because needs only a constant. For a fully-connected layer instead, the input and output neurons are, looking the most powerful networks, maximum few thousands but the number of connections (calculated like input multiplied by output neurons) increases very fast. It is, therefore, necessary to make a deep analysis of the neural network that will be implemented and find the best trade-off.
In my architecture, the dimensions are set to store 2^{18} weights and 2^{16} neurons in input and equal in output.

4.2.5 Sampler

This block is a simple register that is used to store all the partial sums to allow a correct accumulation. Data width of this register in my architecture is equal to one hundred ninety-two (the result of the multiplication between parallelism and Bit for precision states).

4.2.6 Comparator

This element is not a simple comparator but performs the activation function ReLU like in figure 4.8.

4.2.7 Parallel Adders

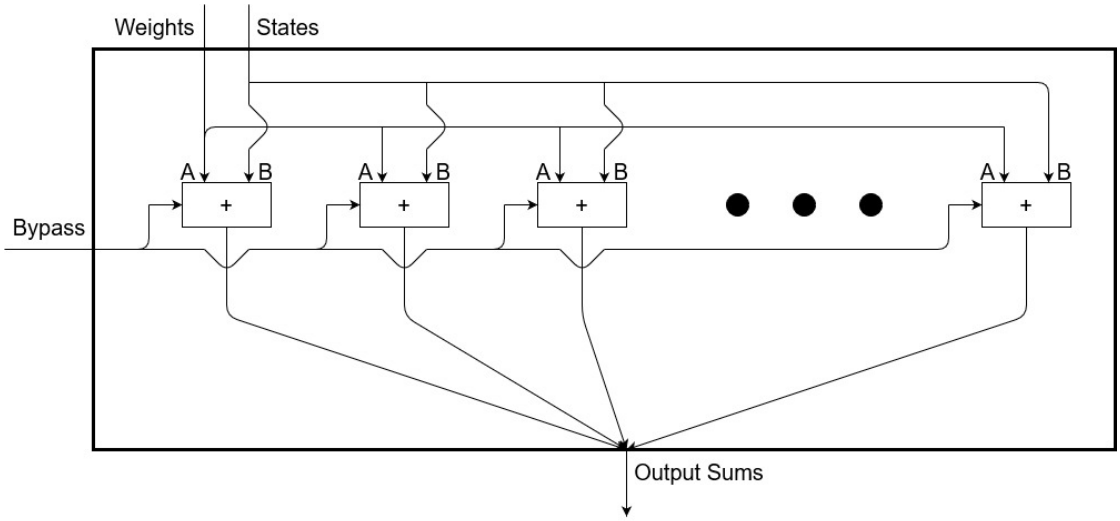

Figure 4.29: Parallel Adders

This block, shown in 4.29, represents, considering my choices, sixteen parallel adders but this value can easily be increased or reduced. Every single adder can elaborate sums between signed data with different data-widths. This last parameter must be set in the IP configuration of the adder present in IP Catalog of Vivado. On port B of parallel adders is present a multiplexer used to select between The data coming from the States Memory or the data coming from sampler where are stored the partial sum. On port A of parallel adders is present a multiplexer that selects between the value coming from Weights Memory or data coming from the other multiplexer as appropriate. The inputs of this other multiplexer are three constants that model respectively the leak of neurons and the two different constant values

used for pooling filter. The two different constants are used in case of a 2x2 pooling filter (the most common in neural network) and for max-pooling respectively.

Max pooling The transfer function of max pooling brings in output the input with the largest number of spikes during the single period. A single period could be composed of ten or twenty step time depending on the accuracy required. Implement this function in hardware means that for each neuron, in input and output, all the value during the period must be stored. This affirmation is true also for all the other layers that so must be executed for an entire period and not for a single step time. This is a problem because the accelerator to reduce the communication with external memory already need big memories internally, due to the huge amount of data of a single layer. The approximate solution that has been adopted for this accelerator is the one explained in paper [7]. In a few words, instead to analyze all the period we take only data from single step and if at least one of the input is equal to "1" the output neurons emit a spike. This approximation produces a greater number of spikes as compared to the correct number but the complexity of circuit greatly reduce. The behaviour could be seen in figure 4.30.

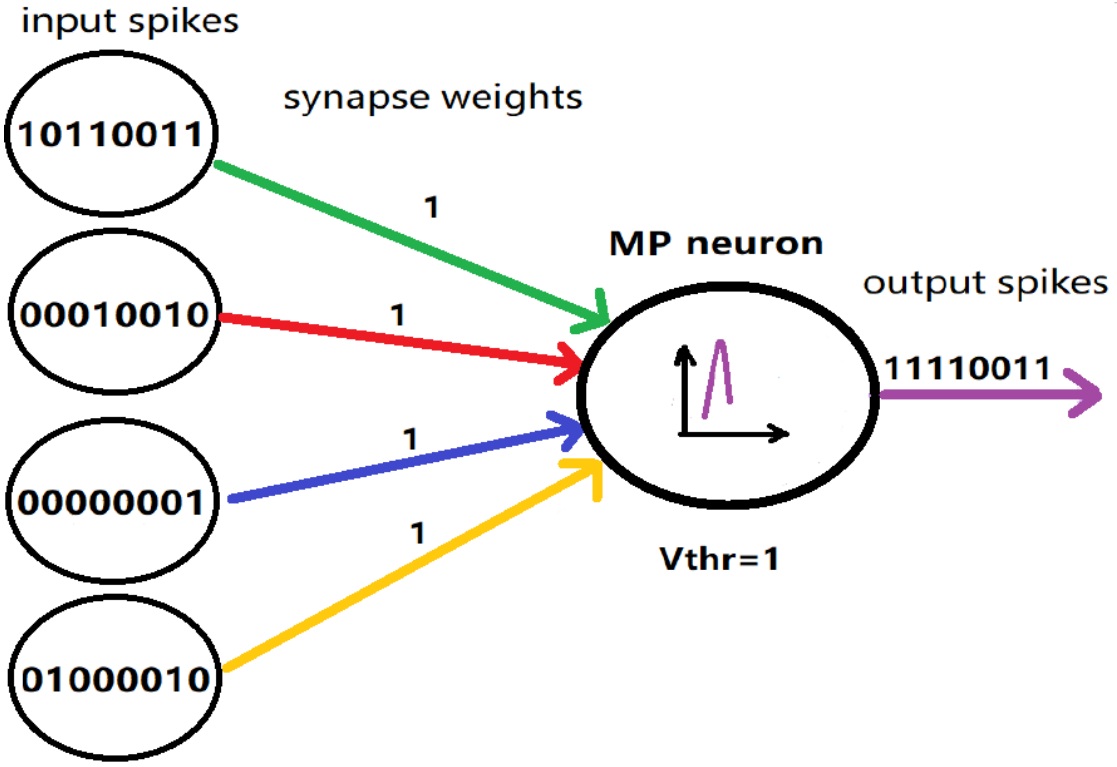

Figure 4.30: Max pooling approximation

Bypass Every adder has also a specific bit of bypass that is fundamental for this accelerator. When the bypass signal is equal to "1" output value of the adder is the value present on the input B without executing the sum. This just above described is the case when the input neuron's value equal to "0". We could say that the bypass bit is obtained as the NOT of input neuron value. For a correct connection between the input memory and the bypass port are inserted some multiplex. This multiplexers controlled by the FSM perform the task to send the correct string of bit for bypass port. The string, indeed, depends on which layer and which operation. For example in case of a convolutional layer, where the chosen input is one by one the bypass bit is equal to that neuron for each adder. On the contrary in the case of pooling the neurons selected in input are equal to parallelism so each adder has is own reference bypass bit. In case of application of leaking to neurons, all the bypass bits are "0" independently on type of layer.

Each adder has a saturation control to avoid overflow and it acts in case of both positive and negative results.

4.2.8 Countroller

This is a particular entity that I named in this way because it is a mix between a simple counter and a controller. It acts like a program counter in a CPU. It is composed of several counters that control the position of filters and position of the computed value in that exact moment. It implements almost all the functions described in the flowcharts shown in algorithm section 4.1. The only functions that will not depend on it are the selection of muxes in input of Parallel Adders and the Write Enable signal for the memories related to output neurons (states and output memories). These two jobs are done by the specific FSM. The addressing of memories different from the Input one is not done by the Countroller but by simple counters. The process that describes the elaboration of the new row in case of convolutional and pooling layer is shown of figure4.31. For the fully-connected layer the change of row is linear so use a normal counter.

```
MovingRow:process(Clk)
begin
if(clk'event and Clk='1') then
if Reset='0' then
Count<=(others=>'0');
elsif Enable="0" or LastColumnScans="0" then
Count<=Count;
elsif TypeLayer=3 then
Count<= Count + 1;
elsif LastRowSingleValue="0" then
Count<= Count + 1;
elsif LastColumnFilter="0" then
Count<= Count + ChangeColumn;
elsif LastRowFilter="0" then
Count<= Count + ChangeRow;
elsif LastFilterSeries="0" then
Count<= Count + NotRightShift;
elsif LastColumnResult ="0" then
Count<= Count + ShiftRight;
elsif LastRowResult ="0" then
Count<= Count + ShiftDown;
else Count<=(others=>'0');
end if;
end if;
end process;
AddressCP<=Count;
EnableScan<=Enable;
```

Figure 4.31: Countroller main process

4.2.9 Interface

At the end of the project was clear that the input and the output pins of this accelerator would exceed the pin capability provided by the FPGA. So to make

possible the synthesis has been realized an interface where all the input and output are serialized in frames of 32 bit. This interface will be managed by the processor or from a DMA (Direct Access Memory) connecting in the right way the DDR and accelerator.

4.2.10 User file

To simplify the configuration and reconfiguration of the accelerator a file has been created. In this file, called NeuralNetworkInfo, are present all the parameters related to the entire network or a specific layer. The user that wants to implement this accelerator can do it modifying only these parameters reported on file and adding the layers with related parameters. This file is reported in appendix A

Chapter 5

Testing the accelerator

In this chapter will be shown some sections of the different simulations to verify the correct behaviour. A complex architecture with several signals requires an in-depth analysis and consequently a huge amount of time.

5.1 Convolutional layer testing

The first algorithm that was tested is the convolutional one with several simulations changing the parameters. For the sake of simplicity, only the extracts of one simulation are reported for the convolutional part. The parameters chosen for this simulation are listed below.

- Image Dimensions= 32x32
- Input Channels= 18
- Filter Dimensions = 5x5
- Number of filters = 32
- Stride(Horizontal and vertical) = 1

In the first figure 5.1 we can evaluate the proper functioning of the Countroller. First of all, the value of Type Layer signal, equal to "0" ensures that we are simulating a convolutional layer. So, in the upper image, it is possible to notice that while the yellow signal shows the address of Input Memory last signal in green select the bit of the word that will be analyzed starting from MSB. At the end of the row, this count will restart for the other and stop after two iterations. This happens because the eighteen values of input channels are stored in two rows. So after the scan of the first sixteen stored in the first row, the other two will be scanned in the second one. So the control that avoids scanning unnecessarily all

the row properly works.
Going to the figure 5.2 we can visualize in more detail the signal in violet, blue and grey that represent the terminal count respectively the of input channels, column of filter and row of filter. From that sequence is also possible to understand that the filter executed is 5x5 indeed the terminal counts are five both for input channels and for the column of filters. Once that the filter count is stopped in order to apply the leaking value, the results will be stored. After this operation, the count will restart. In this case, it will restart from position zero. This happens because the filters present in this layer are thirty-two and this first cycle execute the first sixteen (value of parallelism). So after a second sequences where the filters selected are the second sixteen will be applied the stride.
This could seen in figure 5.3. In fact, at the begin of the image, it is possible to notice near address 108 in yellow is present address 0. Stride is not applied yet. At the end of the image, so after the computation of all 32 filter, from address 108 simulation goes to address 2. It is therefore applied the stride. To be clear simulation jumps to position 2 with stride equal to one because architecture needs two rows to store the input channel.

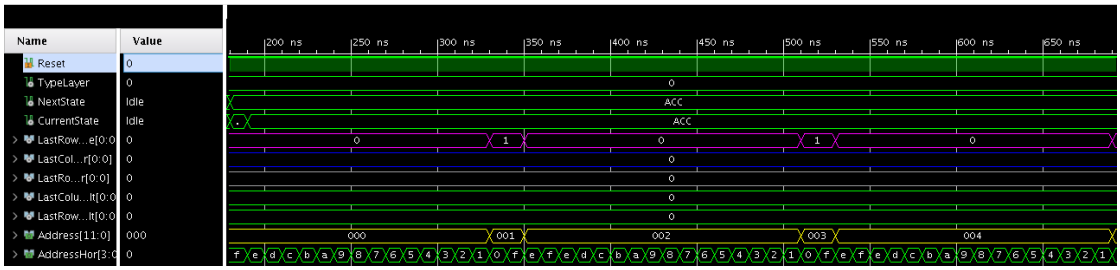

Figure 5.1: Zoomed Simulation for check AddressHor

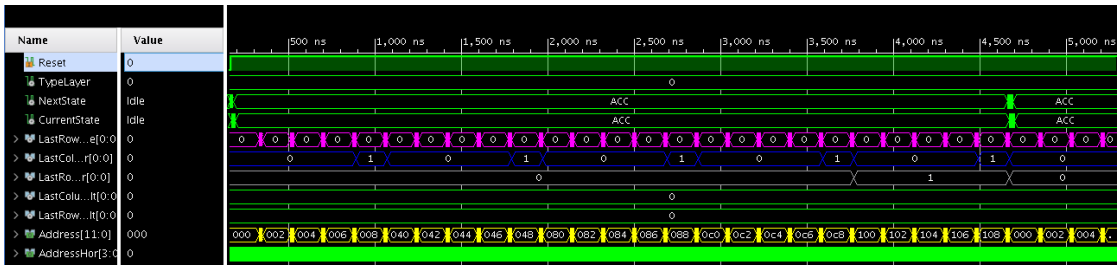

Figure 5.2: Simulation for check Address

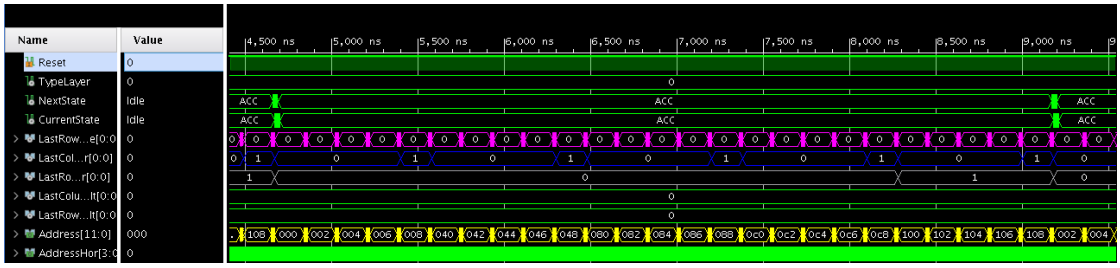

Figure 5.3: Simulation for check jumps in Address

To do a complete simulation of all single blocks in the architecture have been defined some filters. These particular values maybe are not useful in practical terms but could stimulate different aspects of the architecture. In particular, the first and the last of the filters shown in figure 5.4 are designed to verify the correctness of the saturation of states and the properly working of the ReLU activation function. They are stored in Weights Memory in signed integer.

-125	-125	-125	-125	-125
-100	-100	-100	-100	-100
-125	-125	-125	-125	-125
-100	-100	-100	-100	-100
100	100	100	100	100

-100	-50	0	-50	-100
-50	0	0	0	-50
0	0	0	0	0
-50	0	0	0	-50
-100	-50	0	-50	-100

1	1	1	1	1
1	2	2	2	1
1	2	3	2	1
1	2	2	2	1
1	1	1	1	1

3	2	1	2	3
3	2	1	2	3
3	2	1	2	3
3	2	1	2	3
3	2	1	2	3

0	0	1	0	0
0	0	1	0	0
1	1	1	1	1
0	0	1	0	0
0	0	1	0	0

125	125	125	125	125
100	100	100	100	100
125	125	125	125	125
100	100	100	100	100
-100	-100	-100	-100	-100

Figure 5.4: Filters for test

The input values stored in Input Memory are instead random.

After this information and with the certainty of the correct addressing, we could move to check the adders operations.

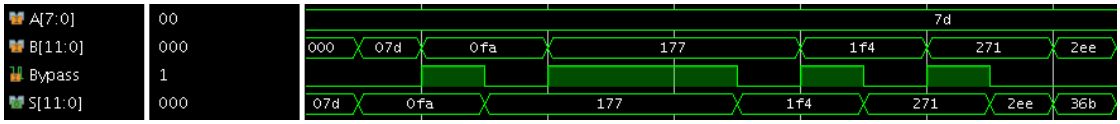

Figure 5.5: Simulation for Adders

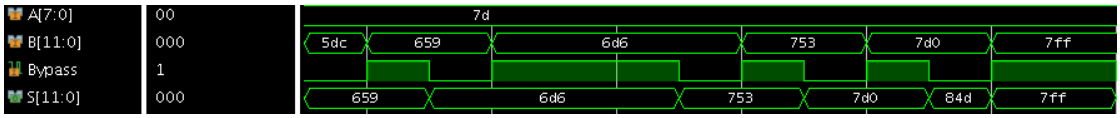

Figure 5.6: Simulation for Adders with positive overflow

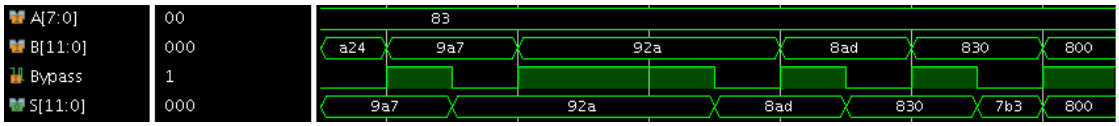

Figure 5.7: Simulation for Adders with negative overflow

As we can see in the figure 5.5 the state located in B signal and the filter value located in A signal are summed if the Bypass signal is "0". When the Bypass is at "1" the value in B is transferred to the output. It is possible also to notice that the previous result is reported in B port. It means that the accumulation path works correctly. The 7d value is the hexadecimal representation of value 125 located in the first position of the last filter in figure 5.4. The initial value of B instead is zero because neurons was not updated before. In the figure 5.6 it is possible to see the case of overflow of the same adder. The addition between 2000(7d0) and 125(7d) bring has result -1971(84d), but the next value of B is correctly the max value admitted 2047(7ff). This condition but in case of a negative number can be seen in another adder and is reported in the screenshot in figure 5.7. In fact, the sum between -125(83) and -2000 (830) bring has result 1971(7b3), but the next value of B is correctly the most negative value admitted -2048(800).

The parts that remain to verify is the addition of the leak to the state value and the subsequent application of activation function before the store. These can be understood in the next figure 5.8

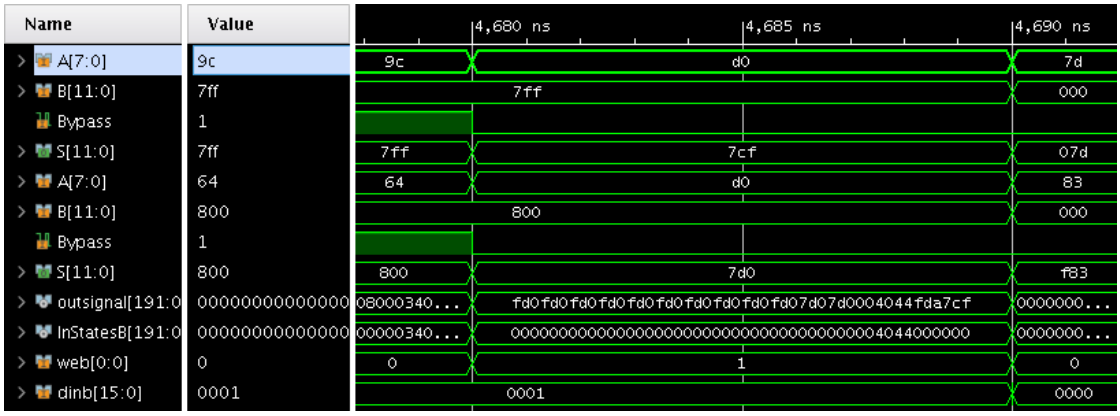

Figure 5.8: Leak and store

In fact, it is possible to see that when the end of a series of filters is reached on all ports A is present a value d0 (-48 in decimal) that, in this simulation, is used to model the leak. Usually, the leak is a small negative value. The outsignal present below the adders ports is the concatenation of all the sum results. It is possible to find 7cf and 7d0 between that long string of numbers. For example, 7cf are the LSBs of the signal. All the sums go in input to the activation function and is result is on signal InStatesB. InStatesB is connected to the input port of States Memory. The threshold of the neurons in this simulation is fixed to 400 (1024 in decimal). The ReLU, illustrated in figure 4.8, works correctly. This can be seen in the differences between outsignal and InStatesB and variation in dinb (input signal is connected to the input port of Output Memory). For example, the three least significant digits of outsignal 7cf (value greater than 400) become 000 in InStatesB and the dinb is equal to "1". In this case, the only neuron that emits a spike is that one computed on that adder. The other neurons' negative states become "000" like in the case of the most significant digits of InStatesB. The positive value below the threshold are instead kept equal of digits 044 that are transferred directly in InStatesB. The web signal that represents the Write Enable of port B of both memories and it is at "1" in the exact moment with correct data.

5.2 Pooling layer testing

For the pooling layer test, the parameters used are itemized below.

- Image Dimensions= 28x28
- Input Channels= 32
- Filter Dimensions = 2x2
- Stride(Horizontal and vertical) = 2

In the first screenshot in figure 5.9 it is possible to see how the addressing the Input Memory. There is no changing value on AddressHor because the mux downstream the memory should not work. This happens because, differently from the convolutional layer, all the sixteen bits from Input Memory are used in parallel. The signal LastFilterSeries assumes the meaning of the end of input channels in this type of layer. when it is at "1", and the filter finishes, it is applied the stride.

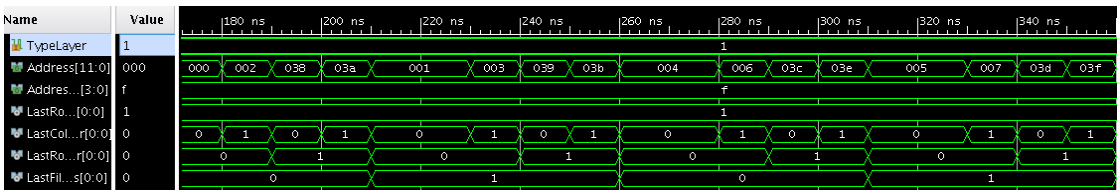

Figure 5.9: Simulation of Countroller in Pooling layer

The screenshot in figure 5.10 shows a frame where the second adder has bypass always at zero. This situation brings the neuron to overcome the threshold so it emits the spike. This could be seen looking at the dinb signal where the only spike presents is the one computed on that specific adder. The other adder reported, for example, reaches the value of 0ca that is not enough to send spike to the next layer. Also, in this case, it possible to see the leak value and contemporary the storing of the spike and next states just after the computation of the partial result of a specific neuron. In the second screenshot of 5.11 is more zoomed so can be understood the value that will be stored in states due to the application of ReLU function and the spikes that are emitted.

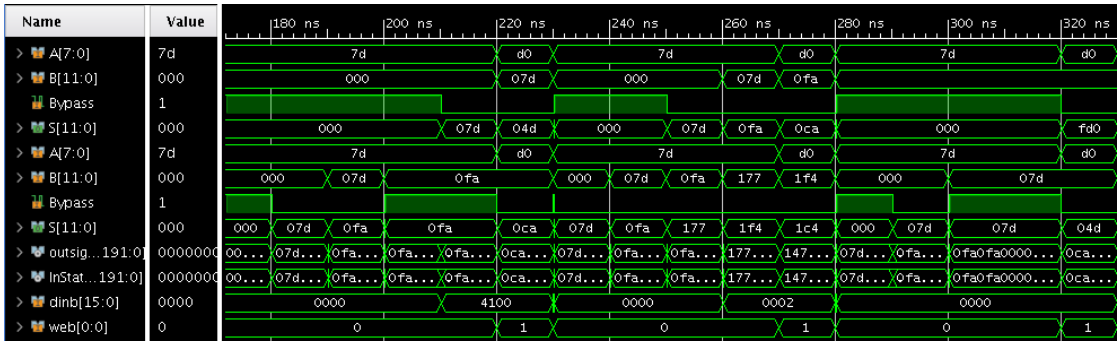

Figure 5.10: Simulation of adders and storing in Pooling layer

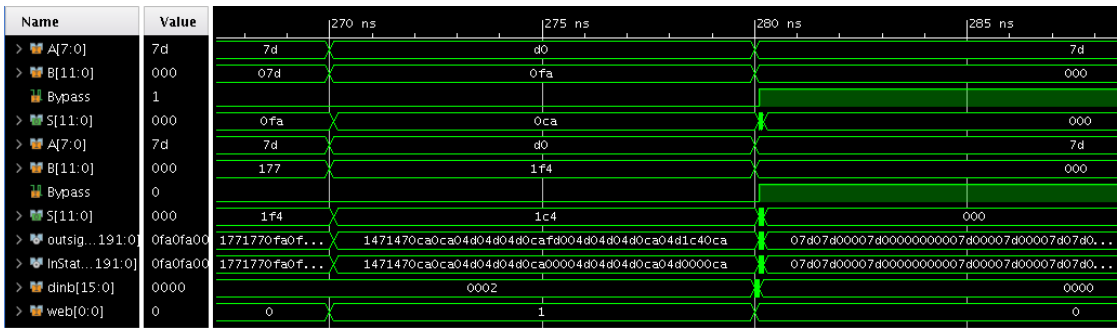

Figure 5.11: Simulation of adders and storing in Pooling layer with zoom

5.3 Fully-Connected layer testing

For the Fully-Connected layer test, the parameters used are itemized below.

- Input Neurons= 512
- Input Neurons= 256

As already said in the previous chapter, the fully connected layer does not use the Countroller to execute jumps of the input address because the scan of input neurons is linear but the signals to speed up the execution must be checked. For example in figure 5.12 can be seen the situation when the row of Input Memory is not empty so the scan of it will start from MSB (f on AddressHor) and going on, after the scan of all output neurons. It is present a neuron that not emit a spike in the third position (d in AddressHor) so the address after this check will skip to the next neuron without scan and update the output neurons. In figure 5.13 can be visualized the case of an empty row and consequently it is possible to observe that the Address skips to the next, from 3 (row where no neuron emits a spike) to 4.

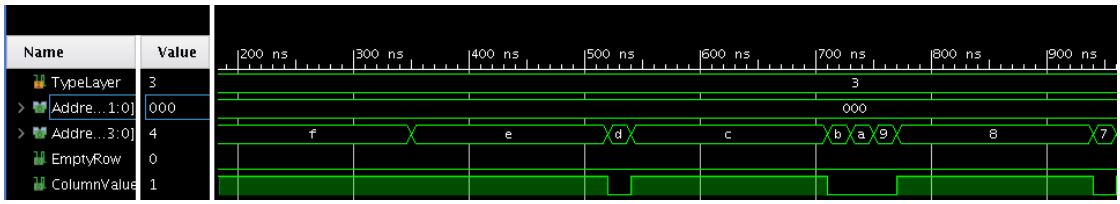

Figure 5.12: Simulation of normal addressing in Fully-Connected layer

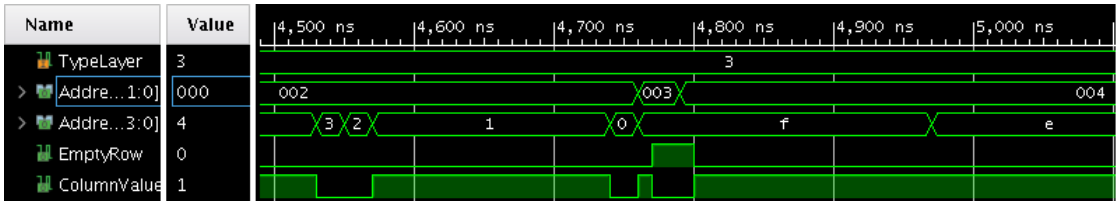

Figure 5.13: Simulation of addressing in Fully-Connected layer with empty row

After the check of addressing, as usual, the next part of the simulation is about the sums.

The adders, in this case, do not use the bypass bit indeed is fixed to "0" and all the sum result will be stored. There is a delay necessary to avoid conflicts in States Memory. All of this is shown in figure 5.14. When all the neurons in input have been analyzed can be finally subtracted the leak from the states and store the eventual spikes in output. In figure 5.15 can be seen that the web of Output Memory is at one and store the outputs where in this specific case, due to the number imposed, all the neurons computed on the last adder emit the spike.

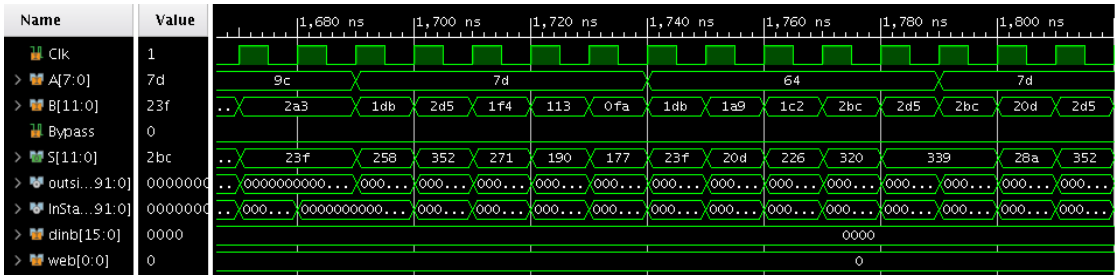

Figure 5.14: Addition in Fully-Connected layer

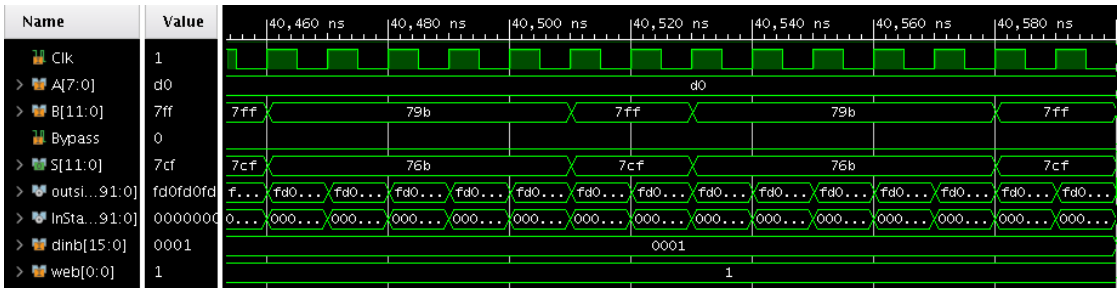

Figure 5.15: Leak and store in Fully-Connected layer

The same analysis have been done also for simulation post-synthesis and implementation and everything works correctly. In these last simulation is to hard to find the specific signal due to the compilation but was used as reference the name of signals present in the schematic generated by Vivado.

Chapter 6

Results

In this chapter are reported the results obtained in terms of speed, area, power and performance.

6.1 Speed

The critical path of this accelerator go from the Weight Memory to the Sampler register, as expected, and the total delay is equal to 10.9 ns that include also the skew. In my design is used a clock period equal to 11 ns. It was interesting to notice that the computed critical path after the synthesis and after the implementation shown a difference over to 1 ns. This means that the huge amount of nets in this circuit reduce the speed of the 10 % with respect to the estimated value in synthesis phase that was below the 10 ns. In the top ten critical delay, shown in figure 6.1 is present one (path 10) with a very low logical delay but width an high net delay. This fact confirms that the huge number of nets increase the length of their paths.

Name	Slack	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	0.048	41	Accelerator/...m/CLKARDCLK	Accelerator/.../Q_reg[38]/D	10.901	5.685	5.216	11.0
Path 2	0.077	41	Accelerator/...m/CLKARDCLK	Accelerator/.../Q_reg[44]/D	10.875	5.685	5.190	11.0
Path 3	0.078	41	Accelerator/...m/CLKARDCLK	Accelerator/.../Q_reg[37]/D	10.873	5.685	5.188	11.0
Path 4	0.120	25	Accelerator/...am/CLKARDCLK	Accelerator/.../Q_reg[33]/D	10.801	5.685	5.116	11.0
Path 5	0.132	41	Accelerator/...m/CLKARDCLK	Accelerator/.../Q_reg[42]/D	10.815	5.685	5.130	11.0
Path 6	0.164	41	Accelerator/...m/CLKARDCLK	Accelerator/.../Q_reg[54]/D	10.805	5.658	5.147	11.0
Path 7	0.193	25	Accelerator/...am/CLKARDCLK	Accelerator/.../Q_reg[26]/D	10.725	5.685	5.040	11.0
Path 8	0.198	25	Accelerator/...am/CLKARDCLK	Accelerator/.../Q_reg[25]/D	10.722	5.685	5.037	11.0
Path 9	0.217	25	Accelerator/...m/CLKARDCLK	Accelerator/.../Q_reg[81]/D	10.740	5.639	5.101	11.0
Path 10	0.217	26	Accelerator/D...unts_reg[7]/C	Accelerator...ARDADDR[10]	9.943	0.419	9.524	11.0

Figure 6.1: Critical paths

6.2 Power

The power consumed by this accelerator with interface included is equal to 0.209 W. This value as can be seen in figure 6.2 is equally divided between dynamic and static power. This is due to two factors. One is the static power contribution from memory used or the other entity instantiated. The second one is the remaining cells unused that consume static power. The dynamic power instead is mainly consumed by the memories followed with margin by signal, so the capacitance of nets, and from DSP (where are present the 16 adders). To go in detail of the consume in figure 6.3 it is possible to notice that the most consuming component is the States Memory with 0.59 W.The idea of store only final results for convolutional and pooling layer therefore mitigate this result. At second place it is possible to find the other big memory, Weights Memory, with 0.12 W. It changes several bits when a new filter value or connection is computed. At third place we find the parallel adders with 0.16 W.

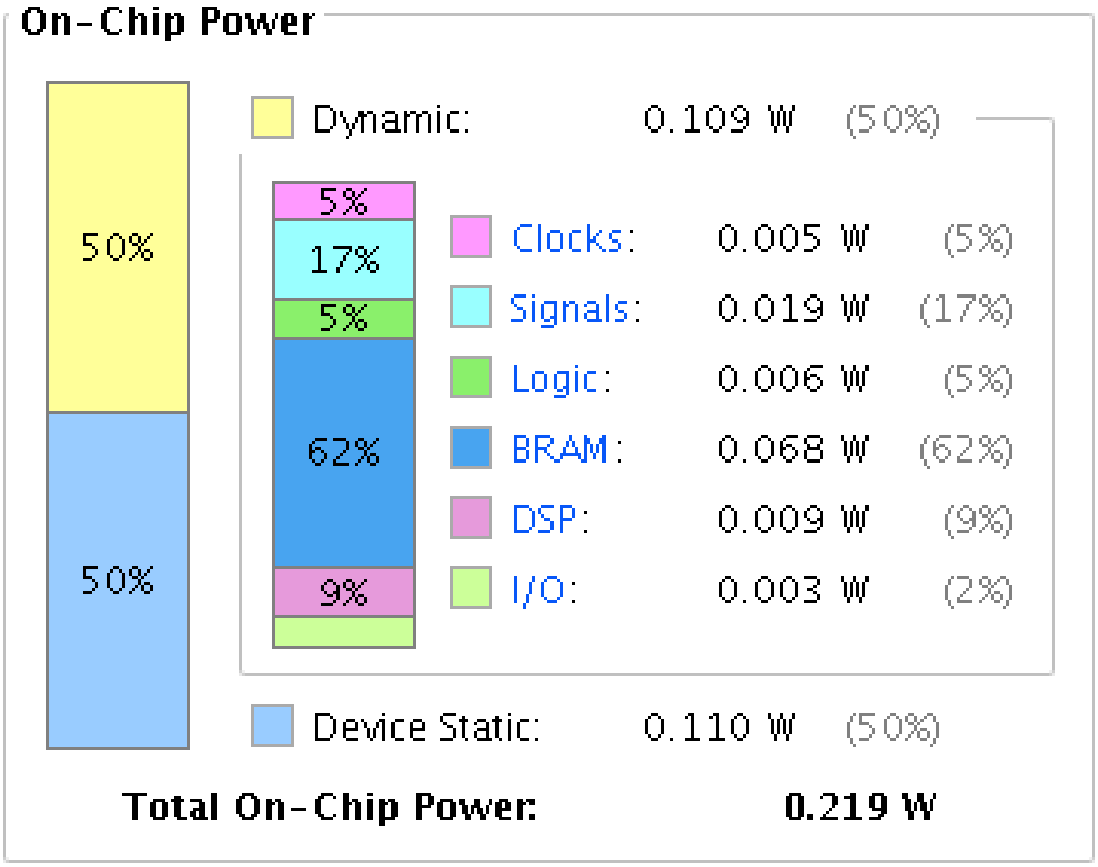

Figure 6.2: Power breakdown

Utilization	Name	Clocks (W)	Signals (W)	Data (W)	Logic (W)	BRAM (W)	DSP (W)
0.109 W (50% of total)	Interface						
0.106 W (48% of total)	Accelerator (Wrapper)	0.005	0.018	0.018	0.006	0.068	0.009
0.105 W (48% of tot...	DP (DatapathSNN)	0.005	0.018	0.018	0.005	0.068	0.009
0.059 W (27% of total)	MemStates (StatMem...	<0.001	0.006	0.006	<0.001	0.053	<0.001
0.018 W (8% of total)	MemWeights (ConnFi...	0.001	0.003	0.003	0.001	0.012	<0.001
0.016 W (7% of total)	AddsA (ParallelAdde...	<0.001	0.003	0.003	0.004	<0.001	0.009

Figure 6.3: Power hierarchy

6.3 Area

To better understand the utilization of FPGA is reported the table 6.1.

Site Type	Used	Available	Utilization %
LUT	746	53200	1.4
Slice Registers	314	106400	0.3
RAMB36	82	140	58.57
RAMB18	1	280	0.36
DSPs	16	220	7.27
IO	76	200	38

Table 6.1: Utilization Table

It is clear that can be used even more BRAMs because the utilization is just below the 59%. I prefer do not instantiate more memories to avoid that the increase of nets will bring the synthesis to fail. It is possible also to notice that the IO used that thanks to the introduction of the interface are equal to 76 with an utilization of 38%. DSPs utilization is equal to about 7% so it means that the parallelism can be increased. This change must be done taking into account that linearly increase the word of the memories and a consequent reduction of depth. This is not a problem in terms of area because it is possible to store the same number of neurons. This could be, instead, a problem in terms of interface because the segmentation of word requires more time. The images below show the place and route executed by Vivado. In the first one (fig. 6.4) are shown in light blue the used components (memories, LUT, DSPs,...). It is possible to notice the high number of BRAM used, the more bigger component, and the low utilization of LUT, the square component. In the second one (fig. 6.5) are shown also the interconnections between the components in green.

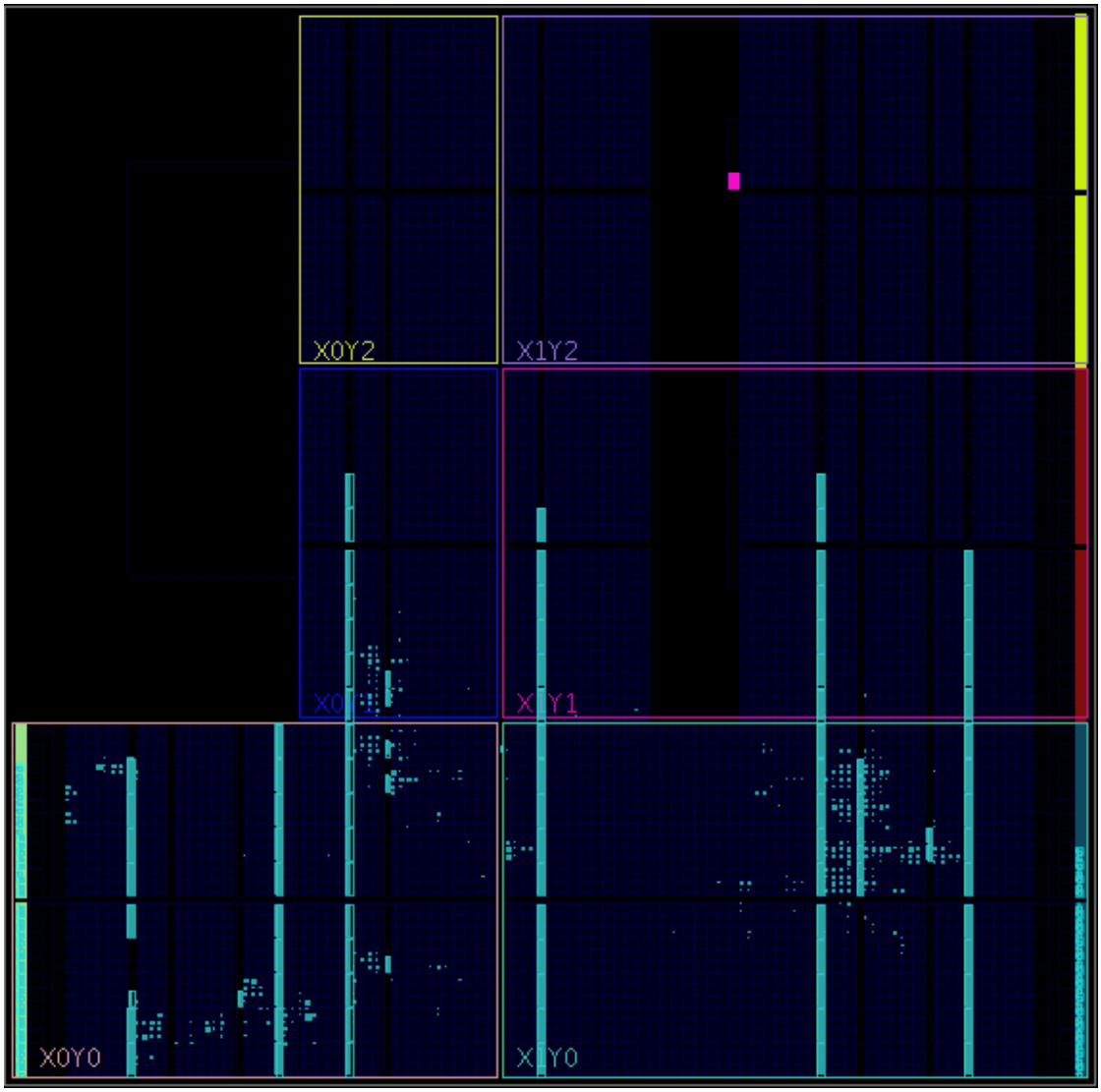

Figure 6.4: Place and route

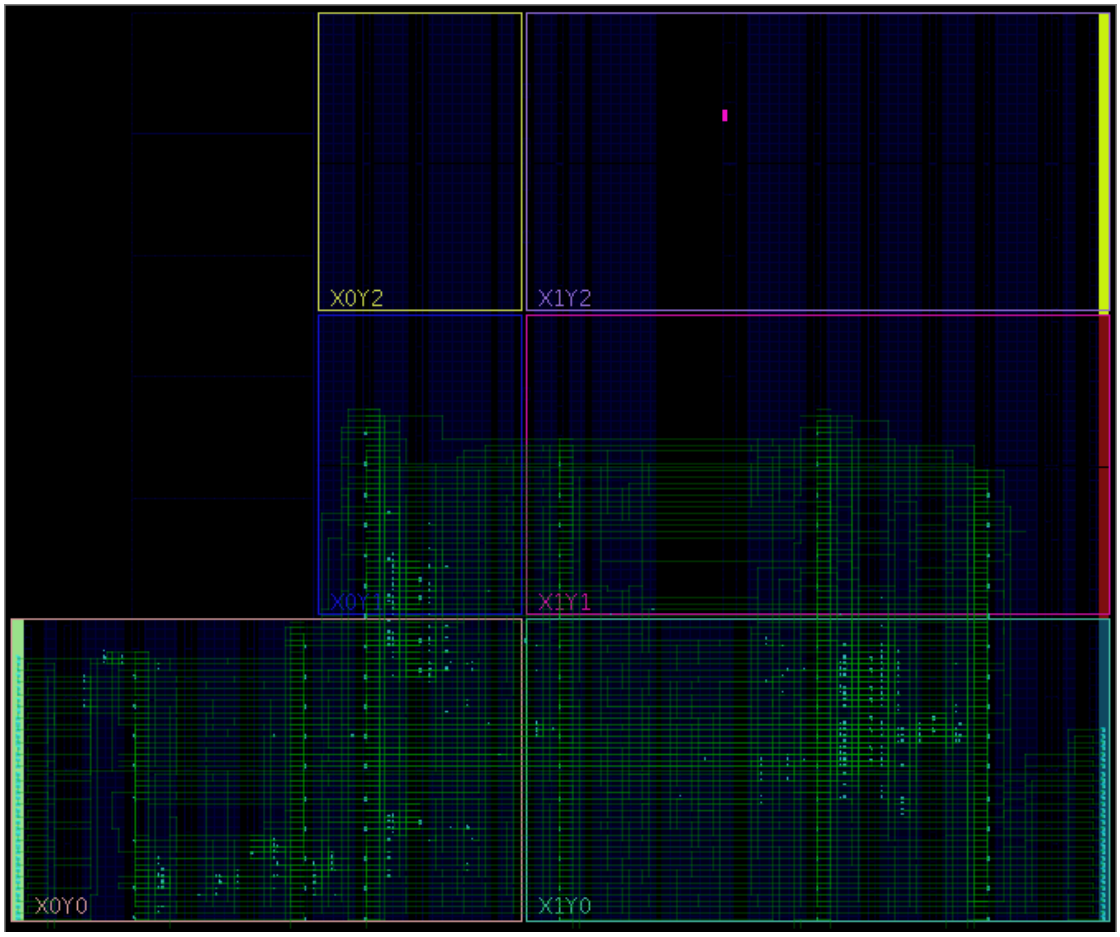

Figure 6.5: Place and route with connections

6.4 Performance

This accelerator, considering the configuration chosen in terms of memories dimensions and parallelism can execute several operations without the necessity to exchange data with the DDR. In fact, after loading data from external, a fully-connected layer with 2^{18} weights like a layer with 512 input neurons and 512 output neurons, or maybe with 1024 input and 256 output is executed in a full cycle of the accelerator. For the convolutional layer is possible for example to compute a frame 32x32 with 64 input channels and 64 filters. In convolutional and pooling layer the number of clock periods depend on filter dimension, filter stride and parallelism. Considering, for example, a parallelism equal to sixteen, filter equal to 5x5 and 64 input channels. With these parameters sixteen output neurons needs (25+1)*64=1664 clock periods. The plus one is due to the leak that must be added to the state of the neuron. In a fully connected layer instead, the number of period of clock necessary depend on the spikes in input and their distribution. In the worst case (each input neuron emits a spike) is equal to 2^{14} but this case is obviously unrealistic.

Chapter 7

Conclusion

In this thesis have been shown all the potential of implementing an SNN acceleration of FGPA. All the advantages from the high level of flexibility and reconfigurability to the good results obtained in terms of speed, power and so on. The next step will be to make this accelerator well integrated with a complete SoC. Must be therefore ensured a fast transfer of data between it and the external DDR and eventually to define a specific control, a sort of DMA, to reduce the efforts of the CPU. This specific control can be easily placed on FPGA looking the LUT utilization. It is important to have a good bandwidth between DDR and accelerator because, otherwise, this bottleneck will reduce the performance of all the SoC.

Appendix A

Code

NeuralNetworkInfo.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

use IEEE.NUMERIC_STD.ALL;

package NeuralNetworkInfo is
 constant NumberOfLayers: integer :=3;
 constant BitInterface : integer := 32;
 constant Parallelism : integer :=16;
 Constant BitForChannels: integer :=2;--ceil(log2(NRowForValue)) max value between layers
 constant BitForAddressSpike: integer := 12;--ceil(log2(MaxNumberOfColumns))
 constant BitForCF: integer := 3;--ceil(log2(ColumnFilter1))
            max value between layers
 constant BitForRF: integer := 3;--ceil(log2(RowFilter1))
            max value between layers
 constant BitForCR: integer := 5;--ceil(log2(ColumnResult1));
            max value between layers
 constant BitForRR: integer := 5;--ceil(log2(RowResult1));
 constant BitForLayers: integer := 2;--ceil(log2(NumberOfLayers));
 constant PrecisionWeights : integer := 8;
 constant PrecisionStates : integer := 12;
 constant Quart: integer :=320;--valore da definire--
 constant Leakage: integer :=-48; --valore da definire--
 constant Threshold: integer :=1024;--valore da definire--
 constant BitForPar : integer:=4;
 constant BitForWeights: integer:=14;--ceil(log2(RowFilter1*ColumnFilter1));));
 Constant TotPrecisionWeights: integer := PrecisionWeights*Parallelism;
```

```
25 constant TotPrecisionStates: integer := PrecisionStates*Parallelism;
26 --Layer 1--
27 constant TypeLayer1: integer :=0;
28 constant RowImage1 : integer := 32;
29 constant ColumnImage1 : integer := 32;
30 constant NChannelsIn1: integer :=18;--never (par*N)+1 with N
     different from 0 (doesn't work)--
31 constant NRowForInputValue1: integer := ((NChannelsIn1-1)/
     Parallelism)+1;--
32 constant ColumnsExtra1: integer := NChannelsIn1 mod Parallelism;
33 constant NFilters1: integer :=32 ;--never 1?
34 constant NRowForResultsValue1: integer := ((NFilters1-1)/Parallelism
     )+1;--
35 constant RowFilter1 : integer := 5;
36 constant ColumnFilter1 : integer := 5;
37 constant StrideH1: integer := 1;
38 constant StrideV1: integer := 1;
39 constant DimensionF1: integer:= RowFilter1*ColumnFilter1*
     NRowForResultsValue1;
40 constant DimensionImageUp1: integer:= ((RowImage1)*(ColumnImage1))
     /2;
41 constant DimensionImage1: integer:= (RowImage1)*(ColumnImage1);
42 constant ChangeColumn1: integer:= 1;-- for conv is fixed to 1--
43 constant ChangeRow1 : integer := (NRowForInputValue1*(ColumnImage1))
     -(NRowForInputValue1*ColumnFilter1)+1;
44 constant NotRightShift1 :integer :=  -NRowForInputValue1*(
     ColumnImage1)*(RowFilter1-1)-(NRowForInputValue1*ColumnFilter1-2)
     -1;
45 constant ShiftRight1 : integer := -NRowForInputValue1*ColumnImage1*(
     RowFilter1-1)-(NRowForInputValue1*ColumnFilter1-2)+(
     NRowForInputValue1*StrideH1-1);
46 constant ShiftDown1 : integer := -NRowForInputValue1*ColumnImage1*(
     RowFilter1-1-(StrideV1-1))+1;
47 constant RowResult1: integer := (RowImage1-RowFilter1)/StrideH1+1;
48 constant ColumnResult1: integer := (ColumnImage1-ColumnFilter1)/
     StrideV1+1;
49 constant DimensionM1:integer:= RowResult1*ColumnResult1*
     NRowForResultsValue1;
50 --Layer 2-- pool
51 constant TypeLayer2: integer :=1;
52 constant RowImage2 : integer := 28;
53 constant ColumnImage2 : integer := 28;
54 constant NChannelsIn2: integer :=1;--in pool must be fixed at one
     and the real value must be gived only in NFilters#--
55 constant NRowForInputValue2: integer := ((NChannelsIn2-1)/
     Parallelism)+1;--
56 constant ColumnsExtra2: integer := NChannelsIn2 mod Parallelism;
57 constant NFilters2: integer :=32 ;--must have this value to keep the
     same order of row in output,obviusly in reality the value is 1 --
```

```
58  constant NRowForResultsValue2: integer := ((NFilters2-1)/Parallelism
      )+1;--
59  constant RowFilter2 : integer := 2;
60  constant ColumnFilter2 : integer := 2;
61  constant StrideH2: integer := 2;
62  constant StrideV2: integer := 2;
63  constant DimensionF2: integer:= RowFilter2*ColumnFilter2*
      NRowForResultsValue2;
64  constant DimensionImageUp2: integer:= ((RowImage2)*(ColumnImage2))
      /2;
65  constant DimensionImage2: integer:= (RowImage2)*(ColumnImage2 );
66  constant ChangeColumn2: integer:= NRowForResultsValue2;
67  constant ChangeRow2 : integer := ( NRowForResultsValue2*(
      ColumnImage2))-( NRowForResultsValue2*ColumnFilter2)+
      NRowForResultsValue2;
68  constant NotRightShift2 :integer :=  - NRowForResultsValue2*(
      ColumnImage2 )*(RowFilter2-1)-( NRowForResultsValue2*(
      ColumnFilter2-1))+1;
69  constant ShiftRight2 : integer := - NRowForResultsValue2*(
      ColumnImage2)*(RowFilter2-1)-( NRowForResultsValue2*(ColumnFilter2
      -1))+( NRowForResultsValue2*StrideH2)-( NRowForResultsValue2-1);
70  constant ShiftDown2 : integer := - NRowForResultsValue2*ColumnImage2
      *(RowFilter2-1)- NRowForResultsValue2*(ColumnImage2-1)-(
      NRowForResultsValue2-1)+(StrideV2* NRowForResultsValue2*
      ColumnImage2);
71  constant RowResult2: integer := (RowImage2 -RowFilter2)/StrideH2+1;
72  constant ColumnResult2: integer := (ColumnImage2-ColumnFilter2)/
      StrideV2+1;
73  constant DimensionM2: integer:= RowResult2*ColumnResult2*
      NRowForResultsValue2;
74  --layer 3-- fully
75 constant TypeLayer3: integer :=3;
76 constant NeuronsInput3: integer :=512;
77 constant NeuronsOutput3: integer := 256;
78 --constant BitForNeuIn: integer:=5;--ceil(log2(NeuronsInput/
     parallelism));--
79 constant BitForNeuOut3: integer:=5;--ceil(log2(NeuronsOutput/
     parallelism));--
80 --BitForNeuIn+BitForParBitForNeuOut less or equal to BitForWeights--
81 constant RemainingBit3: integer :=BitForWeights-BitForNeuOut3-
     BitForPar;
82 constant ThreshVertScan3: integer:=(NeuronsInput3-1)/Parallelism+1;
83 constant ThreshWeightsScan3: integer:=(NeuronsOutput3-1)/Parallelism
     +1;
84
85 end package NeuralNetworkInfo;
```

Bibliography

[1] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. «Gradient-Based Learning Applied to Document Recognition». In: *Proceedings of the IEEE*. 1998, pp. 2278–2324 (cit. on p. 4).

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «ImageNet Classification with Deep Convolutional Neural Networks». In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (cit. on p. 5).

[3] Karen Simonyan and Andrew Zisserman. «Very Deep Convolutional Networks for Large-Scale Image Recognition». In: *CoRR* abs/1409.1556 (2014). URL: http://arxiv.org/abs/1409.1556 (cit. on p. 6).

[4] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. «Going Deeper with Convolutions». In: *Computer Vision and Pattern Recognition (CVPR)*. 2015. URL: http://arxiv.org/abs/1409.4842 (cit. on p. 7).

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. «Deep Residual Learning for Image Recognition». In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385 (cit. on p. 8).

[6] Eugene Izhikevich. «Simple model of Spiking Neurons». In: *IEEE transactions on neural networks / a publication of the IEEE Neural Networks Council* 14 (Feb. 2003), pp. 1569–72. DOI: 10.1109/TNN.2003.820440 (cit. on p. 14).

[7] S. Guo, L. Wang, B. Chen, and Q. Dou. «An Overhead-Free Max-Pooling Method for SNN». In: *IEEE Embedded Systems Letters* 12.1 (2020), pp. 21–24. ISSN: 1943-0671. DOI: 10.1109/LES.2019.2919244 (cit. on p. 60).