Hardware Design and Implementation of Machine Learning Acceleration Network

(Squeeze-Next Architecture)

By

Ahmed Hany Abd-Elrahman

Ahmed Yasser Elbishbishy

Ahmed Mohamed Makram

Abdullah Mohamed Rajab

Mohamed Sayed Mohamed

Mohamed Hamed Mohamed

A graduation project Sponsored by

Siemens EDA (Mentor Graphics) and ***ONELAB***

Under the Supervision of

Prof. Hassan Mostafa

A Graduation Project thesis

Submitted to the Faculty of Engineering at Cairo University

In Partial Fulfillment of the Requirements for the Degree of Bachelor of

Science in Electronics and Communications Engineering

Faculty of Engineering, Cairo University Giza, Egypt

TABLE OF CONTENTS

List Of Figures

List Of Tables

List of Symbols and Abbreviations

ASIC	Application-Specific Integrated Circuit
BRAM	Block Random Access Memory
CIFAR	Canadian Institute For Advanced Research
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CTS	Clock Tree Synthesis
DC	Design Compiler
DPU	Deep learning Processing Unit
DRC	Design Rule Check
DSP	Digital Signal Processing
EEPROM	Electrically Erasable Programmable Read-Only Memory
ESD	Electro-static discharge violations
FC	Fully Connected
FPGA	Field Programmable Gate Array
FPN	Feature Pyramid Network
GDS	Graphic Design System
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High-Level Synthesis
HW	Hardware
I/O	Input/Output
IC	Integrated Circuit
ICC	Integrated Circuit Chip
ICG	Integrated Gated Clock
ILA	Integrated Logic Analyzer
IP	Internet Protocol
LED	Light-Emitting Diode

LEF	Library Exchange Format
LLVM	Low Level Virtual Machine
LUT	LookUp Table
LVS	Layout versus Schematic
MAC	Multiply and Accumulate
MMCM	Mixed Mode Clock Manager
MSE	Mean Squared Error
PAR	Place and Routing
PDP	Power Delay Product
PLD	Programmable Logic Device
PLL	Phase-Locked Loop
PM	Pattern Match
RAM	Random-Access Memory
RE	Read Enable
RELU	Rectified Linear Unit
ResNet	Residual Neural Network
ROM	Read-Only Memory
RTL	Register Transfer Language
SC	Standard Cell
SqNxt	Squeeze-next
SVF	Serial Vector Format
SW	Software
TANH	Tangent hyperbolic
TLU	Table Look Up
VGG	Visual Geometry Group
VHSIC	Very High-Speed Integrated Circuit
VIO	Virtual Input/Output
WE	Write Enable
YOLO	You Only Look Once

Acknowledgments

First and Foremost we thank Allah for all the opportunities, trials, and strength that have been showered on us. Our praise is to Allah, the Almighty, the greatest of all, on whom ultimately we depend for sustenance and guidance.

Second, our appreciation also goes out to our families and friends for their encouragement and support all through our studies.

We are incredibly grateful to our supervisor Dr. Hassan Mostafa for his invaluable advice, continuous support, patience and his caring about following up each stage in the project, also for providing us all tools and equipment we needed

Finally, we want to thank Siemens EDA team represented in Eng. Ziad Ibrahim, QA/test engineer at Questa static for his encouragement and for providing his time and experience to help us overcome some obstacles we faced during some stages. We got from him great experience and knowledge that helped us all the time of the project.

Abstract

The Convolutional Neural Network (CNN) becomes one of the most trending fields of research nowadays because it gives the power to machines to interact with surrounding environment and paving the way to computer vision applications to detect and take actions, organize and take decisions based on this data in all fields of life. It is obvious that GPUs have high power consumption and relatively huge area that make them unable to fit in mobile devices. CNN architecture that uses FPGA implementation proved to be faster when it gets compared to traditional GPU and CPU, allowing it to be used in real time applications. Using Squeeze net CNN architecture will make FPGA speed go higher, which is needed with today's environment as it allows for fast automation process and better application learning in many fields.

The aim of this project is to make, implement and design Squeeze-Next CNN architecture on a Virtex-7 FPGA that is available on VC709 board and to finish the ASIC flow with the best possible throughput.

Chapter 1: Introduction

1.1 Motivation

Deep convolutional Neural Network (CNN) has a lot of ways to be used in many fields like computer vision. Squeeze next [1], Alex net [2], VGG16 [3] and Squeeze net [4] have gotten better with image recognition the introduction of better computation power and continuous research in the deep learning algorithms. To work out the details of an image is a tough task for researchers lately. In the past, they had to rely on hard codes and computer vision to make the image detection work, but the newer deep learning algorithms have been taken their place instead. They have proven themselves to be better at dealing with higher resolutions. The latter allowed also to make the model shrink in size and higher inference time as the network became even more in depth, which is important as embedded systems gives so little resources to begin with. The engineer must be considerate with the resources given to from small computational power to limited memory. For the distributed training on parallel servers, the overhead to train the model is directly proportional to the number of parameters and complexity of the model. The model size of CNN can play a vital role when it comes to transporting the model wirelessly in terms of the firmware or software updates.

The CNN [5] consists of various layers such as the convolution layer, pooling layer, RELU layers, and Fully Connected layer (FC). The convolution layers take high computational power as compared to the other layers because of complex matrix multi plication. Several network compression techniques are available such as architectural compression, pruning, quantization and encoding techniques (Human encoding). However, many of these methods reduce the model size but results in a significant accuracy drop of the model.

1.2 Problem Definition

Deep Neural network is one of the hot researches nowadays, there are attention begun to turn on this powerful field and may companies aspire to achieve as deep as they can in this field of technology, however it's very important to specify where to implement this Network on. There is a challenge between companies to implement Convolution Neural Network (CNN) that can be fast as it can and more accurate, and this Neural Network one of the most commonly used network in the Deep neural network, which made a great achievement in the image processing field for their high accuracy and low numbers of computation it takes to processes an image if compared with deep neural network, however there is a huge number of computation it takes to process one single image due to the convolution operation for multiple dimensions data and this reflects the high power consumption, large amount of resources and memories, so there is a tradeoff between the accuracy and power consumption, amount of resources and memories, if you need more accurate network all you have to do is to pay the price, however now the network's training phase which take place on a computer cluster are not a big deal in power consumption so the power consumed in the testing phase only.

Any Machine learning algorithm can be implemented on GPUs, FPGA and ASIC, there are challenges among those three platforms in order to get the fast response (high throughput), less delay, low power consumption and more efficient, since advanced machine learning are powered with Deep learning so the Deep learning define which platform is better.

The GPUs the most powerful among the three platforms because of having a huge number of cores and large external bandwidth, it is too easy to perform the training, testing and deploying the models using various framework and in floating-point representation which gives the highest accuracy and high performance for the model itself, however the GPUs are the most consuming power among the three, and it's not the best thing for all application that depends on its cost and amount of resources, the throughput of GPUs are high, however they cannot reach the maximum throughput due to performing other applications in parallel with the model itself.

The next one is ASIC, from the advantage of implementing the CNN model on ASIC is you can achieve the highest power efficiency and best performance, nowadays researches are held on building CNN various models on ASIC to accelerate the CNN, However ASIC models are not flexible enough to cope up with the evolution of CNN models, also the cost and time for design, verification and fabrication.

The last one is FPGA, it is obvious that FPGA compensate between the advantages of the GPUs and ASIC, like low power consumption, low area, fast response and high performance it does not add too much cost like GPUs in the application, however it is limited with finite amount of resources and memories and also not the best on mass production but still the most commonly used for development and testing.

The most important thing in this field is the model cope up with the real time application many application depends on camera photos with large number of pixels that translated to huge data and these application need to respond as fast as they can to avoid any catastrophic or financial lost so that they need fast response with optimum accuracy.

1.3 Organization

CHAPTER2, in this chapter we will give a summary of different architectures, a brief background of convolution neural network, terms that might be used in this chapter and a discussion of Squeeze Next FPGA ASIC.

CHAPTER3, in this chapter provides a discussion about the main idea of our architecture, how to use parallelism, how to use pipelining, adder tree to implement convolution, batch normalization implementation and optimization and main idea of our memory.

CHAPTER4, this chapter provides a discussion about our detailed design, integration idea, implementation and optimization in small and huge layers, skip connection merging idea, skip connection memory optimization, responsibilities of controller and its problems, average pooling and fully connected layer.

CHAPTER5, this chapter discusses the software part discusses the different versions of the squeezenext networks and choosing the suitable one. Also we discussed the batch normalization problem and its solution. And finally the generation of test vectors for each layer.

CHAPTER6, this chapter discusses different optimization techniques and their effect on different fields, we applied the timing enhancements to increase the operating frequency and the total throughput, the area aware enhancements used to reduce the number of resources used, hence reduce the consumed power, Power reduction techniques are used to decrease the total dynamic power and the Accuracy Enhancements make the HW model accuracy reach to the SW model accuracy.

CHAPTER7, this chapter provides a discussion on the synthesis flow, the used constraints, the implementation flow and the best strategies for our design then generating and downloading the bitstream on FPGA.

CHAPTER8, this part introduces the ASIC part for the project and why we choose to follow the ASIC flow and the problems we have faced in ASIC and how we solve them like memories, the method and steps we followed in the ASIC flow.

CHAPTER9, this chapter shows the RTL verification methodology and the calculated accuracy then the calculated results including utilization, power, timing, throughput and energy at different frequencies. Finally we compared our design with previous works which implemented on FPGA with only 10 classes.

CHAPTER10, in this chapter, extra modifications that can be done to implement Squeeze-Next more efficiently with reduction in used clock cycles and FPGA resources will be presented.

Chapter 2: Background And Related Work

2.1 Introduction To Neural Network

Deep Learning is a subfield of machine learning concerned with algorithms inspired by the structure of the brain which is called artificial neural networks.

The purpose of artificial neural networks is to achieve a very simplified model of the human brain. By having the artificial neural networks try to learn tasks mimicking the brain's behavior. The brain consists of a large set of neurons which are specialized cell elements. These neurons are activated in response to the input, the activation of the neurons allows the network to detect and classify the patterns. Depending on certain input data, a neural network will try to calculate the probability that the data belong to a certain class (e.g., an object in a specific image). The neural network can be trained to recognize different classes by being provided a set of labeled training data which is called supervised learning[6].

2.2 Network Training

Convolutional Neural Networks (CNNs) are a special type of Neural Networks which are commonly used with visual data, which have shown state-of-the-art performance on various competitive benchmarks. The powerful learning ability of deep CNN is largely due to the use of multiple feature extraction stages (hidden layers) that can automatically learn representations from the data. The topology of CNN is divided into multiple learning stages composed of a combination of the convolutional layer, non-linear processing (RELU) units, and subsampling (Pooling) layers. Each layer performs multiple transformations using a bank of convolutional kernels (filters). Convolution operation extracts locally correlated features by dividing the image into small slices (similar to the retina of the human eye), making it capable of learning suitable features. Output of the convolutional kernels is assigned to non-linear processing (RELU) units, which not only helps in learning abstraction but also embeds non-linearity in the feature space. This non-linearity generates different patterns of activations for different responses and thus facilitates in learning of semantic differences in images. Output of the non-linear

function (RELU) is usually followed by subsampling (Pooling), which helps in summarizing the results and also makes the input invariant to geometrical distortions [7].

2.3 Training Process

The basic learning that has to be done in neural networks is training neurons when to get activated. Each neuron should activate only for particular type of inputs and not all inputs. Therefore, by propagating forward, it is noticed how well the neural networks are behaving and find the error. After finding out that the network has error, backpropagation is applied and a form of gradient descent is used to update new values of weights. Then, forward propagation is applied again to see how well those weights are performing and then the weights are updated using backpropagation. This will go on until reaching some minima for error value.

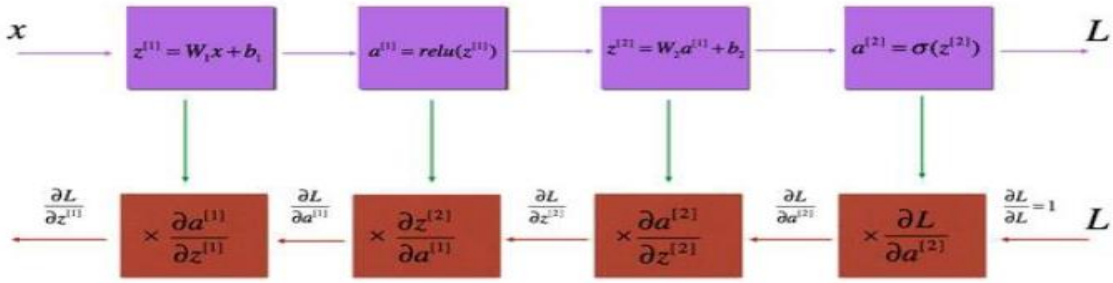

Figure 1: Training process of neural network

2.3.1 Forward Propagation

In forward Propagation as in Figure, in 1st row, input X is provided to each neuron and two functions are calculated, one is linear multiplication i.e. $Z=W\times X+b$ and the other is activation function a=RELU(z), different activation functions can be used, then it will forward through every layer and predicted output is obtained.

2.3.2 Backward Propagation

Back propagation is a technique to reduce the loss i.e. (Actual o/p-predicted o/p) by updating the parameters weight, bias by using an algorithm called Gradient descent. For example in 2nd row last column, Loss(L) is partially differentiated w.r.t a[2] but a[2] depends on z[2], again z[2] depends on weight w[1], activation a[1], and bias b[1], so gradients of a[1], w[1] and b[1] is calculated w.r.t Loss(L). Then by using Gradient

Descent algorithm, weight and bias in that layer are updated but again a[1] depends on calculation of z[1]. Above procedure will be repeated till first layer. Finally, it will propagate backside to reduce the loss by calculating all parameters gradients w.r.t Loss(L), and update them by using Gradient Descent algorithm.

2.3.3 Loss Function

A loss function can be defined in many different ways but a common one is MSE (Mean Squared Error), which are half times (actual - predicted) squared. E_{total} =Σ $\frac{1}{2}(target-output)^2$.

The predicted label (output of the CNN) must be the same as the training label (This means that the network got its prediction right). In order to achieve this, it's a must to minimize the amount of loss (error). It just an optimization problem in calculus to find out which inputs (weights) most directly contributed to the loss (or error) of the network as shown in Figure.

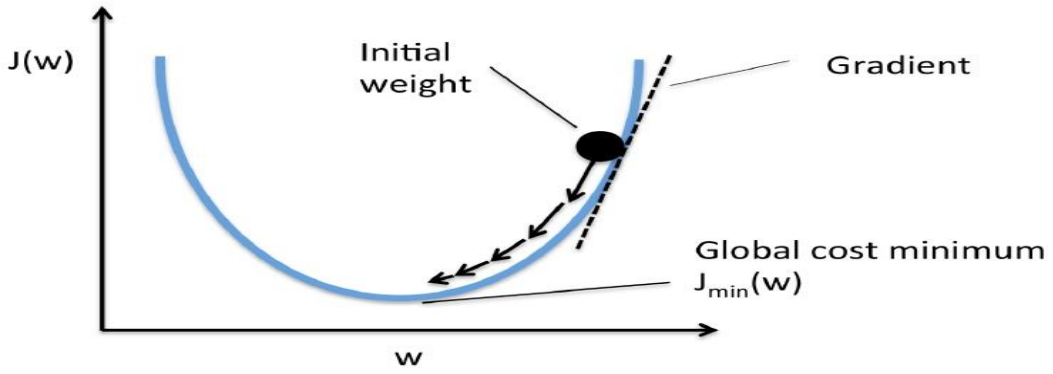

Figure 2: Cost function of CNN

2.4 Introduction To CNN

A Convolutional Neural Network (Conv-Net/CNN) is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a Conv-Net is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, Conv-Nets have the ability to learn these filters/characteristics.

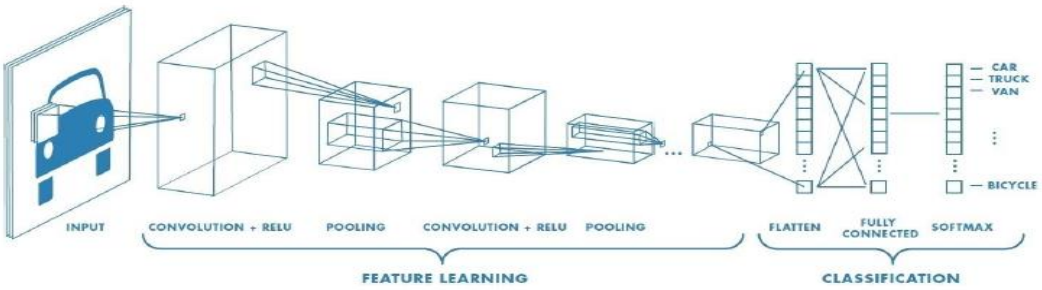

Figure 3: CNN classification example

Convolutional neural networks are constructed by stacking a number of generic network layers, which transform the input feature maps of dimension $(in* w_{in}*ch_{in})$. Into output feature maps of dimension $(h_{out}* w_{out}*ch_{out})$. A typical CNN path consists of two parts:

1. The feature extractor which extracts features across the CNN layers which are; Convolutional (Conv), Pooling (Pool) Rectified Linear Unit (RELU).
2. The Classifier, which is implemented using fully connected layers, takes these features and decides on the output class.

In order to understand the proposed hardware implementation, the CNN detailed layers will be discussed in this section.

2.4.1 Convolution Layer

The Conv layer is the main block of a CNN that does most of the computations. It works by dividing the image into small regions (known as receptive field) and convolving them with a specific filter (multiplying weights of the filter or kernel (weights) with corresponding receptive field elements), then sliding these filters over the input feature maps as shown in figure. Each of these weight filters can be thought of as feature identifiers. [8]

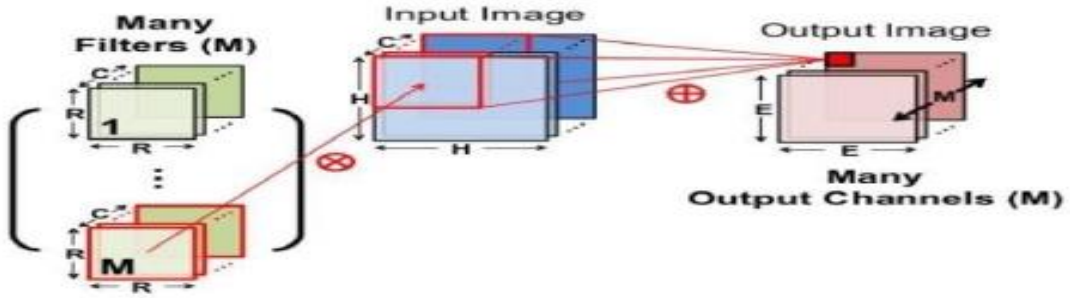

Figure 4: Convolution Layer process example

There are 2 parameters in conv layers which are listed below:

1. **Filters:** The Conv layer's parameters consist of a set of learnable filters which work as feature detector (edges, simple colors, and curves).
2. **Stride:** Stride is the number of pixels by which the filter matrix slides over the input matrix.

2.4.2 Nonlinearity Layers

These layers are used to introduce the non-linearity in the model. This helps to train the model faster and more accurately, and it helps to reduce the overfitting of the model on the training dataset. Rectified linear unit (RELU) is one of the favorite activation functions used in CNN. RELU function converts all negative numbers to 0 and keeps the positive number. RELU helps to reduce the computational complexity, and it helps in convergence while training.

2.4.3 Normalization Layer (Batch Normalization)

Normalization layer is used to normalize the responses from the adjacent output channels. It normalizes the output distribution to zero mean with unit variance.

2.4.4 Pooling Layer

In CNN, it is used to down-sample the input features with non-linearity. It reduces the resolution of the input feature map but keeps the features intact. [9]

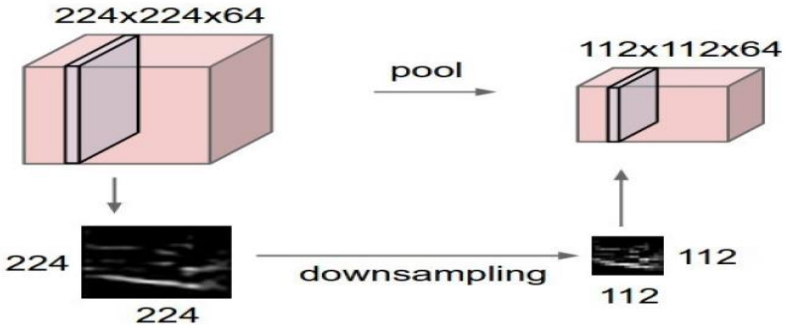

Figure 5: Pooling Layer process example

There are two popular choices of pooling:

1. *Max pooling:* Suppose you have 4×4 input and you want to apply max-pooling. It is quite simple to take 4×4 break it into different regions. The output is 2×2 each of the outputs will just be the max from the correspondingly shaded region.

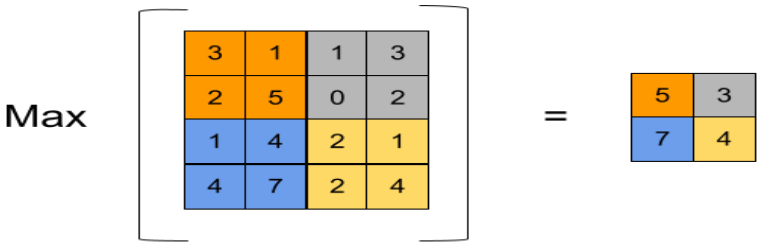

Figure 6: Max pooling example

2. *Average pooling:* instead of taking the max within each filter you take the average.

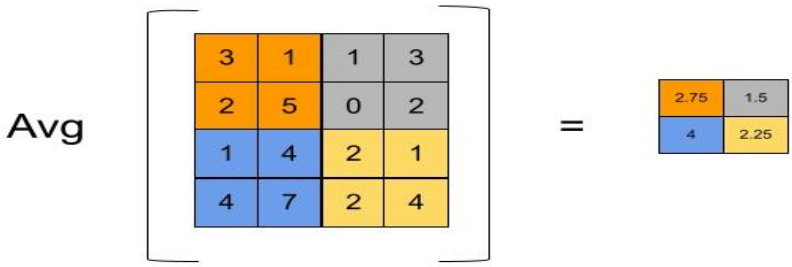

Figure 7: Average pooling example

2.4.5 Fully Connected Layers (FC)

These layers are often used as the last layers in the CNN to compute the class score of the classification. This layer has a full connection to all previous activations. FC layers act as a classifier on the top of high-level features. [6]

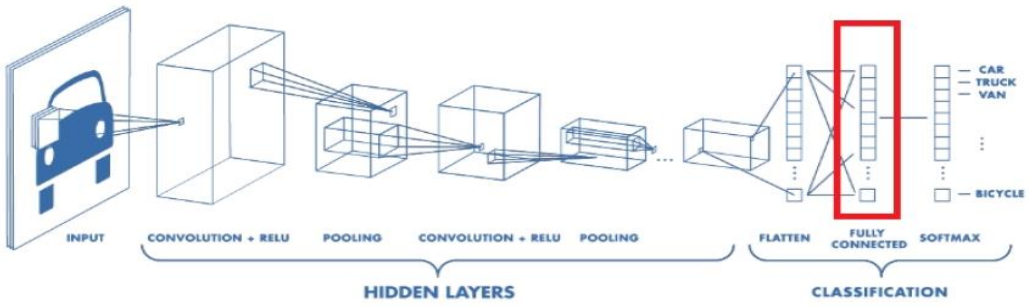

Figure 8: Fully Connected Layers

2.4.6 Activation Layer

Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated ("fired") or not, based on whether each neuron's input is relevant for the model's prediction. It's important that activation functions be

computationally efficient because they are calculated several times (thousands or even millions) for each data sample. [10]

2.4.6.1 Binary Step Function

The binary step function outputs one if the input is positive and zero otherwise. Step function cannot support classifying the inputs into one of several categories. As they produce don't support multi-value output.

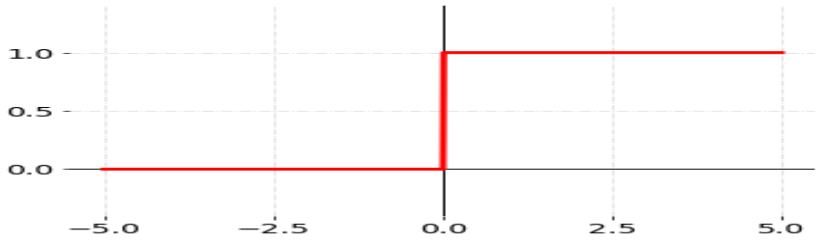

Figure 9: Binary Step Function

2.4.6.2 Linear Activation Functions

It takes the inputs, multiplied by the weights for each neuron, and creates an output signal proportional to the input as shown in figure In one sense, a linear function is better than a step function because it allows multiple outputs, not just yes and no.

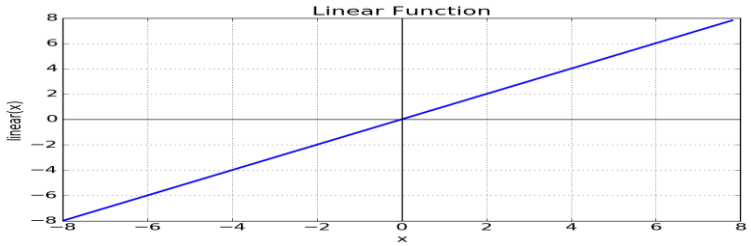

Figure 10: Linear Activation Functions

However, a linear activation function has two major problems:

1. Not possible to use backpropagation (gradient descent) to train the model since the derivative of the function is a constant, and has no relation to the input, X.
2. A linear activation function makes the neural network unable to fit non-linear data. They turn the neural network into just one layer; the last layer will be a linear.

2.4.6.3 Nonlinear Activation Function

Networks use a non-linear activation functions which help them fit non-linear data such as images, video, audio, etc. [10]

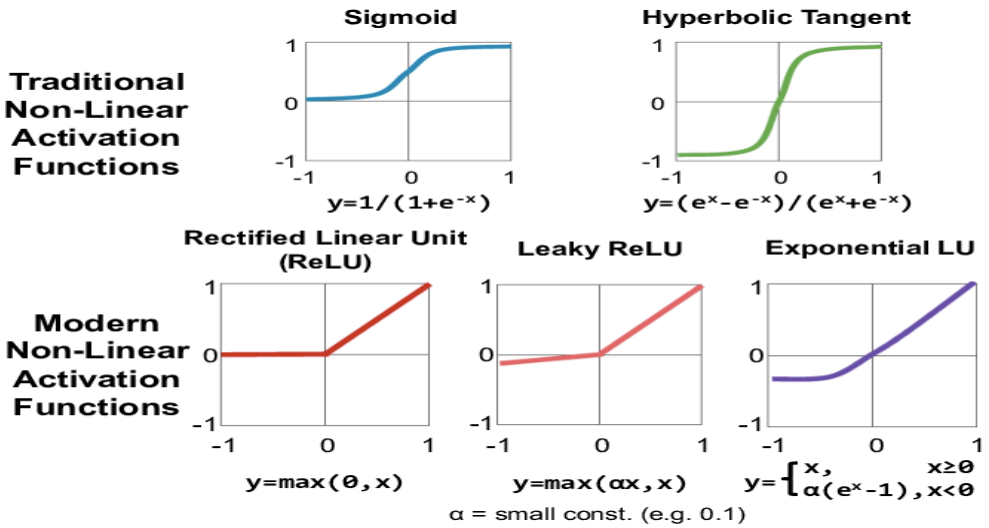

Figure 11: Nonlinear Activation Functions examples

Non-linear functions fix the problems of a linear activation function:

1. They have a derivative function which is related to the inputs and so allow backpropagation.
2. They allow stacking of multiple layers of neurons to create a deep neural network. Deep neural network with multiple hidden layers are needed to learn complex data with high levels of accuracy.

There are seven types of Nonlinear Activation functions and here are the most popular nonlinear activation functions:

1. SIGMOID /LOGISTICS FUNCTIONS:

Advantages

- Smooth gradient, preventing “jumps” in output values.
- Output values between 0 and 1, normalizing the output of each neuron.
- Clear predictions—for X above 2 or below -2, tends to bring the Y value (the prediction) to the edge of the curve, very close to 1or0.

Disadvantages

- Vanishing gradient: For very high or very low input values, the derivative is almost zero, this can result in the network refusing to learn, or being too slow to reach an accurate prediction.
- Outputs not zero centered.
- Computationally expensive due to the exponential function.

2. TANH /HYPERBOLIC TANGENT:

Advantages

- Zero centered: which make it easier to model inputs that have negative, neutral, and positive values.
- Otherwise like the sigmoid function.

Disadvantages

- Like the Sigmoid function.

3. RELU (RECTIFIED LINEAR UNIT): The RELU is a piece-wise linear function. It outputs the same input if it's positive and outputs zero for negative values.

Advantages

- Computationally efficient
- No vanishing gradient problem for positive inputs

Disadvantages

- For negative inputs the gradient becomes zero, the network cannot perform backpropagation and cannot learn. This issue is called the dying RELU.

RELU and its variants are preferred over others activations as it helps in overcoming the vanishing gradient problem.

2.5 Different CNN Architectures

In recent years, the world witnessed the birth of numerous CNNs. These networks have gotten so deep that it has become extremely difficult to visualize the entire model. We stop keeping track of them and treat them as black box models. These illustrations provide a more compact view of the entire model of each architecture.

2.5.1 ALEXNET (2012)

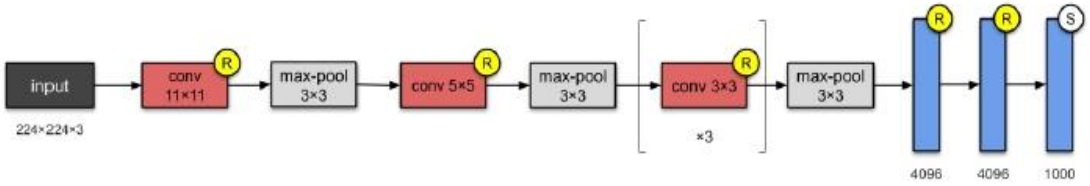

Figure 12: ALEXNET

Alex-Net is one of most famous CNN architecture because it was the first CNN to win the ILSVRC in 2012.AlexNet consist of 5 conv layers and 3 max-pooling layers in addition to 3 fully-connected layers in the end of architecture.

It has a huge size and 60 million parameters; Alex-Net was the first to implement:

1. Rectified Linear Units (RELUs) as activation functions.
2. Overlapping pooling in CNNs.

2.5.2 VGGNET (2014)

In this architecture the designers from Visual Geometry Group (VGG) depend on the most straightforward way to improve the accuracy of CNN is to go deeper and increase the number of layers and parameters.as shown in figure, VGG-Net has 13 convolutions layers and 5 max-pooling layers followed by 3 fully-connected layers in near to output. And it depends also on RELU the same activation function of Alex-Net. VGG-Net has **138M parameters** and takes up about 500MB of storage space.it has another version VGG-19 with a greater number of layers and higher accuracy.

Figure 13: VGGNET

2.5.3 GOOGLENET (INCEPTION V1)

This CNN architecture is by Google. It was the winner of ILSVRC 2014 challenge. The main highlight of this architecture was the inception model which

dramatically reduced the number of parameters of the model. It has 4 million parameters compared to AlexNet with 60 Million. It uses average pooling instead of fully connected layers at the top layer. The most recent version of GoogLeNet is inception-v4.[11]

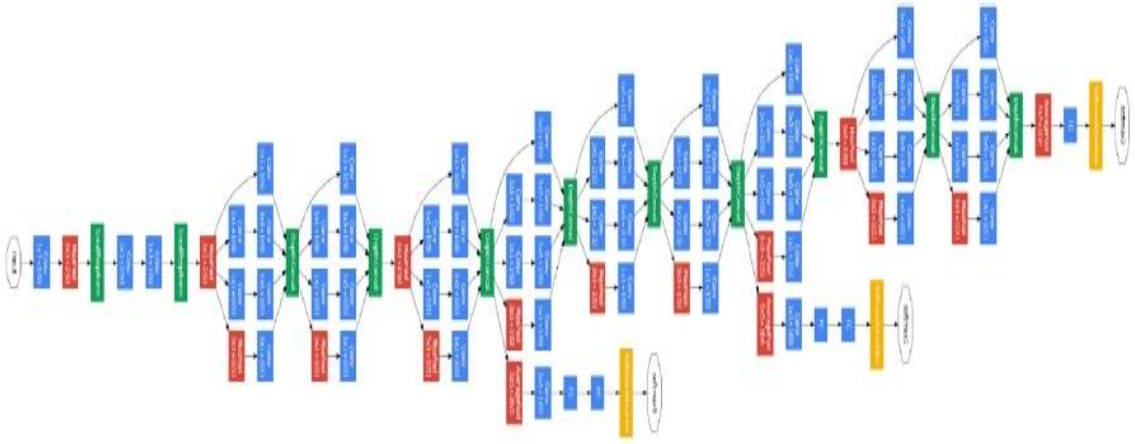

Figure 14: GOOGLENET

2.5.4 RESNET-50 (2015)

Residual Neural Network (ResNet)[12] in Figure was introduced by Kaiming He et al from Microsoft Research, skip connections were used in which helped in making CNN much deeper, there are multiple versions of ResNet architectures but most commonly used is ResNet152 which won 2015 ILSVRC competition and consists of 152 layers, it achieves top-5 error rate of 3.57% and contains only 60 million parameter which is considered small number its huge number of layers.

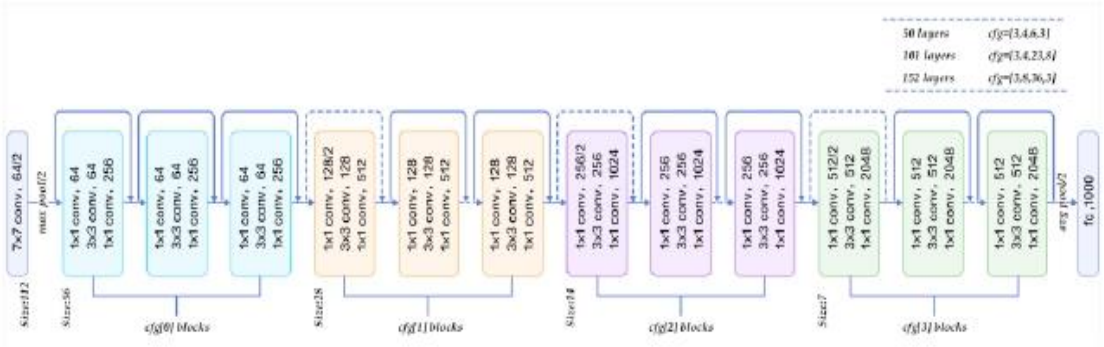

Figure 15: RESNET

2.5.5 SQUEEZE-NET

This is a small CNN architecture which achieves the AlexNet-level accuracy on ImageNet with 50x fewer parameters. This type of small architecture is beneficial for efficient distributed training; it has very little overhead while exporting new models to clients and it is feasible for FPGA and embedded deployment.

In the SqueezeNet paper, the author outlines three main strategies:

1. Replace 3 x 3 kernels with 1 x 1 for smaller network: To reduce the network parameters, they used 1 x 1 filters instead of 3 x 3. Since 1 x 1 kernels have 9x less parameters than 3 x 3 kernels.

2. Decrease the number of input channels to 3x3 kernels: Using the 1x1 kernels as bottleneck layer called squeeze layer as shown in the Figure 5.5 to reduce the depth of the model and reduce the computation for the following 3x3 kernels.

3. Down-sampling later in the network to keep the large activation maps to preserve the feature maps: The intuition is that, the large activation map is kept due to delayed down-sampling in the network, which results in higher classification accuracy.

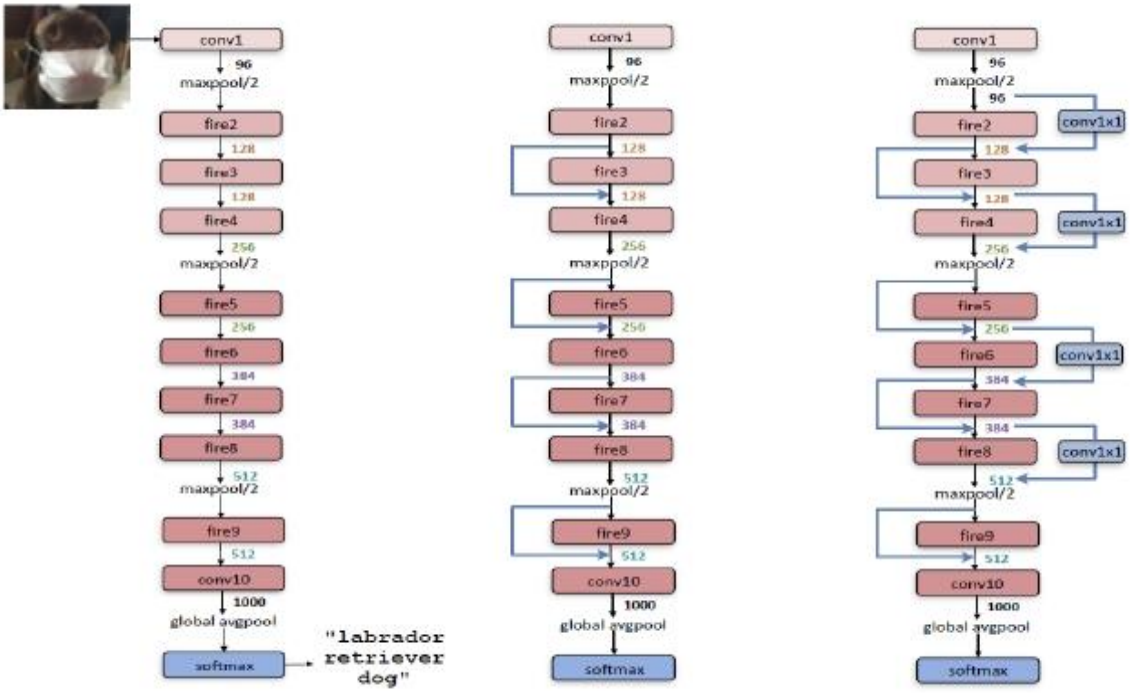

Figure 16: SQUEEZE-NET

2.5.6 SQUEEZE-NEXT

SqueezeNext is a new family of neural network architectures whose design was influenced by simulation outcomes on a neural network accelerator as well as by taking into account earlier structures like SqueezeNet. With 112 less parameters, this new network can match AlexNet's accuracy on the ImageNet test, while one of its deeper variations can match VGG-19 accuracy with only 4.4 million parameters—31 fewer than VGG-19. In addition, SqueezeNext outperforms MobileNet's top-5 classification accuracy while utilizing 1.3 less parameters, although it eliminates the use of depthwise-separable convolutions, which are ineffective on some mobile processor architectures. Depending on the resources available on the target hardware, the user can make a tradeoff between speed and accuracy thanks to this wide range of accuracy. We were able to construct versions of the baseline model that are 2.59/8.26 quicker and 2.25/7.5 more energy efficient than SqueezeNet/AlexNet without any accuracy loss by using hardware simulation findings for power and inference speed on an embedded system.

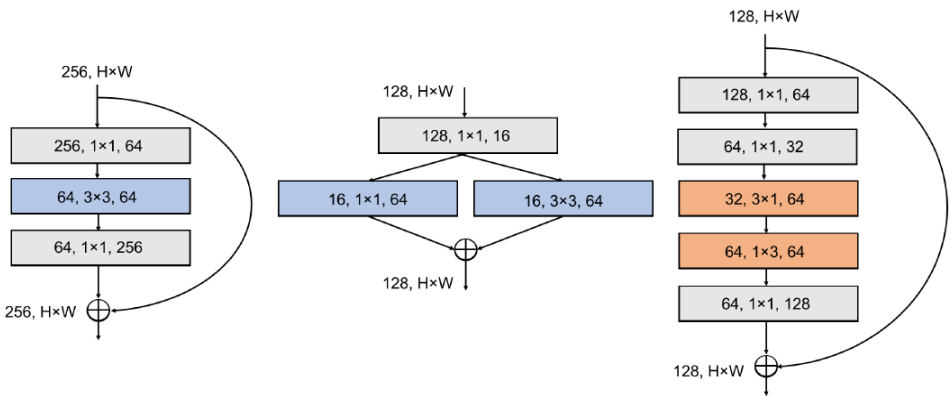

Figure 17:ResNet block on the left, SqueezeNet block in the middle, SqueezeNext block on the right

ResNet, SqueezeNet, and SqueezeNext (SqNxt) blocks are shown in the diagram on the left, middle, and right, respectively. To limit the number of input channels to the 3 * 3 convolution, SqueezeNext employs a two-stage bottleneck module. In order to further minimize the number of parameters (orange sections), the latter is further divided into separable convolutions, which is followed by a 1 * 1 expansion module.

Model	Params (× 1E+6)	MAC	Top-1	Top-5	Depth	8x8, 32KB		16x16, 128KB	
						Time	Energy	Time	Energy
AlexNet	60.9	725M	57.10	80.30	—	x5.46	1.6E+10	x8.26	1.5E+10
SqueezeNet v1.0	1.2	837M	57.50	80.30	—	x3.42	6.7E+09	x2.59	4.5E+09
SqueezeNet v1.1	1.2	352M	57.10	80.30	—	x1.60	3.3E+09	x1.31	2.4E+09
Tiny Darknet	1.0	495M	58.70	81.70	—	x1.92	3.8E+09	x1.50	2.5E+09
1.0-SqNxt-23	0.72	282M	59.05	82.60	[6,6,8,1]	x1.17	3.2E+09	x1.22	2.5E+09
1.0-SqNxt-23v2	0.74	228M	58.55	82.09	[6,6,8,1]	**x1.00**	2.8E+09	x1.13	2.4E+09
1.0-SqNxt-23v3	0.74	228M	58.18	81.96	[4,8,8,1]	x1.00	2.7E+09	x1.08	2.3E+09
1.0-SqNxt-23v4	0.77	228M	59.09	82.41	[2,10,8,1]	x1.00	2.6E+09	x1.02	2.2E+09
1.0-SqNxt-23v5	0.94	**228M**	**59.24**	82.41	[2,4,14,1]	x1.00	**2.6E+09**	**x1.00**	**2.0E+09**

Table 1: Simulated hardware performance results to some CNNs

Observe how the SqueezeNext model's changes improve inference and power usage over the original model. For instance, compared to the baseline model, the 1.0-SqNxt-23v5 model is 12 percent quicker and 17 percent more energy efficient.

2.6 Point Representation

2.6.1 Fixed-Point Representation

A fixed-point representation of a number consists of integer and fractional components. The bit length is defined as: WL=IWL+FWL+1.

With this representation the range of the number is $[-2^{IWL}, 2^{IWL}]$ and a step size (resolution) of 2^{-FWL}. [13]

The fixed-point number system shows the best trade-off between accuracy and computational complexity in hardware-based applications. Fixed-point has two main advantages. Firstly, the smaller hardware implementation of fixed point-based system allows for more modules to be instantiated in same area with same logic gates that increases the opportunities of parallelism and pipelining. Secondly, the smaller data representation of parameters or input pixels reduces the required memory, enabling larger CNN models to fit within the given memory capacity and reducing the power of memory-access because more data fit within same memory bandwidth. [14]

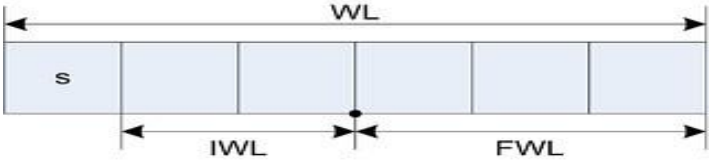

Figure 18: Fixed-Point REPRESENTATION

2.6.2 Floating-Point Representation

Floating point number has 3 parts: sign, exponent and mantissa.

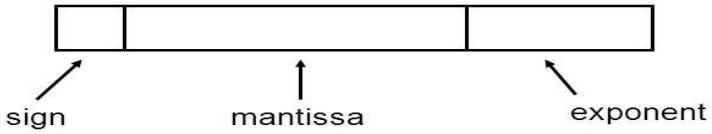

Figure 19: Binary Floating-Point REPRESENTATION

Floating point math offers a wide range of numbers and more precision than fixed point math. The problem with floating point numbers is that floating point operations is slow .They also have to deal with very large registers. [15]

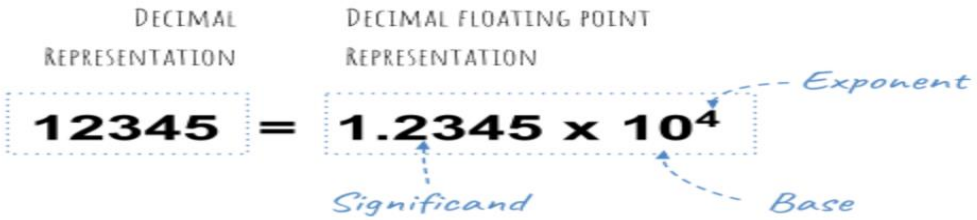

Figure 20: Decimal Floating-Point REPRESENTATION

2.7 Parallelism

Parallelism[16] means many calculations or the execution of processes are carried out simultaneously, that will decrease the time of execution of CNN architectures. For example, two-dimensional convolution is computed between sliding windows of input feature maps and kernels, and consumes most computation time of CNN as shown in Figure. Parallelism included inside an output feature map of each layer, known as intra-output parallelism.

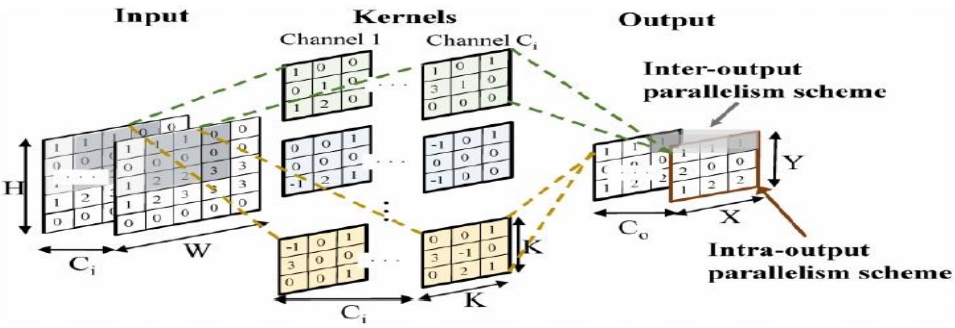

Figure 21: Parallelism process

2.7.1 Inter Layer Parallelism

Data between the layers of CNN architecture are dependent so no layer can be executed before another and layers can't work in parallel.

2.7.2 Inter Output Parallelism

Every output feature map is result of convolution between the input feature map and filter. All filters are independent and different so the output feature maps are totally independent of each other. The convolution between the input volume and different filters can be performed in parallel without problems.

2.7.3 Inter Kernel Parallelism

The output feature map is result of swapping filter on all input feature map and performing convolution. Each convolution is independent since input pixels are independent from each other so it is possible to compute all of output pixels in output feature map concurrently.

2.7.4 Intra Kernel Parallelism

The convolution is set of multiplication and addition operations. Every multiplication operation between the input pixel and corresponding parameter in filter and all of operations are independent and the addition operation of results from multiplication to calculate the output pixel can be performed in parallel also.

2.8 Pipelining

The approach is to rearrange the algorithm into a pipeline, where each stage can operate simultaneously with the other stages as shown in Figure. Pipelining tends to be faster and it can even be more resource efficient.

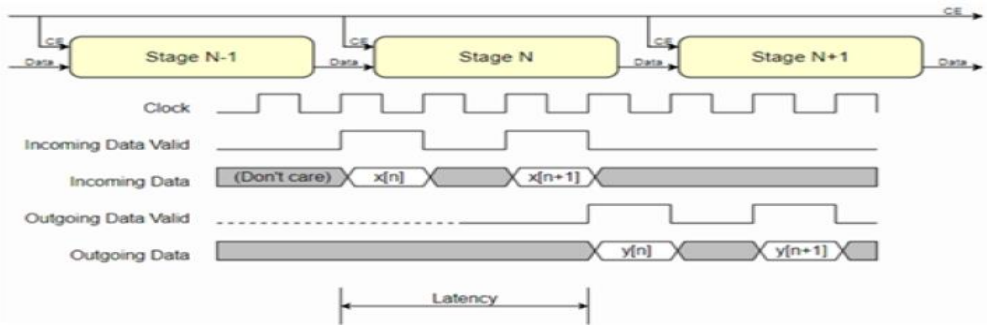

Figure 22: PIPELINING example

2.9 Tensorflow

Machine Learning applications that is Accelerated on Field-Programmable Gate Arrays (FPGAs) has many advantages over other computing platforms. While the machine learning architecture is often developed in a high-level language such as Python, the manual transformation of the algorithm to C code for High-Level Synthesis (HLS) or to Register Transfer Level (RTL) code for synthesis consumes many times and requires digital design engineers.

In order to show how we can make FPGAs more accessible to software developers, LeFlow, an open-source tool which allows machine learning software developers to automatically synthesize TensorFlow numerical computation models to FPGA hardware. The flow uses Google's XLA compiler to generate LLVM code from a TensorFlow specification, which is then delivered to a high level synthesis tool such as LegUp to generate RTL. So, we can write the TensorFlow code and it will be automatically converted to RTL code. [17]

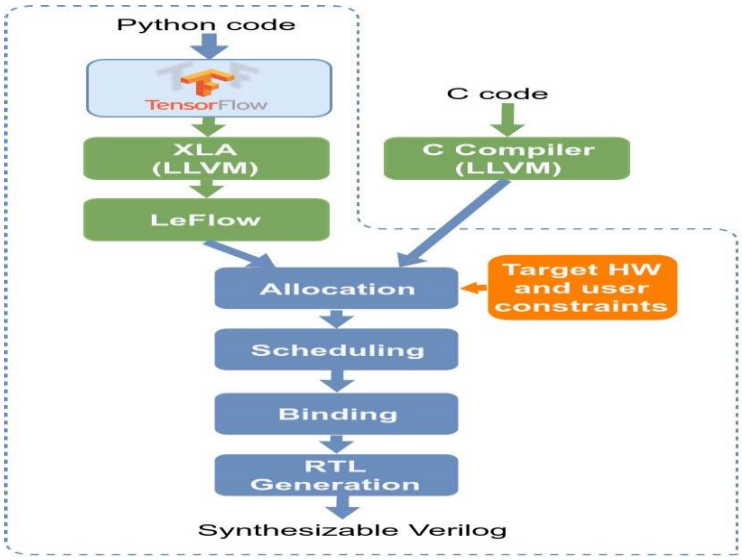

Figure 23: TENSORFLOW algorithm

2.10 DPU

The DPUCZDX8G is the deep learning processing unit (DPU) designed to support the Zynq UltraScale+ MPSoC. It is a configurable engine which is optimized for the computation of convolutional neural networks. There is many options and parameters

that can describe the parallelization utilized in the engine. Where this options can be selected according to the target device and application. It supports many common convolutional neural networks, such as VGG, ResNet, GoogLeNet, YOLO, SSD, MobileNet, FPN, and others, since it contains a set of optimized instructions.

The Xilinx® DPUCZDX8G is a programmable engine optimized for convolutional neural networks. It is consisting of a scheduler module with high performance, a hybrid computing array module, a fetching unit module for the instructions, and a global memory pool module. Due to this available powerful hardware that the DPU provides for the CNNs it became easier to develop CNN network on the hardware.

There are two flows that can be used for integrating the DPUCZDX8G into the project: the VIVADO® flow and the Vitis™ flow. [18]

Figure 24: Zynq UltraScale+ MPSoC

2.11 FPGA

2.11.1 Overview

An integrated circuit that comprises of internal hardware blocks with user-programmable interconnects to tailor operation for a particular application is called a field-programmable gate array (FPGA) [19]. The interconnects are easily

reprogrammable, enabling an FPGA to enable a new application or accept design modifications over the course of the part's lifetime.

A basic FPGA architecture consists of thousands of fundamental elements called configurable logic blocks (CLBs) surrounded by a system of programmable interconnects, called a fabric, that routes signals between CLBs. Input/output (I/O) blocks interface between the FPGA and external devices.

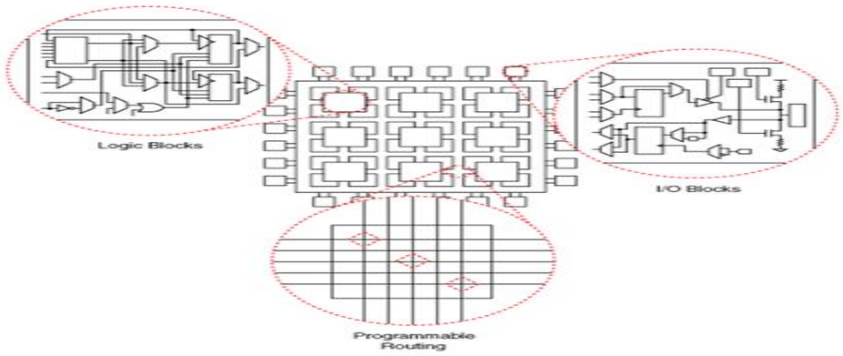

Figure 25: Basic FPGA architecture

Multiple logic blocks make comprise a single CLB. An FPGA's lookup table (LUT) is one of its distinguishing characteristics. For any combination of inputs, a LUT holds a preset list of logic outputs; LUTs with four to six input bits are common. Multiplexers (mux), full adders (FAs), and flip-flops are examples of common logic.

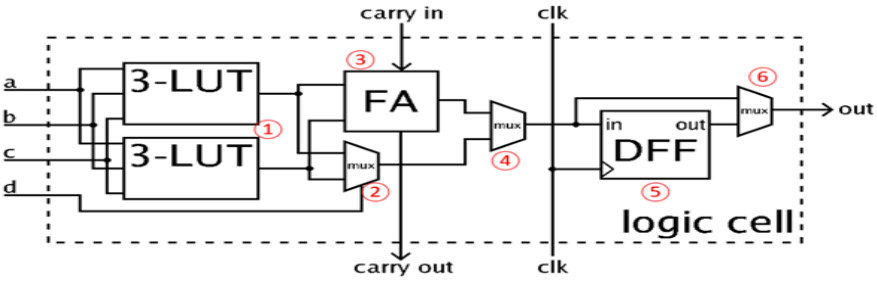

Figure 26: Logic block components

Each device's CLB has a different number and configuration of parts; this condensed CLB has two modes of operation. In arithmetic mode, the LUT outputs are provided as inputs to the FA along with a carry input from another CLB. In normal mode, the LUTs are coupled with Mux 2 to produce a four-input LUT. The FA output and the LUT output are chosen by Mux 4. Through the D flip-flop, Mux 6 determines if the operation is asynchronous or synchronized to the FPGA clock.

The more advanced CLBs found in current-generation FPGAs can do many operations with only one block, and they can be combined to perform more intricate tasks like multipliers, registers, counters, and even digital signal processing (DSP) operations.

In order to create an FPGA-based design, the necessary computing jobs must first be defined in the development tool before being combined into a configuration file that specifies how to connect the CLBs and other modules. The procedure resembles a software development cycle, but instead of creating a set of instructions to run on a predetermined hardware platform, the objective is to construct the hardware itself.

To create an FPGA setup, designers have historically used a hardware description language (HDL) like VHDL or Verilog.

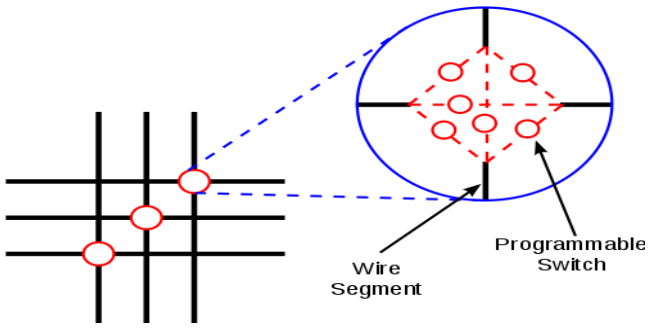

Figure 27: FPGA Wiring

The FPGA routing is typically not segregated. That is, before ending in a switch box, each wiring segment only crosses one logic block. Longer routes can be created by activating several of a switch box's programmable switches. Some FPGA systems use longer routing lines that span many logic blocks for faster performance connection. There is a switch box wherever a vertical and a horizontal channel cross. In this system, a wire can connect to three other wires in adjacent channel segments through three programmable switches that are present when it enters a switch box. The planar or domain-based switch box topology is the switch pattern or topology utilized in this architecture. In this switch box layout, a track number one wire only links to other track number one wires in neighboring channel segments, a track number two wire only connects to other track number two wires, and so on. The connections in a switch box are shown in the right figure. Typically, the width of each routing channel is the same

(number of wires). In the array, more than one I/O pad may fit within a row's height or a column's breadth.

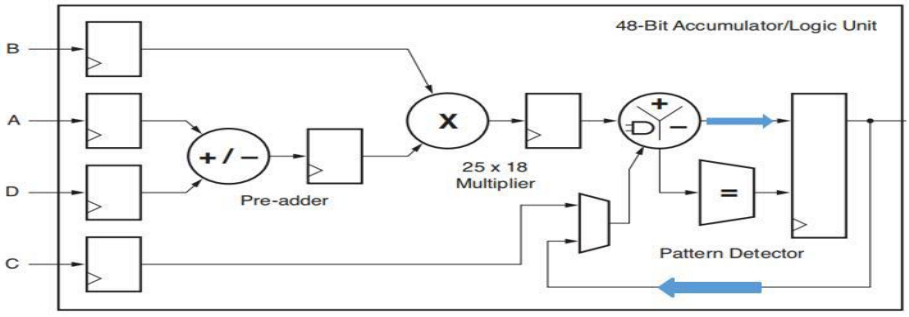

Figure 28:48-bit logic unit

An additional popular form of core that is available as an IP core or an embedded core is digital signal processors (DSPs), as seen in Figure. To manipulate analogue signals, these are essentially specialized CPUs. Multiply-Accumulate block, or MAC, is implemented as a DSP slice and is primarily utilized as a building block for complicated DSP applications. They are frequently used for filtering and compression of video or audio data.

2.11.2 FPGA Flow

Figure 29 shows the steps of the design flow.

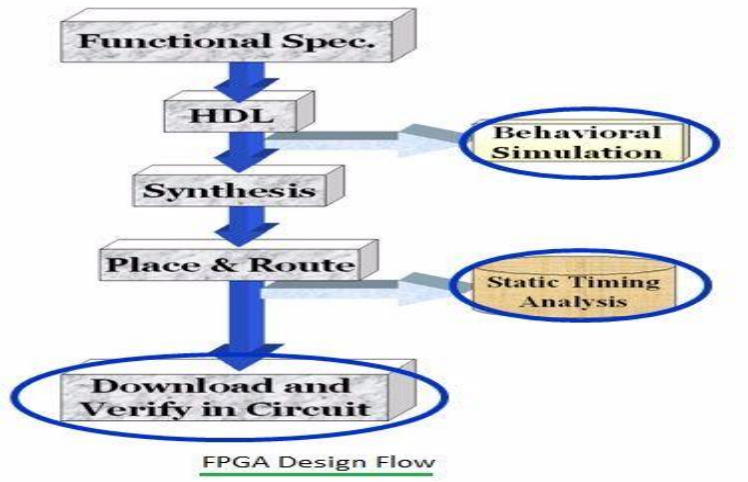

Figure 29: FPGA Flow

1-Functional Specifications: in this step, all specifications for the application are determined along with good understanding of function of this application**.**

2-HDL: the HDL code that describes that function is written, and then Behavioral Simulation is done to make sure that the HDL describes the function needed correctly.

3-Synthesis: HDL is converted into logic gates and other cells present in the FPGA itself, Static timing analysis is done to approximately calculate the maximum clock delay of the application and calculate the maximum clock speed achieved for the application.

4-Place & Route: The logic blocks and cells in the FPGA are connected together, and Static Timing Analysis is done again to calculate the exact delay model of the application.

5-Download & Verify in circuit: The HDL code is burned on the FPGA.

2.12 ASIC

2.12.1 Overview

ASIC[20] is an integrated circuit(IC or chip) customized for a particular use or application (not for general purpose like CPU).ASIC is any integral circuit or any piece of hardware or microchip.(CPU is also an ASIC but it's not meant for any particular task rather its universal). IC can be meant for image processing, audio processing, video processing, mobile application, IOT application, (CPU, RAM chip, ROM, EEPROM, FLASH memory in your mobile or pc).

ASIC is combination of analog function (clock, amplification, noise suppressing), digital function (adder, mux, registers, CPU), programmable logic, different type of memory, power management in a single chip. So here chip is dedicated for a particular application.

Initial investment for ASIC development is lot but it costs less for high volume. It takes several months to design, verify and produce and require a very methodology and requires very expensive tools for development. So it's targeted for high volume to reduce the cost (mostly for consumer electronics).

ASIC are 3 types:

1- *SEMI CUSTOM:* it uses predesigned logic cell (AND, OR, MUX, FLOPS, etc.) known as standard cell. The ASIC designer uses all these standard cells stored in the form of a library and does placement of these cells and interconnection. These cells can be placed anywhere on silicon as per the requirement. All these cells are predesigned, pretested and pre characterized so fully optimized. This semi-custom design reduces design time and risks. Here we can also change the transistor size in a standard cell to optimize it for speed and performance (each standard cell is designed in ASIC library provided by ASIC Company using full custom).

2- *FULL CUSTOM:* Here the designers design some or all the logic cell, circuit or layout, interconnection of its own for a specific application. It is approached if there are no suitable existing cell libraries available. It's used when ASIC technology is new or so specialized that the circuit must be custom designed (optimized analog circuit, memory cell, processor). It's costly, time consuming but provides max performance, minimized area and highest flexibility as designed with own efforts.

3- *PROGRAMMABLE ASIC :* FPGA,PLD

2.12.2 ASIC Flow

Figure 30 shows the steps of the design flow.

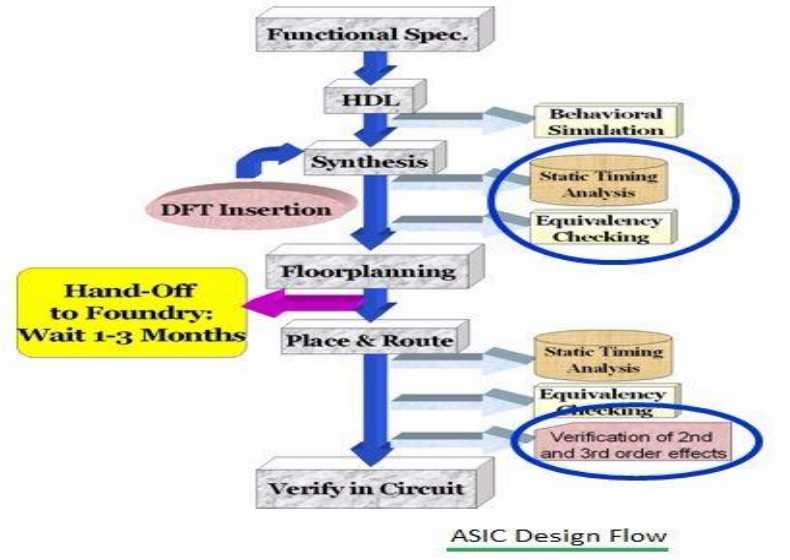

Figure 30:Asic Flow

1-Functional Specifications: in this step, all specifications for the application are determined along with good understanding of function of this application**.**

2-HDL: the HDL code that describes that function is written, and then Behavioral Simulation is done to make sure that the HDL describes the function needed correctly.

3-Synthesis: reads in the RTL code (.v , .vhd or .sv files) with the standard cells physical libraries which contain delay information (.lib files), physical dimensions and metal layer information within the cell (.lef files) to convert the behavioral RTL code into real physical standard cell gates (gate level netlist).

4-Floorplanning: formalizes and refines the floorplan during the architecture planning step. In this step while keeping in mind the area requirements, we divide the whole die area into physical partitions, and mold their shapes. Pins and ports are assigned to a location that can then be refined depending on the results of PAR.

5-Placement: In this step, all standard cells are placed in legal locations on site rows, with minimum wire length, while ensuring optimal placement which helps timing convergence to be faster.

6-Clock Tree Synthesis: In all the previous steps we assumed clock network is ideal. During clock tree synthesis, clocks are propagated and then using clock buffers the clock tree will be synthesized. The aim of this step is to get optimize the clock latency and minimizing clock skew. Many algorithms used to optimize the clock latency - H Tree, Steiner Tree, Clock Tree Mesh, Multi-source Clock Tree Synthesis and traditional Single Point Clock Tree Synthesis.

7-Routing: In this step we route the signal nets as all instances placed and clocks are now routed. Modern process supports ten to twelve metal layer stacks, but M0-M1 only are used for standard cell routing. For detail routing we usually use glorified maze router algorithm and to ensure faster run time we add some constraints. The metal resources are

the legal locations for metal routes. The main goal of detail routing is to minimize detours as they can cause issues on timing, and to minimize Design Rule Check (DRC) violations, i.e opens, shorts, etc. This step does from ten to twelve multiple search and repair loops to minimize the overall DRC count.

8-Physical and Timing Verification: Physical verification ensures that the layout is correct. Physical Verification checks have been increased recently and now includes- DRC (Design Rule Checks), LVS (Layout versus Schematic), , Electro-static discharge violations (ESD), Electromigration, Antenna violations, Pattern Match (PM) violations, Shorts, Opens, Floating nets etc. Tracking these violations is important in parallel with the Place and Route flow. Timing Verification ensures that the chip works at the required frequency by verifying setup time and hold time are met in all timing paths in the design.

2.14 Summary

In this chapter a background about convolution neural networks, different architectures, detailed discussion about Squeeze-Next, FPGA, ASIC and useful terms such as parallelism and fixed-point representation were presented.

Chapter 3: Design Architecture

In this chapter, we present two different architectures for implementing the CNN (Squeeze-Next) on" Virtex7 FPGA" to understand what led to our design.

3.1 First Architecture Approach

The first one "Time-shared architecture" is designing a very fast 1 block using available FPGA's resources and reusing the block in different layers. But the problem with this architecture is that it won't be fast enough. So, we have to think about another architecture.

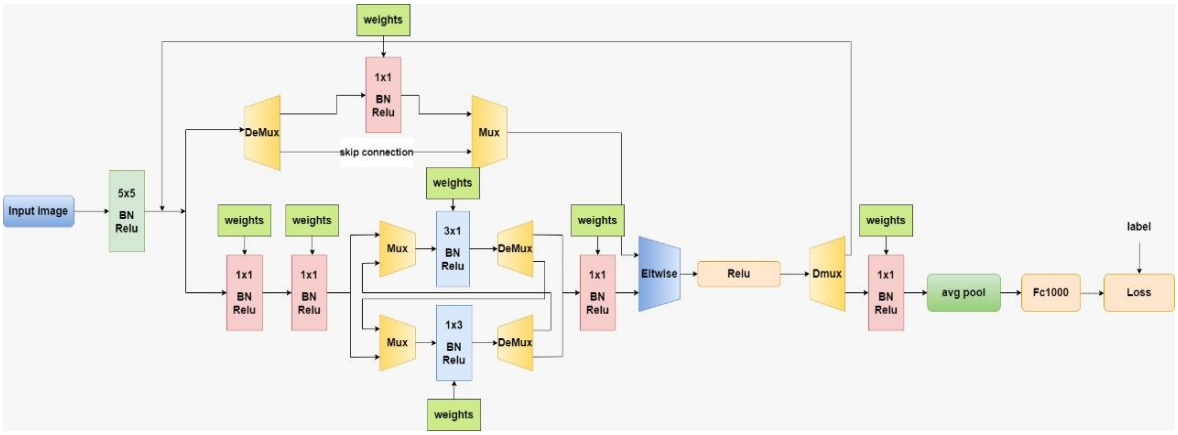

Figure 31: Time-shared architecture

3.2 Second Approach Architecture

The second one "Pipelined architecture" targets high-speed implementation of Squeeze-Next by pipelining all the layers together and deploying the whole architecture.

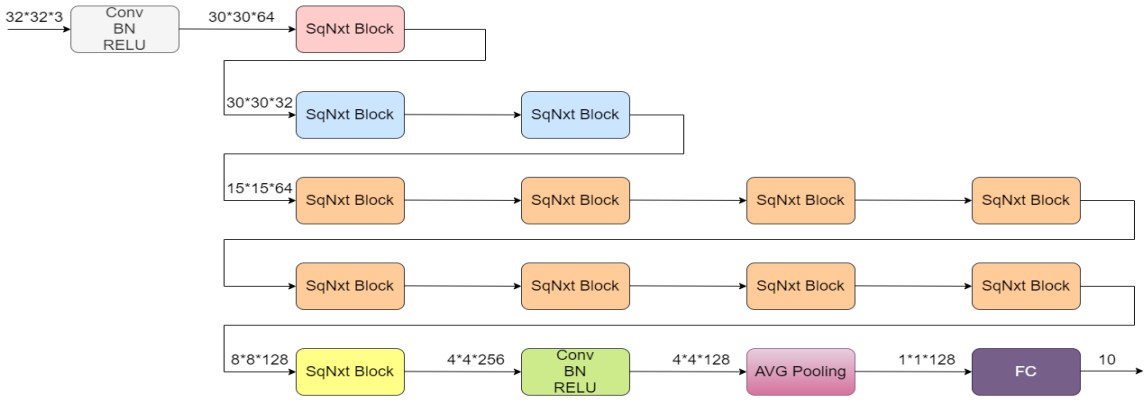

Figure 32: Pipelined architecture

The number of parameters and MACs (Multiply and Accumulate) in every layer is needed to make a good implementation of this architecture. Breakdown of the 1.0-SqNxt-14 architecture. Here, Wi, Hi, and Ci are the input width, height, and channels, Kw and Kh are the convolution filter sizes, Wo, Ho, and Co are the output width, height, and channels, S is the stride, Pw and Ph is padding, Repeat refers to how many times a particular module is repeated, Par is the number of parameters for each layer, MAC is the number of multiply and add instructions.

Name	Type	W_i	H_i	C_i	K_w	K_h	W_o	H_o	C_o	S	P_w	P_h	Repeat	Par	mac
conv1	Conv	32	32	3	3	3	30	30	64	1	0	0	1	1792	1555200
conv2	Conv	30	30	64	1	1	30	30	32	1	0	0	1	2080	1843200
conv3	Conv	30	30	64	1	1	30	30	16	1	0	0	1	1040	921600
conv4	Conv	30	30	16	1	1	30	30	8	1	0	0	1	136	115200
conv5	Conv	30	30	8	1	3	30	30	16	1	0	1	1	400	345600
conv6	Conv	30	30	16	3	1	30	30	16	1	1	0	1	784	691200
conv7	Conv	30	30	16	1	1	30	30	32	1	0	0	1	544	460800
Conv8	Conv	30	30	32	1	1	15	15	64	2	0	0	1	2112	460800
Conv9	Conv	30	30	32	1	1	15	15	32	2	0	0	1	1056	230400
conv10	Conv	15	15	32	1	1	15	15	16	1	0	0	1	528	115200
conv11	Conv	15	15	16	1	3	15	15	32	1	0	1	1	1568	345600
conv12	Conv	15	15	32	3	1	15	15	32	1	1	0	1	3104	691200
conv13	Conv	15	15	32	1	1	15	15	64	1	0	0	1	2112	460800
conv14	Conv	15	15	64	1	1	15	15	32	1	0	0	1	2080	460800
Conv15	Conv	15	15	32	1	1	15	15	16	1	0	0	1	528	115200
Conv16	Conv	15	15	16	3	1	15	15	32	1	1	0	1	1568	345600
Conv17	Conv	15	15	32	1	3	15	15	32	1	0	1	1	3104	691200
Conv18	Conv	15	15	32	1	1	15	15	64	1	0	0	1	2112	460800
Conv19	Conv	15	15	64	1	1	8	8	128	2	0	0	1	8320	524288
Conv20	Conv	15	15	64	1	1	8	8	64	2	0	0	1	4160	262144
Conv21	Conv	8	8	64	1	1	8	8	32	1	0	0	1	2080	131072
Conv22	Conv	8	8	32	1	3	8	8	64	1	0	1	1	6208	393216
Conv23	Conv	8	8	64	3	1	8	8	64	1	1	0	1	12352	786432
Conv24	Conv	8	8	64	1	1	8	8	128	1	0	0	1	8320	524288
Conv25	Conv	8	8	128	1	1	8	8	64	1	0	0	7	57792	3670016
Conv26	Conv	8	8	64	1	1	8	8	32	1	0	0	7	14560	917504
Conv27	Conv	8	8	32	3	1	8	8	64	1	1	0	7	43456	2752512
Conv28	Conv	8	8	64	1	3	8	8	64	1	0	1	7	86464	5505024
Conv29	Conv	8	8	64	1	1	8	8	128	1	0	0	7	58240	3670016
Conv60	Conv	8	8	128	1	1	4	4	256	2	0	0	1	33024	524288
Conv61	Conv	8	8	128	1	1	4	4	128	2	0	0	1	16512	262144
Conv62	Conv	4	4	128	1	1	4	4	64	1	0	0	1	8256	131072
Conv63	Conv	4	4	64	1	3	4	4	128	1	0	1	1	24704	393216
Conv64	Conv	4	4	128	3	1	4	4	128	1	1	0	1	49280	786432
Conv65	Conv	4	4	128	1	1	4	4	256	1	0	0	1	33024	524288
Conv66	Conv	4	4	256	1	1	4	4	128	1	0	0	1	32896	524288
Fc10	FC	1	1	128	1	1	1	1	10	1	0	0	1	1290	1280
									4645					532231	32593920

Table 2: The number of parameters and MACs in every layer

3.3 Methods Of Parallelism

The “Pipelined architecture” pipelines all the layers together but all layer stages are done respectively. So, we have to implement different hardware for each layer but the stages of one layer share the same hardware.

To increase the speed of one layer, we use inter kernel parallelism and inter output parallelism. For every layer fixed number of parallel channels and a fixed number of parallel filters and the multiplication of them is the number of used DSPs for this layer.

Virtex-7 Series DSPs are equal to 3600. The used DSPs in this way of parallelism are 3504. For each layer, we are careful that the number of $\frac{MACs}{DSPs}$ will be similar to other layers because it represents the number of cycles in each layer which we need to be similar in our Pipelined architecture.

Layer	*Parallel Ch*	*Parallel Filter*	*Used DSPs*	mac	Mac/DSPs
Layer 1	3	64	192	1555200	8100
Block1 and skip connection Layer 2	16	16	256	2764800	10800
Block2 Layer 2	4	4	16	115200	7200
Block3 Layer 2	4	8	32	345600	10800
Block4 Layer 2	8	8	64	691200	10800
Block5 Layer 2	8	8	64	460800	7200
Layer 3	16	16	256	2304000	9000
Layer 4	16	16	256	2073600	8100
Layer 5	16	16	256	2621440	10240
Layer 6	16	16	256	2359296	9216
Layer 7	16	16	256	2359296	9216
Layer 8	16	16	256	2359296	9216
Layer 9	16	16	256	2359296	9216
Layer 10	16	16	256	2359296	9216
Layer 11	16	16	256	2359296	9216
Layer 12	16	16	256	2359296	9216
Layer 13	16	16	256	2621440	10240
Layer 14 and FC10	8	8	64	525568	8212
			3504		

Table 3: The calculated DSPs units

3.3.1 16-Channel Parallelism

We need to implement 16 channels parallelism with 16 filters parallelism at 12 times.

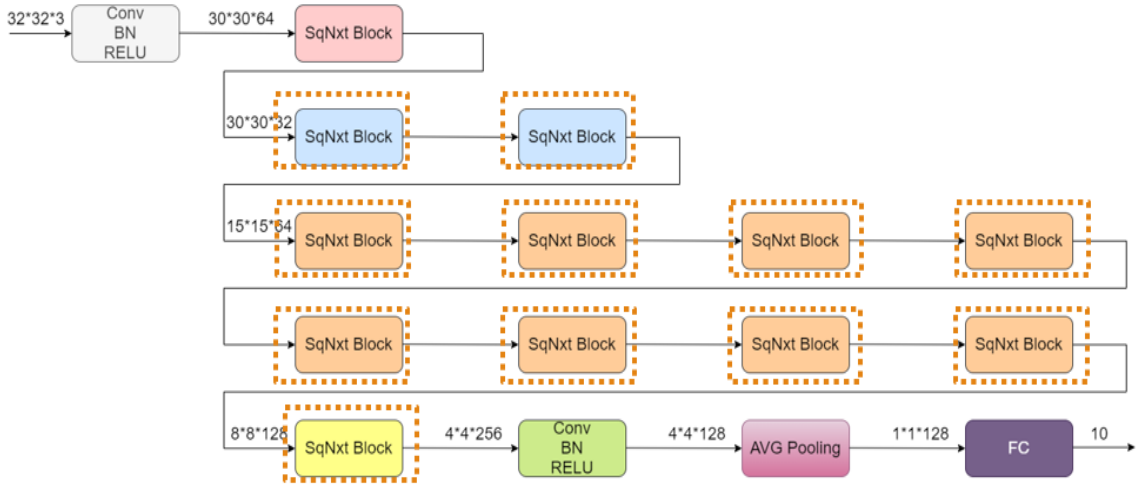

Figure 33: 16-channel parallelism

We need to implement pipelined 3-input adder for each block using 25 registers and 8 adders.

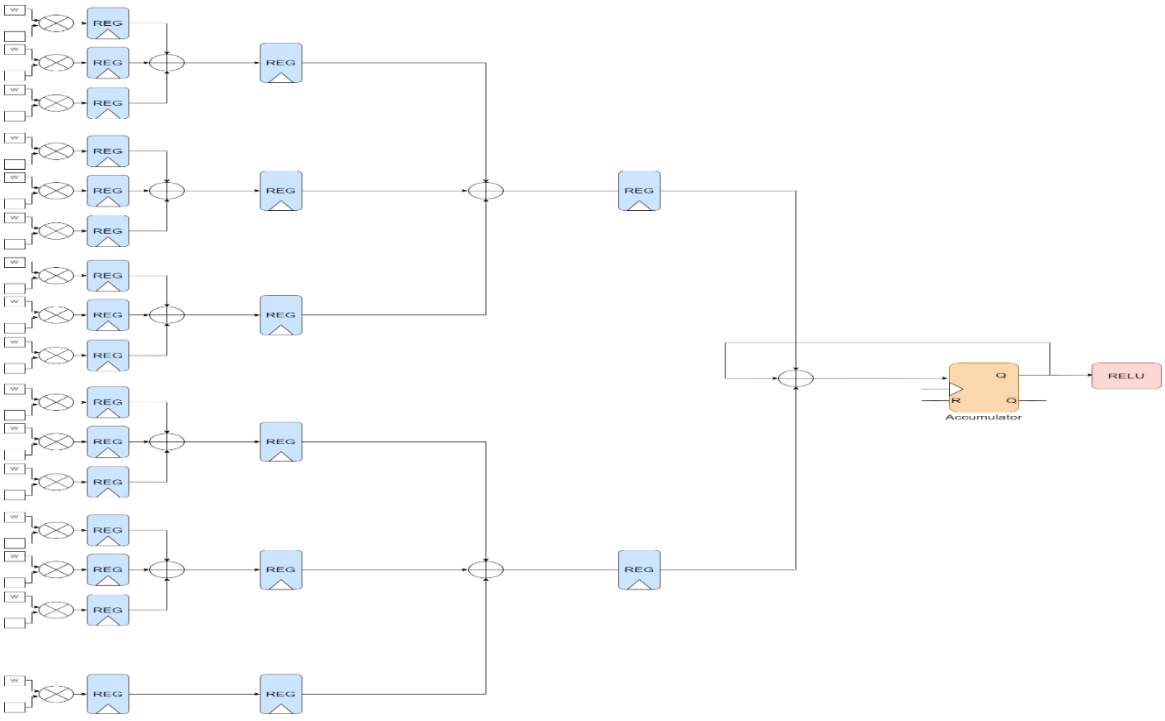

Figure 34:16-channel pipelined adder

3.3.2 8-Channel Parallelism

We need to implement 8 channels parallelization with 8 filters parallelization at 3 times.

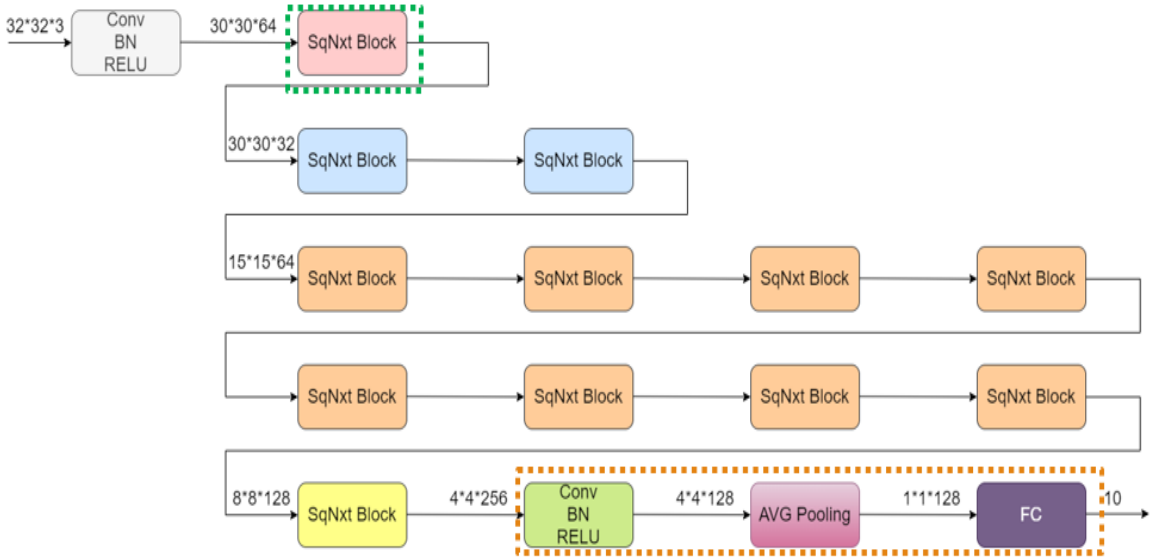

Figure 35: 8-channel parallelism

We need to implement pipelined 3-input adder for each block using 13 registers and 5 adders.

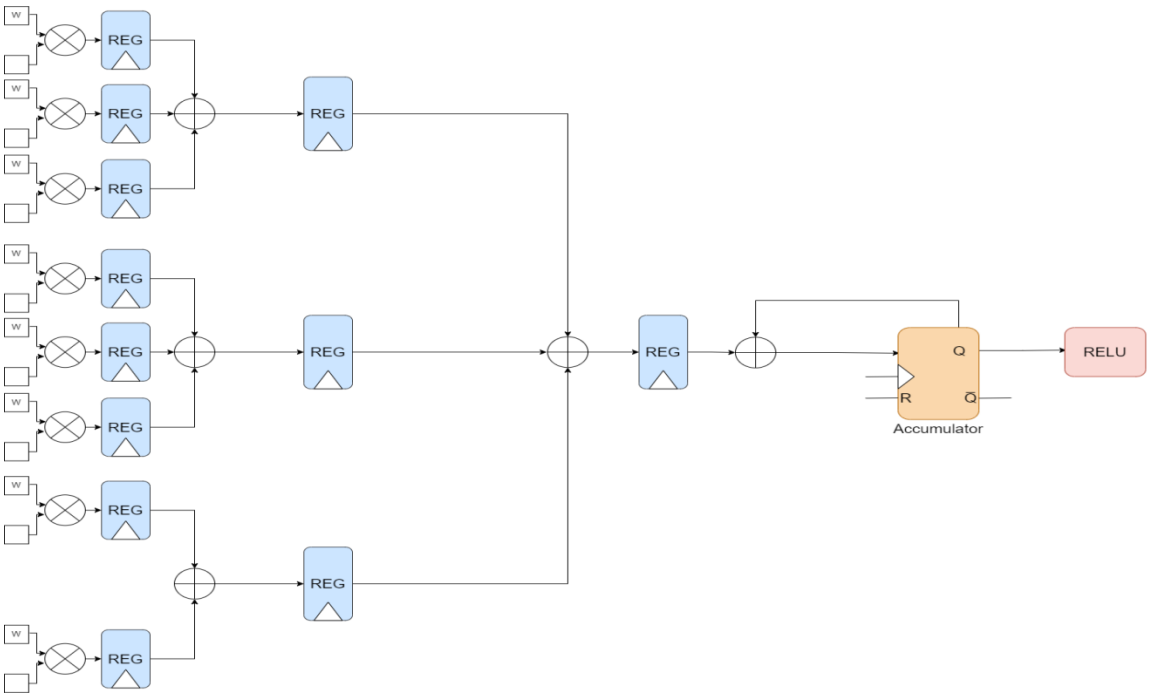

Figure 36: 8-channel pipelined adder

3.3.3 4-Channel Parallelism

We need to implement 4 channels parallelization with 4 filters parallelization at 1 time. And 4 channels parallelization with 8 filters parallelization at 1 time.

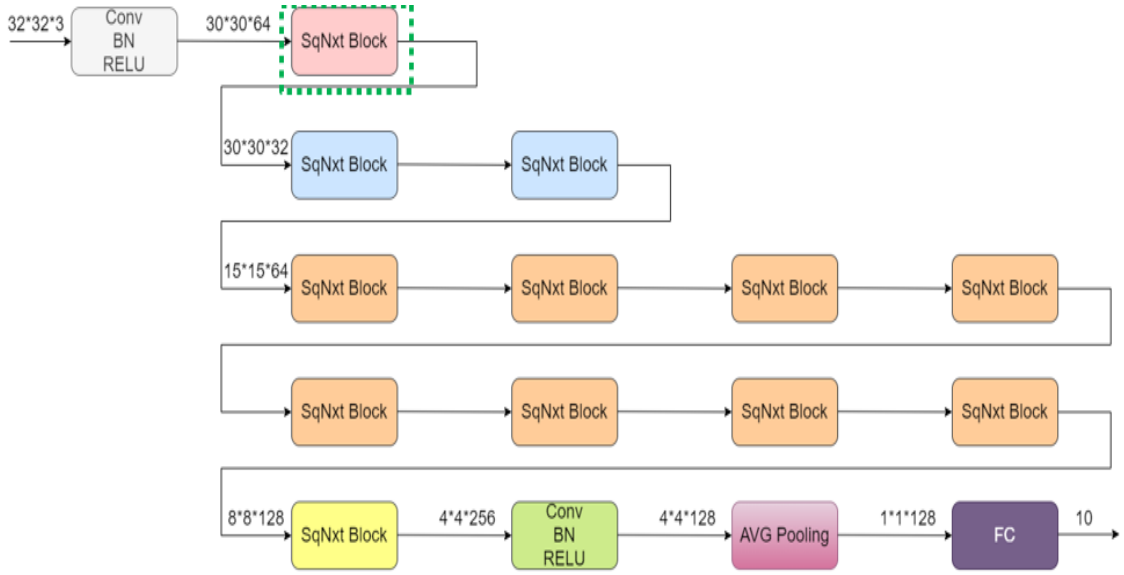

Figure 37:4-channel parallelism

We need to implement pipelined 3-input adder for each block using 6 registers and 2 adders.

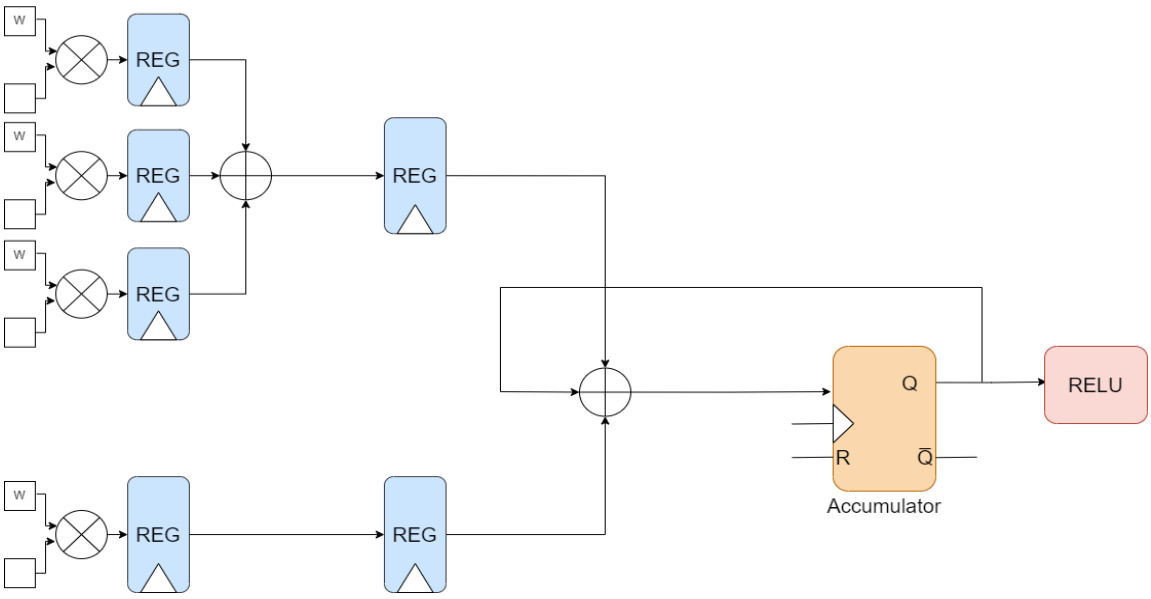

Figure 38: 4-channel pipelined adder

3.3.4 3-Channel Parallelism

We need to implement 3 channels parallelization with 64 filters parallelization at 1 time.

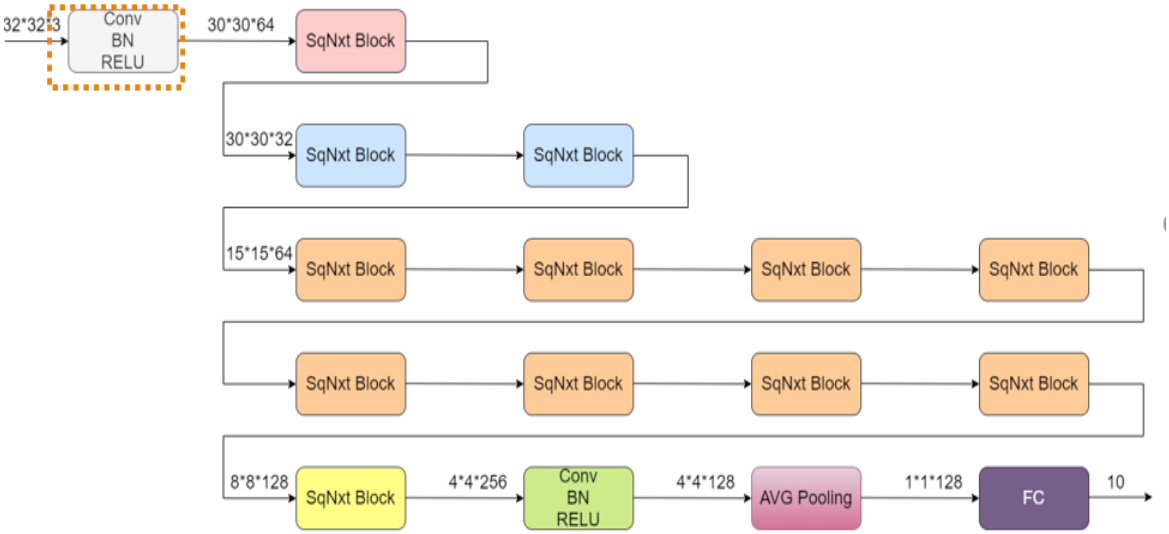

Figure 39:3-channel parallelism

We need to implement pipelined 2-input adder for each block using 5 registers and 2 adders.

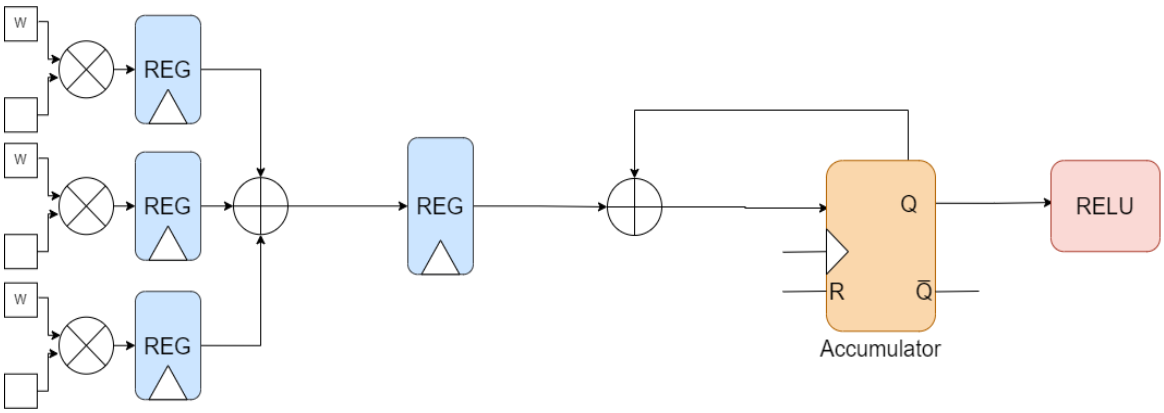

Figure 40: 3-channel pipelined adder

3.4 Batch Normalization

We have one mini-batch for each filter

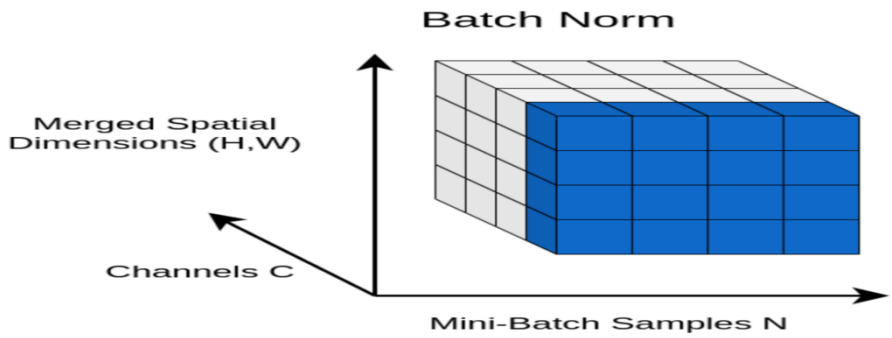

Figure 41: Batch normalization method

In every mini-batch, we have Two learnable parameters called beta (β) and gamma (γ) and two non-learnable parameters (Mean Moving Average (μ_{mov}) and Variance Moving Average ($\sigma^2{}_{mov}$)) are saved as part of the 'state' of the Batch Norm layer.

The equation of one pixel of mini-batch is $Y = \frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}} \times \gamma + \beta$. where ε is epsilon which equals to 10^{-5} in our training, x is the output of convolution, y is the output of one pixel of mini-batch, μ is the mean of all pixels of mini-batch and σ^2 is the variance of all pixels of mini-batch.

During Training, Batch Norm starts by calculating the mean and variance for a mini-batch. However, during Inference, we have a single sample, not a mini-batch. Here is where the two Moving Average parameters come in.

The ones that we calculated during training and saved with the model. We use those saved mean and variance values for the Batch Norm during Inference.

The equation of the Mean Moving Average of mini-batch is: $\mu_{mov} = \alpha \times \mu_{mov} + (1-\alpha) \times \mu$ where α is momentum which equals to 0.1 in our training

The equation of Variance Moving Average of mini-batch is: $\sigma^2{}_{mov} = \alpha \times \sigma^2{}_{mov} + (1-\alpha) \times \sigma^2$ where α is momentum which equals to 0.1 in our training.

3.4.1 Batch Normalization First Approach

The first approach is describing the equation of one pixel of mini-batch as it is. The equation is Y $=\frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}} \times \gamma + \beta$

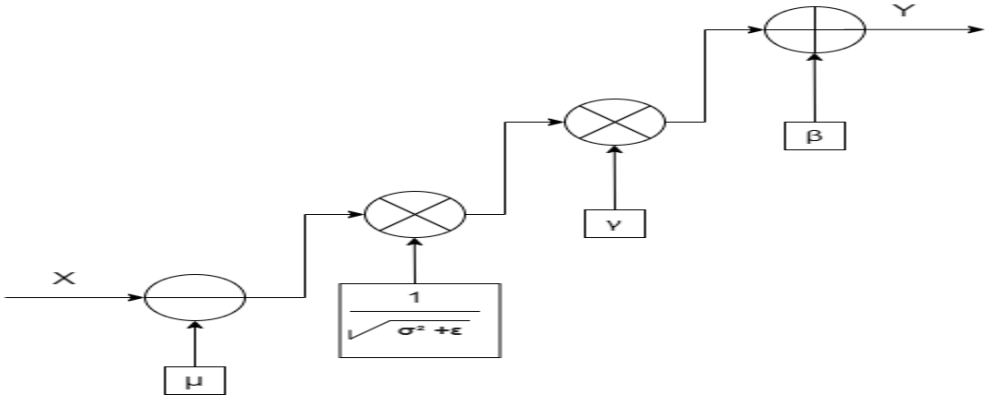

Figure 42: First approach of batch normalization

3.4.2 Batch Normalization Second Approach

The second approach is describing the equation of one pixel of mini-batch but by some simplifications to use 1 multiplier and 1 adder only. So, we can use 1 DSP for every block.

The equation is Y=$\frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}} \times \gamma + \beta = \frac{\gamma}{\sqrt{\sigma^2+\varepsilon}} \times x + \beta - \frac{\mu \times \gamma}{\sqrt{\sigma^2+\varepsilon}} = ax + b$

Where: a=$\frac{\gamma}{\sqrt{\sigma^2+\varepsilon}}$ and b= $\beta - \frac{\mu \times \gamma}{\sqrt{\sigma^2+\varepsilon}}$

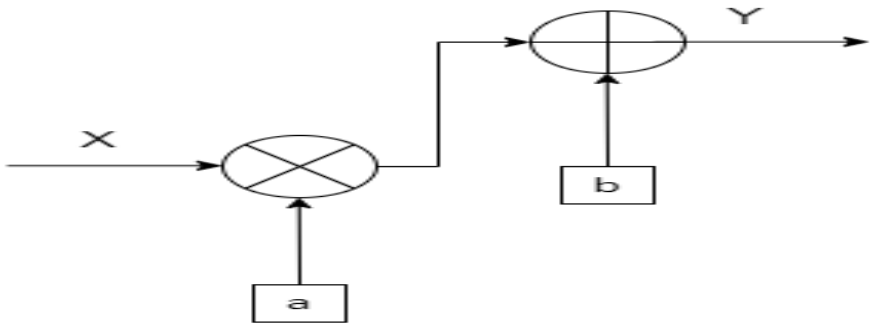

Figure 43: Second approach of batch normalization

3.4.3 Batch Normalization Third Approach

We notice that we can make fusing batch normalization. The equation of convolution is Y=$\sum x * w_{conv}$ and the equation of batch is $\bar{y}=ax+b$. [37]

So, we can say $\bar{y}$=b + $\sum x * w_{conv}$ * a. we make it using only convolution.

3.5 Layer Storage

We have 5 stages in every layer. If the input data is stored in “memory 1”, the output of the first stage will be stored in “memory 3”. At the second stage, the input will be in “memory 3” and the output will be in “memory 1” and so on. We still have the problem that we need to make element-wise addition between the input of the first stage and the output of the fifth stage. And the question is where is the data input of the first stage? We don’t have storage for it. So, we will use “memory 2”.

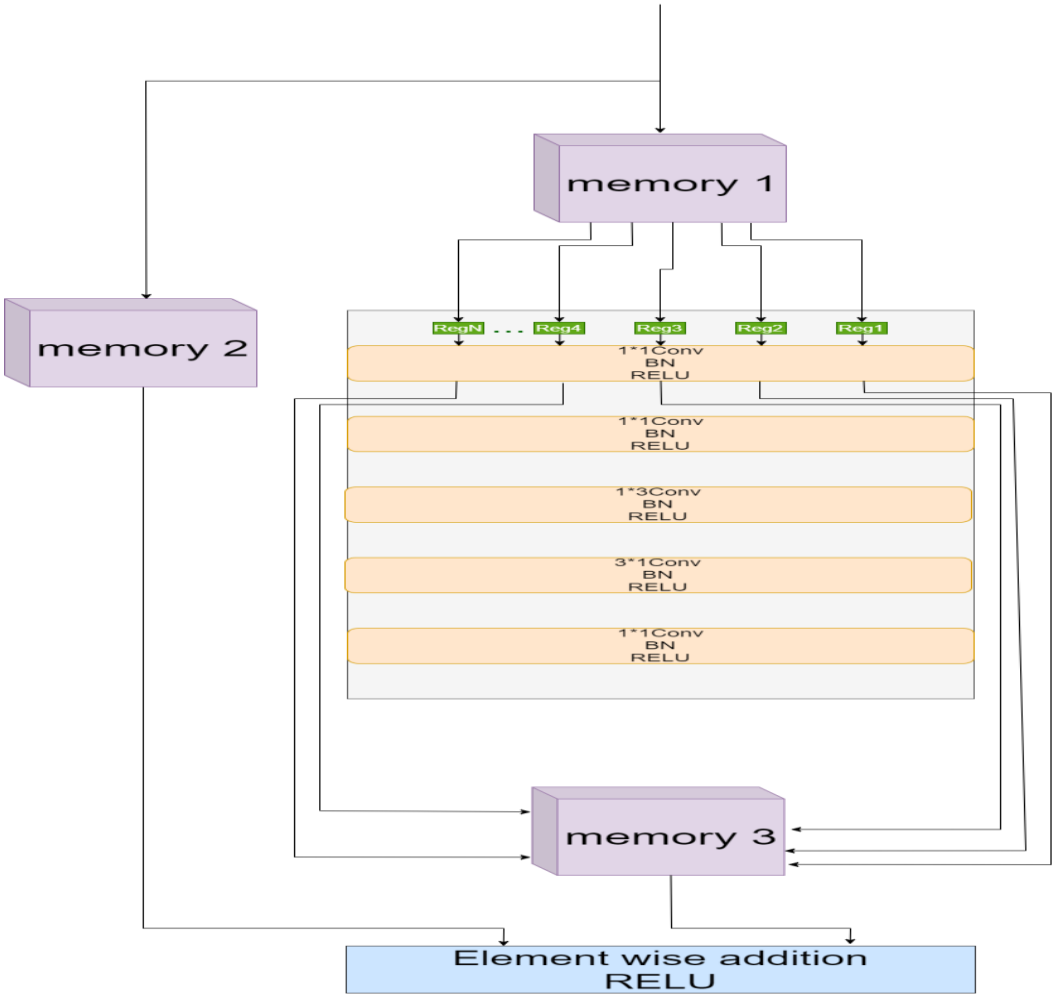

Figure 44: Memories of one layer

3.6 Memories

3.6.1 FPGAs on-chip Memory:

The FPGA fabric includes embedded memory elements that can be used as random-access memory (RAM), read-only memory (ROM), or shift registers. These elements are block RAMs (BRAMs), LUTs (distributed RAM), and Registers.

3.6.1.1 BRAM

FPGAs normally have an on-chip BRAM matrix, which could be configured as FIFO, RAM, or ROM. The targeted device (Virtex-7 x690t) contains about 2940 BRAM 18Kb instances, each can be configured to 4Kb x 4, 8Kb x 2, or 16Kb x 1. So, it can hold 1024 parameters. BRAMs can have dual ports for the same instances, allowing the performance of half the latency.

3.6.1.2 LUTS

The LUT is a small memory in which the contents of a truth table are written during device configuration. Due to the flexibility of the LUT structure in Xilinx FPGAs, these blocks are commonly referred to as distributed memories. Each LUT can be configured as 6 input 1-bit ROM (64x1). This is the fastest kind of memory available on the FPGA, because it can be instantiated in any part of the fabric that improves the performance of the implemented circuit, which makes them a good approach to SqueezNext implementation.

3.6.1.3 REGISTER:

Register memory is the smallest and fastest memory in a computer. It is not a part of the main memory and is located in the CPU in the form of registers, which are the smallest data holding elements.

3.6.2 WEIGHT MEMORY:

Not only each layer has its weight memory, but each conv has its weight memory. As an example (conv2) in layer2 which has kernel size1x1x64x32. It can be configured as a block ROM with a size 32kb. This memory only can read one parameter in each cycle, so this big memory can’t be good for our parallelization technique so, we must

divide this memory into small blocks for each parallelization technique. Each small block with a size of [8x16] 8 numbers 16bit for each. Every 16 small blocks are dedicated for two filters and one adder tree.

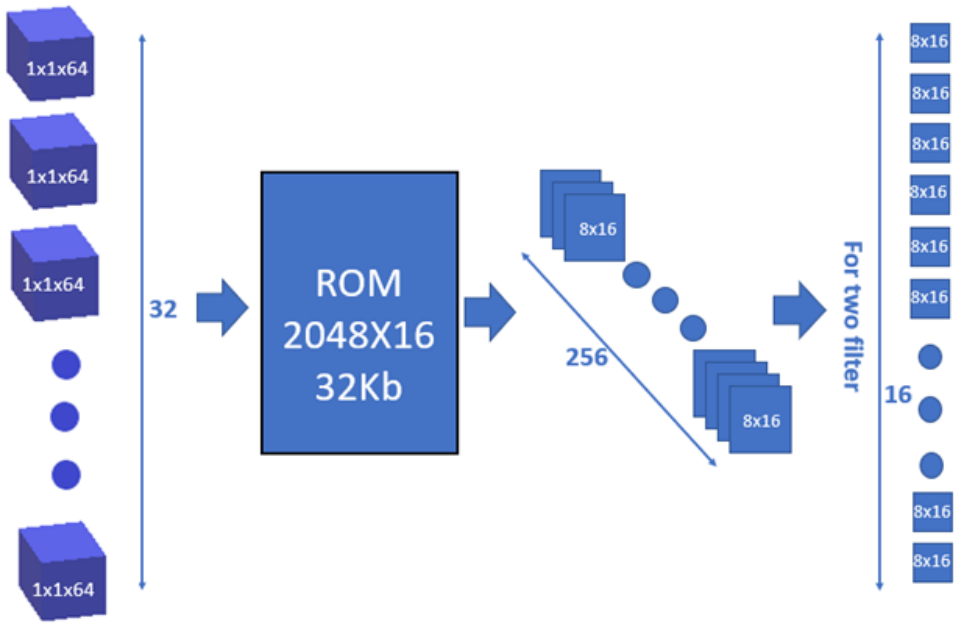

Figure 45: Division of weight memory into small block

For memory optimization, we used for weight memory Distributed RAM which the best choice for memory optimization where the size of small block is about 128 b which will be not efficient if we use BRAM.

3.6.3 BIAS MEMORY:

For each conv, there are a small number of biases which is the number of filters. Therefore, we use the register as a memory for bias. Each bias is eighteen bits.

3.6.4 DATA MEMORY:

We use BRAM for image data memory. Because each BRAM can only read and write maximum two addresses every cycle (dual port) So, we need to use the number of BRAM equal to the number of the input of the adder tree. If we take an example of data [15x15x32] which converted to RAM of the size of 112.5 kb with 7200 number, but this size can't be divided into several BRAM equal to the input of the adder tree. Therefore, we use RAM of size 128kb which is divided into 8 BRAM each having 1024 free spaces.

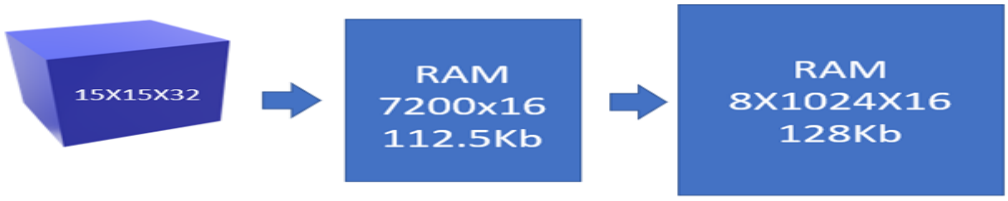

Figure 46: Data memory using BRAM

There is an overhead that doesn't affect the design because we use the spaces for padding. And we fill this places with don't care that so, it doesn't impact on area or power.

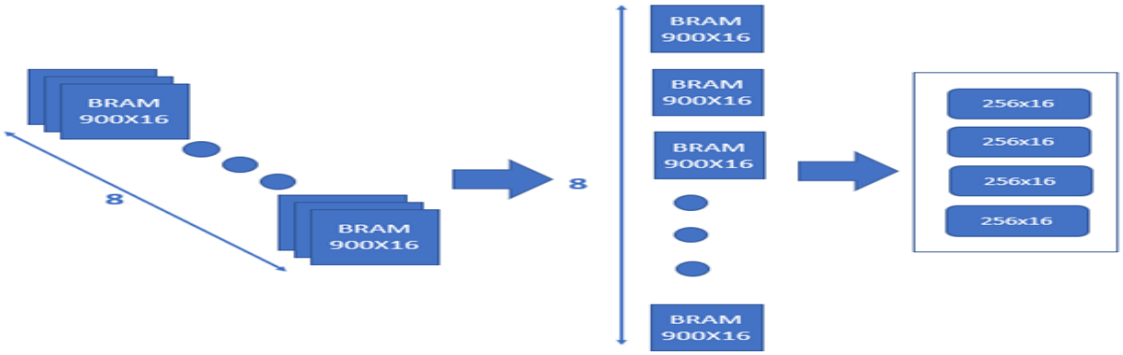

Figure 47: Division memory into BRAM with the same size

As shown in figure 47, we divide the RAM memory into some of BRAM with size of 16 Kb filled with 900 number .In this example, we use 16 adder trees thus, we use dual-port memory for each BRAM. Each BRAM is divided into 4 blocks of size 4kb that help us when writing and reading with simple address.

3.7 Intermediate Storage

We need batch-separated memories in each layer in Pipelined architecture.

3.8 Summary:

This chapter provides a discussion about the main idea of our architecture, how to use parallelism, how to use pipelining, adder tree to implement convolution, batch normalization implementation and its optimization and main idea of our memory.

Chapter 4: Detailed Design

4.1 Transition Between Layers That Have 1 Convolution

Due to our architecture, we have 6 layers consists of 1 convolution: 1 layer is convolution from the origin and 5 was 1 layer but we split it into 5 layers because of its huge delay.

4.1.1 Transition Traditional Pipelined

To make traditional pipelining between these 6 layers: we need input memory and output memory for each layer. After end of each layer, we will pass its data from its output memory to the input memory of next layer. We have waste of delay, area and power in this method because the output memory has the same data in the input memory in next layer (waste of area) and this transition take time and spend power.

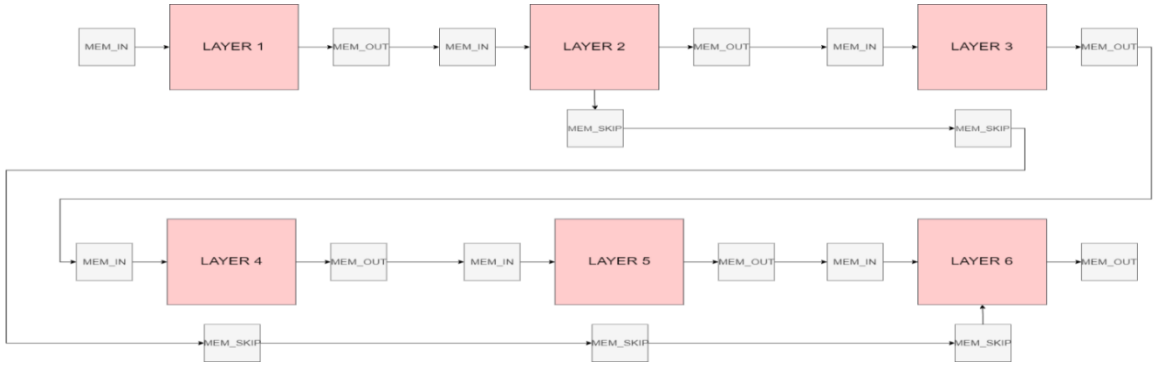

Figure 48: Traditional pipelining in first 6 layers

4.1.2 Transition New Method

In this new method, we will replace each 2 memories (output and input memory) with 1 memory which we called transition memory. The new method depends on transition in specific time.

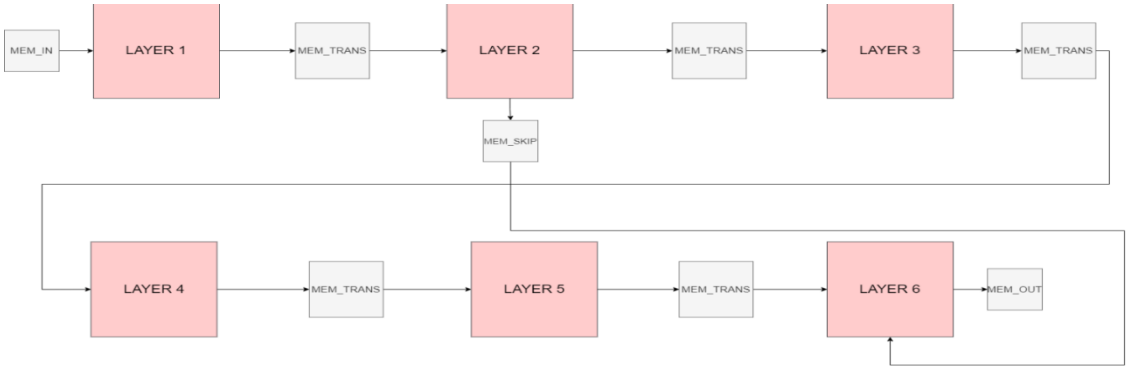

Figure 49: New transition method in first 6 layers

To specify this time, we will see if we pass data from slow layer to fast layer or from fast layer to slow layer. If we pass data from fast layer to slow layer, we will start this transition after first output data in fast layer and the fast layer will end before slow layer so we will freeze it until the end of slow layer. If we pass data from slow layer to fast layer, we will start this transition after we make sure there won't be any confliction in data so we will freeze the fast layer from the beginning. Using this method, we nearly decreased the number of used BRAMs to its half as shown in table 4:

TARGET OPTIMIZATIONS		Before optimizations	After optimizations
BRAMs	Numbers	841 BRAMs	471 BRAMs
	Utilizations	28%	16%

Table 4: BRAM Number Optimization

Now we need to design new memory which we called “Simultaneous Memory”.

4.2 Simultaneous Memory

It is a modified dual-port memory. By using 1st data input with (addr1, WR) and 2nd output data with (addr2, RE) and canceling out other input and output data. Therefore, we can read and write (simultaneously). While the layer(n) stores data by (addr1, WR) the layer(n+1) reads data by (addr2, RE).

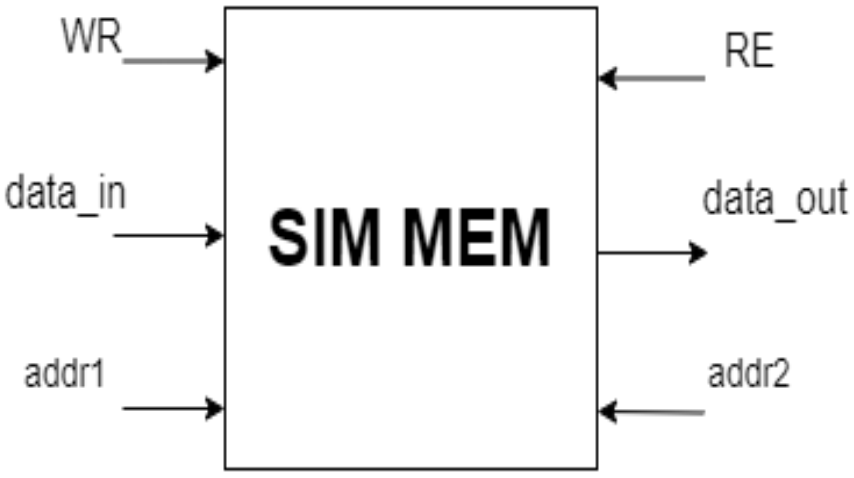

Figure 50: Simultaneous Memory

4.3 Optimization In Skip Connection Memories

We notice that in traditional pipelining we have 5 skip connection memories, but in new method we have only 1 skip connection memory. So how that happened? These 5 layers was 1 layer originally and these layers has skip connection convolution. So, because of pipelining we should have 5 skip connection memories. But our new method isn't like traditional pipelining so we don't need to 5 skip connection memories as all 5 layers will be in the same data in same time not in different data like pipelining.

4.4 Skip connection merging

We can't assume it as a slow layer then it will be freeze in this layer which we don't need. To assume this layer as a slow layer and save the time, we should merge the normal convolution with the skip connection convolution by out 1 output from normal convolution then 1 output from skip connection convolution as the images show.

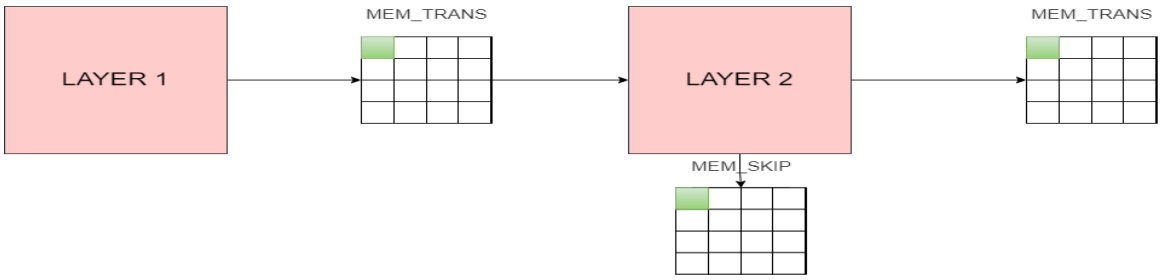

Figure 51: Skip connection merging (step 1)

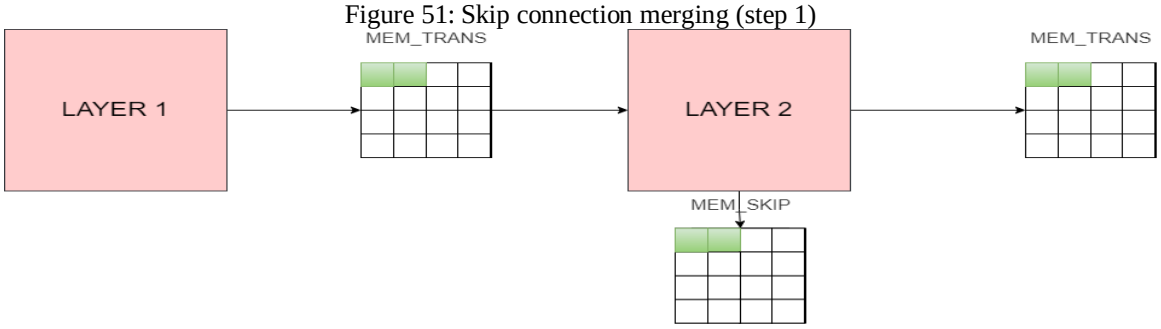

Figure 52: Skip connection merging (step 2)

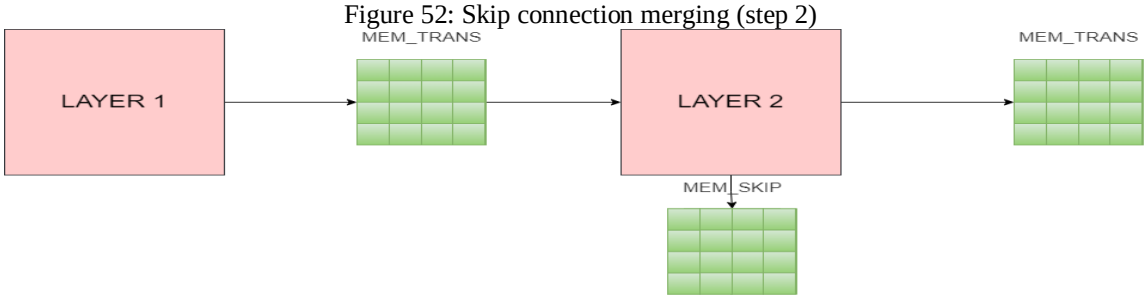

Figure 53: Skip connection merging (last step)

4.5 Transition Between Layers That Have 5 Convolutions

It is different case in layers that have 5 convolutions because they have 3 internal memories so we need to know which method is better for these layers

4.5.1 Transition traditional pipelined between layers that have 5 convolutions

In traditional pipelining, we will store output of fifth convolution in memory 3 in layer 1 then we will move it to memory 1 and memory 2 in layer 2 then we will start layer and so on

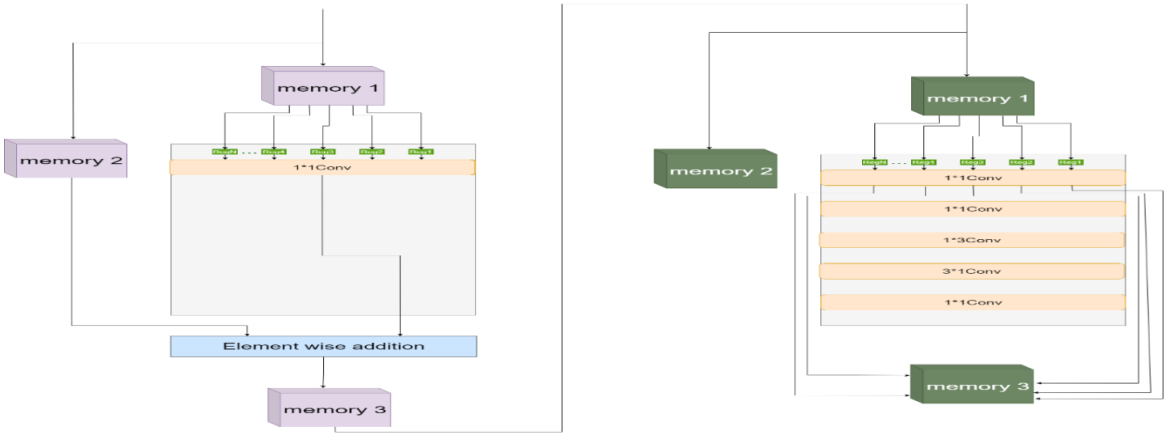

Figure 54: Traditional pipelining between big layers

4.5.2 Transition new method between layers that have 5 convolutions

In transition new method, we will store output of fifth convolution in memory 1 and memory 2 in layer 2 directly and will start layer 2 simultaneous as layers that have 1 convolution.

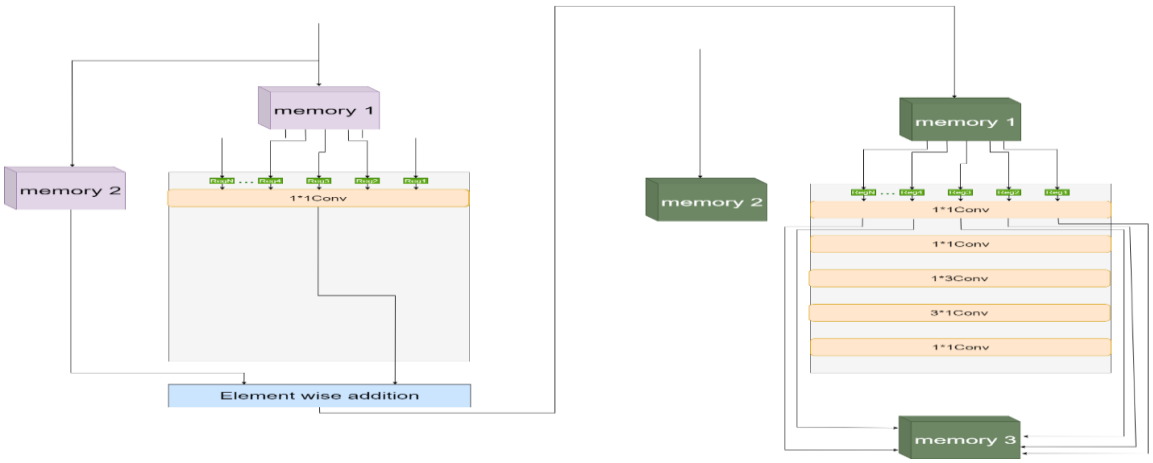

Figure 55: Transition new method between big layers

Now we can see that the new method is better here because it saves area and power. But we have special case in layers from 10 to 16 because layers from 10 to 16 is the same layer with different bias and weights so we need to use 1 controller for all of them and we need to use 8 dual port BRAM in each memory of them so we need traditional pipelined here because we haven't BRAM can take 4 addresses.

4.6 Stall controller

Stall controller is controller which we design to freeze the layer in the specific time we determined and run the layer in the specific time we determined. This schedule represents the start time and freeze time of each layer.

Layer Number	Start Time	Freeze Time
Layer1	At start and after 10805 clk cycles again	After 8104 clk cycles
Layer2	After 13 clk cycles from layer1 starting	Never freezing after starting
Layer3	After 3613 clk cycles from layer2 starting	At start and after 10805 clk cycles from layer2 starting
Layer4	After 8 clk cycles from layer3 starting	Never freezing after starting
Layer5	After 12 clk cycles from layer4 starting	Never freezing after starting
Layer6	After 3613 clk cycles from layer5 starting	At start and after 10805 clk cycles from layer5 starting
Layer7	After 4503 clk cycles from layer6 starting	At start and after 10805 clk cycles from layer6 starting
Layer8	After 7204 clk cycles from layer7 starting	At start and after 10805 clk cycles from layer7 starting
Layer9	After 9208 clk cycles from layer8 starting	At start and after 10805 clk cycles from layer8 starting
Layer10	After Ending of layer9	After 10805 clk cycles from its start
Layer11	After Ending of layer9 in case that layer 10 is started before	After 10805 clk cycles from its start
Layer12	After Ending of layer9 in case that layer 11 is started before	After 10805 clk cycles from its start

Layer13	After Ending of layer9 in case that layer 12 is started before	After 10805 clk cycles from its start
Layer14	After Ending of layer9 in case that layer 13 is started before	After 10805 clk cycles from its start
Layer15	After Ending of layer9 in case that layer 14 is started before	After 10805 clk cycles from its start
Layer16	After Ending of layer9 in case that layer 15 is started before	After 10805 clk cycles from its start
Layer17	After Ending of layer9 in case that layer 16 is started before	After 10805 clk cycles from its start
Layer18	After Ending of layer9 in case that layer 17 is started before	After 10805 clk cycles from its start

Table 5: Layers start and stop time

4.7 Images controller

Due to pipelining between layers and the STALL controller design, we can now Pipeline inference of multiple images and to make a prediction of the class of a specific image we can load a set of images on the FPGA not just one image, also pipelining in inferring the images improved the total throughput. So we can conclude that our design support inferring multiple images one by one into the model (Pipeline inference).

4.8 Layer Controller

4.8.1 Controller Responsibilities

Our controller consists of some up-counters and combinational logic to do some responsibilities as follow:

4.8.1.1 Determine the address of input data memory and weight memory

We use 4 counters: j (for number of kernal rows), k (for number of kernal column), x (for number of data rows), y (for number of data column). The address of data memory will be (x+j) for row and (y+k) for column. The address of weight memory will be j for row and k for column. The counter k will count to end then j then y then x and so on.

4.8.1.2 Determine the address of output data memory and weight memory

We need to take advantage of address of input data memory. We can input the old address in Register before its change by one clock.

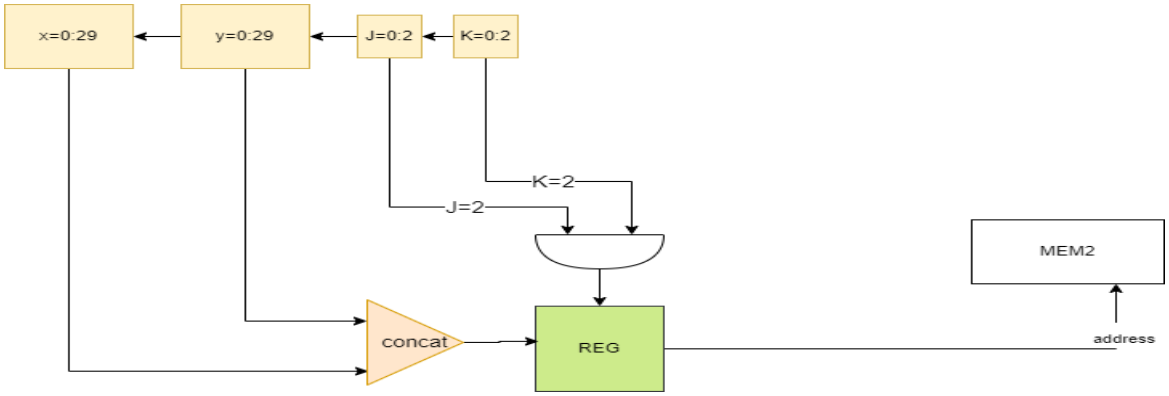

Figure 56: Determine the address of output memory

4.8.1.3 Determine when we should write data and restart the accumulator

Assume that last data should go to adder tree when j=max(j) and k=max(k). So, we should wait the number of stages in adder tree (as each stage ends in one clock cycles) after last data should go to adder tree to write. So, we use counter R to count the number of stages after j and k get their max values. Note that in this image max(j)=max(k)=2 and the number of stages in adder tree=4.

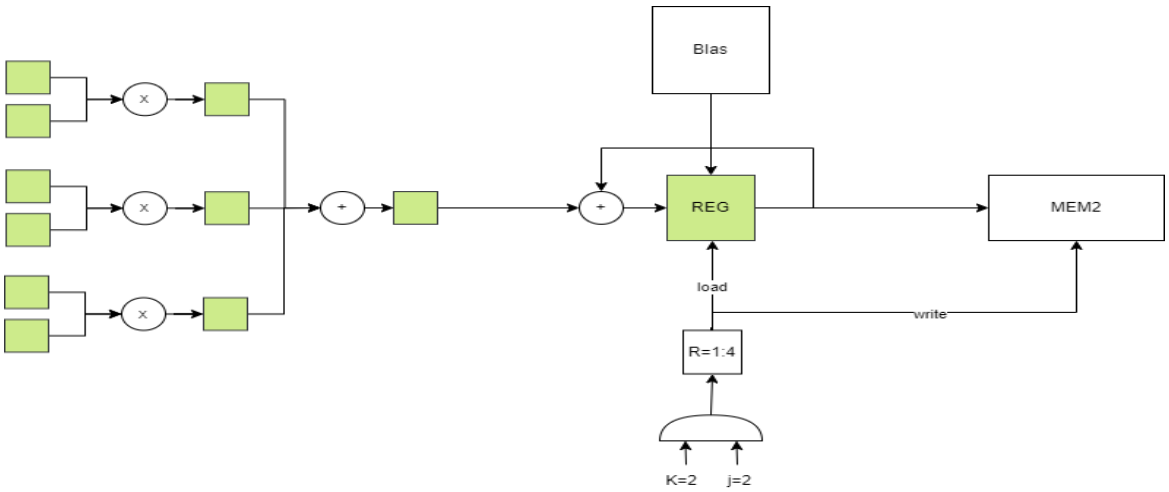

Figure 57: Determine when we should write data

4.8.2 Controller Problems

When all counters go to their final values, they will come back to zero. The problem is that we need to wait for the number of stages in the adder tree (as each stage ends in one clock cycles) to store final data in memory. So, we use temp counter.

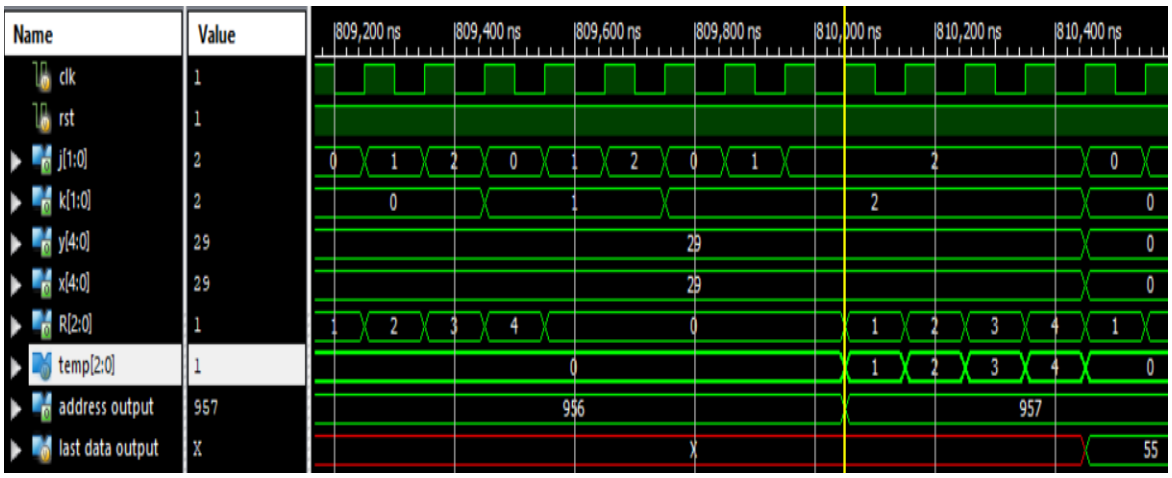

Figure 58: Problem of last output data

When we go to repeat the same layer but with new data, we still have false number in the adder tree. We need R counter to count new number of stages in the adder tree (as each stage ends in one clock cycle) in new repeated layer to restart the adder tree. So, if temp counter goes to its maximum, it will make counter R go to one not zero.

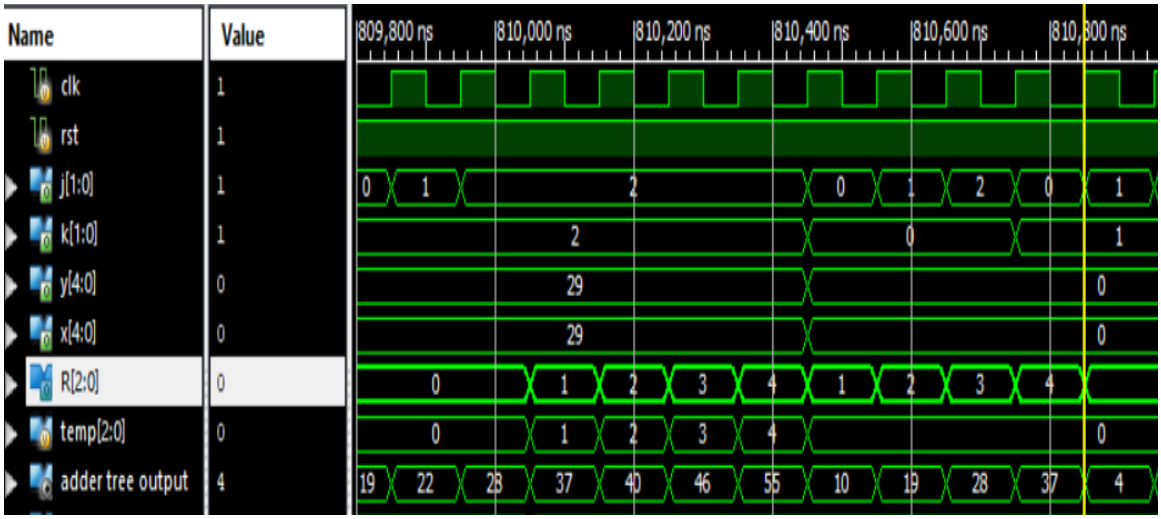

Figure 59: Problem of first output data

4.8 Average Pooling

Average pooling is the one of the important parts at the end of the layers after finishing all convolutions the output patch is 4x4 and before the output goes to Fully connected we need to reduce the size of the patch to be 1x1, so that we take the mean value from each patch which contains 16 value (4x4), so the input that enters the FC becomes 1x1x128.

Average pooling in our design is implemented at the last layer (Layer18) and this layer has a special filter for the Average pooling, this filter is consisted of one accumulator then shift block.

The accumulator block each cycle take one value from the 16 value of each patch then accumulate until reach 16 cycles then the output from the accumulator is shifted to the right by 4 bits so the digital value is divided by 16 then it stored inside register file to be ready for the next level.

As we use parallelism technique, we do not have one accumulator, instead we have 8 accumulator blocks to gives me 8 data each 16 cycle. Figure 60 shows the filter of the average pooling.

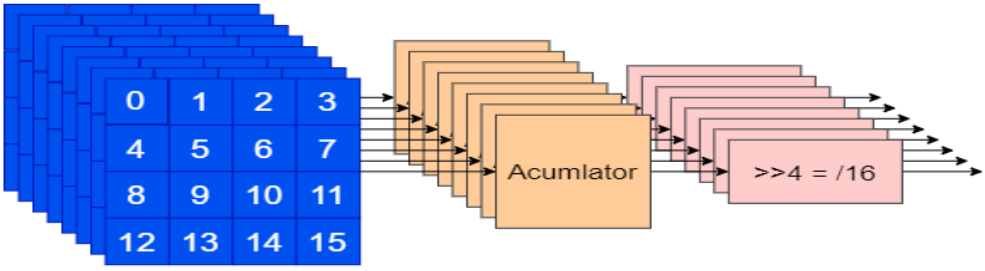

Figure 60: Average pooling

4.9 Fully connected layer (FC)

Fully connected layer the last layer that decide the type of class of input image, the fully connected layer has an input of 1x1x128 and it should reduce the 128 numbers to 10 number each number gives like a contribution for each class from 10 classes, the maximum number indicates the type of image the model predicts, the FC is just like the

convolution layer but it differs from it that it has no RELU at the end of the convolution, so the output can be negative.

The Fully connected layer implemented in the last layer (Layer 18) which use the same filter of the normal convolution filter this filter takes its input data from register file that stored its data from average pooling (the layer before FC) and takes the weight value from other rom and the accumulator of the filter is loaded with bias from FC bias rom, the output of the convolution block stored inside another register file (small memory), there are 10 values from register file enter comparator tree block to determines the largest value and output 4 bit signal which determine the index of the predicted type of the input image. Figure 61 shows the comparator tree.

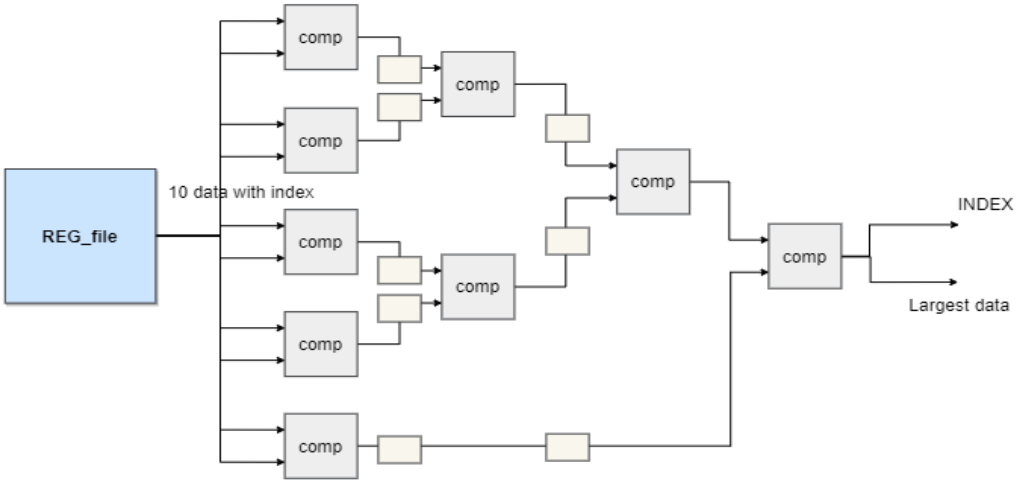

Figure 61: Fully connected layer

4.10. Summary

This chapter provides a discussion about our detailed design, integration idea, implementation and optimization in small and huge layers, skip connection merging idea, skip connection memory optimization, responsibilities of controller and its problems, average pooling and fully connected layer.

Chapter 5: Model Training And Parameters Generation

5.1 Introduction To Squeezenext Training

Squeeze has many different versions that vary in accuracy, number of parameters, size, and complexity.

To choose a suitable version for our needs and to confirm the results found in the art about the model, we used a python code used to implement the 23-layer version then modified it to create several other versions as shown in table 6.

The data set used was CIFAR10, which consists of 60,000 32x32 colored images, divided into 10 classes having 6000 images in each class. Training and testing of the data set were done on a 1660ti Graphics card and Intel core i7-9750h processor. We used 50,000 images for training and 10,000 images for testing.

After comparing results, we found that the squeezNext-14-1x version in Fig.62 is the most suitable for our needs as has about 200,000 fewer parameters than the 23-layer version, also its size is significantly less coming at 2.05 Mb Compared to the 23-layer version while taking a hit of only 0.28% in accuracy.

Moreover, the 6-layer versions were not chosen as they suffered from a big accuracy loss from the 23-layer version despite having excellent size and number of parameters.

Architecture	*accuracy*	*Time of 1 Epoch (sec)*	Parameter No.(100K)	MAC NO. (1M)	Memory size(MB)
SqueezeNext-23-1x-v5	91.68	66	7,87	54.50	3.00
SqueezeNext-14-1x	91.4	41	5,37	33.63	2.05
SqueezeNext-14-1x (Relu then BN)	89.01	41	5,37	33.63	2.05
SqueezeNext-14-1x (with max pooling)	86.98	35	5,37	10.07	2.05
SqueezeNext-6-1x	87.14	29	2,62	14.50	1.00
SqueezeNext-6-1x (using Elu)	85.88	26	2,62	14.50	1.00

Table 6: Comparison between SqueezeNext versions

To further our research other versions of the 14-layer model was tested where the RELU was executed before the batch normalization, but this led to less accuracy (despite finding

papers claiming otherwise). Also, the 14-layer version with max-pooling wasn't chosen due to the big decrease in accuracy.

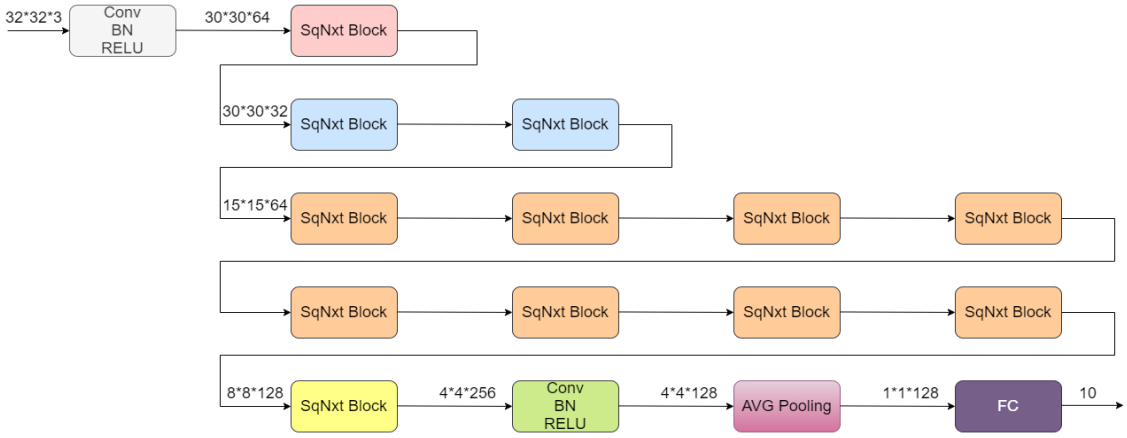

Figure 62: SqueezeNext 14-layer model

5.2 Fixed Point Representation:

A great deal of effort has been expended in the design of CNNs training algorithms under various applications. Training CNNs incorporates many complex calculations and experiments different sets of parameters/settings are very common. It is therefore very effective to do training using high-level programming languages, e.g., Python and MATLAB because the parameters can be changed very easily. Moreover, many of the calculations are independent of each other therefore can be done in conjunction. Thus, graphics processing units (GPUs), which can perform a large number of simple independent calculations, are widely accepted to speed up the training and testing processes.

At all times the above-mentioned training statistics is done using floating-point arithmetic, real applications may require the trained CNNs to operate with fixed-point calculations to reduce delays and improve throughput, whatever the application is. Using fixed-point values, neural networks can be then implemented on Hardware such as FPGAs and ASICs [21]. Various representations are shown in fig.63.

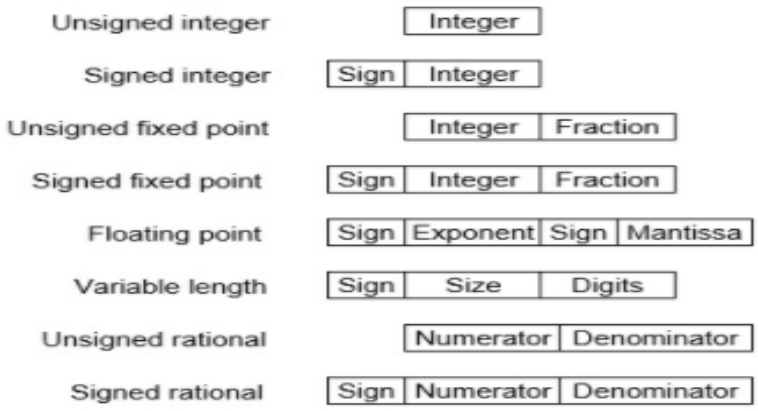

Figure 63: Fixed Point Various Representations

In fixed-point representation, we need to define the integer and fractional word length for our parameters.

To decrease the cost and the power consumption, the fixed-point arithmetic has to be used. Nevertheless, the application fixed-point specification has to be determined. This specification defines the integer and fractional word length for each data. The data dynamic range has to be calculated for determining the position of the binary point with respect to the word length of the integer data. The fractional part word length depends on the operator's word length Fixed point representation divides the number into 3 fields: sign bit field, integer field, and fraction field Signed Fixed-point representation has a range of numbers from $-(2^{(k-1)}-1)$ to $(2^{(k-1)}-1)$, for K bits [22].

Dealing with fixed point numbers takes special measures to enable its use in different mathematical operations such as multiplication, addition and subtraction, which we will discuss now. Fixed-point multiplication is similar to 2's compliment multiplication but it needs the location of the binary point to be deduced after the multiplication to obtain the correct result as shown in fig.64. The deduction of the binary point's location is a task for designers only. But the implementation doesn't know where the binary point is located.

This is due to the fact that the fixed-point multiplication is similar to a 2's complemented multiplication; where there is no extra hardware is required. The bits required for the product is the multiplicand's length added to the multiplier's length.

```
0110.1001  == 6.5625
000100.01  == 4.25
          01101001
        x 00010001
       ------------
          01101001
         00000000
        00000000
       00000000
      01101001
     00000000
    00000000
   00000000
 --------------------
  x000011011111001   ==  0000011011.111001  ==  27.890625
```

Figure 64: Fixed point multiplication

Addition is harder because the points need to be aligned before executing the addition.

When we add more than two numbers all of the same length, the number of bits required for the addition is: the word length plus $\log_2(N)$where N is the number of elements being summed as shown in fig.65 [23].

```
0110.1001  == 6.5625
000100.01  == 4.25
            0110.1001
          + 000100.01
            ------------
              001010.1101  ==  10.8125
```

Figure 65: Fixed point addition

Converting a number to fixed point can be done in software in various methods, We used a simple mathematical formula for the conversion that depends on multiplying the number to be converted by the weight (in binary representation) corresponding to the number of fraction bits used and flooring the answer then dividing it by the same aforementioned weight.

Hence, we notice that using fixed-point representation comes with a cost however as it leads to a decrease in accuracy that varies with the total number of bits used and the portion of these bits designated to the integer and fraction fields.

Choosing the correct number of bits for fraction, and integer parts were done by observing the values of the parameters after training and choosing a number that is satisfactory enough to represent them.

For example, knowing that all the parameters are values below one only one bit can be assigned to the integer part to be fully represented (not including the sign bit), After converting the numbers, we test the model to measure the loss in accuracy as shown in table 7, and decide whether a suitable number of bits were chosen.

After careful examination of the inputs and output of each layer and output parameters the design chosen for fixed-point was: 16-bit numbers with 1 sign bit, 1 integer bit, and 14 fraction bits. Except for the linear layer which needed 5 bits for the integer part as it contained big numbers.

This gave an accuracy of **89.93**:

Chosen sizes	Accuracy
Without using Fixed point	**91.45%**
Total bits:64 Fraction bits:60	**91.4%**
Total bits:32 Fraction bits:29	**91.38%**
Total bits:20 Fraction bits:17	**91.12%**
Total bits:16 Fraction bits:13	**89.93%**
Total bits:8 Fraction bits:5	**79.34%**

Table 7: Accuracy for different representations

After further inspection of the network, it was noticed that some layers may need a different fixed-point configuration as shown in fig.66.

These changes improved the accuracy from 89.93 To 90.8, while keeping stages where multiplication occurs, of same configuration to ease the multiplication process in hardware.

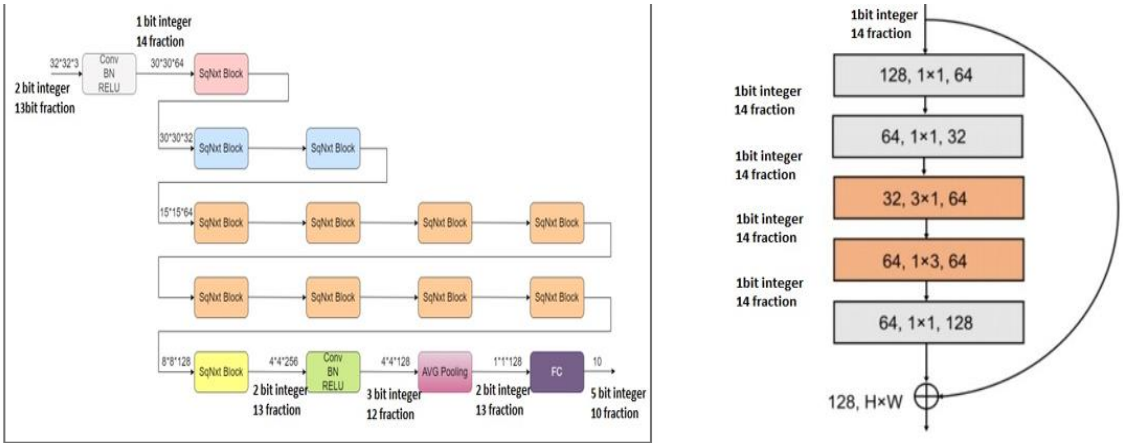

Figure 66: Different fixed-point configuration within layers

5.3 Batch Normalization And Convolution Merging:

To increase the efficiency of the hardware used for batch normalization, new parameters were deduced from the normal batch normalization equation as shown in fig.67. The new parameters called a and b, where a was multiplied by the convolution weight the precedes it, and b was used as the bias of the convolution.

Then a new network was created without a separate batch normalization layer and using convolution layers with biases.

After testing the new model, we verified that the accuracy remained nearly the same.

This enabled us to greatly reduce the size of the hardware used, while also saving power and having less delay.

$$\text{the equation is } Y=\frac{x-\mu}{\sqrt{\sigma^2+\varepsilon}}\times\gamma+\beta=\frac{\gamma}{\sqrt{\sigma^2+\varepsilon}}\times x+\beta-\frac{\mu\times\gamma}{\sqrt{\sigma^2+\varepsilon}}=ax+b$$

$$\text{where } a=\frac{\gamma}{\sqrt{\sigma^2+\varepsilon}} \text{ and } b=\beta-\frac{\mu\times\gamma}{\sqrt{\sigma^2+\varepsilon}}$$

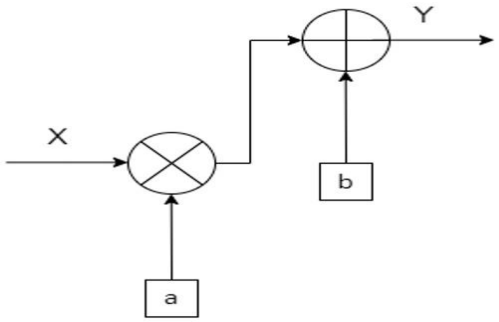

Figure 67: New batch normalization parameters

5.4 Test Vectors Generation:

In order to be able to use the parameters generated in the hardware, and also test the hardware using the inputs and outputs generated, test vector files were generated.

These files contain the output parameters, and the output and input of each stage in fixed point representation where each number has its own line inside the files to make reading the numbers and dealing with them easier, and each layer has its own file.

5.5 Summary:

This chapter discusses the software part where we trained different squeezeNext networks with different modifications and chose the suitable one. Also we discussed the batch normalization problem and its solution by merging the batch normalization with convolution. Finally, we covered the software model we developed according to our design to generate test vectors for each layer.

Chapter 6: Optimizations For Squeezenext Architecture

In this chapter, the goal is to enhance the design in terms of timing, area and power along with fixing the synthesis issues and solve any simulation and synthesis mismatch.

The following aspects were categorized to make the most out of the required optimizations:

- Timing and Pipelining
- Area aware optimizations
- Power Reduction Techniques
- Accuracy enhancements

6.1 Timing Enhancements

We hardly met the 100 MHz constraints so we did many improvements to increase the frequency used and finally it can be in range between 150MHz to 200MHz. Also we applied other modifications to increase the throughput and reduce the latency.

6.1.1 Break the critical paths to increase the operating frequency.

We broke the paths which have the highest combinational logic delay, most of these paths were at the end points of the adder trees, BRAMs output pins, DSPs input pins and some paths were suggested using Vivado Timing Analysis. This modification improved the operating frequency from 75MHz to above 140MHz.Also we used the Register Retiming option in synthesis to shorten the clock cycle which will be discussed later in details.

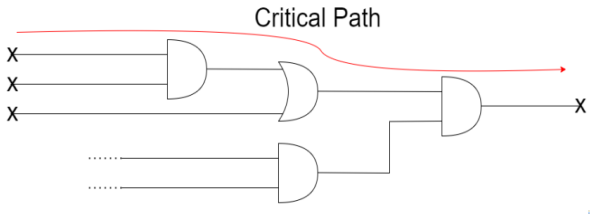

Figure 68: Combinational logic before breaking the critical path

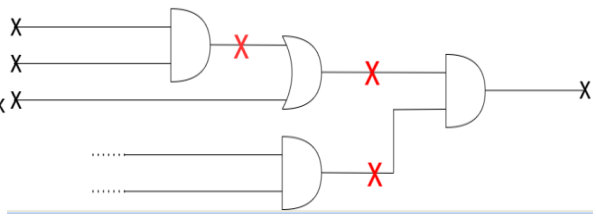

Figure 69: Combinational logic after breaking the critical path

6.1.2 Pipelining:

We used pipelining techniques to increase the throughput of the network while sacrificing some area for the added registers.

6.1.2.1 *Pipelining within the layers:*

We applied pipelining in layers through using pipelined adder trees to perform the convolution with higher throughput.

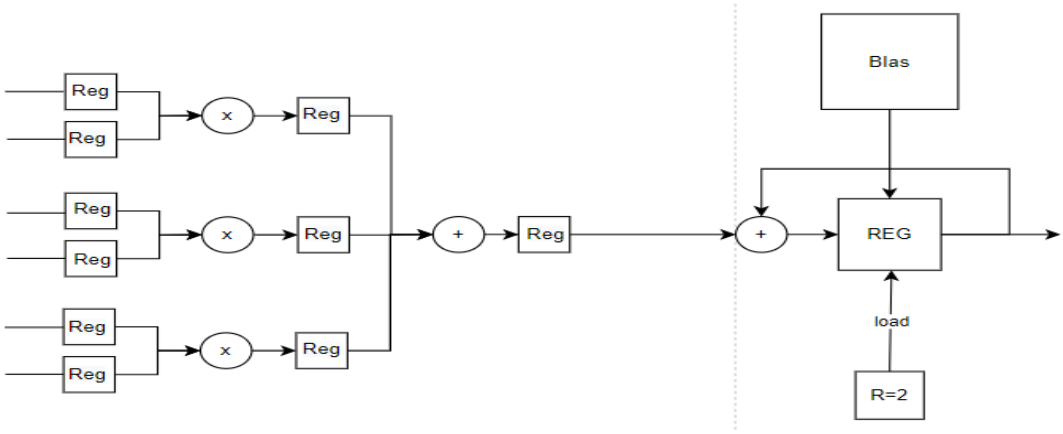

Figure 70: Three input pipelined adder tree with 3 channels

6.1.2.2 *Pipelining between layers:*

Pipelining between layers is done in the data transition between layers, as the different layers can now do processing on different images at the same time, also we improved this by not using the Transition traditional pipelined as discussed before, but we improved the pipelining through the system controller and the transition dual port memories which makes the current layer not have to wait the previous layer to complete the data processing and store it in memory, but the current layer can start after a certain number of data is stored in memory, this method has much improved the total throughput and the time taken to do processing on an image from the first layer to the last layer.

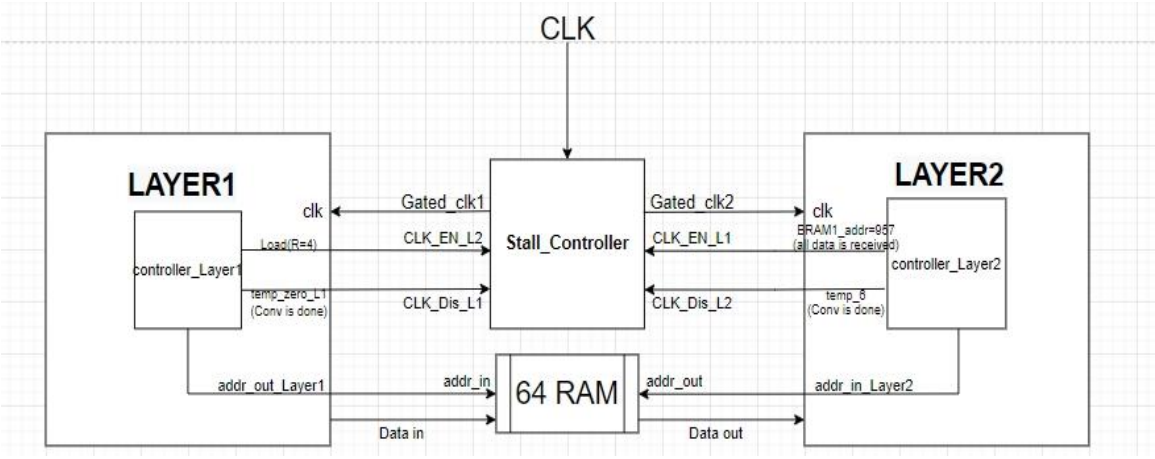

Figure 71: Connections between the first two layers

The next table shows the optimization effect on the total throughput:

	Before Optimization	After Optimization
Total number of Clock cycles needed to get the first output	194490 Clock Cycles	129678 Clock Cycles
Total number of Clock cycles needed to get the rest outputs	10805 Clock Cycles	10805 Clock Cycles

Table 8: Optimization in Clock cycles after Pipelining between layers

Improvement percentage for the first output= $\frac{194490-129678}{194490} * 100\% = 33.324\%$

6.1.3 Parallelism inside layers:

Parallelism means many calculations or the execution of processes are carried out simultaneously, that will decrease the time of execution of CNN architectures. We used parallelism in filters by repeating the block N times to increase throughput. We used parallelism in channels by repeating the multiplier in one block M times to increase throughput. We say we have M×N parallelism.

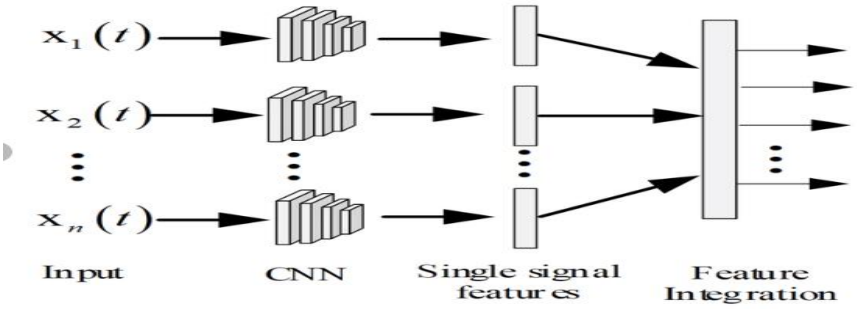

Figure 72: Parallelism process for multi-channel network

6.1.4 Merging some Convolutions in layers:

This is done by applying the convolutions at some layers which have skip connection memory to work at the same time, this method reduced the number of clock cycles needed by merging the normal convolution with the skip connection convolution.

6.1.5 Pipelining inference of multiple images

Our design support inferring multiple images one by one into the model (Pipeline inference) due to pipelining between layers, to make a prediction of the class of a specific image we can load a set of images on the FPGA not just one image which increases the total throughput of the system.

The next table shows the optimization effect on the total throughput:

	Before Optimization	After Optimization
Total number of Clock cycles needed to get the first output	127016 Clock Cycles	129678 Clock Cycles
Total number of Clock cycles needed to get the rest outputs	127016 Clock Cycles	10805 Clock Cycles

Table 9: Optimization in Clock cycles after Pipelining inference

Improvement percentage= $\frac{127016-10805}{127016} * 100\% = 91.4932\%$

6.2 Area Enhancement

6.2.1 Using Counter-Based Controller instead of FSM

This method has reduced the resources taken nearly to half, we thought in a more structural way about the resources needed and we didn't use any additional resources, i.e. the controller of the first layer consists of two 2-bits counters and two 5-bits counters.

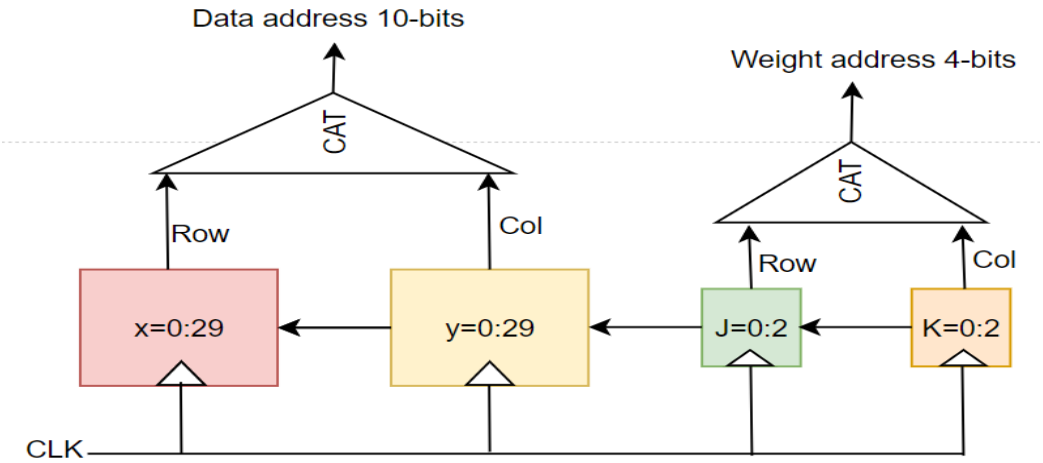

Figure 73: Layer 1 controller

6.2.2 Using one controller to some layers:

We used the traditional pipelining for the layers from layer 10 to layer 16 as they work the same speed and have identical architectures, so we used only one controller for the seven layers instead of seven controllers.

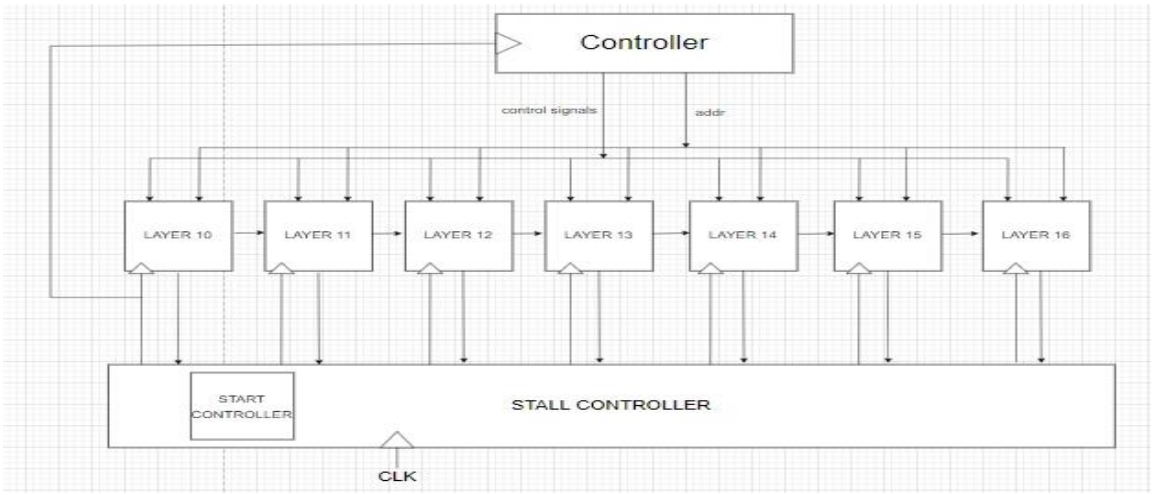

Figure 74: Network layers from layer 10 to layer 16

6.2.3 Stride Optimization

Stride value in CONV8 in table 10, is equal 2, but we can make it equal 1 by making stride value in CONV2 and CONV7 is equal 2. Now we can make the size of skip connection memory in layer2 and output memory layer7 reduce to its half.

Layers	#DSP	Name	Wi	Hi	Ci	Kw	Kh	Wo	Ho	Co	S
Layer1	192	conv1	32	32	3	3	3	30	30	64	1
Layer2	256	conv2	30	30	64	1	1	30	30	32	1
		conv3	30	30	64	1	1	30	30	16	1
Layer3	16	conv4	30	30	16	1	1	30	30	8	1
Layer4	32	conv5	30	30	8	1	3	30	30	16	1
Layer5	64	conv6	30	30	16	3	1	30	30	16	1
Layer6	64	conv7	30	30	16	1	1	30	30	32	1
Layer7	256	Conv8	30	30	32	1	1	15	15	64	2
		Conv9	30	30	32	1	1	15	15	32	2
		conv10	15	15	32	1	1	15	15	16	1
		conv11	15	15	16	1	3	15	15	32	1
		conv12	15	15	32	3	1	15	15	32	1
		conv13	15	15	32	1	1	15	15	64	1

Table 10: Size of network layers from 1 to 7

6.2.4 DSPs Optimization

Due to the previous optimization in stride, we notice that layer2 and layer7 are faster than before. Layer2 was ended at 10800 clock cycles, but now it ends in 5400 only due to optimization in stride. So, we don't need 256 DSPs to this layer. We only need

164. And Layer6 finished after 7200 clock cycles, but now it finished after only 1800 due to optimization in stride. So, we don't need 64 DSPs in this layer, we only need 16 DSPs. The next table shows the optimizations effect on the DSPs:

TARGET OPTIMIZATIONS		Before optimizations	After optimizations
DSPs	Numbers	3504 DSPs	3328 DSPs
	Utilizations	97%	92%

Table 11: DSPs Optimization

Improvement percentage = 5%

6.3 Power Reduction Techniques

These techniques effects on the dynamic power only and have no relation with the static power.

6.3.1 Clock gating in the Stall Controller

The Stall Controller is responsible for starting and stopping the layers, We used clock gating with enable and disable signals to achieve this, and instead of applying the clock gating to the Registers Enable Pins we applied it to the clock pins.

The Clock gating main advantages are to reduce the dynamic (switching) power consumption and decrease the FAN OUT of the main clock, and simplify the routing.

We faced glitch problems in the Latch Free Clock Gating which causes:

- Pulse Clipping : that leads to Pulse Width Violation
- Spurious Clocking: that leads to hold time and setup time violations

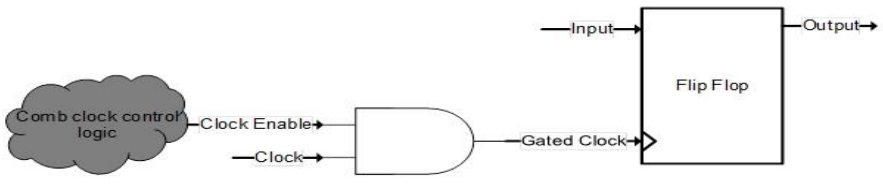

Figure 75: Latch-Free clock gating

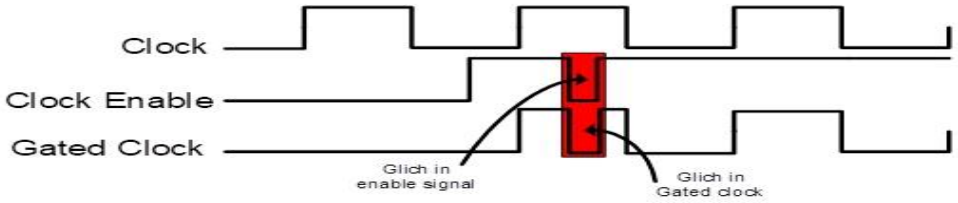

Figure 76: Glitch problem in the gated clock

So to avoid these problems we used Latch Based Clock Gating which consists of Active Low Latch with *AND* Gate and put them together in Integrated Gated Clock (ICG) Cell to avoid the different routing delay.

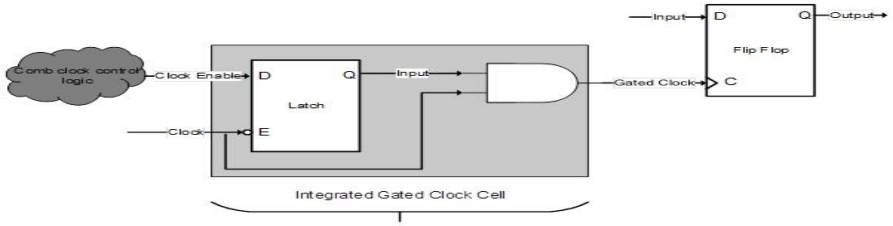

Figure 77: Latch-based clock gating

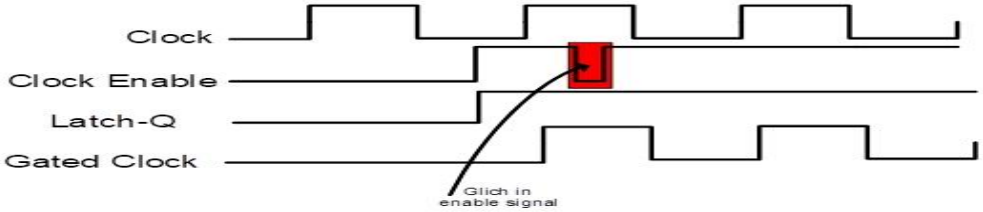

Figure 78: No Glitch problem in the gated clock

Clock gating location should be near to the clock source to avoid the power consumption in the buffers in the clock distribution network, and the buffers are placed after the clock gate.

6.3.2 Operand Isolation:

In case of doing some processing on data buses, we need to isolate or register the inputs to avoid unnecessary glitching power due to different arrival times of the bits of the data buses.

Figure 79: Registering 32-bits multiplier inputs

6.3.3 Avoid XOR Gates:

In XOR gate, any transition at any of its inputs will always cause a transition at the output which leads to high switching power consumption so it's called Power Hungry.

Comparator operations involve a lot of XORs, so we reduced our dependence on the comparisons in our design.

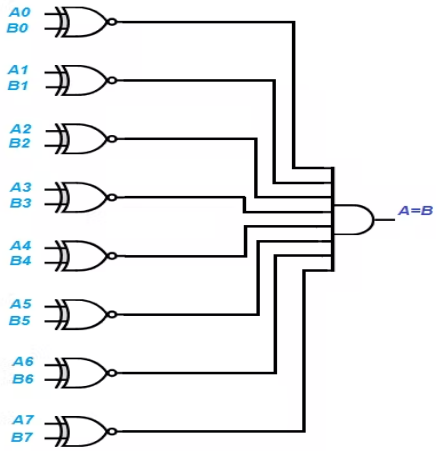

Figure 80: 8-bits comparator

Also the methods used in the Area Enhancements lead to reduce the consumed power.

6.4 Accuracy Enhancements

In Fixed-Point Multiplication and Addition, We used a fixed 16-bit data size (3 bits for integer part and 13 bits for fraction part) for all the layers in the network, and we got accuracy less than the software with approximately 1.5% so we applied some modifications which make the HW accuracy reach nearly to the SW floating point accuracy.

6.4.1 Change data size within the layer

By increasing the number of bits to be 18-bits inside the adder trees only and trimming the extra bits before storing the data in memory. We increased the size to increase the number of fraction bits which effects on the accuracy, i.e. as shown in the next figure, if we used 5-bits output instead of 6-bits, the least fraction bit will be neglected and the output will be 111.101 instead of 111.1011.

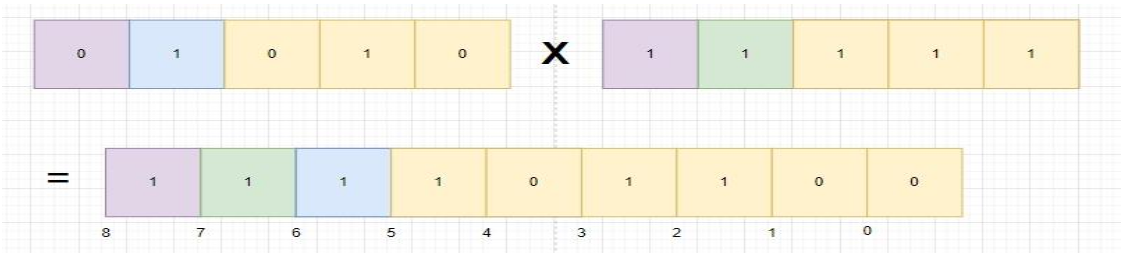

Figure 81: Increasing the size of the output fraction bits

6.4.2 Moving the Radix Point in the fixed point representation

We did some software statistics to know the range of numbers in all layers (Max,Min and Avg), We found that we don't need to use the same number of integer and fraction bits in all layers, and then we found a way to move the radix point.

6.4.2.1 Moving the Radix Point to Left:

We use this method to increase the number of fraction bits (only if there's no overflow in the output integer part).

6.4.2.2 Moving the Radix Point to Right:

We use this method to increase the number of integer bits by taking advantages of the overflow, this is done by applying Signed/Unsigned multiplication then calculate the radix point position.

6.5 Summery

This chapter discusses different optimization techniques and their effect on different fields, we applied the timing enhancements to increase the operating frequency and the total throughput, the area aware enhancements used to reduce the number of resources used, hence reduce the consumed power, Power reduction techniques are used to decrease the total dynamic power and the Accuracy Enhancements make the HW model accuracy reach to the SW model accuracy.

Design optimization is considered as an incremental process which applies increasing engineering effort and tool computational time to make the design meet timing constraints. Good HDL coding can positively influence the ability of the design implementation tools to achieve the desired timing performance.

But these techniques have tradeoffs, so we need to get the best strategy for our design.

Chapter 7: Synthesis And Implementation

7.1 Synthesis Flow

Synthesis is the process of transforming an RTL-specified design into a gate-level representation. Vivado® synthesis is timing-driven and optimized for memory usage and performance. FPGA synthesis is a vital phase in the deployment of Squeeze-Next CNN architecture. Xilinx Vivado offers a variety of options and switches for synthesis, and even ready-to-use strategies that help designers employ Vivado Synthesizer to achieve a target for their RTL, such as area, timing and so on. Now we are going to offer some strategies that affect our design in vivado. [24]

7.1.1 flatten_hierarchy:

Determines how Vivado synthesis controls hierarchy through these strategies:

- **none:** The output of synthesis and the original RTL have the same hierarchy.
- **full:** Leaving only the top level and do completely flatten the hierarchy.
- **rebuilt:** The synthesis tool is allowed to flatten the hierarchy and perform synthesis, then based on the original RTL , it rebuilds the hierarchy.

7.1.2 gated_clock_conversion:

Enables and disables the synthesis tool to convert the clocked logic with enables. Performing this conversion helps the tool to use the dedicated clocking resource, but it also uses different clock enables and it will need more control sets in the design which can have other effects.

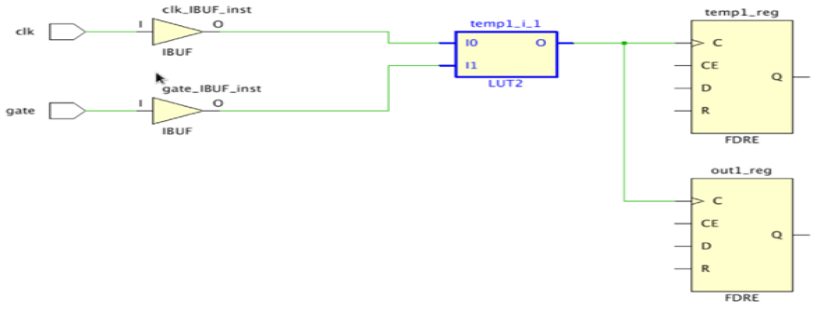

Figure 82: Clocking structure with LUT without gated clocks conversion

One way to fix these problems and to remove the gates is to rewrite the RTL code. However this needs a lot of work, and if the design is being prototyped in FPGAs, the RTL is not allowed to be changed. Another way to fix this is to make the synthesis tool allow converting those gates, then the clock will directly drive the register clock pin and the gating logic will go to the clock enable pin.

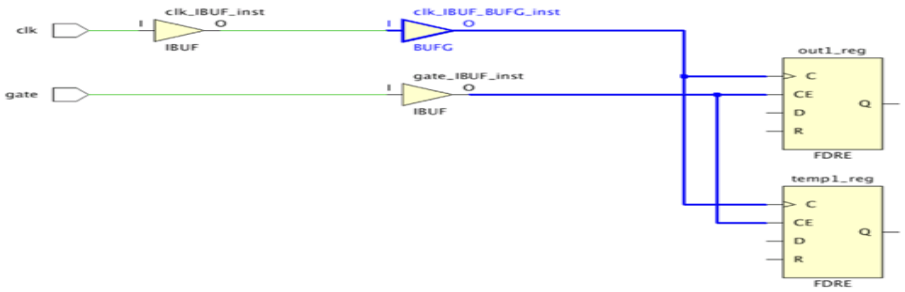

Figure 83: Same structure with gated clocks converted

7.1.3 bufg:

Controls how many BUFGs the tool infers in the design. If the *-bufg* option is set to 12 and there are three BUFGs instantiated in the RTL, the Vivado synthesis tool infers up to nine more BUFGs.

7.1.4 fanout_limit

Specifies the maximum number of loads driven by a signal before it starts to replicate the logic.

7.1.5 directive

This option runs Vivado synthesis with different optimizations:

- Default: performs the default settings in Vivado synthesis tool.
- RuntimeOptimized: Reduces the synthesis run time.
- AreaOptimized_high:
- AreaOptimized_medium:
- AlternateRoutability: Algorithms to improve route-ability
- AreaMapLargeShiftRegToBRAM:
- AreaMapLargeShiftRegToBRAM:
- FewerCarryChains:

7.1.6 retiming:

The aim of this procedure is to shorten the clock cycle or reduce circuit area, as Register Retiming is an optimization technique in timing that moves registers forward or backward across combinational elements in a circuit to reduce the critical path. [25]

As our target is throughput, we can sacrifice some area, because the total number of registers will increase, to serve the throughput by decreasing the critical paths.

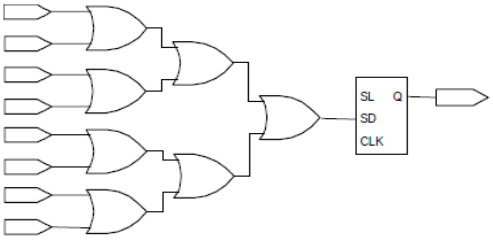

Figure 84: Before Retiming

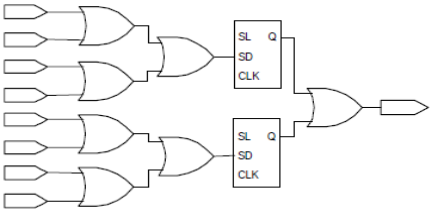

Figure 85: After Retiming

7.1.7 fsm_extraction:

Controls how synthesis extracts and maps finite state machines with FSM Encoding protocols. We didn't use this option as we used Counter-based controller instead of FSM as we found that it is better in area.

7.1.8 keep_equivalent_registers:

Prevents merging of registers with the same input logic, it increases the number of registers in design, hence the consumed power. So it's kept U*nchecked*.

7.1.9 resource_sharing:

Sets the sharing of arithmetic operators between different signals. As our target is throughput, we didn't use it and kept it OFF.

7.1.10 Synthesis Options

Flow_AreaOptimized_high
-flatten_hierarchy : rebuilt
-gated_clock_conversion : off
-bufg : 12
-fanout_limit : 10000
-directive : AreaOptimized_high
-retiming : Checked

7.2 Constraints:

The requirements that must be met by the compilation is defined by the design constraints, they are used to ensure that the design is functional on the board. Not all constraints are used by all steps in the compilation flow, some of them are done in synthesis flow and others are critical in PAR. For example, physical constraints are used only during the implementation steps. [26]

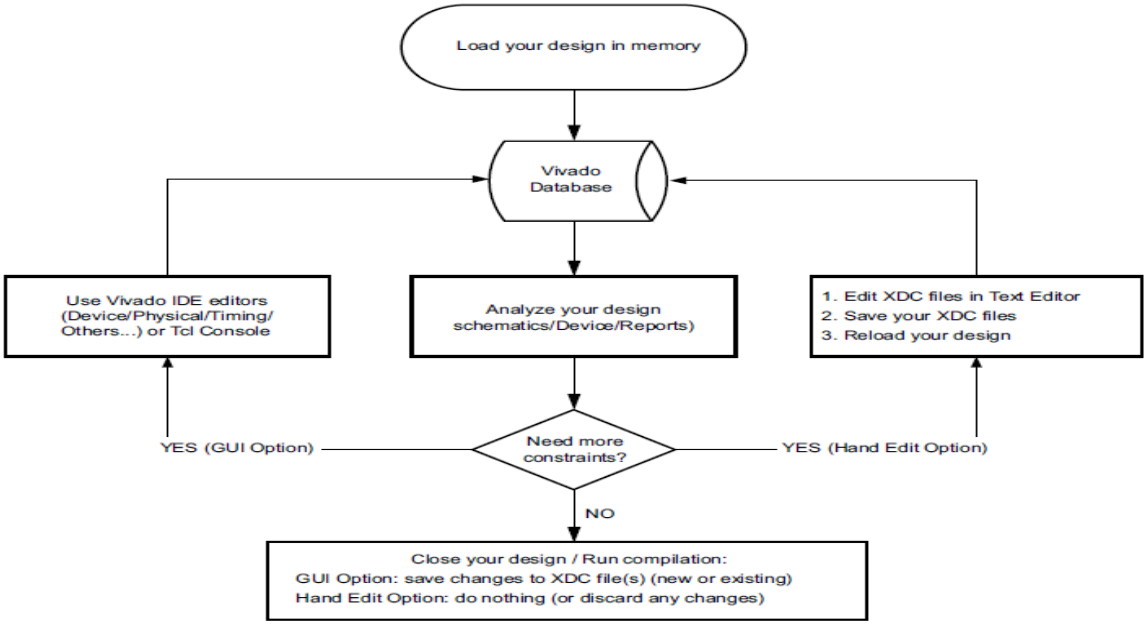

Figure 86: Constraints Editing Flow

The four primary types of constraints are synthesis constraints, I/O constraints, timing constraints and area/location constraints. [27]

- **Synthesis constraints:** They influence the details of how the synthesis tool will translate the HDL code to RTL.
- **I/O constraints** (pin assignment): They are used to assign a signal to a specific I/O (pin) or I/O bank on the target board.
- **Timing constraints:** They are used to specify the timing characteristics of the design. Timing constraints effect on all internal timing interconnections, and they can be either path-specific or global.
- **Area constraints:** They are used to map specific design element or specific fixed resource to a range of resources within the FPGA.

Timing analysis for FPGA designs is easier than ASIC designs, as the FPGAs have Clock distribution network with minimum skew also Xilinx Vivado timing analysis engine has timing models for each cell in Virtex-7 FPGA on VC709 board (Our target FPGA).

7.3 Implementation Flow

The Vivado Design Suite implementation process transforms a logical netlist and constraints into a placed and routed design, ready for bitstream generation. The implementation process walks through the following sub-processes [28]:

7.3.1 Opt Design:

By optimizing the logical design to make it easier to fit onto the target Xilinx device. Directives provide different modes of behavior for the **opt_design** command. The available directives are described in the next table:

Directive	Description
Explore	Runs multiple passes of optimization.
ExploreArea	Runs multiple passes of optimization with emphasis on reducing combinational logic.
AddRemap	Runs the default logic optimization flow and includes LUT remapping to reduce logic levels.
ExploreSequentialArea	Runs multiple passes of optimization with emphasis on reducing registers and related combinational logic.
RuntimeOptimized	Runs minimal passes of optimization, trading design performance for faster run time.
NoBramPowerOpt	Runs all the default opt_design optimizations except block RAM Power Optimization.
ExploreWithRemap	Same as the Explore directive but includes the Remap optimization.
Default	Runs opt_design with default settings.

Table 12: Opt Design directives descriptions

From these directives we try to choose the one help us to improve timing problems, so we choose **ExploreWithRemap** directive combine between reducing combinational logic and LUT remapping to reduce logic levels.

7.3.1 Power Opt Design (optional):

Optimizes design elements to reduce the power demands of the target Xilinx device. Power optimization is an optional step that optimizes dynamic power using clock gating. The command also performs intelligent clock gating to optimize power.

7.3.2 Place Design:

Places the design onto the target Xilinx device and performs Fan-out replication to improve timing. The available directives are described in the next table:

Directive	Description
Explore	Higher placer effort in detail placement and post-placement optimization.
WLDrivenBlockPlacement	Wirelength-driven placement of RAM and DSP blocks. Override timing-driven placement by directing the Placer to minimize the distance of connections to and from blocks. This directive can improve timing to and from RAM and DSP blocks.
EarlyBlockPlacement	Timing-driven placement of RAM and DSP blocks. The RAM and DSP block locations are finalized early in the placement process and are used as anchors to place the remaining logic.
ExtraNetDelay_high	Increases estimated delay of high fanout and long-distance nets. This directive can improve timing of critical paths that meet timing after place_design but fail timing in route_design due to overly optimistic estimated delays. Two levels of pessimism are supported: high and low. ExtraNetDelay_high applies the highest level of pessimism.
ExtraNetDelay_low	Increases estimated delay of high fanout and long-distance nets. This directive can improve timing of critical paths that have met timing after place_design but fail timing in route_design due to overly optimistic estimated delays. Two levels of pessimism are supported: high and low. ExtraNetDelay_low applies the lowest level of pessimism.
ExtraPostPlacementOpt	Higher placer effort in post-placement optimization.
ExtraTimingOpt	Use an alternate set of algorithms for timing-driven placement during the later stages.
RuntimeOptimized	Run fewest iterations, trade higher design performance for faster runtime.
Quick	Absolute, fastest run time, non-timing-driven, performs the minimum required for a legal design.
Default	Run place_design with default settings.

Table 13: Place Design directives descriptions

From these directives we choose the one achieve highest fanout by **ExtraTimingOpt** directive that uses an alternate set of algorithms to improve the timing of critical paths.

7.3.3 Post-Place Power Opt Design (optional):

It is an additional optimization to reduce power after placement.

7.3.4 Post-Place Phys Opt Design (optional):

By optimizing the logic and placement using estimated timing based on placement by including the replication of high fan-out drivers. The directives provide different modes of behavior for the **phys_opt_design** command. Only one directive can be specified at a time, and the directive option is incompatible with other options. The available directives are described in the next table:

Directive	Description
Explore	Run different algorithms in multiple passes of optimization, including replication for very high fanout nets, SLR crossing optimization, and a final phase called Critical Path Optimization where a subset of physical optimizations are run on the top critical paths of all endpoint clocks, regardless of slack.
ExploreWithHoldFix	Run different algorithms in multiple passes of optimization, including hold violation fixing, SLR crossing optimization and replication for very high fanout nets.
ExploreWithAggressiveHoldFix	Run different algorithms in multiple passes of optimization,including aggressive hold violation fixing, SLR crossing optimization and replication for very high fanout nets.
AggressiveExplore	Similar to Explore but with different optimization algorithms and more aggressive goals. Includes a SLR crossing optimization phase that is allowed to degrade WNS which should be regained in subsequent optimization algorithms. Also includes a hold violation fixing optimization.
AlternateReplication	Use different algorithms for performing critical cell replication.
AggressiveFanoutOpt	Uses different algorithms for fanout-related optimizations with more aggressive goals.
AddRetime	Performs the default phys_opt_design flow and adds register retiming.
AlternateFlowWithRetiming	Perform more aggressive replication and DSP and block RAM optimization, and enable register retiming.
Default	Run phys_opt_design with default settings.

Table 14: Post-Place Phys Opt Design directives descriptions

By choosing *AggressiveExplore* directive which allowed to degrade WNS and includes a hold violation fixing optimization, so it is the best option to improve timing problems.

7.3.5 Route Design:

The Vivado router performs routing on the placed design, and performs optimization on the routed design to resolve hold time violations. When routing the entire design, directives provide different modes of behavior for the **route_design** command. Only one directive can be specified at a time. The directive option is incompatible with most other options to prevent conflicting optimizations.

The available directives are described in the next table:

Directive	Description
Explore	Allows the router to explore different critical path placements after an initial route.
AggressiveExplore	Directs the router to further expand its exploration of critical path routes while maintaining original timing budgets. The router runtime might be significantly higher compared to the Explore directive because the router uses more aggressive optimization thresholds to attempt to meet timing constraints.
NoTimingRelaxation	Prevents the router from relaxing timing to complete routing. If the router has difficulty meeting timing, it runs longer to try to meet the original timing constraints.
MoreGlobalIterations	Uses detailed timing analysis throughout all stages instead of just the final stages, and runs more global iterations even when timing improves only slightly.
HigherDelayCost	Adjusts the internal cost functions of the router to emphasize delay over iterations, allowing a tradeoff of run time for better performance.
RuntimeOptimized	Run fewest iterations, trade higher design performance for faster run time.
AlternateCLBRouting	Chooses alternate routing algorithms that require extra runtime but may help resolve routing congestion.
Quick	Absolute, fastest compile time, non-timing-driven, performs the minimum required for a legal design.
Default	Run route_design with default settings.

Table 15: Route Design directives descriptions

From these directives we try to choose the one help us to improve timing problems, so we choose **NoTimingRelaxation** that make the router never release until meeting the original timing constraints.

7.3.6 Post-Route Phys Opt Design (optional):

By optimizing the logic, placement, and routing using actual routed delays.

The available directives in **Post-Route Phys Opt Design** are described in previous tables. By choosing the **AggressiveExplore** option where the router uses more aggressive optimization thresholds to attempt to meet timing constraints.

7.3.7 Write Bitstream:

Typically, bitstream generation follows implementation. Although not technically part of an implementation run, bitstream generation is available as an incremental step for Xilinx device configuration.

7.4 Downloading The Bitstream Into The FPGA

7.4.1 Bitstream Overview

A bitstream is a binary sequence that comprises a sequence of bits. These are used in FPGA applications for programming purposes and to establish communication channels. FPGA bitstream is a file containing the programming data associated with your FPGA chip. It is a file containing binary data that codes for all of the configuration information of a given FPGA. Not only that, but it also has the commands needed to control the functionality of the chip. All of this information is arranged in frames which make up the fundamental blocks for the FPGA configuration memory space. The bitstream file is generated after an elaborate process that comprises a lot of designing, synthesis, and validation.

7.4.2 Frequency Synthesis

The clock on the virtex-7 board is differential with frequency 200MHZ, where we want different output frequencies than the 200MHZ, so we used the IBUFGDS primitive to give single output clock that can be used then by the PLL or MMCM block to give the required frequency.

7.4.2.1 IBUFGDS

The IBUF and IBUFG primitives are the same. IBUFGs are used when an input buffer is used as a clock input. In the Xilinx tools, an IBUFG is automatically placed at the clock input sites.[]

And the same is valid for IBUFDS and IBUFGDS for differential IO.

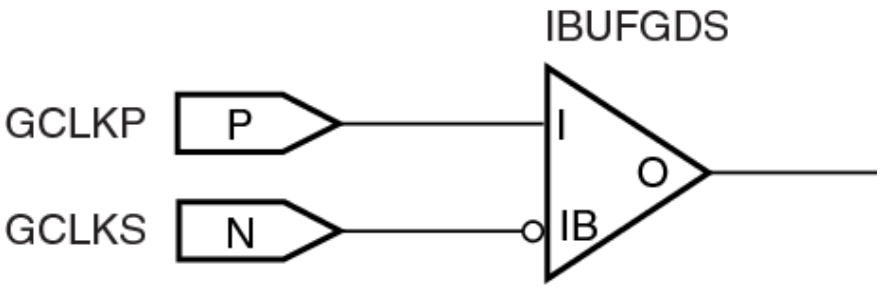

Figure 87: IBUFGDS differential input clock

7.4.2.2 MMCM

Mixed Mode Clock Manager is a PLL with some small part of a DCM tacked on to do fine phase shifting, so its mixed mode , the PLL is analog, but the phase shift is digital. Thus the MMCM can do everything the PLL can do and the phase shifting from the DCM block.

We can choose any output frequency to be synthesized; synthesizing frequencies higher than 200MHZ is available. Also we can choose different clock outputs with different frequencies and phases. MMCM is generated using the IP CORE Clocking Wizard.

The IP CORE Clocking Wizard. This LogiCORE ™ IP simplifies the creation of HDL source code wrappers for clock circuits customized to the clocking requirements. The Wizard guides in setting the appropriate attributes for the clocking primitive, and allows overriding any wizard-calculated parameter. In addition to providing an HDL wrapper for implementing the desired clocking circuit, the Clocking Wizard also delivers a timing parameter summary generated by the Xilinx ® timing tools for the circuit. [29]

7.4.3 Output Observation

We used VIO and ILA for output observation.

7.4.3.1 VIO (Virtual Input Output)

The LogiCORE™ IP Virtual Input/Output (VIO) core is a customizable core that can both monitor and drive internal FPGA signals in real time. The number and width of the input and output ports are customizable in size to interface with the FPGA design. Because the VIO core is synchronous to the design being monitored and/or driven, all design clock constraints that are applied to your design are also applied to the components inside the VIO core. Run time interaction with this core requires the use of the Vivado® logic analyzer feature. [30]

The VIO provides virtual LEDS and many input ports for the observation of the output and changing the inputs from the software instead of the FPGA in/out ports.

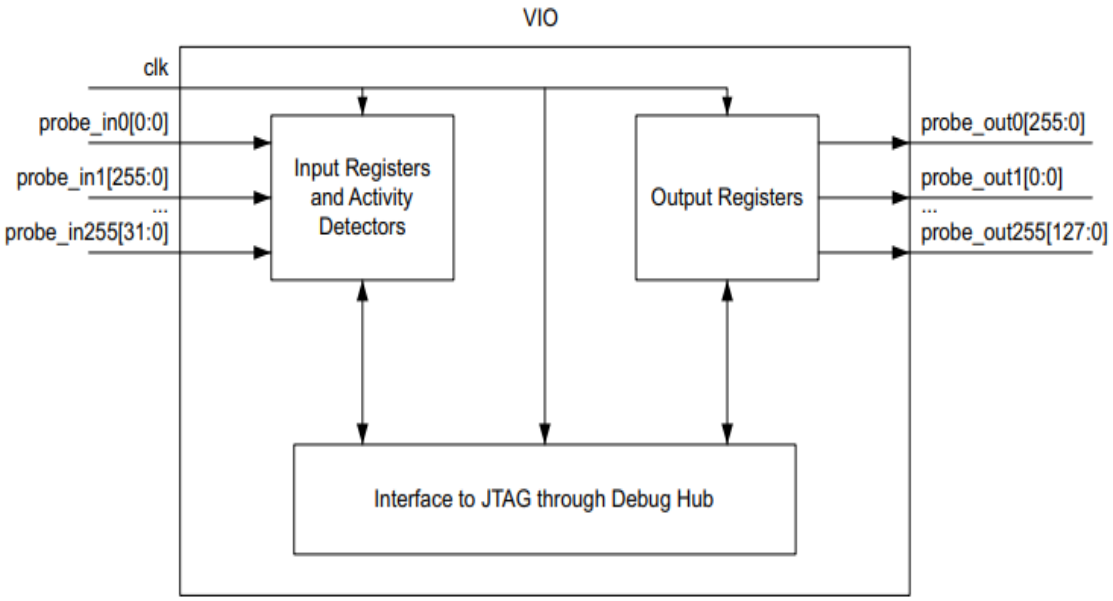

Figure 88: VIO block diagram

7.4.3.2 ILA (Integrated Logic Analyzer)

The ILA IP block is customized for monitoring the internal signals of the design; the ILA shows the signals in timing waveforms and the state of each signal. The ILA is as VIO, since it's synchronous to the design so all the design constraints related to the clock is also applied to the ILA IP. [31]

7.4.3 Top Module Setup

For downloading the bitstream and testing the code we used the following setup in **fig1.**

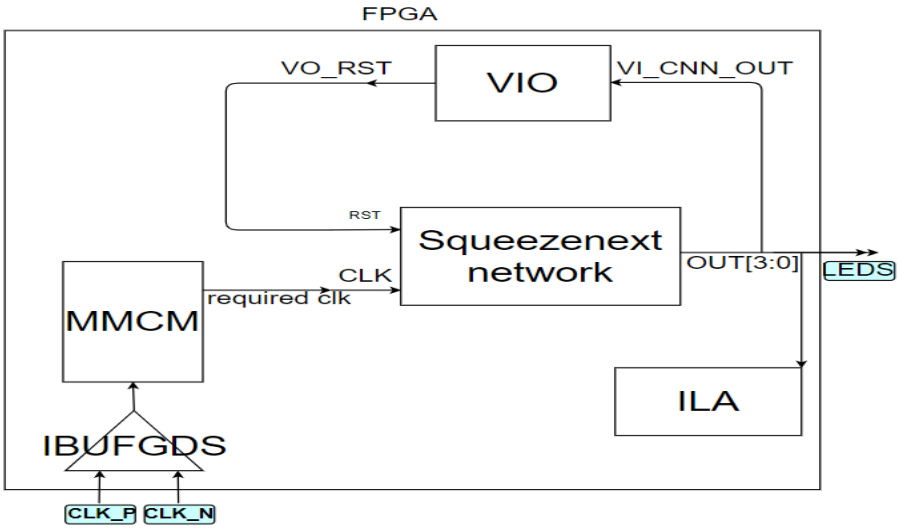

Figure 89: The used setup for downloading the code

7.4.4 I/O Constraints

The following table shows the I/O constraints and those we chose from the datasheet of the VIRTEX-7 VC709 FPGA.

Signal	IOSTANDARD	GPIO Pin	Pin	PULLTYPE
Clk_N	DIFF_SSTL15	__	G18	__
CLK_P	DIFF_SSTL15	__	H19	__
reset	LVCMOS18	SW2.10	BA30	PULL_UP
OUT[3:0]	LVCMOS18	DS2.2, DS3.2, DS4.2, DS5.2	AM39, AN39, AR37, AT37 respectively	PULL_UP

Table 16: I/O constraints

7.5 Summary:

This chapter provides a discussion on the synthesis flow and which strategies give the best results for our design, the constraints including timing constraints and physical constraints, the implementation flow and the best strategies for our design, generating and downloading bitstream on FPGA.

Chapter 8: ASIC Flow

In this part we will introduce many sections, the first section is introducing the purpose why we want to complete in ASIC flow and what problem we face in ASIC implementation, the Methodology we followed to test and implement the whole system.[32]

8.1 Purpose for ASIC flow implementation

At first our project aim is to choose and implement a CNN network for image detection in FPGA, but we thought it would be better if we add more goals to the project to enhance the project and increase our experiences like completing the ASIC flow.

So, for that purpose we want to participate in this field and have more experiences in this field that is rich with knowledge, and have the highest opportunity to cooperate in this important field of technology. We also need to make our specific design to squeeznext network to achieve better delay and power.

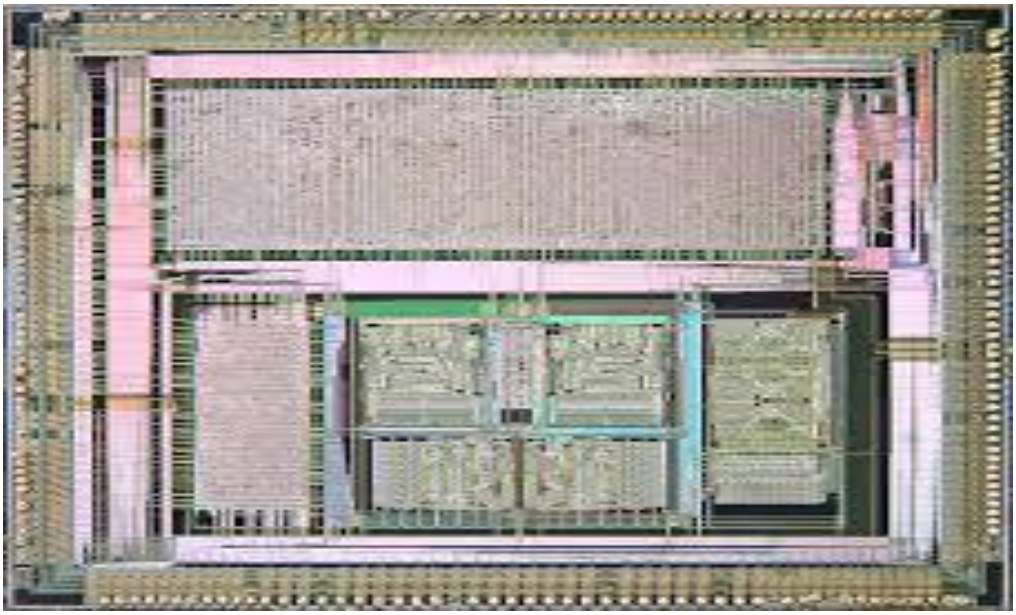

Figure 90: Application Specific Integrated Circuit example

8.2 Problems in ASIC Flow Design

In this section, we will shed a light on some problems we faced when we start ASIC flow until we finish, and how we solve those problems.

8.2.1 Arithmetic Blocks

There are many differences between any digital design RTL implemented in FPGA and ASIC, especially memories and arithmetic blocks and this is because the differences in the internal designs in both FPGA and ASIC, FPGA's building block is LUTS and Registers, while ASIC's building block is standard cells, each one of them differs in their delay, area and the internal design, Also the tool complexity and algorithm is differing from FPGA than ASIC, as FPGA IC is limited with a number of resources and special blocks, while ASIC is unlimited, so it's important to optimize resources usage in FPGA, also there is a high performance and high speed Arithmetic block called DSP which is more important for multiplication operation in the convolution block.

So, we faced a problem when we merge between our RTL design that targeting FPGA to a design that target ASIC. For an example we use DSP block to implement multiplication block which is essential in convolution block, so we need to replace the DSP with a block as efficient as the DSP, fortunately there are many multiplications algorithms like Booth multiplier, and at last we use the multiplier implemented with tool itself as it's fast and suitable for large size data width.

8.2.2 Memory Blocks

The main important block in any design RTL is the memory especially if this design is pipelined, our design use pipeline between 18 layers and we need separate numbers of memory for data and for weight essential for each convolution, we use BRAMs when we implement the RTL design of memory, but in ASIC we need to implement an efficient memory with low power usage and high throughput but it's difficult to implement memories in ASIC, especially for large size memories, so we search a lot until we found an open source that implement memories in ASIC which called OPEN SOURCE RAM, and replace each memory in the design with the new memory.

We faced problem from the size of memory we want to implement, at first, we want to implement memory has size same as the size of BRAMs which is 1024 word, However, this size is not suitable as the propagation delay increases especially the last element in the memory, to solve this problem we divide this memory that includes 1024 word into 32 parts and their output is connected with multiplexer and this way reduce the worst propagation delay taken by the memory path.

8.2.3 Setup Violation

One of the most important targets is the clock speed we need the whole system to be faster and have as much throughput as it can, however the digital design engineer is more cautious from the timing and violation, and there are two types of violation and they are independent on each other, which are setup violation and hold violation.

Setup time is an amount of time that data of a synchronous input must be stable before the active edge, and if the input change at this window it causes Meta-stability, while Hold time is an amount of time the data should of a synchronous input must be stable after the active edge, and also cause Meta-stability if the input data violates.

At first there are many specifications that limit some constrains, like clock frequency, output and input delays and clock uncertainty, then after the design tool elaborate the RTL file into netlist of simple logic blocks the propagation delay (for worst and best delay) is no longer unknown, and after Place and Route is finished the routing delay becomes known, so the whole delays are all known especially so we can now calculate and estimate the critical path that identify the worst case scenario that could have setup violation, and the how much slack remains.

But if we have setup violation, then we can either reduce the clock speed which is not applicable or we can break the critical path, so that the propagation delay reduced and we can now have plenty of positive slack to increase the clock frequency.

Many critical paths exist in the paths contains complex compositional circuit like arithmetic circuit and also in the paths contain large memory size so to avoid any violation we should increase number of pipelines inside these paths and break the largest paths, in our design critical paths exist in path from filter to memory or from memory to

filter due to large memory that lead to large propagation delay also the filter itself contains sequential layers of adders preceded with layer of multipliers that have largest propagation delay, also in the last layer there is comparator tree which have sequential layers of comparators to get the index of the predicted input image, so how we solve the violation at these paths? For memories we break it to mini memories and their output connected to multiplexer and for the filter and comparator tree we separate each layer from the following layer with layer of registers,

What about hold time violation? At first hold time violation occurs in the path that have small propagation delay like path between two register and there is no combinational block exist between them, and this path can be found inside controller and also inside the filter itself, so this problem can be solved if we put some large buffers to increase the propagation delay and avoid the hold time violation. Both violations can be accurately calculated after the PNR (Place and Route) after all delays becomes no longer unknown like standard cell delay, routing delay and parasitic delay.

8.3 Step and Method

At first, we have RTL design of 18 layers each layer may have one or more memories and they have all controllers that have multiple of counters and there is big controller that control the stall action inside layers, and each layer have filters contains layers of multiplier and adders.

8.3.1 Method

So at first we did not start to follow the ASIC flow for the whole system that include 18 layers, instead we need to start as easy as simple so we began with small blocks like memory of small size, multiplier and even adder all with same size in the design, and after that we began with large blocks like filter of one output then the real filter (of several outputs contains parallism) controller alone and even large memories, then we go higher with a whole layer but the small one then we start with large one like layers from layer7 to 18, after that we start with the whole system and all that to avoid getting violation and to be clear to trace. But we face problems when the layer is bigger

because of the performance of our processors, so not all layers are performed, but this the way we need to follow.

8.3.2 Steps

What about steps, first step after design the RTL code we began with synthesis in this step its input is RTL design code written in Verilog or VHDL language and the output is a Netlist of logic gates connected and mapped with standard cells in the library.

8.3.2.1 Synthesis

In Synthesis step we began to define the top module name, the paths of the standard cells library, then we define the work library and the paths of the RTL files after that we began with elaboration which change the RTL complex design into small logic gates connected together which perform the same functionality of the RTL design after this we link between the standard cells and the logic gates after optimization occurs and then check constrains are met and there are no violations and we choose the effort to be medium, after this we generate the *sdc* file and all reports (area, power , resources and timing analysis of worst 10 paths) then we go to the next step PNR.

In Synthesis we input the RTL, constrains and *SC.libs*. Then we get *netlist.v* and *.svf files*. We used DC shell to synthesis our design.

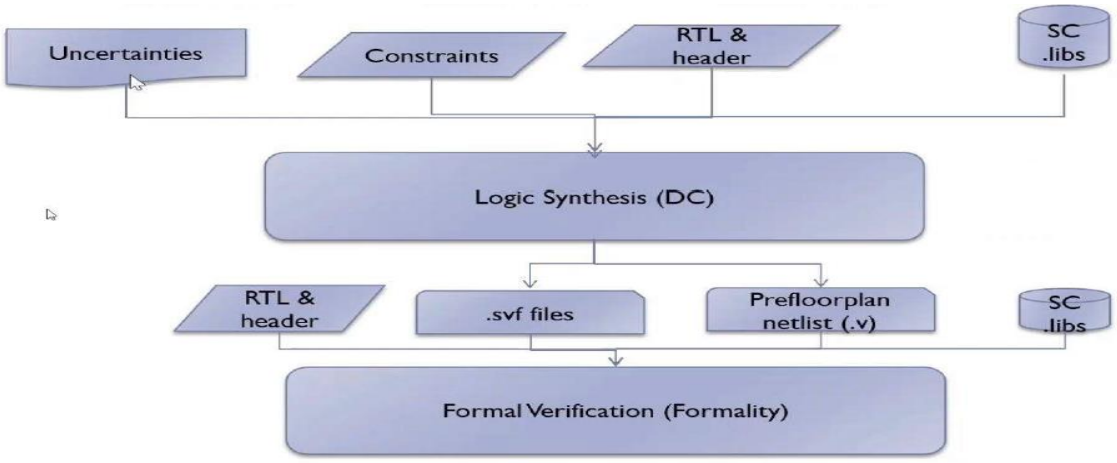

Figure 91:Logic synthesis inputs and outputs

8.3.2.2 PNR

In PNR which is Backend digital design engineer are specialized with the whole magic occurs here, first we identify the top module name and the target and the link library and search path, then we identify the library of routing wires delay model, and we pass the constrain file.

After this we go in the next step in the PNR which is **floorplan** in this step we want to calculate the width and height of the IC chip from the sizes of cells used and we identify a parameter which is the utilization which identify the ratio between the size of the cells to the IC size, after this we go to the next step which is **power network**.

The **power network** step is to identify the metal layer positions vertical or horizontal from layer 6 to layer 10 and here we identify also the VSS source network and how it's connected to the whole cells and also the ground signal and then we define the VSS and GND pins, after floor planning and power network we check for the ignored layers to make sure they wanted and also chick congestion and physical constrains and then go to the next step which is **placement**.

Placement, at first, we start with initial placement for the cells and then the tool checks for legality and timing and estimate them and then do some optimization to get the optimum place for each cell so that the routing becomes easier, according to an algorithm and the effort we give to the tool, it goes into finite loop until it founds the optimum places for each cell and met the placement constrains with low congestion and then go to the next step **CTS**.

CTS, here we want to identify all things that related to the input clock to the IC chip and define the type of tree network and the types of buffer used in the clock tree, we define the clock tree clock, early delay, skew, max capacitance, max fan out, maximum transition and the buffer size, then synthesize, optimize and route the clock tree network, so after that we generate all previous reports in addition to clock tree timing, clock tree summary and timing reports, after that we go to the next step which is **Routing.**

Routing, before routing we define some spare cells and tell the tool not to touch them, after that we start routing at first the tool starts with routing each cell with target cell until

the routing fails and then try again from the previous routing until route each cell with the target cells and then check if there are violation in timing and try to reduce the violation as much as it can, the routing becomes easier if there are several metal layers and low utilization and can be easier also if exert more effort in the placement step.

At the end, we check all constrain are met and generate the last timing report in addition to the parasitic report and get the GDS file to be ready to for fabrication.

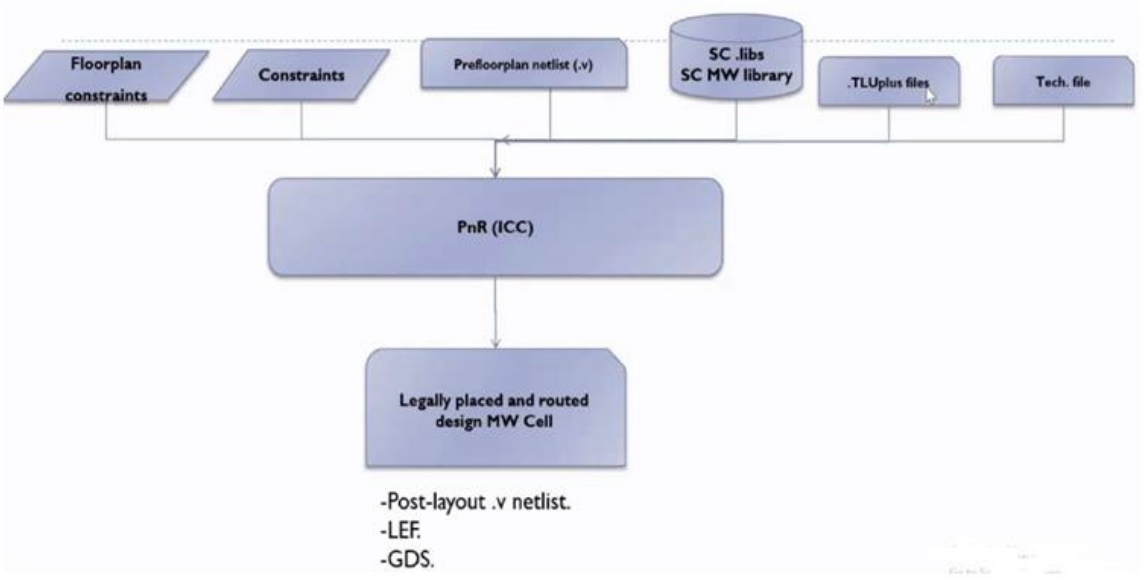

Figure 92: PNR inputs and outputs

In PNR we input the prefloorplan netlist, floorplan constrains, constrains, TLUplus files and SC.libs. Then we get LEF files and .GDS file. We used ICC shell to place and route our design.

8.4 ASIC Results

An average power of 1000 mw for each layer was needed to satisfy the required constraints:

- Clk period: 6ns
- Clock uncertainty: 0.09 ns
- Input delay: 2 ns
- Output delay: 2 ns

P.O.C	Layer 1	Layer 2	Layer3	Layer 4	Layer 5	Layer 6
Power(mw)	1000	1000	1000	1000	1000	1000
Area (um^2)	626157.879	372314.88	50727.529	122876.571	246560.71	101758.8307
Combinational Cell Count	216051	124244	18864	45972	93279	39441
Sequential Cell Count	54047	29599	3883	10786	21054	9395

Table 17: ASIC Results

8.4 Summary

This part introduces the ASIC part for the project and why we choose to follow the ASIC flow and the problems we have faced in ASIC and how we solve them like memories, the method and steps we followed in the ASIC flow.

Chapter 9: Design Results

9.1 Testing Functionality and RTL Verification Methodology

9.1.1 Verification for each layer

First, we developed a python system model with the details of the design and input output memories (BRAMS) as described previously. Then we generated test vector files for each convolution layer. Files containing the inputs and the expected output in each BRAM in the inputs and outputs BRAMS, so we can get the RTL output of each layer alone and compare it with the test vectors to track the source of the errors in the RTL faster.

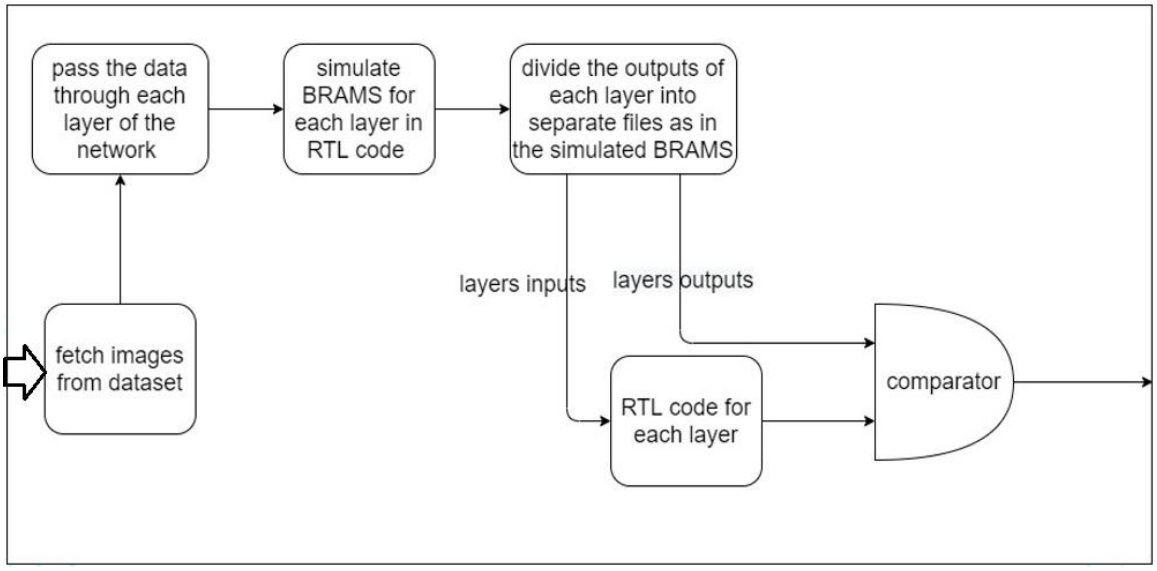

Figure 93: RTL verification methodology

After the input data passes through the RTL layers the output files are generated then taken to be compared with the system model output files of the layer.

9.1.2 Verification of the overall system and results

First, we tested the Fixed-point representation impact on the accuracy using the python software model of the squeeze-Next CNN network where we found the accuracy decreased from 91.45% to 90.8%. This test is a result of 10,000 testing images, where 9080 images of the 10000 are detected right as in the dataset.

Then we Fetched 1040 images from the dataset to test the accuracy of the hardware implemented system. The accuracy was 90.96% for the 1040 images, where 946 images of the 1040 images were detected right as in the dataset. And the 1040 images outputs are the same as the software python CNN model outputs.

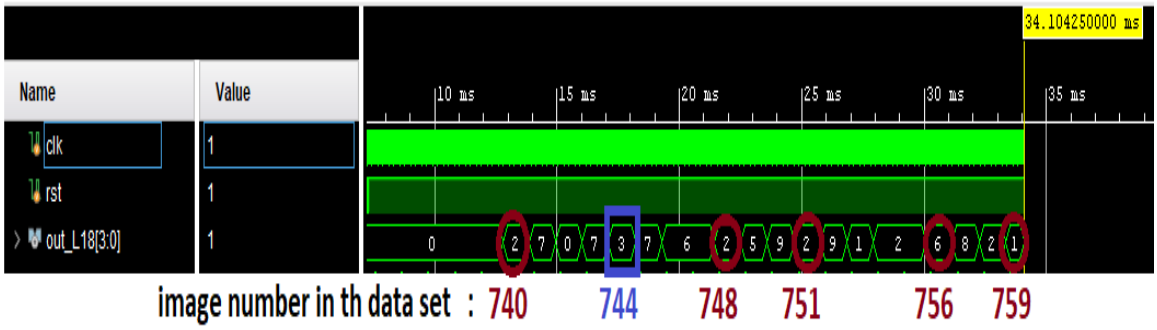

Figure 94: 20-image example for the tested images

image_no	dataset_label	hardware_output
image740(bird)	= 2	2
image741(horse)	= 7	7
image742(airplane)	= 0	0
image743(horse)	= 7	7
image744(truck)	= 9	3(WRONG)
image745(horse)	= 7	7
image746(frog)	= 6	6
image747(frog)	= 6	6
image748(bird)	= 2	2
image749(dog)	= 5	5
image750(truck)	= 9	9
image751(bird)	= 2	2
image752(truck)	= 9	9
image753(automoile)	= 1	1
image754(bird)	= 2	2
image755(bird)	= 2	2
image756(frog)	= 6	6
image757(ship)	= 8	8
image758(bird)	= 2	2
image759(automoile)	= 1	1

Figure 95: 20-image example dataset vs hardware output

Figure 96: image number 741 from the dataset

As shown in the figures, the output of random 20 images from the hardware versus the dataset labels. All the images are detected right except the image number 744 is wrong, and the RTL detect the image number 741 from the dataset correctly which is a horse with size of 32*32 pixels.

9.2 FPGA Results

First we implemented the network at a frequency of 100MHz then we did many optimizations to enhance the design in terms of timing and area and breaking all the critical paths in the design, and that made the design to work at a frequency of 200MHz after synthesis and 150MHz after implementation. Our target is throughput so we sacrificed some area to achieve our target.

The FPGA used is Virtex-7 on VC709 board and all results are calculated using Vivado 2018.3 Software.

We will discuss the design utilization on the target FPGA with the timing information and power dissipation at frequencies of 100MHz, 125MHz, 166.6MHz and 200MHz.

9.2.1 FPGA Results at 100MHz

9.2.1.1 Utilization Results

Resources Utilizations after Synthesis using the Vivado Synthesis tool:

LUT	FF	BRAMs	DSP	IO	BUFG	MMCM
325689	**346037**	**245.50**	**3328**	**7**	**12**	**0**

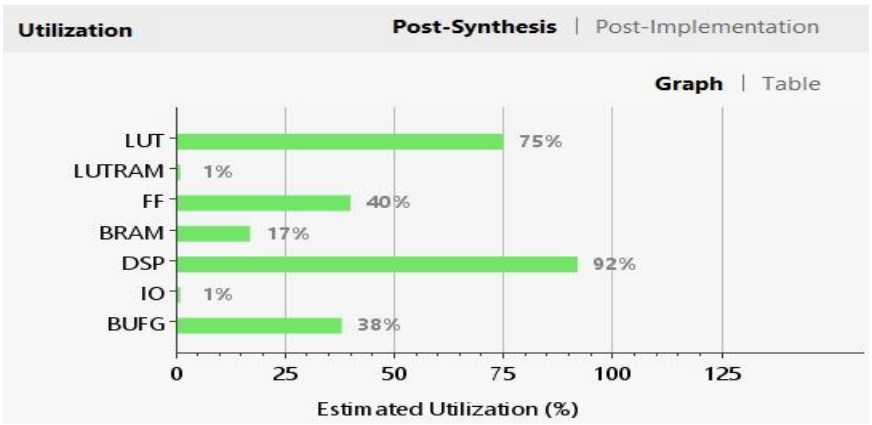

Figure 97: Resources Utilizations after Synthesis

Resources Utilizations after implementation using the Vivado implementation tool:

LUT	FF	BRAMs	DSP	IO	BUFG	MMCM
322508	348224	245.50	3328	7	14	1

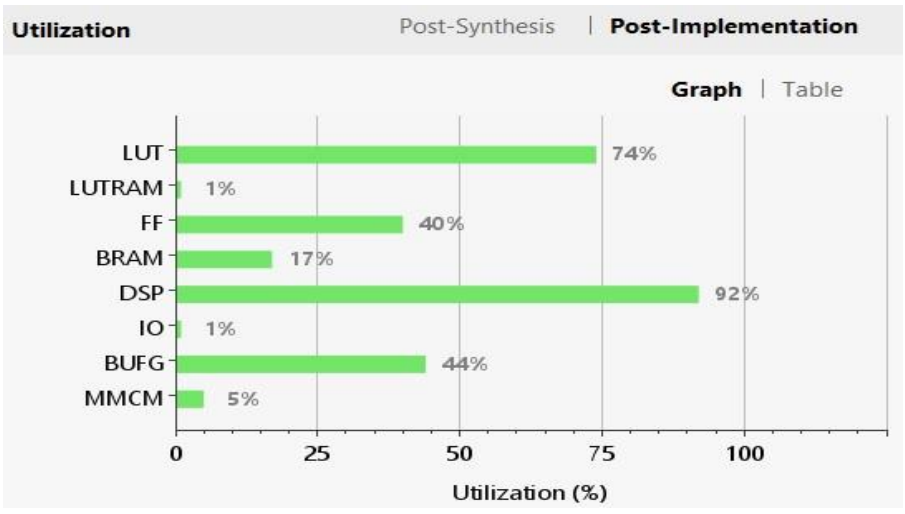

Figure 98: Resources Utilizations after implementation

After implementation the number of BUFGs and FFs has increased but the number of LUTs is reduced due to the optimization strategies we used in the implementation flow.

The difference in resources before and after implementation is shown in the next table:

Resources	After Synthesis		After implementation	
LUT	325689	75%	322508	74%
FF	346037	40%	348224	40%
BRAM	491	17%	491	17%
DSP	3328	92%	3328	92%
BUFG	12	38%	14	44%
MMCM	0	0%	1	5%

Table 18: Resources before and after implementation

9.2.1.2 Timing information

After Synthesis:

All user specified timing constraints are met and there's no hold or setup time violation. The next figure shows the worst 10 paths and there's a positive slack of 4.286 which means we can increase the operating frequency.

Name	Slack ^1	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	4.927	1	1	L10_TO_L18/L...eg_reg[3]/C	out_L18[3]	2.933	2.466	0.466	10.000	clk
Path 2	4.931	1	1	L10_TO_L18/L...eg_reg[2]/C	out_L18[2]	2.929	2.463	0.466	10.000	clk
Path 3	4.953	1	1	L10_TO_L18/L...eg_reg[1]/C	out_L18[1]	2.907	2.440	0.466	10.000	clk
Path 4	4.953	1	1	L10_TO_L18/L...eg_reg[0]/C	out_L18[0]	2.907	2.440	0.466	10.000	clk
Path 5	4.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	10.000	clk
Path 6	4.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	10.000	clk
Path 7	4.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	10.000	clk
Path 8	4.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	10.000	clk
Path 9	4.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	10.000	clk
Path 10	4.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	10.000	clk

Figure 99: worst 10 paths after synthesis

After Implementation:

Still there's no violations in setup or hold time and there's positive slack in both of them, the slack is lower than the one after synthesis as the wiring delay is considered. We can reduce the routing delay by optimizing the routing in implementation steps but this will take much higher run-time.

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.096 ns	Worst Hold Slack (WHS):	0.051 ns	Worst Pulse Width Slack (WPWS):	1.100 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	553160	Total Number of Endpoints:	553160	Total Number of Endpoints:	352501

All user specified timing constraints are met.

Figure 100:Setup and hold timing information after implementation

Also all nets in the design are routed successfully and there's no routing violation.

Implementation	Summary | **Route Status** | Failed Nets
Conflict nets:	0
Unrouted nets:	0
Partially routed nets:	0
Fully routed nets:	550339

Figure 101: Fully routed nets

```
Router Utilization Summary
  Global Vertical Routing Utilization    = 50.1376 %
  Global Horizontal Routing Utilization  = 43.3178 %
```

Figure 102: Router Utilization Summary

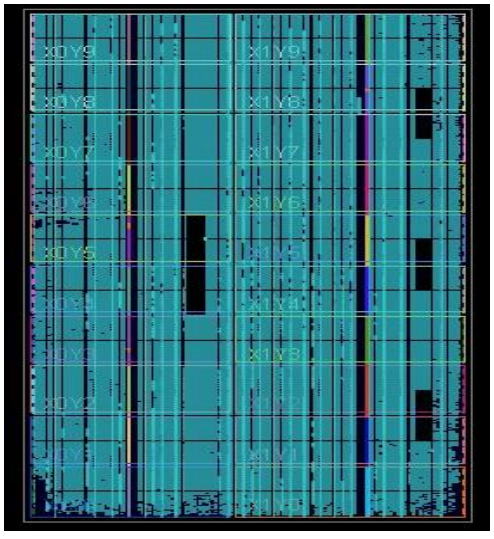

Figure 103: Device implementation

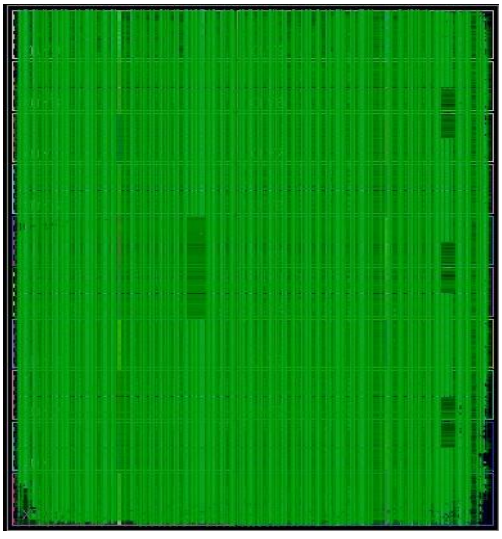

Figure 104: Global Vertical/Horizontal routing utilization

9.2.1.3 Power Analysis

After Synthesis:

The following Figure shows the power consumption of the design after Synthesis.

Total power on chip equals to 15.575 Watt.

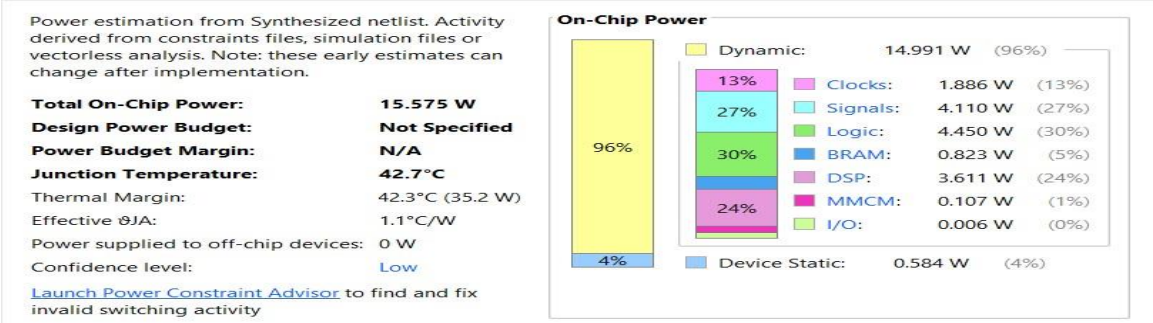

Figure 105: Total power on chip after synthesis

<u>After Implementation:</u>

The following Figure shows the power consumption of the design after Implementation on Virtex 7 FPGA.

Total power on chip equals to 15.96 Watt.

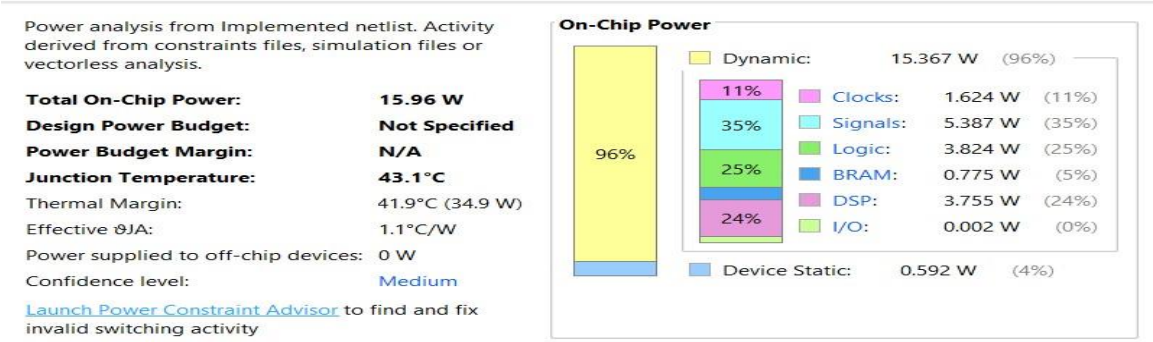

Figure 106: Total power on chip after implementation

9.2.1.4 Throughput Calculations

The following table shows the total number of clock cycles to get the outputs:

Total number of Clock cycles needed to get the first output	127016 Clock Cycles
Total number of Clock cycles needed to get the rest outputs	10805 Clock Cycles

Table 19: The total number of clock cycles to get the outputs

<u>For 100MHz :</u> The Total throughput= $\frac{100*10^6}{10805} = \mathbf{9254.975}\ \boldsymbol{image/second}$ **(FPS)**

Latency =**108.05** $\boldsymbol{\mu}$ ***Second***

9.2.1.4 Efficiency (Energy Per Image)

It's also called power delay product (PDP) or figure of merit. The small number of Power delay product indicates how the design is good.

After Synthesis: Power delay product= 108.05 µ * 15.575 = **1.68** ***m joule***

After Implementation: Power delay product= 108.05 µ * 15.96= **1.72448** ***m joule***

9.2.2 FPGA Results at 125 MHz

9.2.1.2 Timing information

All user specified timing constraints are met and there's no hold or setup time violation. The next figure shows the worst 10 paths and there's a positive slack of 2.927 nsec which means we still can increase the operating frequency.

Name	Slack ^1	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement
Path 1	2.927	1	1	L10_TO_L18/L...eg_reg[3]/C	out_L18[3]	2.933	2.466	0.466	8.000
Path 2	2.931	1	1	L10_TO_L18/L...eg_reg[2]/C	out_L18[2]	2.929	2.463	0.466	8.000
Path 3	2.953	1	1	L10_TO_L18/L...eg_reg[1]/C	out_L18[1]	2.907	2.440	0.466	8.000
Path 4	2.953	1	1	L10_TO_L18/L...eg_reg[0]/C	out_L18[0]	2.907	2.440	0.466	8.000
Path 5	2.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	8.000
Path 6	2.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	8.000
Path 7	2.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	8.000
Path 8	2.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	8.000
Path 9	2.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	8.000
Path 10	2.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	8.000

Figure 107: worst 10 paths

9.2.1.3 Power Analysis

After Synthesis:

The following Figure shows the power consumption of the design after Synthesis.

Total power on chip equals to 18.606 Watt.

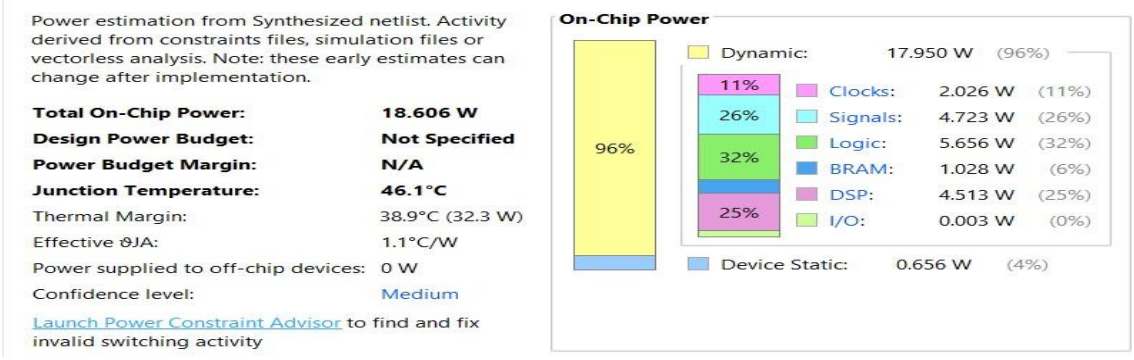

Figure 108: Total power on chip after synthesis

After Implementation:

The following Figure shows the power consumption of the design after Implementation on Virtex 7 FPGA.

Total power on chip equals to 19.899 Watt.

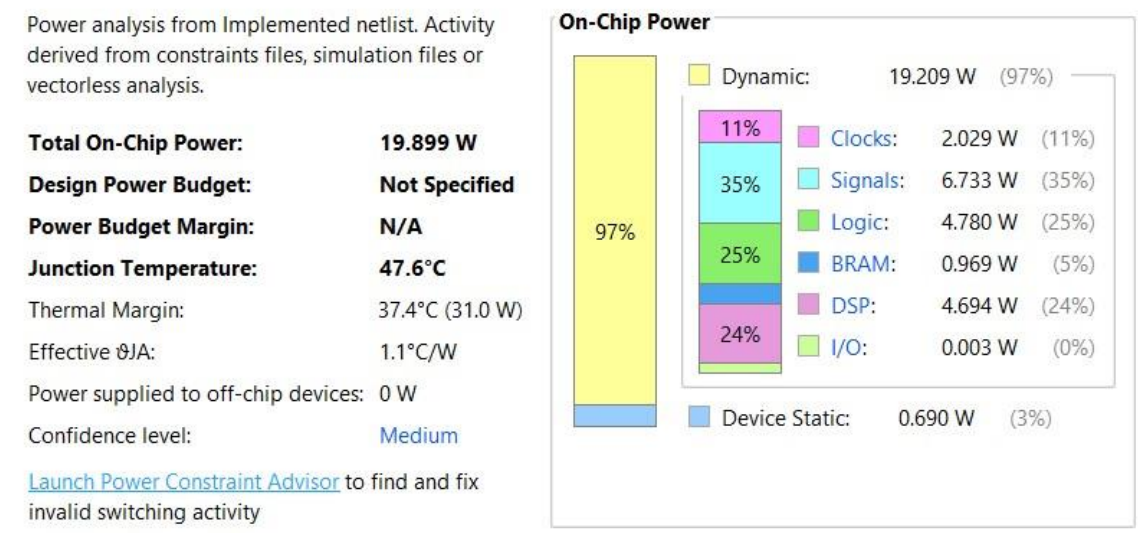

Figure 109: Total power on chip after implementation

9.2.1.4 Throughput Calculations

For 125MHz : The Total throughput= $\frac{125*10^6}{10805} = \mathbf{11568.72}\ \boldsymbol{image/second}$ **(FPS)**

Latency =**89.44** $\boldsymbol{\mu}$ ***Second***

9.2.1.4 Efficiency (Energy Per Image)

It's also called power delay product (PDP) or figure of merit. The small number of Power delay product indicates how the design is good.

After Synthesis: Power delay product= 89.44 µ * 18.606 = **1.664** ***m joule***

After Implementation: Power delay product= 89.44 µ * 19.899= **1.7798** ***m joule***

9.2.2 FPGA Results at 166.6 MHz

9.2.1.2 Timing information

All user specified timing constraints are met and there's no hold or setup time violation. The next figure shows the worst 10 paths and there's a positive slack of 0.927 nsec which means we still can increase the operating frequency.

Name	Slack ^1	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	0.927	1	1	L10_TO_L18/L...eg_reg[3]/C	out_L18[3]	2.933	2.466	0.466	6.000	clk
Path 2	0.931	1	1	L10_TO_L18/L...eg_reg[2]/C	out_L18[2]	2.929	2.463	0.466	6.000	clk
Path 3	0.953	1	1	L10_TO_L18/L...eg_reg[1]/C	out_L18[1]	2.907	2.440	0.466	6.000	clk
Path 4	0.953	1	1	L10_TO_L18/L...eg_reg[0]/C	out_L18[0]	2.907	2.440	0.466	6.000	clk
Path 5	0.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	6.000	clk
Path 6	0.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	6.000	clk
Path 7	0.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	6.000	clk
Path 8	0.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	6.000	clk
Path 9	0.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	6.000	clk
Path 10	0.961	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	6.000	clk

Figure 110: worst 10 paths

9.2.1.3 Power Analysis

<u>After Synthesis:</u>

The following Figure shows the power consumption of the design after Synthesis.

Total power on chip equals to 24.771 Watt.

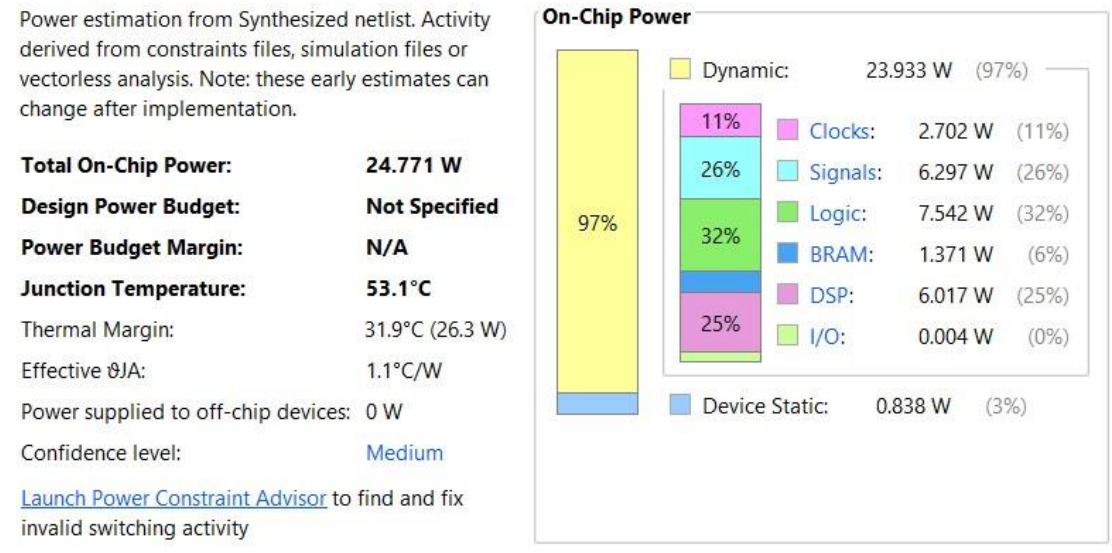

Figure 111: Total power on chip after synthesis

After Implementation:

The following Figure shows the power consumption of the design after Implementation on Virtex 7 FPGA.

Total power on chip equals to 26.513Watt.

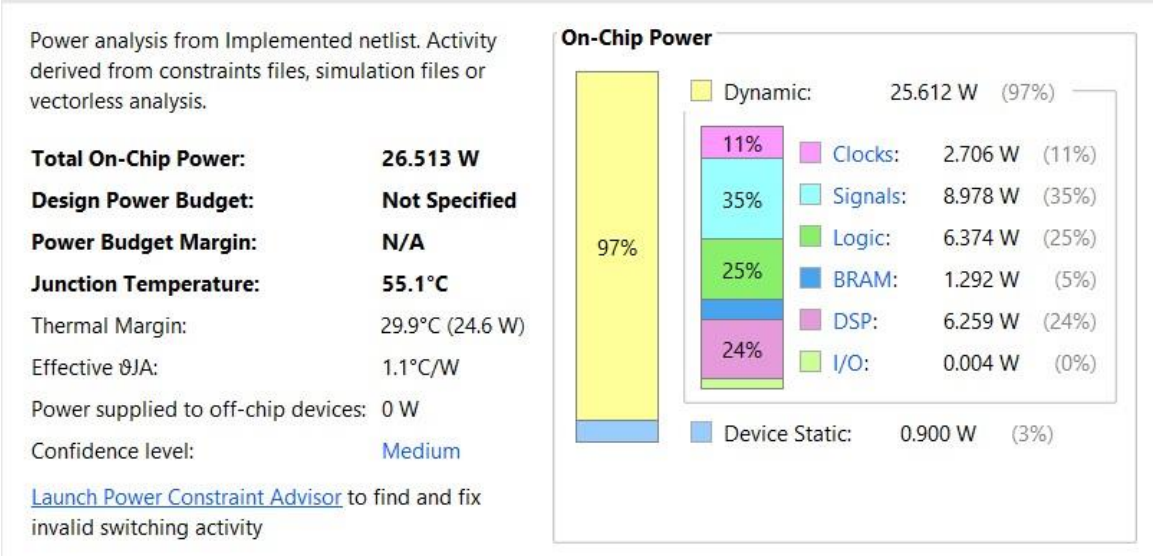

Figure 112: Total power on chip after implementation

9.2.1.4 Throughput Calculations

For 166.6MHz : The Total throughput= $\frac{166.6*10^6}{10805} = \mathbf{15418.788}\ \boldsymbol{image/second}$ **(FPS)**

Latency =**64.8559** $\boldsymbol{\mu}$ ***Second***

9.2.1.4 Efficiency (Energy Per Image)

It's also called power delay product (PDP) or figure of merit. The small number of Power delay product indicates how the design is good.

After Synthesis: Power delay product= 64.8559 µ * 24.771 = **1.60656** ***m joule***

After Implementation: Power delay product= 64.8559 µ * 26.513= **1.7195** ***m joule***

9.2.2 FPGA Results at 200 MHz

9.2.1.2 Timing information

user specified timing constraints are not met and there's setup time violation. The next figure shows the worst 10 paths and there's a negative slack of 0.073 nsec which means we can't work at this operating frequency after implementation.

Name	Slack ^1	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	-0.073	1	1	L10_TO_L18/L...eg_reg[3]/C	out_L18[3]	2.933	2.466	0.466	5.000	clk
Path 2	-0.069	1	1	L10_TO_L18/L...eg_reg[2]/C	out_L18[2]	2.929	2.463	0.466	5.000	clk
Path 3	-0.047	1	1	L10_TO_L18/L...eg_reg[1]/C	out_L18[1]	2.907	2.440	0.466	5.000	clk
Path 4	-0.047	1	1	L10_TO_L18/L...eg_reg[0]/C	out_L18[0]	2.907	2.440	0.466	5.000	clk
Path 5	-0.039	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	5.000	clk
Path 6	-0.039	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	5.000	clk
Path 7	-0.039	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	5.000	clk
Path 8	-0.039	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	5.000	clk
Path 9	-0.039	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	5.000	clk
Path 10	-0.039	7	4	L1_TO_L9/I.../CLKBWRCLK	L10_TO_L18/...g/DIADI[13]	4.134	2.556	1.578	5.000	clk

Figure 113: worst 10 paths

9.2.1.3 Power Analysis

<u>After Synthesis:</u>

The following Figure shows the power consumption of the design after Synthesis.

Total power on chip equals to 29.748 Watt.

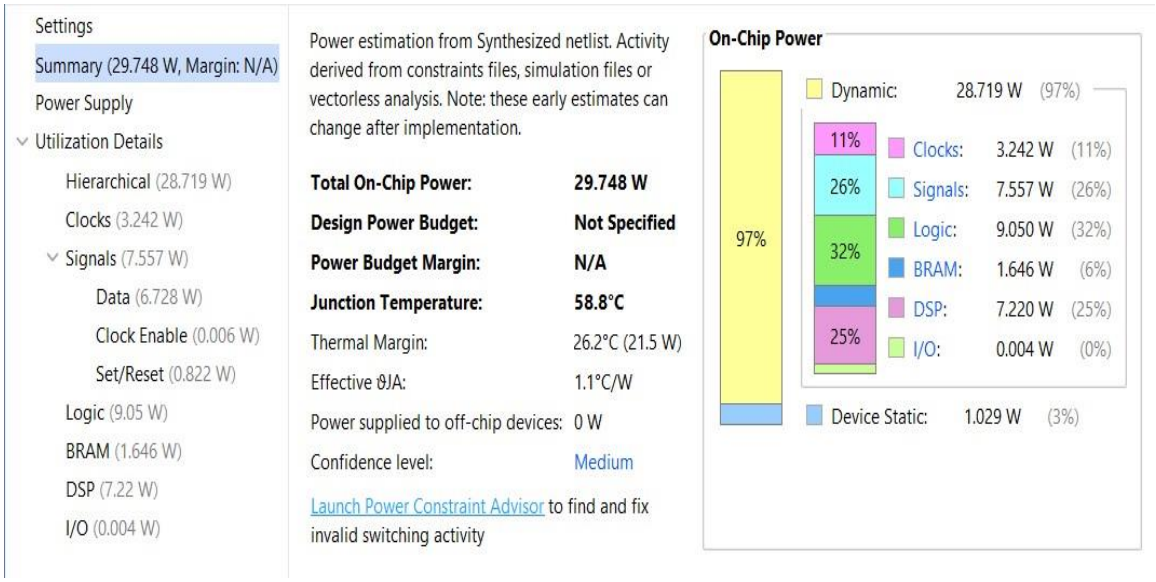

Figure 114: Total power on chip after synthesis

After Implementation:

The following Figure shows the power consumption of the design after Implementation on Virtex 7 FPGA.

Total power on chip equals to 31.858 Watt.

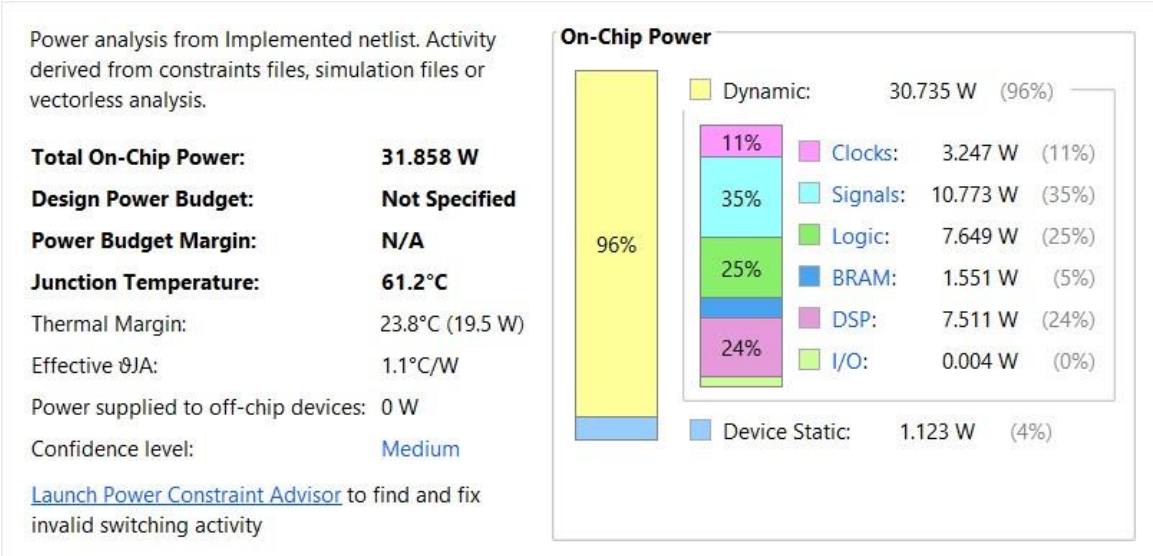

Figure 115: Total power on chip after implementation

9.2.4 Result Comparison At Different Frequencies

We can summarize all the previous results in the next tables:

After synthesis:

	At 100 MHz	At 125 MHz	At 167 MHz	At 200 MHz
Power(watt)	15.575	18.606	24.771	29.748
Time (µ s)	108.05	89.44	64.8559	-
Throughput(image/sec)	9254.975	11568.72	15418.788	-
Energy (mJ)	2.105	1.664	1.60656	-
Utilization BRAMS	17%	17%	17%	17%
Utilization DSPs	92%	92%	92%	92%
Utilization FF	40%	40%	40%	40%
Utilization LUTs	74%	74%	74%	74%

Table 20: FPGA results at different frequencies after synthesis

After implementation:

	At 100 MHz	At 125 MHz	At 167 MHz	At 200 MHz
Power(watt)	15.96	19.899	26.513	31.858
Time (µ s)	108.05	89.44	64.8559	-
Throughput(image/sec)	9254.975	11568.72	15418.788	-
Energy (mJ)	1.68	1.7798	1.7195	-
Utilization BRAMS	17%	17%	17%	17%
Utilization DSPs	92%	92%	92%	92%
Utilization FF	40%	40%	40%	40%
Utilization LUTs	75%	75%	75%	75%

Table 21: FPGA results at different frequencies after implementation

The accuracy is same in all cases.

9.3 Hardware Testing

In the following Figures the test of two random images on the FPGA using the setup shown in Fig 107.

1st image : Testing the image205 in the dataset CIFAR10 with output = 3

```
image203(horse)          =  7
image204(automobile)     =  1
image205(cat)            =  3
image206(airplane)       =  0
image207(dog)            =  5
```

Figure 116: Image205 label in the dataset CIFAR10

Figure 117: Image 205 in the dataset CIFAR10 (cat)

The output on the FPGA GPIO LEDS:

Figure 118: The output on the FPGA LEDs

VIO output:

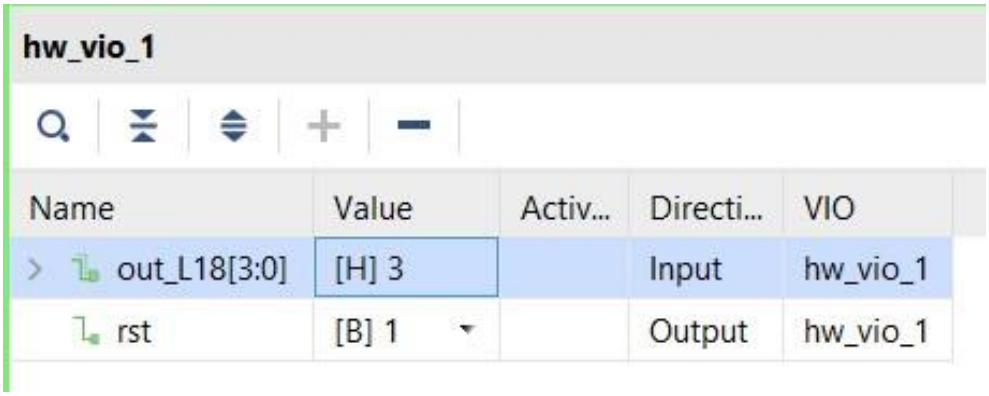

Figure 119: VIO output

The outputs of the FPGA GPIO and the VIO are the same as the dataset output.

2nd image: Testing the image1015 in the dataset CIFAR10 with output = 4

```
image1013(automobile)    =   1
image1014(cat)           =   3
image1015(deer)          =   4
image1016(automobile)    =   1
image1017(frog)          =   6
```

Figure 120: Image1015 label in the dataset CIFAR10

Figure 121: Image 1015 in the dataset CIFAR10 (dear)

VIO output:

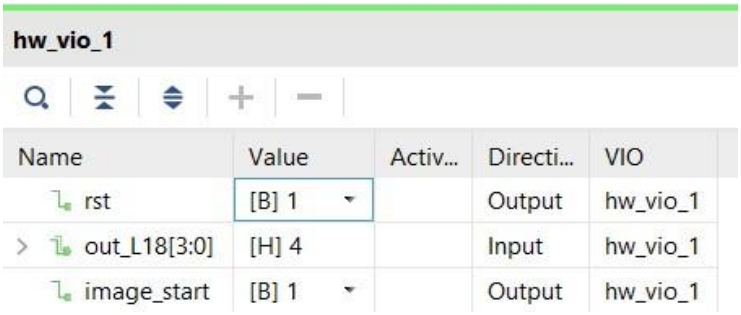

Figure 122: VIO output

ILA Output:

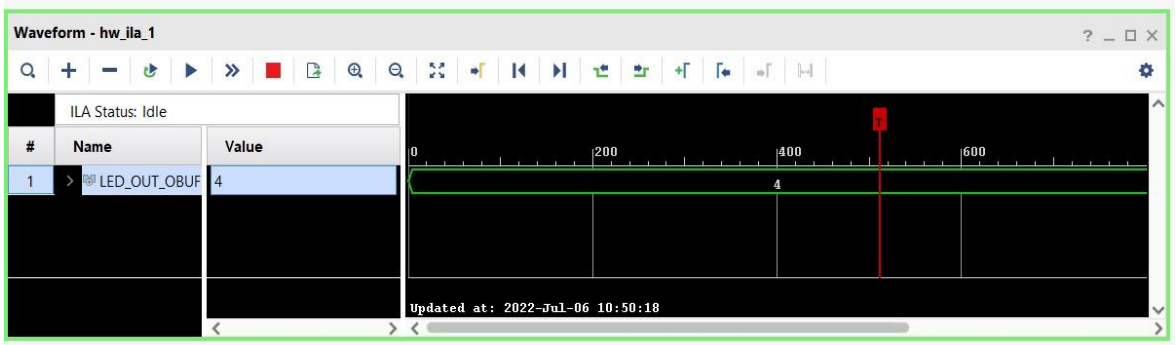

Figure 123: ILA output

The outputs of the ILA and the VIO are the same as the dataset output.

9.4 Other Works

We compared our network implementation (16-bit fixed point representation) with other works that implemented one of CNNs with only 10 classes for a certain application on FPGA, as shown in the next table:

	[33]	[34]	[35]	[36]	This implementation
CNN	PC-BNN	RAd-MobileNet	Squeeze-Net	Zynq-Net	**Squeeze-Next**
FPGA	PYNQ Z1 XC7Z020	Virtex-7 xc7vx980	Virtex-7 VC709	Virtex-7 VC709	**Virtex-7 VC709**
Data Set	Cifar-10	Cifar-10	Image-Net	Image-Net	Cifar-10
No. of Classes	10	10	10	10	**10**
Accuracy	86%	72.61%	66.8%	-	**90.96%**
Frequency (MHz)	143	225	100	125	**100**
Power(watt)	2.4	**3.25**	8.9	11.5	**19.483**
Throughput (image/second)	930	-	248.756	1.5625	**9254.975**
Time (ms)	21.84	-	4.02	640	**0.10805**
Energy (mJ)	2.58	-	35.78	736	**1.68**
Utilization BRAMS	-	-	992 (67.48%)	1413.5 (96.12%)	**245.5 (17%)**
Utilization DSPs	-	937	2658 (73.8%)	3552 (98.67%)	**3328 (92%)**
Utilization FF	-	79,327	159772 (18.44%)	184723 (17.5%)	**348224 (40%)**
Utilization LUTs	23,436	57,438	96990 (22.39%)	338922 (65.3%)	**322508 (75%)**

Table 22: Comparison with different implementations for CNNs on FPGA

As shown in the previous table, our implementation requires the highest power that is equal 19.483 watts, but the latency is the smallest ,the throughput is the highest among the other networks and the energy is the smallest as well.

9.5 Summary

In this chapter we shows the RTL verification methodology and calculated the accuracy for our implementation then calculated the results including utilization, power, timing, throughput and energy using VIVADO 18.3 at different frequencies. Finally we compared our design with other networks which implemented on FPGA with only 10 classes for a certain application and we found that our design has the highest power and area but it's the best in timing, throughput and energy.

Chapter 10: Future Work

10.1 Change Fixed-Point Representation to reduce the Computational Power

We can change the fixed-point representation with lower number of bits so we can reduce the computational power as we found that the most of the dissipated power is due to the multiplication. This may affect the accuracy and this is a tradeoff and we need to make the best strategy to get the best design. But it will be acceptable if the accuracy is decreased 1 or 2% (to become 89% or 88%) with great reduction in the power.

10.2 Using Multiple Clock Domains

We can use clocks with different frequencies in layers to increase the throughput and reduce the latency instead of working with the critical path frequency in the entire network, we can work with critical path frequency in each layers and solving the metastability and synchronization issues.

10.3 Do The Verification Flow To The Network

Performing the verification different techniques to all the network and check if there are any corner cases we missed or any issues, that will make our design more reliable.

10.4 Complete ASIC flow for whole network

We can complete the ASIC flow for whole network to have .GDS file contain our network and doing formal verification for our design.

References

[1] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, Kurt Keutzer “SqueezeNext: Hardware-Aware Neural Network Design”[Submitted on 23 Mar 2018] .[Online].. Available: https://arxiv.org/abs/1803.10615

[2Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size” [Online]. Available: https://arxiv.org/abs/1602.07360

[3] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton “ImageNet Classification with Deep Convolutional Neural Networks” [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[4] Karen Simonyan, Andrew Zisserman “Very Deep Convolutional Networks for Large-Scale Image Recognition” [Online]. Available: https://arxiv.org/abs/1409.1556

[5] Sumit Saha “A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way” [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[6] IBM Cloud Education “Supervised Learning” [released on 19 August 2020] [Online]. Available: https://www.ibm.com/cloud/learn/supervised-learning

[7] A. Khan, A. Zahoora, and A. Qureshi," A Survey of the Recent Architectures of Deep Convolutional Neural Networks", 2019

[8] E. Adel, R. Magdy, S. Mohamed, M. Mamdouh, E. El Mandouh, and H. Mostafa, Bachelor Thesis Report " Accelerating Deep Neural Networks Using FPGA", 2018

[9] "Convolutional neural networks", SlideShare, 2015. [Online]. Available: https://www.slideshare.net/perone/deep-learning-convolutional-neural-networks

[10] Pragati Baheti, “12 Types of Neural Network Activation Functions: How to Choose?”, v7labs. [Online]. Available: https://www.v7labs.com/blog/neural-networks-activation-functions

[11] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich “Going Deeper with Convolutions” [Submitted on 17 Sep 2014] [Online]. Available: https://arxiv.org/abs/1409.4842v1

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun “Deep Residual Learning for Image Recognition” [Submitted on 10 Dec 2015]. [Online]. Available: https://arxiv.org/abs/1512.03385

[13] Hayden So , “Introduction to Fixed Point Number Representation”, Spring 2006. [Online]. Available: https://inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html

[14] Philipp Gysel, Mohammad Motamedi, Soheil Ghiasi “Hardware-oriented Approximation of Convolutional Neural Networks” [Submitted on 11 Apr 2016]. [Online]. Available: https://arxiv.org/abs/1604.03168

[15] Kriti Kushwaha “Introduction of Floating Point Representation”,2021. [Online]. Available: https://www.geeksforgeeks.org/introduction-of-floating-point-representation/

[16] S. Sombatsiri, et al. "Parallelism-flexible Convolution Core for Sparse Convolutional Neural Networks.", 2018. [Online]. Available: https://www.jstage.jst.go.jp/article/ipsjtsldm/12/0/12_22/_pdf

[17] Daniel Holanda Noronha; Kahlan Gibson; Bahar Salehpour; Steven J.E. Wilton “LeFlow: Automatic Compilation of TensorFlow Machine Learning Applications to FPGAs”,IEEE,2019.

[18] PG338 (v3.2) “Zynq DPU v3.2 Product Guide”, July 7, 2020.

[19] articles and videos “FPGA Basics: Architecture, Applications and Uses”, arrow, 24 Sep 2018. https://www.arrow.com/en/research-and-events/articles/fpga-basics-architecture-applications-and-uses

[20] KOMAL CHAUHAN “ASIC Design Flow in VLSI Engineering Services – A Quick Guide”,2020. [Online]. Available: https://www.einfochips.com/blog/asic-design-flow-in-vlsi-engineering-services-a-quick-guide/

[21] Chun Yan & Lau, Francis & Sham, Chiu-Wing. (2018). Fixed-Point Implementation of Convolutional Neural Networks for Image Classification. 105-109. 10.1109/ATC.2018.8587580.

[22] Arjun Thakur,Tutorialspoin , Fixed Point and Floating Point Number Representations, Jun,2020.

[23] Christopher Felton, “A Fixed-Point Introduction by Example” blog, April, 2011.

[24] XILINX,Vivado Design Suite User Guide Synthesis, UG901 (v2019.2), January 27, 2020.

[25] Design Compiler Reference Manual: “Introduction to Register Retiming”, Dec. 2003, Synopsys.

[26] XILINX,Vivado Design Suite User Guide Using Constraints, UG903 (v2019.2), December 12, 2019.

[27] Design Constraints and Optimization,CHAPTER 9.

[28] XILINX,Vivado Design Suite User Guide Implementation, UG904 (v2022.1),May 24, 2022.

[29] Clocking Wizard v6.0 LogiCORE IP Product Guide,E-2012.03

[30] XILINX, Vivado Design Suite, Virtual Input/Output v3.0 , LogiCORE IP Product Guide, April 4, 2018.

[31] XILINX, Vivado Design Suite, Integrated Logic Analyzer v6.2, LogiCORE IP Product Guide, October 5, 2016.

[32] Physical Design Essentials: An ASIC Design Implementation Perspective Softcover reprint of hardcover 1st ed. 2007.

[33] Yang, Li, "Exploring FPGA Implementation for Binarized Neural Network Inference" (2018). Electronic Theses and Dissertations, 2004-2019. 6205. https://stars.library.ucf.edu/etd/6205

[34] Bouguezzi, S.; Fredj, H.B.; Belabed, T.; Valderrama, C.; Faiedh, H.; Souani, C. An Efficient FPGA-Based Convolutional Neural Network for Classification: Ad-MobileNet. Electronics **2021**, 10, 2272. https://doi.org/10.3390/electronics10182272

[35] A. Tarek,A. Mohamed,A. Adel,A. Mohamed, F. Khaled,F. Khaled, E. El Mandouh, and H. Mostafa, Graduation Project thesis ”Accelerating Aware Machine Learning Algorithms Design And Verfication”, August 2020.

[36] A. Gamal,A. Hesham , G. Saied ,M. Ayman, O. Essam, S. Mustafa,E. El Mandouh, Z. Ibrahim , S. Mohamed , H. Mostafa, Graduation Project thesis “Accelerated Deep Neural Networks Using FPGA ZynqNet Architecture” August 2020.

[37] V. Thakkar, S. Tewary and C. Chakraborty, "Batch Normalization in Convolutional Neural Networks — A comparative study with CIFAR-10 data," 2018 Fifth International Conference on Emerging Applications of Information Technology (EAIT), 2018.