Machine Learning Hardware Acceleration for Advanced Driving Assistance Systems (ADAS)

A Graduation Project Report Submitted to the faculty of engineering at Cairo university in Partial fulfilment of The Requirements for the Degree of Bachelor science

In

Electronics and Electrical communications Engineering

By

Mohamed Ayman Mohamed

Mahmoud Ahmed Fouad

Mahmoud Yasser Ibrahim

Mostafa Nasser Zaki

Moataz Mohamed Gharib

Under supervision of

Dr. Hassan Mostafa

Faculty of Engineering, Cairo university

Giza, Egypt

July, 2021

Table of Contents

Chapter 1: Introduction to ADAS 9
1.1 ADAS definition 9
1.2 Why is ADAS important? 9
1.3 ADAS applications 10
1.4 How does ADAS work? 11
1.5 The future of ADAS 12
1.6 ADAS and Machine/Deep learning 13
Chapter 2: Image Classification Models 21
2.1 Neural networks overview 21
2.2 Convolutional neural networks overview 21
2.3 General convolution neural network architecture 21
2.3.1 Convolution layer 22
2.3.2 Relu activation function 22
2.3.3 Pooling layer 22
2.3.4 Fully connected layer 23
2.4 CNN architectures and models for image classification 24
2.4.1 SqueezeNet 2016 24
2.4.2 ResNet model 25
2.4.3 AlexNet Model 28
2.4.4 VGGNet 29
2.4.5 MobileNet 30
Chapter 3: Training 33
3.1Training 33
3.1.1Model 1 34
3.1.2Training of model one on GTS 39
3.1.3Disadvantage in model 1 40
3.2Model 2 and Quantization 41
3.2.1Quantization aware Training 41
3.2.2Second Step: Post training quantization 43
3.2.3How quantization is implemented in hardware 43
3.2.4Model 2 Layer 44
Chapter 4: Hardware Design Methodology 46
4.1 FPGA Introduction and main resources 46

4.2 FPGA 7 series internal components 46
4.2.1 Configure Logic Block (CLB) 46
4.2.2 Configurable I/O blocks 47
4.2.3 Clock Driver 48
4.2.4 DSP Block 48
4.2.5 Block Ram 49
4.3 FPGA digital design flow 49
4.4 MobileNet accelerator design 50
4.4.1 First Design Approach 50
4.4.2 Second Design Approach 51
4.4.3 Shared Layer approach 52
4.5 Shifter Block 53
4.5.1 How multiplication is done in hardware 53
4.5.2 Shift Register with one Controller 54
4.6 Adder Tree Block 59
4.6.1 Look ahead with carry save adder 59
4.6.2 Pipelined adder tree 62
4.6.3 Adder tree overflow issue 62
4.7 Standard Convolution 63
4.8 Depthwise Convolution 65
4.8.1 How Depth wise fetch data from memory? 66
4.8.2 How Depthwise fetch weights from memory? 66
4.9 Pointwise Convolution 67
4.9.1 Pointwise hardware complexity 68
4.9.2 Pointwise hardware Structure 68
4.9.3 Input Fetching 69
4.9.4 Core Block: Pointwise Convolution 69
4.9.5 Weights Fetching 70
4.9.6 PW Output Storage 72
4.9.7 Batch normalization 73
4.9.8 Illustrative example 74
4.10 Average Pooling Layer 77
4.11 Fully connected Layer 78
4.11.1 FC architecture 78
Chapter 5: Controllers and Weight distribution 80

5.1 Main controller.. 80
5.2 Standard convolution and Depthwise controller 81
5.2.1 Controller operation sequences (Data Flow) 82
5.2.2 Controller operation sequences (Weight, Bias and M parameter) 83
5.3 Pointwise controller 83
5.4 Fully connected controller 84
5.5 Weight distribution 84
Chapter 6: Testing methodology and functional simulation result 86
6.1 Testing methodology 86
6.2 Testing Standard convolution layer 87
6.3 Testing Depthwise convolution layer.. 87
6.4 Testing Pointwise layer.. 88
6.5 Testing Pooling and Fully connected Layer.. 89
Chapter 7: Results, Future Work and Conclusion.. 91
7.1 Results... 91
7.2 Power optimization 93
7.3 Benchmark 96
7.4 Future Work 97
7.4.1 Increase throughput by time sharing between photos 98
7.4.2 Make design ready for ASIC flow 98
7.4.3 Experimental Work... 98
7.5 Conclusion 98
References.. 99

List of Figures

Figure 1: Type of information ADAS provide to Driver 9
Figure 2: Autonomous levels 10
Figure 3: ADAS application 11
Figure 4: Main parts of any ADAS 12
Figure 5: vehicle to vehicle communication. 13
Figure 6: Performance Vs Amount of data 14
Figure 7: Object detection 14
Figure 8: Major methods of object detection 15
Figure 9: YOLO architecture 15
Figure 10: Point pillars architecture 17
Figure 11: Lane detection 18
Figure 12: DeepLane architecture 18
Figure 13: VPGNet architecture 19
Figure 14: Semantic segmentation 19
Figure 15: ADAS semantic segmentation image 20
Figure 16: General CNN architecture 21
Figure 17: Convolution Layer 22
Figure 18: Relu function 22
Figure 19: Max Pooling 23
Figure 20: Average pooling 23
Figure 21: Fully connected layer 24
Figure 22: SqueezeNet architecture 24
Figure 23: Squeeze Net fire module 25
Figure 24: Performance Vs iterations 25
Figure 25: Short connection in ResNet 26
Figure 26: ResNet50 model 27
Figure 27: Performance of some ResNet versions 28
Figure 28: AlexNet architecture 29
Figure 29: VGG-16 architecture 30
Figure 30: Depthwise Convolution 31
Figure 31: Standard Convolution 31
Figure 32: Pointwise Convolution 31
Figure 33: MobileNet architecture 32
Figure 34: Signs from GTS 33
Figure 35: Model 1 Layers 38
Figure 36: Mobile Net trained on image net 39
Figure 37: Insert Fake Quant layer during training 41
Figure 38: Finding zeros 42
Figure 39: Models are trained on image net 42
Figure 40: Final mode 2 Layers 45
Figure 41: Configurable logic block 47

Figure 42: Input output configurable block ... 47
Figure 43: Input output configurable block ... 48
Figure 44: Layer pipeline architecture ... 50
Figure 45: Series 7 FPGA resources ... 50
Figure 46: Shared layer architecture ... 51
Figure 47: Operation overlapping between depth and point wise modules ... 51
Figure 48: MobileNet accelerator Block diagram ... 52
Figure 49: Image 6*6 and filter 3*3 multiplication ... 53
Figure 50: Bram in virtex- 7 ... 54
Figure 51: Input Image with required Padding ... 55
Figure 52: Values in Registers ... 55
Figure 53: One instance of module ... 58
Figure 54: 32 instances controlled by one counter and controller ... 59
Figure 55: Carry save block ... 59
Figure 56: generation and propagation truth table ... 59
Figure 57: carry lookahead block diagram ... 60
Figure 58: pipelined carry lookahead adder tree adding 9 numbers ... 62
Figure 59: Standard convolution architecture ... 63
Figure 60:Standard convolution adder tree ... 64
Figure 61: Depthwise core parallelism ... 65
Figure 62: Depthwise block diagram ... 65
Figure 63: Memory weights fetching ... 66
Figure 64: Pointwise convolution ... 67
Figure 65: Pointwise filters ... 67
Figure 66: Pointwise hardware structure ... 68
Figure 67: Input buffer ... 69
Figure 68: Convolution parallelism ... 69
Figure 69: Pointwise sequence of operation ... 70
Figure 70: Weights fetching ... 70
Figure 71: How weights are stored in ROM ... 71
Figure 72: Pointwise output buffer ... 72
Figure 73: MobileNet PW layers ... 73
Figure 74: Batch norm weights storing ... 74
Figure 75: illustrative example ... 75
Figure 76: Kernels are applied ... 75
Figure 77: Combing the 2 parts ... 76
Figure 78: Storing the result in output buffers ... 76
Figure 79: Average Pooling ... 77
Figure 80: Average Pooling architecture ... 77
Figure 81: Fully connected layer architecture ... 78
Figure 82: FC weights ROM ... 79
Figure 83: Main parts of digital system ... 80
Figure 84: FSM of main controller ... 80
Figure 85: Layer controller entity ... 81

Figure 86: State diagram ... 82
Figure 87: PW controller ... 83
Figure 88: Fully connected controller ... 84
Figure 89: Weight distribution flow ... 84
Figure 90: Weight distribution files ... 85
Figure 91: Testing methodology ... 86
Figure 92: File's comparison of pooling layer ... 87
Figure 93 : Depth wise Results ... 87
Figure 94: File comparing and expected errors ... 88
Figure 95 : pooling results ... 89
Figure 96 : Classify input image class from fully connected layer ... 89
Figure 97 : classify another input image ... 90
Figure 98 : input image ... 90
Figure 99: Resources Utilization ... 91
Figure 100: Power Analysis ... 91
Figure 101:Timing summary results ... 92
Figure 102: Enable Bram's 100% ... 94
Figure 103 : Power optimization for Brams ... 94
Figure 104 : Signal power if 16 bit is used ... 95
Figure 105 : Signal power if 8 bit is used ... 95
Figure 106:Time diagram of Multiple photos processing ... 97
Figure 107: FPGA board ... 98

List of tables

Table 1: comparison between YOLO and state of art detectors ... 16
Table 2: Comparison between LIDAR methods ... 17
Table 3: ResNet versions ... 27
Table 4: Tuning lambda ... 40
Table 5: Tunning alpha ... 40
Table 6:Resource per layer type ... 68
Table 7: Stages Results ... 92
Table 8: Timing Results ... 93
Table 9 : Comparison among different paper results and our results ... 96

List of Abbreviations

ADAS	Advanced Driver Assistance System
ASIC	Application-Specific Integrated Circuit
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
Concat	Concatenation
Conv	Convolution
CV	Computer Vision
DNN	Deep Neural Network
DSD	Dense-Sparse-Dense training
DSP	Digital Signal Processing
e.g.,	For Example,
FC	Fully Connected layer
FF	Flip-Flop
FIFO	First in First out
FPGA	Field Programmable Gate Arrays
HDL	Hardware Description Language
i.e.,	In Other Words,
I/O	Input/Output
ILSVRC	ImageNet Large Scale Visual Recognition Challenge
LUT	Look Up Table
MAC	Multiply and accumulate
RAM	Random Access Memory
Relu	Rectified Linear Unit
ResNet	Residual Neural Network
ROM	Read Only Memory
SRL	Shift Register LUT
STA	Static Timing Analysis
TNS	Total Negative Slack
VGGNet	Visual Geometry Group network
VIO	Virtual Input/Output
vs.	Versus
WNS	Worst Negative Slack
etc.	And the rest

Abstract

The use of Machine Learning (ML) and Artificial intelligence (AI) is a pillar in ADAS (Advanced Driver Assistance Systems) and self-learning cars. The ML and AI are used in several applications such as images, classifications, and traffic signs detection. One of the main challenges to the AI and ML models is the need for computation resources such as (GPUs) which usually are power-hungry to reduce the required real-time and high accuracy performance metrics.

The target in this project is to implement a CNN hardware Accelerator on an FPGA specified for ADAS applications, and focusing on accelerating the design and achieving high speed. MobileNet is the chosen architecture as it is the most suitable network for ADAS applications due to its simplicity, its high accuracy and its low number of parameters which is much less than other CNN architectures, which makes it possible to be implemented on FPGAs.

In this work, the maxima frame is 4975 fps, the power consumption of the system is 8.118W, the energy per image is 0.0017 J/image while the design is run on the Virtex-7 -VC-709 platform with a 100 MHz work clock frequency

Chapter 1: Introduction to ADAS

A huge improvement is noticed in the domain of automotive safety nowadays to minimize the number of car accidents. We can easily see in our cars an example of a system that helps us while driving to avoid collisions like park systems, cameras to see blind spots, collision warning systems …………etc.

In this chapter, ADAS definition is discussed with some examples, and the relation between ADAS and machine learning is clarified.[1]

1.1 ADAS definition

Advanced Driver Assistance Systems (ADAS) are smart systems that help the car driver with his driving activities to improve the safety level by providing the driver with accurate and sufficient information about the surrounding area of the vehicle such as traffic sign, location of other vehicles, pedestrians locations, Street Lines …………. etc.

As shown in figure (1), according to this information, the car has a real time automated system that take a correct action in a correct sharp time on the vehicle to avoid accidents like braking or steering.[1]

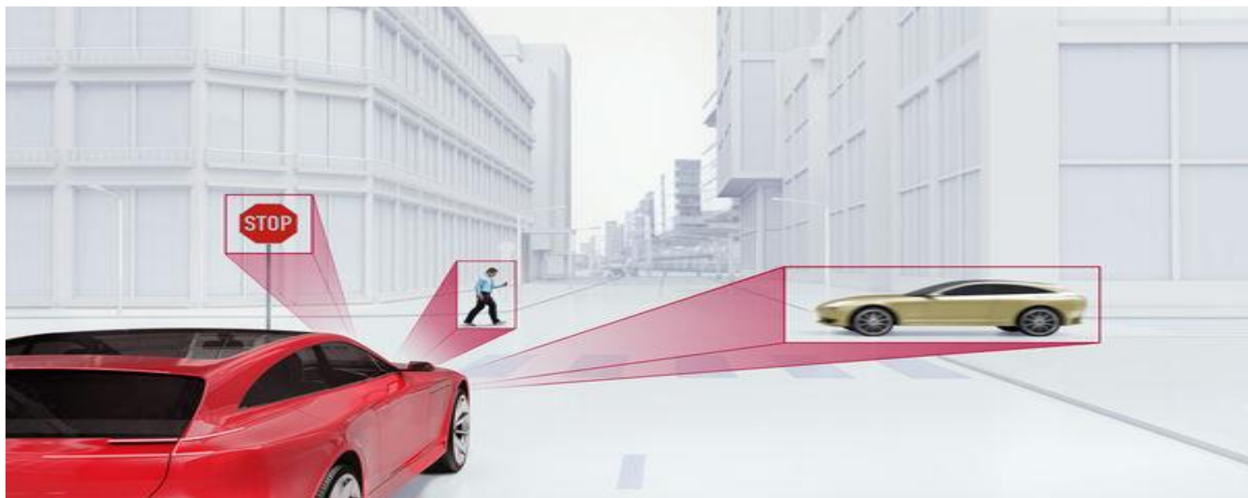

Figure 1: Type of information ADAS provide to Driver

1.2 Why is ADAS important?

ADAS increases the safety on the roads by warning the drivers to prevent accidents or traffic rules violence. So ADAS is very important systems to save lives. According to the statistics by the National Highway Traffic Safety Administration (NHTSA). “The Nation lost 35,092 people in crashes on U.S. roadways during 2015.” This 7.2% increase was “the largest percentage increase in nearly 50 years.” About 94% of those accidents were caused by human error.

ADAS provides many applications such as pedestrian and vehicles detection, Lane assistance, automatic parking, surround view, and driver drowsiness detection that assist the driver, reduce accidents and save lives.[1]

1.3 ADAS applications

As shown in figure (2), The autonomous engineers classified the automation into six levels. Starting from level zero there is no automation at all until we reach level five. In level five we have full automation that doesn't need any assistance from the human.

ADAS is classified as level two in these levels, it just monitors information to the driver and warns him if there's a dangerous situation. It may take some real time actions on the car in very dangerous situations.

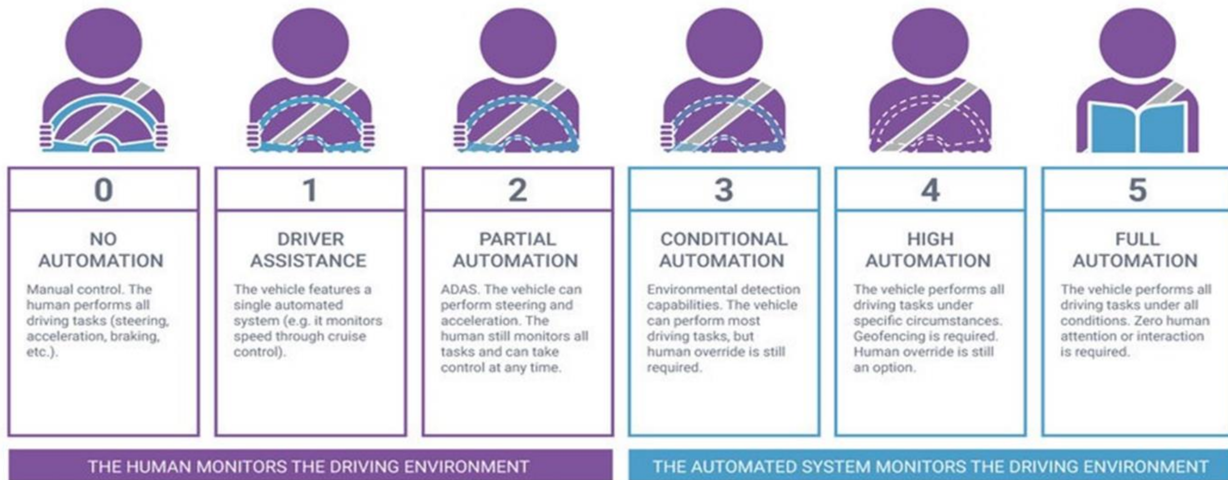

Figure 2: Autonomous levels

Depending on the previous classification we can talk about different ADAS applications, as shown in figure (3):

1) Blind spot monitoring:

Blind spot detection systems use sensors like cameras to provide the driver with important information that is very difficult or impossible to obtain like the blind point that the normal mirrors can't get. Some systems sound an alarm when they detect an object in the driver's blind spot, especially when the driver wants to move into the next lane.

2) Automatic Emergency Braking:

Automatic emergency braking system uses sensors to detect whether the driver is going to hit another object on the road. This application can measure the distance of nearby objects and alert the driver to any danger. Some emergency braking systems can take preventive safety measures, such as tightening seat belts, reducing speed, and adaptive steering to avoid a collision.

3) Lane departure warning system:

Lane departure warning system warns the driver if the car begins to move out of its lane unless turn signals are on this direction. This application can use the data from camera and detect the two lanes that the car should stay within them so if there's a significant shift in the lane in the image it warns the driver.

4) Traffic sign detection and classification:

This system uses the data of the cameras to detect and classify the different traffic signs across the road. The system may just monitor the information to the driver to warn him if any traffic rule is violated. We use the data coming from the camera then detect the location of the traffic sign then classify it.

*In part 2 of this chapter, we will talk about each application and how to implement it using machine/deep learning models.

*The application of Traffic sign detection and classification using camera is our mainly consideration in this project and we will focus on it in the next chapters.[1]

Figure 3: ADAS application

1.4 How does ADAS work?

As shown in figure (4), ADAS in cars depend on three parts:

1) <u>Sensors:</u> they gather information on their immediate environment, such as pedestrians, cars, traffic signs…...etc. they are different sensors can be used like camera that provide the system with 2D image that it can extract different information from it like the type of traffic sign or the existing of cars or people in the image.

We have also lidar that throws laser light at an object on the earth surface and calculate the time it takes to return to the LiDAR source so it can calculate the object's distance from the car. The main advantage of lidar over the camera that it overcomes the lighting conditions problem that affects badly the image taken by the camera.

We have also ultrasonic that send sonic waves and calculate the object's distance from the car like the LiDAR sensor by in shorter range so ADAS depend mostly on it in the parking system.

ADAS can use combined data from different sensors and use them as input to the processor to increase the performance and the accuracy.

2) Processor: it uses a combined data from the sensors to understand it and construct a full understanding of the surrounding area. After processing the data, it may just show the information to the driver to inform him of different traffic signs on the road for example.

The processor also can control some actuators in the car if needed to avoid collision to improve the safety level.

3) Actuators: they are anything that control the car movement like brakes, steering, throttle……… etc.

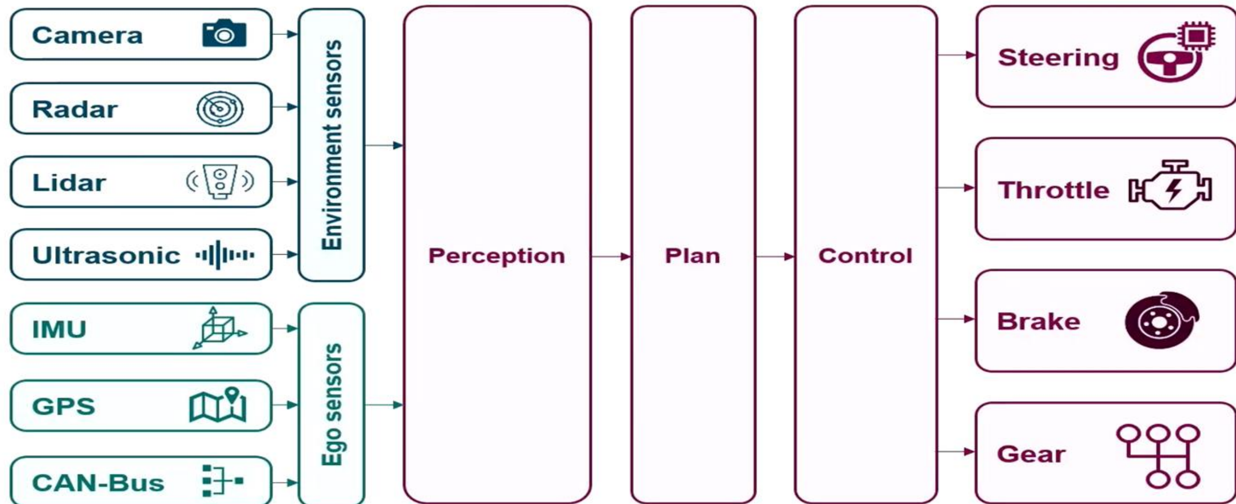

Figure 4: Main parts of any ADAS

1.5 The future of ADAS

ADAS is advancing at a very fast rate. According to Speaking in 2016 chairman and CEO of General Motors, Mary Barras wrote: "The auto industry will change more in the next 5 to 10 years than it has in the past 50."

As Barras said, the next step-change for the advancement of ADAS technologies will be the advent of the so-called connected car, made possible by the widespread adoption of:

- vehicle-to-vehicle (V2V).
- vehicle-to-infrastructure (V2I).
- vehicle-to-everything (V2X) communications.

Current ADAS functions are restricted by what the sensors can detect, which today extends to a useful forward range of around 250 meters.

As shown in figure (5), V2V communication allows vehicles to communicate with each other directly and exchange information, such as positions, relative speeds, directions and even control inputs, like sudden braking, accelerations or directions changes.

As an extension of V2V, V2I provides vehicles with information from the road network's infrastructure, such as traffic lights and signals, variable speed limits and congestion information.

Such information is expected to not only improve safety but also reduce congestion by enabling a freer flow of traffic, and it is also recognized as a key driver towards full autonomy.

V2X, meanwhile, adds data streams from beyond the immediate road network, including cloud-stored information, meteorological updates and possibly cyclists, pedestrians and other vulnerable road users (VRUs).[2]

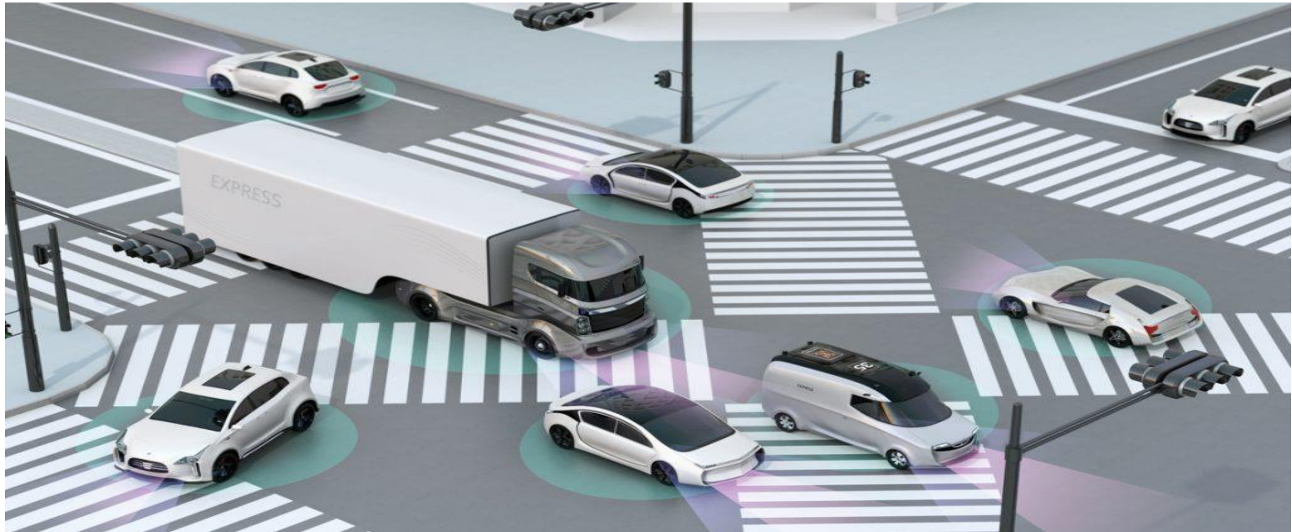

Figure 5: vehicle to vehicle communication.

Moving toward fully autonomous cars—vehicles capable of sensing their environment and operating without human involvement—adds complexity in the electronic architecture of these vehicles.

These complex architectures require an increase in the volume of data. To manage this data, the new integrated domain controllers require higher computing performance, lower power consumption, and smaller packaging.

The adoption of 64-bit processors, neural networks and AI accelerators to handle the high volume of data requires the latest semiconductor features, semiconductor process technologies, and interconnecting technologies to support ADAS capabilities.[1]

1.6 ADAS and Machine/Deep learning

In this part, the different functions that can be implemented using deep learning will be discussed and we will relate them with ADAS applications.

The deep learning is a subset of machine learning that has a breakthrough nowadays. This was due to the huge amount of data that becomes available so, we need to increase the performance of the learning algorithm using this huge data, as shown in figure (6).

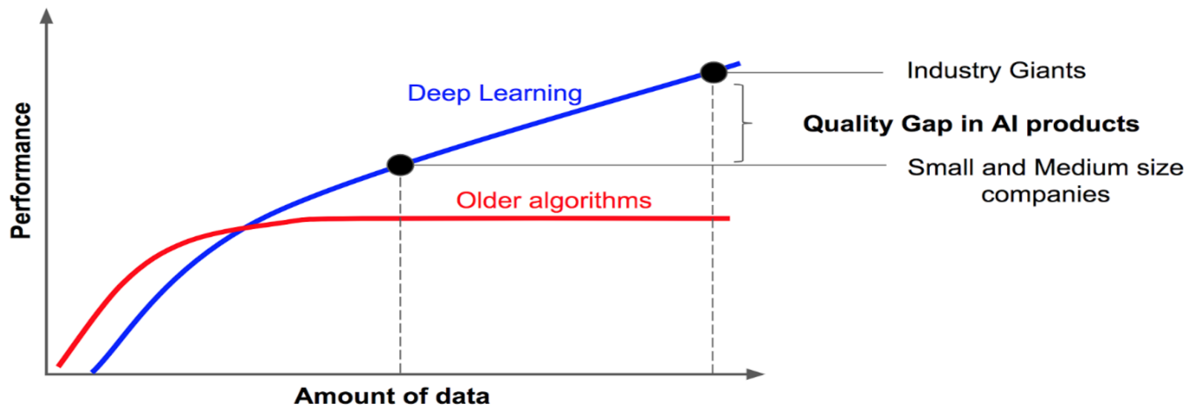

Figure 6: Performance Vs Amount of data

1)Detection

Object detection is a computer vision technique for locating instances of objects in images or videos.

As shown in figure (7), The detection of objects like vehicles, pedestrians, cyclists, animals………...etc. is crucial in advanced driver assistance systems (ADAS) to avoid collisions and accidents.

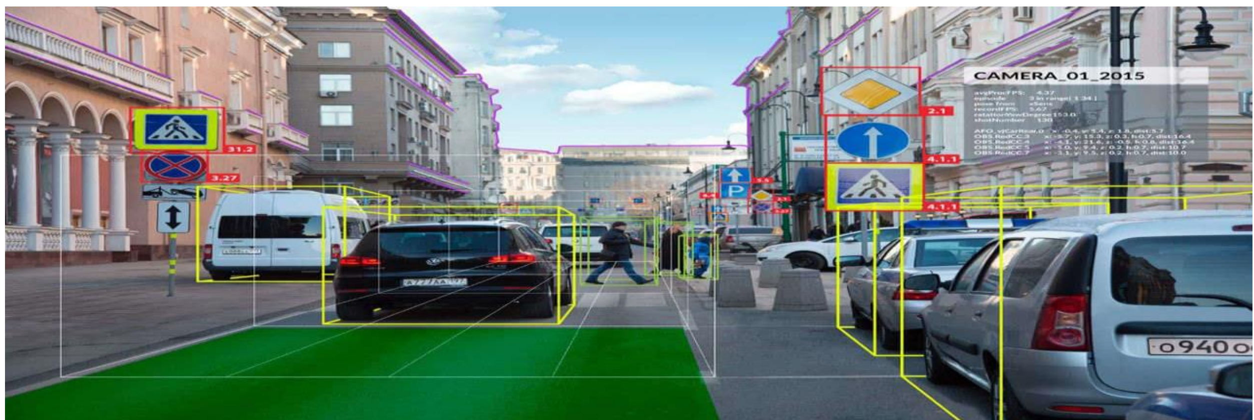

Figure 7: Object detection

Deep Learning for object Detection:

The Data comes from Cameras, Lidars (Light Detection and Ranging) and radars.

We can categorize the deep-learning detectors into two categories: two-stage and one stage methods. Two stages such as R-CNN, Fast R-CNN, etc. that generate the regions by CNN then classify them. one stage like YOLO, SSD, etc. which directly get the probability and position without generating regions stage.[3]

The major methods of object detection are shown in figure (8).

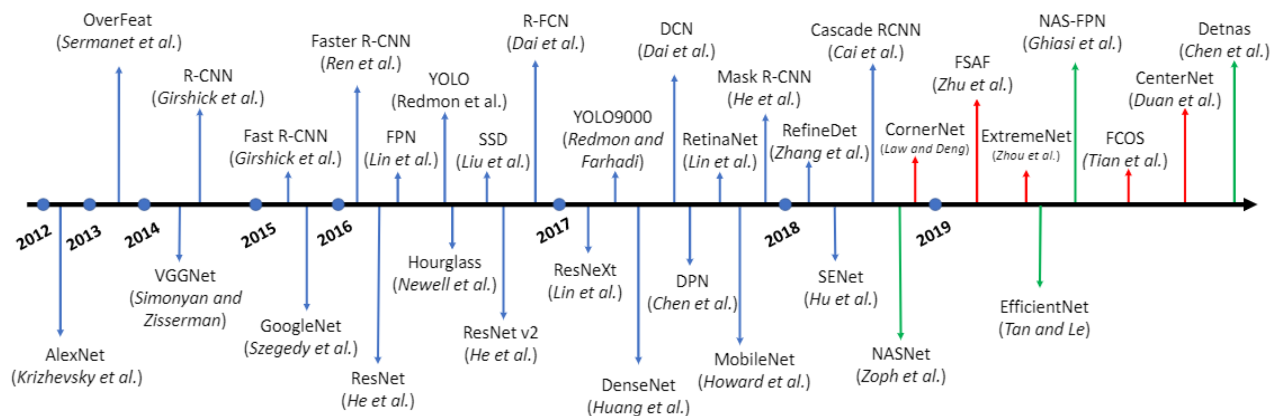

Figure 8: Major methods of object detection

YOLO

You Only Look Once is a state-of-the-art, real-time object detection system. It was produced in 2015.

YOLO has its own neat architecture based on CNN and anchor boxes and is proven to be an on-the-go object detection method for many problems. YOLO has 3 versions: YOLO V1, YOLO V2, YOLO V3

YOLO V2 is more accurate and faster than V1. YOLO V3 more accurate but not faster than V2. [4]

YOLO network architecture is shown in figure (9):

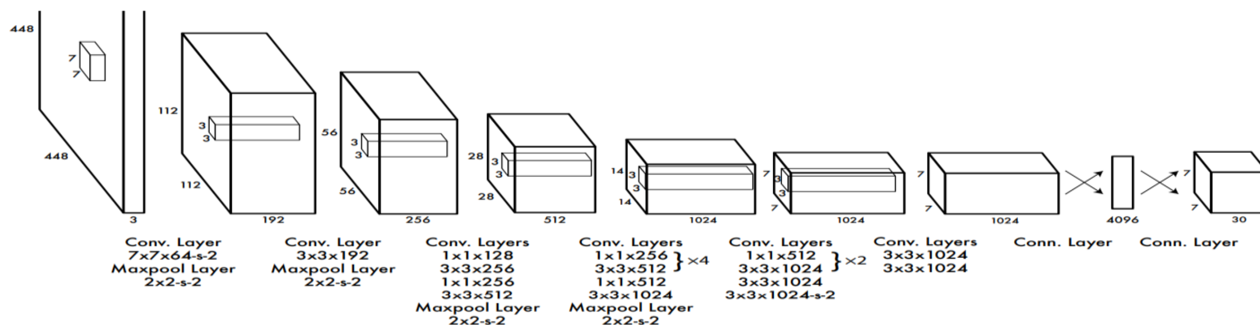

Figure 9: YOLO architecture

It consists of 24 convolutional layers followed by 2 fully connected layers. The model takes the input image and divide it into a grid of 13 by 13 cells: each cell is predicting 5 bounding boxes. A bounding box describes the rectangle that encloses an object. YOLO also obtains a confidence score that gives us information about how certain it is that the predicted bounding box actually encloses some object.

The following table show a comparison (Results on PASCAL VOC 2007 test set) between state of art detectors and YOLO [4]:

Table 1: comparison between YOLO and state of art detectors

Fast R-CNN	2007+2012	70.0	0.5
Faster R-CNN VGG-16	2007+2012	73.2	7
Faster R-CNN ResNet	2007+2012	76.4	5
YOLO	2007+2012	63.4	45
SSD300	2007+2012	74.3	46
SSD500	2007+2012	76.8	19
YOLOv2 288 × 288	2007+2012	69.0	91
YOLOv2 352 × 352	2007+2012	73.7	81
YOLOv2 416 × 416	2007+2012	76.8	67
YOLOv2 480 × 480	2007+2012	77.8	59
YOLOv2 544 × 544	2007+2012	**78.6**	40

LiDARs

LiDARs have the advantage of working in nightlight and give an accurate 3D mapping of the environment. We can group the neural networks for LiDAR data processing into two categories: first category is 2D methods in which the point cloud is projected onto one or more planes, which are then processed by typical convolutional networks. the second is :3D methods – the point cloud is processed without reducing the third dimension, the following subdivision can be made:

1.methods operating on points – these methods perform semantic segmentation or classify the entire cloud as an object – an exemplary method is Point- Nets.

2.methods operating on cells – these methods divide the three-dimensional space into cells (fixed size), aggregate the features of particular points into a features vector for a given cell and process the matrix of cells with 2D or 3D convolutional networks – examples are VoxelNet and PointPillars.

3.hybrid methods – methods partly using both of the above-described approaches – an example is PVRCNN.

We will focus on one of these methods:

PointPillars

PointPillars takes the point cloud (which is a collection of hundreds of millions, or sometimes billions of highly accurate 3-dimensional x,y,z points and component attributes.) as input from LiDAR and generates oriented cuboids denoting the detected objects: Pedestrians, cars, and

cyclists. A “pillar” is a three-dimensional cell, without a user-defined height. The network architecture is shown in the following figure:

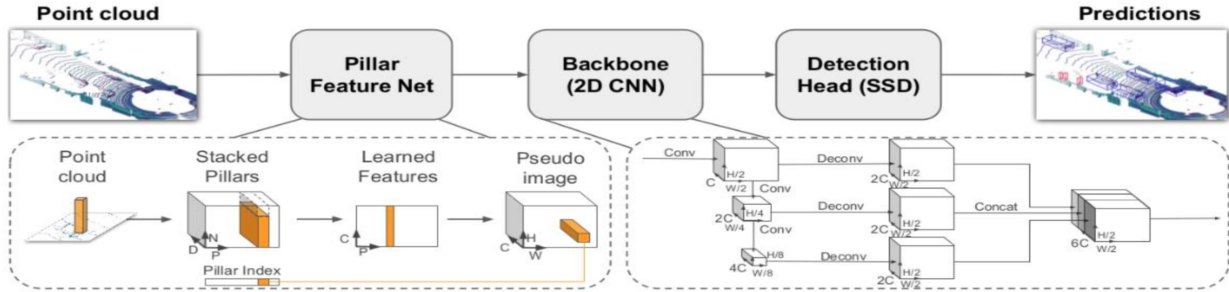

Figure 10: Point pillars architecture.

It consists of three main parts: The first part – Pillar Feature Net (PFN) –whose task is to convert the point cloud into a sparse “pseudo-image”. The second part of the network – Backbone (2D CNN) – processes the “pseudo-image” and extracts high-level features. The last part of the network is the Detection Head (SSD), that detects and regresses the 3D cuboids on the objects.[5]

The following table shows comparison between different methods that takes data from LiDAR (Results on the KITTI test 3D detection benchmark)

Table 2: Comparison between LIDAR methods

Method	Modality	Speed (Hz)	mAP	Car			Pedestrian			Cyclist		
			Mod.	Easy	Mod.	Hard	Easy	Mod.	Hard	Easy	Mod.	Hard
MV3D	Lidar & Img.	2.8	N/A	71.09	62.35	55.12	N/A	N/A	N/A	N/A	N/A	N/A
Cont-Fuse	Lidar & Img.	16.7	N/A	82.54	66.22	64.04	N/A	N/A	N/A	N/A	N/A	N/A
Roarnet	Lidar & Img.	10	N/A	**83.71**	73.04	59.16	N/A	N/A	N/A	N/A	N/A	N/A
AVOD-FPN	Lidar & Img.	10	55.62	81.94	71.88	66.38	50.80	42.81	40.88	64.00	52.18	46.61
F-PointNet	Lidar & Img.	5.9	57.35	81.20	70.39	62.19	51.21	**44.89**	40.23	71.96	56.77	50.39
VoxelNet	Lidar	4.4	49.05	77.47	65.11	57.73	39.48	33.69	31.5	61.22	48.36	44.37
SECOND	Lidar	20	56.69	83.13	73.66	66.20	51.07	42.56	37.29	70.51	53.85	46.90
PointPillars	Lidar	**62**	**59.20**	79.05	**74.99**	**68.30**	**52.08**	43.53	**41.49**	**75.78**	**59.07**	**52.92**

2) Lane Detection:

It is not easy for drivers to find the Lane lines on the road in case of heavy rain fall or there is a snow covering the ground, ADAS helps the drivers by proving Lane detection function. lane detection is a critical component of ADAS. If lane positions are detected, the car will know where to go and avoid the risk of running into other lanes or getting off the road. This can prevent the driver/car system from drifting off the driving lane.[3]

Figure 11: Lane detection

Deep Learning for Lane Detection:

Data also can come from cameras or LiDARs. Lane detection methods can be grouped into two main categories: two-step and one-step methods. One step method gets the detection and gathering results directly from the input image. On the other hand, two-step methods consist of two main steps: feature extracting step and post-processing step.[3]

We will focus of two examples of the networks proposed for Lane detection:

DeepLane is a method based on the idea of classification-based Lane detection method, which combines some prior information to determine lane position. The overall architecture is shown in figure (13).

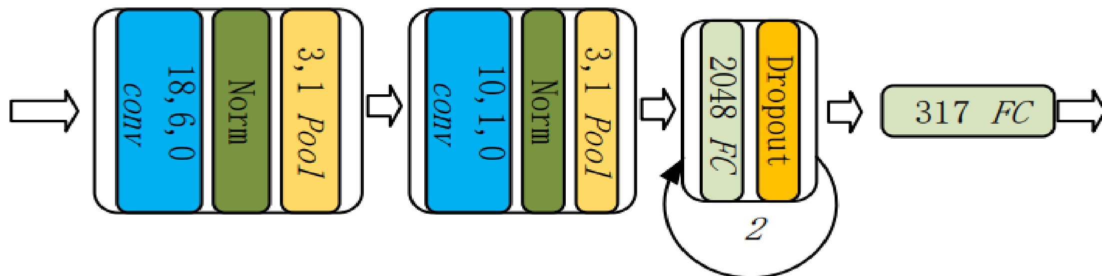

Figure 12: DeepLane architecture

A SoftMax layer is applied to obtain the probability distribution of lane position. The fully connected layer consists of 317 outputs (316 for possible positions and one for the absence of lane marker).[3]

VPGNet (Vanishing Point Guided Network) was proposed by Seokju Lee et al based on the idea of object detection-based lane detection method. It is another method to estimate geometric characteristics by CNN.VPG performs four tasks: grid regression, object detection, multi-label classification, and vanishing point.[3]

The architecture is shown in the following figure:

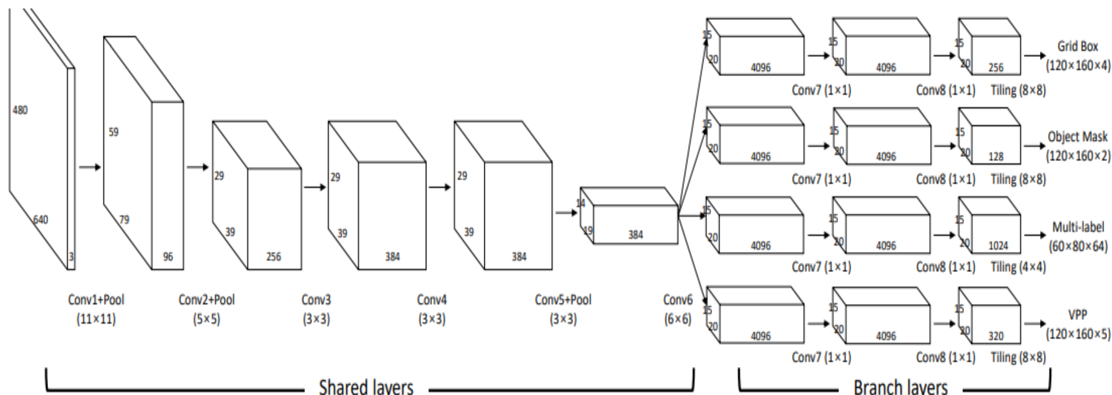

Figure 13: VPGNet architecture

3) semantic segmentation

Semantic Segmentation is the process of assigning a label to every pixel in the image. This is in stark contrast to classification, where a single label is assigned to the entire picture. Semantic segmentation treats multiple objects of the same class as a single entity. On the other hand, instance segmentation treats multiple objects of the same class as distinct individual objects (or instances). Typically, instance segmentation is harder than semantic segmentation.

As shown in figure (14), our main goal is to take RGP image and output a segmentation map where each pixel contains a class label represented as an integer.

Figure 14: Semantic segmentation

An approach towards implementing this application is by constructing a neural network architecture for this task is to simply stack a number of convolutional layers (with same padding to preserve dimensions) and output a final segmentation map (all this will be discussed in chapter 2).

Semantic segmentation is a very important task in ADAS applications, we can use it to classify the regions that contain vehicles and humas all around the car without classifying the humans into men and women or classifying the type of the vehicles, as shown in figure (15).

Figure 15: ADAS semantic segmentation image

4) image classification

Image classification is where a computer can analysis an image and identify the class the image falls under (Or a probability of the image being part of a certain class) A class is a label for a certain instance like identify if image is a car, dog, cat, ………. etc.

The image classification can be used in so many applications in ADAS, we will use it to classify the different traffic signs

This Deep learning application can be implemented using different types of Deep learning Models that will be discussed in chapter (2).

Chapter 2: Image Classification Models

In this chapter, we will focus on the back bone of deep learning (CNN “convolutional neural network”). A general CNN architecture will be discussed, then different models of image classification as an application of deep learning.

2.1 Neural networks overview

Artificial neural networks (ANN) are commuting system inspired by biological neural networks that constitutes human brains.

ANN is based on a collection of connected artificial neurons that transfer data from input to output with some calculations to behave as a human. Humans should move with some situations to learn how to behave correctly and with similar situation human act dependent on the previous learning. Unlike humans, deep learning approaches the machine with a large amount of data and after that machine is able to take a decision itself.

Deep learning is subfield of machine learning which use artificial neural networks. Deep learning is the most accurate approach in machine learning fields because no accuracy saturation occurs with the amount of data in the learning phase.

Our approach is based on deep learning architectures specifically convolution neural networks (CNN).

2.2 Convolutional neural networks overview

Convolutional neural network (CNN) is a spatial type of artificial deep neural networks (DNN) which developed to work with high input futures like RGP high quality photos with good performance and small number of parameters compared to any DNN. CNN uses shared weights to extract the input futures (hidden layers). Each hidden layer consists of filters. Automatically in the train phase, each filter able to extract one feature from the input features using convolution process and after convolution layers the output encoding version of the input feed into fully connected normal DNN to do the functionality of the network like detection or recognition. [7]

2.3 General convolution neural network architecture

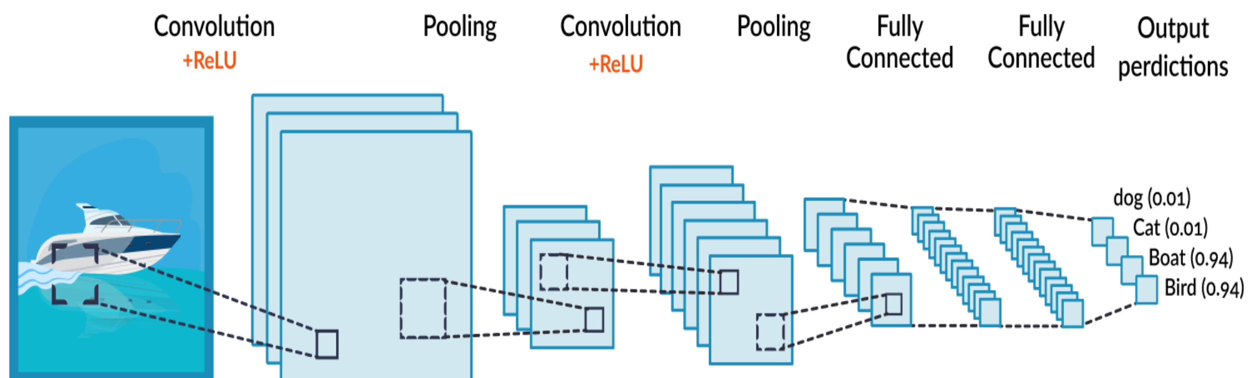

Figure 16: General CNN architecture

2.3.1 Convolution layer

The first layer in any CNN model is convolution layer. As discussed above, convolution layer mission is to generate a small version of input future photo convolution layer simply done by slide down the filter on photo with a constant stride to cover all pixels and multiply the active input features with the filter weights to generate the output as shown in figure (17). [7]

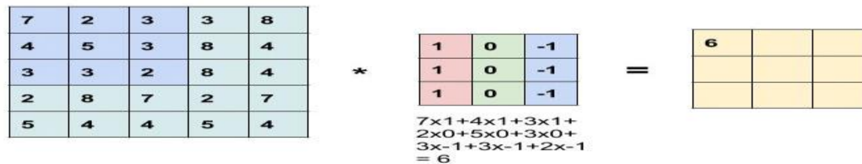

Figure 17: Convolution Layer

2.3.2 Relu activation function

Relu function introduce the nonlinear function. Relu is done after convolution layer to generalize the output in the next layer. There are many non-linear functions can be used, but in CNN Relu is the most used activation function [7].

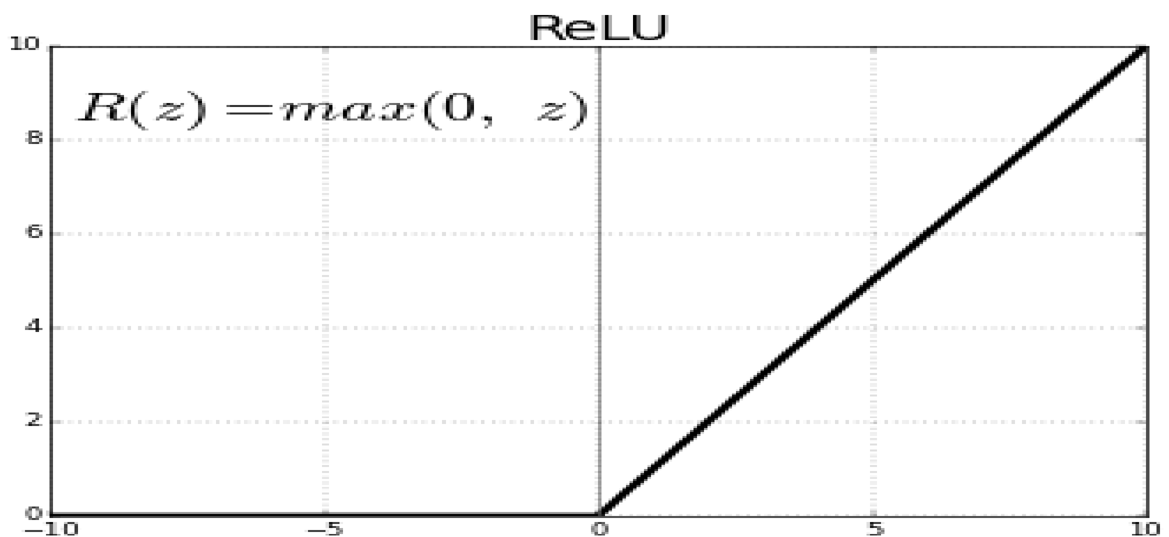

Figure 18: Relu function

2.3.3 Pooling layer

After Relu stage, pooling layer comes next and it is a down sampling layer. The output feature mapping is similar to the input futures but the size of the matrix shrinks. There are 2 main types of pooling which are max polling and average polling [7].

2.3.3.1 Maximum pooling

Max pooling takes the max number between window of pixels and set it as output in the figure (19), the pooling window consists of 4 pixels, so the max of 4 pixels is set as output of these 4 pixels.

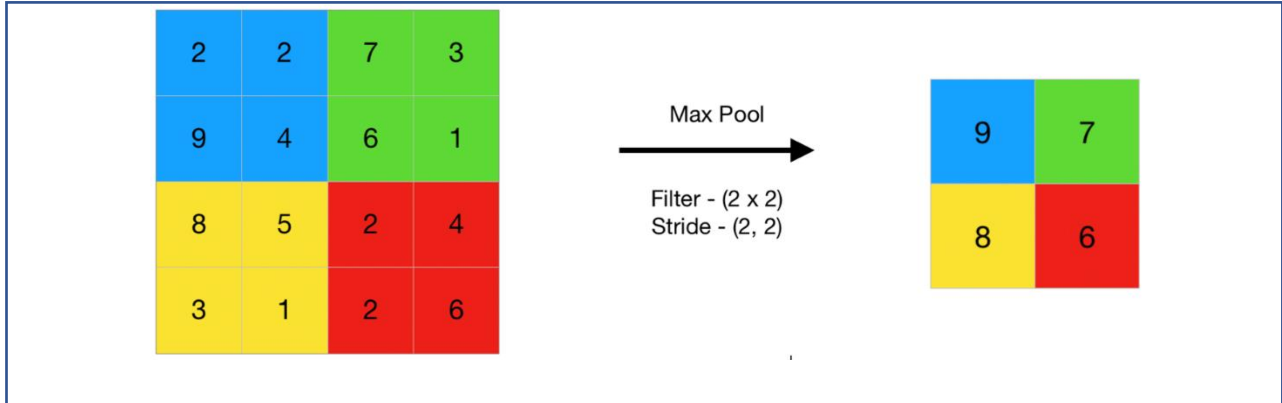

Figure 19: Max Pooling

2.3.3.2 Average pooling

Average pooling same as max pooling, but the output is the average of the pixels as shown in the figure (20).

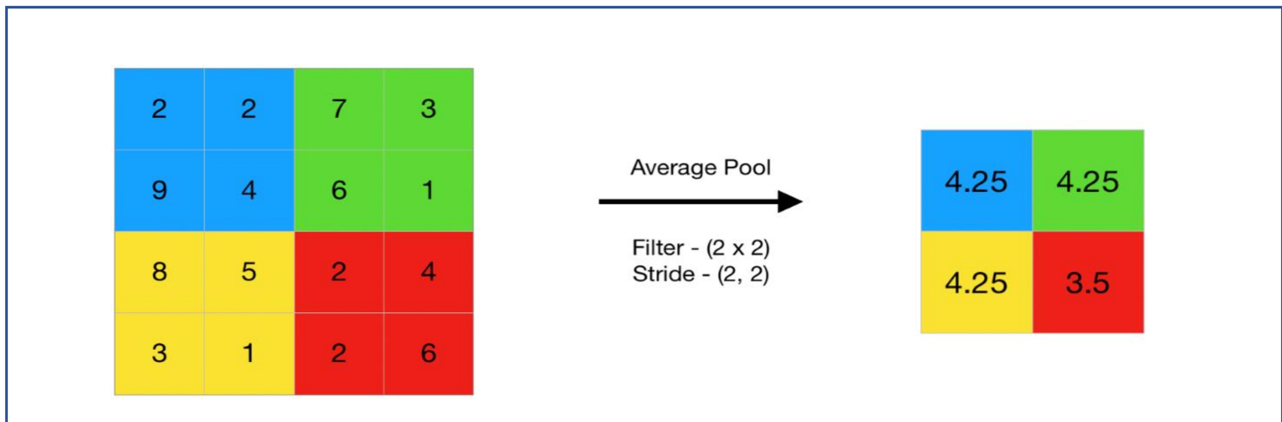

Figure 20: Average pooling

2.3.4 Fully connected layer

Fully connected (FC) layer is an ordinary DNN which used in logical regression or classification like soft max.
FC layer take the output from the convolution layers which represent the original photo features mapping and determine the most feature correlated to a particular class after flatten the output of convolution layers and multiply with FC weights as shown in the figure (21) [7].

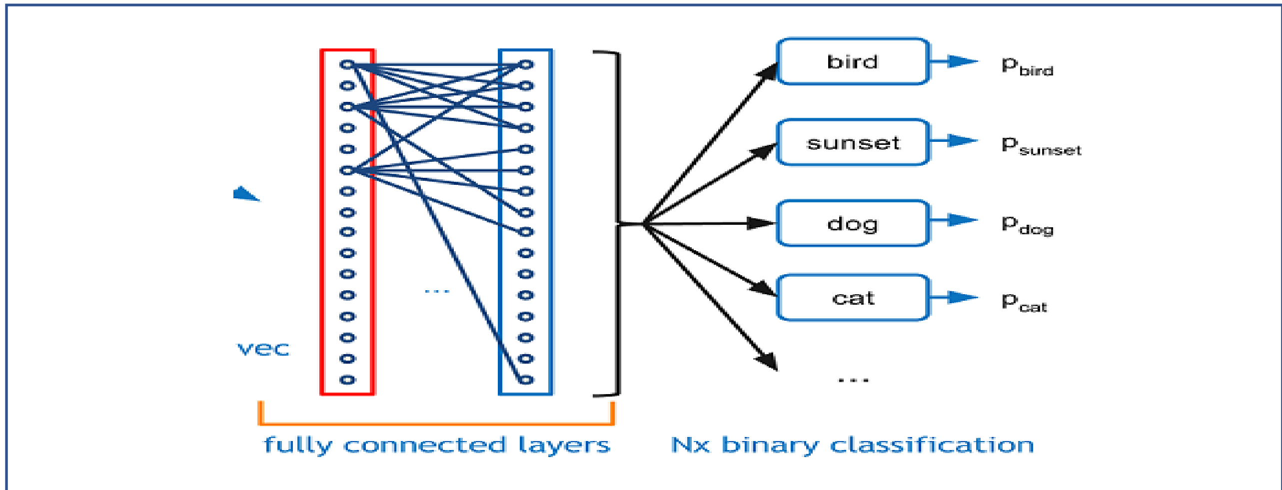

Figure 21: Fully connected layer

2.4 CNN architectures and models for image classification

After the revolution of computer vision and advanced driving assistant systems (ADAS) a lot of CNN models developed to achieve more accuracy and to reduce the number of parameters to implement the network as ASIC or FPGA product not only GPU applications. CNN can categorize into classic CNN which uses multiple of convolution layers to improve accuracy and modern CNN which uses an efficient way of learning to improve accuracy with small number of parameters. We will discuss the CNN architectures found in our survey.

2.4.1 SqueezeNet 2016

SqueezeNet deployed in 2016 by Stanford university, it was designed to achieve high accuracy (equal to AlexNet) with 50 times less parameters than AlexNet in addition to having no fully connected layers. As a result of that, SqueezeNet is very suitable to FPGA products. Figure (22) is an overview of SqueezeNet architecture [8].

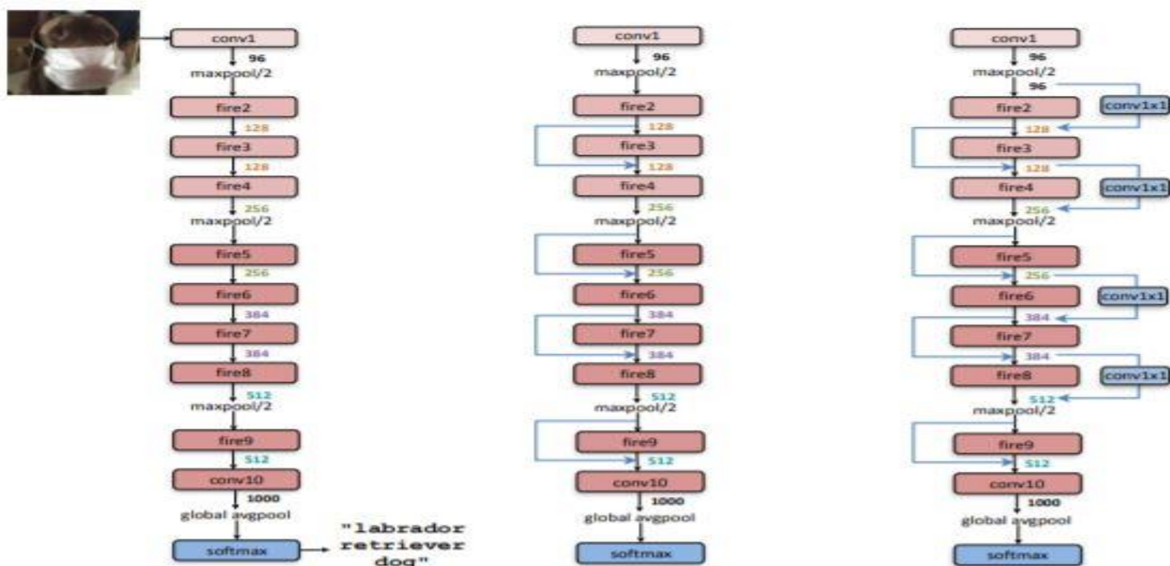

Figure 22: SqueezeNet architecture

Fire module

The Fire module is the foundation of SqueezNet consists of a Squeeze layer which reduces the number of input channels using a small number of 1×1 convolutions and an Expand layer which increases the number of channels of the Squeeze layer output using 1×1 and 3×3 convolutions. This method is called a bottle-neck structure. The Expand layer of the Fire module also has 1×1 convolution filters to reduce the number of parameters further [8].

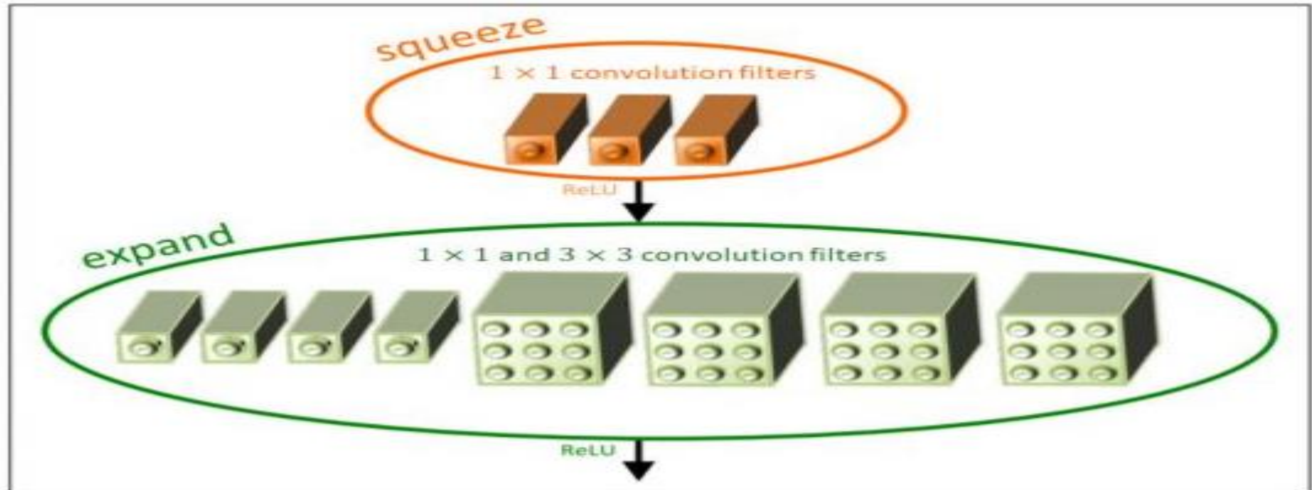

Figure 23: Squeeze Net fire module

2.4.2 ResNet model

2.4.2.1 Introduction

The common trend in research that network architecture community needs is to go deeper. Keep in mind that feedforward network to implement any function but surely overfitting will happen, so "*The deeper the better"* when it comes to convolutional neural networks. However, it has been noticed that after some depth, the performance degrades.

2.4.2.2 Reasons for performance degrades after certain depth:

while trying to avoid overfitting and other problems by using back propagation and increasing network layers, there are another problem will appear. When we increase the number of layers, there is a common problem in deep learning associated with that called Vanishing/Exploding gradient. This causes the gradient to become 0 or too large. Thus, when we increase number of layers, the training and test error rate also increases, as shown in figure (24).

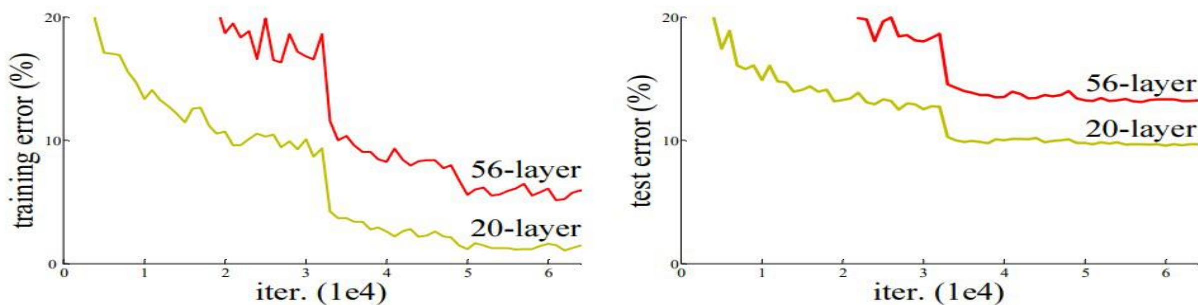

Figure 24: Performance Vs iterations

Note that this is not because of overfitting cause if those 56 layers should have lower training error and this not the case after analyzing more and more the authors were able to reach to the main problem as we mentioned Vanishing/Exploding gradient.

2.4.2.3 Vanishing/Exploding gradient:

Certain activation functions, like the sigmoid function output belong to [0,1], so it converts a large input space into a small input

So $whatever\ \delta x\ was\ huge\ it\ effect\ output\ with\ small\ \delta y,$ Hence the derivative becomes small.

For shallow-few layers- network, this isn't a big problem. However, it is a great issue if more layers are used, it causes the gradient to be too small for training to work effectively.

If M hidden layer uses sigmoid function. A small gradient means that the back-propagation effect which we use to get better weights by subtracting this back-propagation coefficient from initial weights will be too small to remove errors if we can call it an "error "to use better words to describe this "the weights and biases of the initial layers will not be updated effectively with each training session"

Solution:

Before ResNet, there had been several ways to solve the vanishing gradient issue, but none seemed to really tackle the problem once and for all. ResNet introduces "identity shortcut connection" simply skip mean one or more layers as in figure 25.

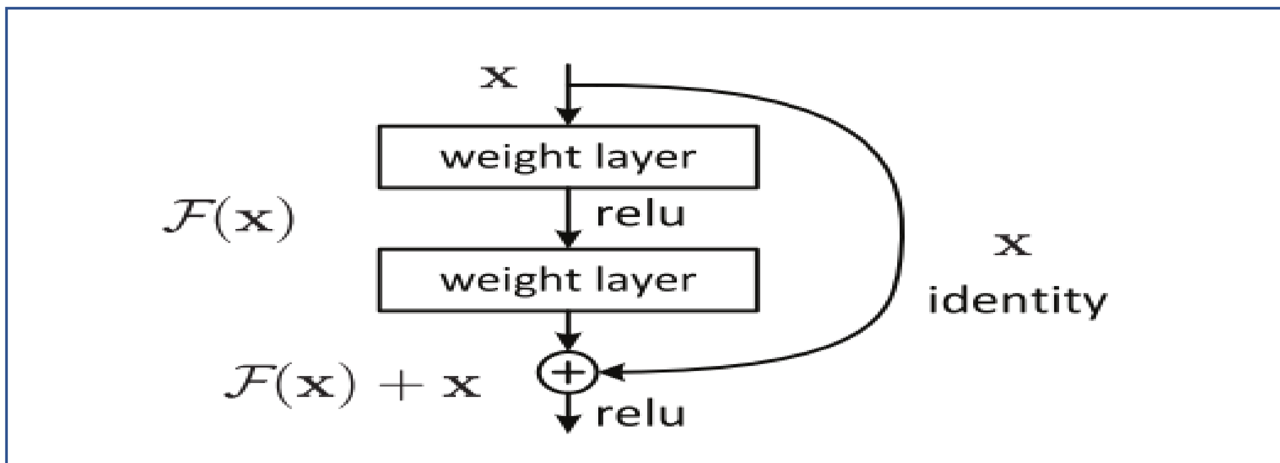

Figure 25: Short connection in ResNet

So, identity x will add to activation(F(x)+x)) to decrease effect of have small derivatives. if any layer hurt the performance of architecture, then it will be skipped by regularization. So, this results in training very deep neural network without the problems caused by vanishing/exploding gradient. The authors of the paper experimented on 100-1000 layers on CIFAR-10 dataset.

Example: ResNet 50 architecture:

The architecture of ResNet50 in figure (26) and deep learning model flowchart. Architecture of ResNet50 is shown and includes convolution layers, max pooling layers, and a fully connected layer.

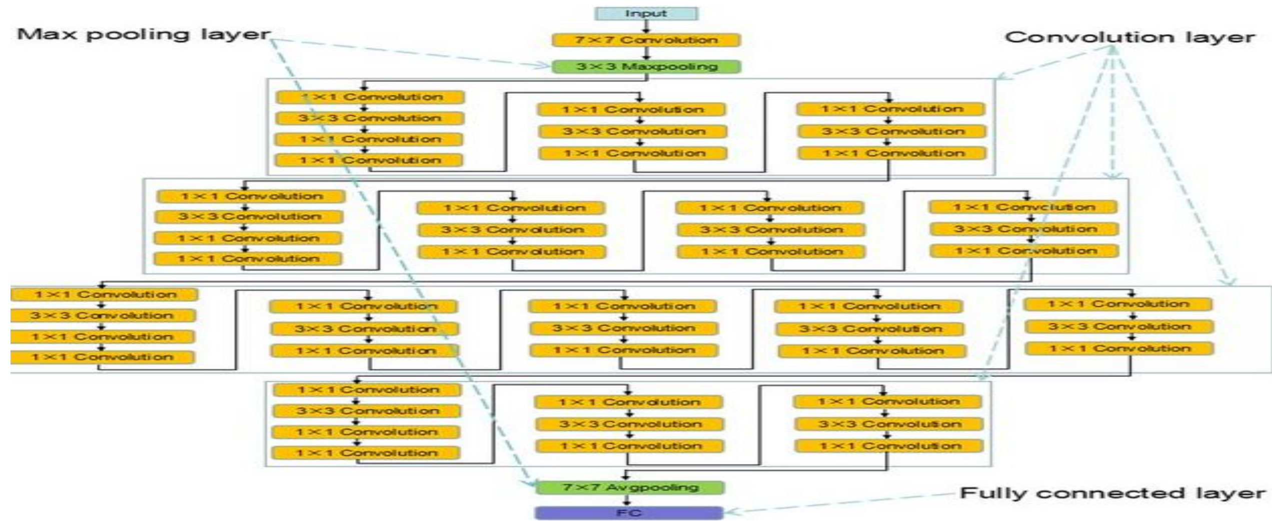

Figure 26: ResNet50 model

ResNet Versions:

Table 3: ResNet versions

Number of Layers	Number of Parameters
ResNet 18	11.174M
ResNet 34	21.282M
ResNet 50	23.521M
ResNet 101	42.513M
ResNet 152	58.157M

In figure (27), we can see the performance of some ResNet versions shown in table (3) training using image net. ResNet-152 achieves a top-5 validation error of 4.49%. A combination of 6 models with different depths achieves a top-5 validation error of 3.57%. Winning the 1st place in ILSVRC-2015.

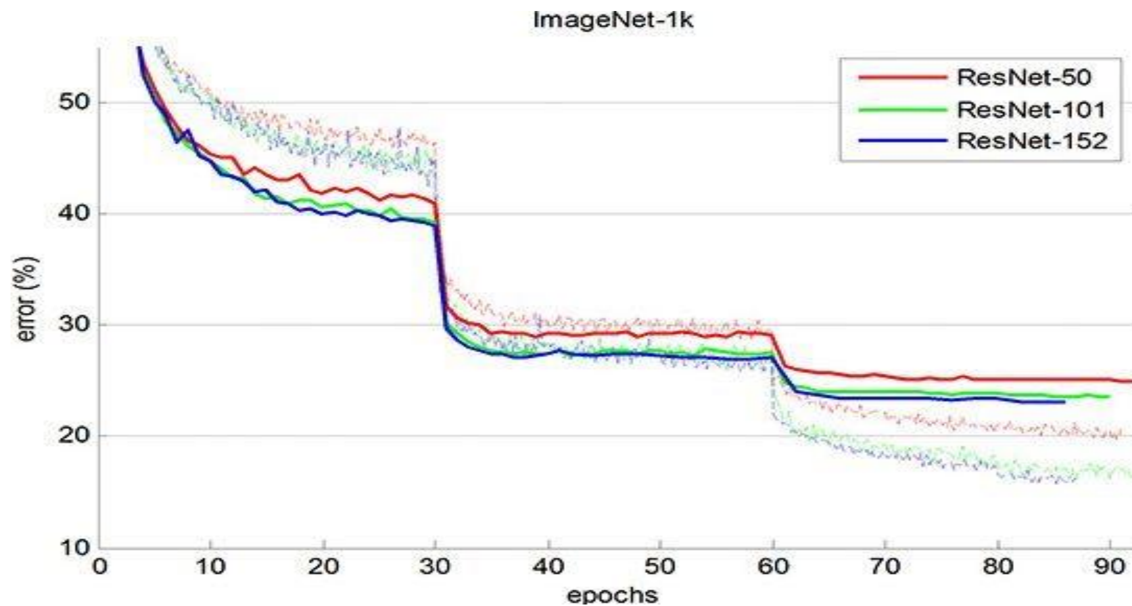

Figure 27: Performance of some ResNet versions

2.4.3 AlexNet Model

2.4.3.1 Introduction

AlexNet is the name of a convolutional neural network which has a large impact on the field of machine learning, specifically in the applications of deep learning. It was primarily designed by Alex Krizhevsky. After competing in ImageNet Large Scale Visual Recognition Challenge 2012, AlexNet shot to fame. It achieved a top-5 error of 15.3%. This was 10.8% lower than that of runner up. This network showed, for the first time, that the features obtained by learning can transcend manually-designed features, breaking the previous paradigm in computer vision.

2.4.3.2 Architecture

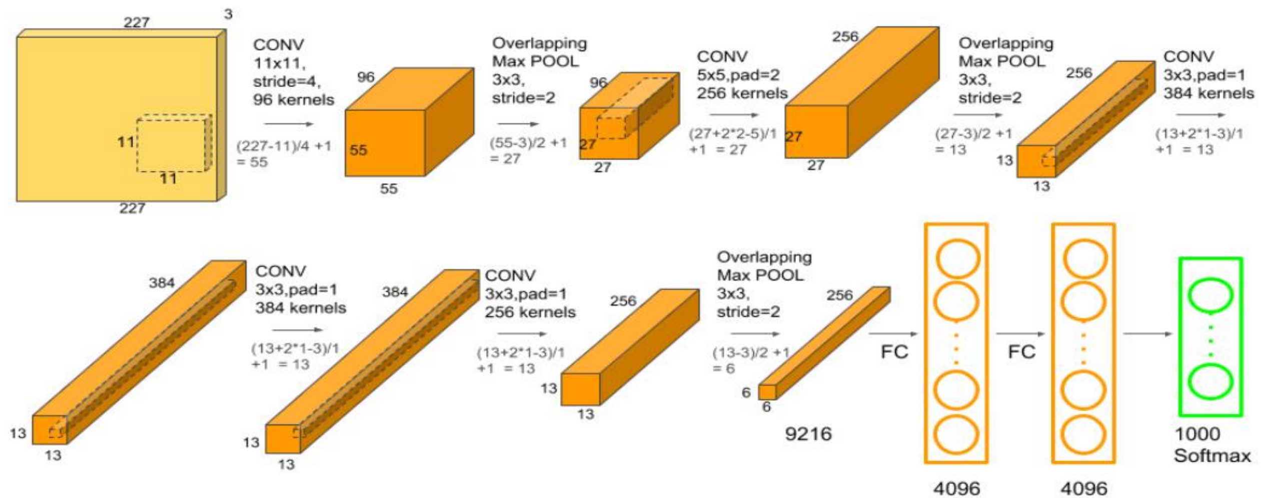

Figure 28: AlexNet architecture

As shown in figure (28), AlexNet architecture consists of 5 convolutional layers, 3 max-pooling layers, 2 normalization layers, 2 fully connected layers, and 1 SoftMax layer. Each convolutional layer consists of convolutional filters and a nonlinear activation function Relu. The pooling layers are used to perform max pooling. The input size is mentioned at most of the places as 224x224x3 but due to some padding which happens it works out to be 227x227x3. AlexNet overall has 60 million parameters and needs 1.1 billion computation units in a forward pass.

AlexNet success was because of new methods used that was not adopted at that time such as:

1-Using Relu as the nonlinearly after the convolution layers instead of Sigmoid and tanh functions that were commonly used which increased the speed greatly.

2-Using maximum pooling instead of traditionally used average pooling.

3-Using dropout method between fully connected layers in order to improve the generalization error instead of using ordinary regularization.

4-Using Data Augmentation by Mirroring and random crop which helped in decreasing overfitting problem [9].

2.4.4 VGGNet

2.4.4.1 Introduction

VGG 16 is a Convolutional Neural Network architecture, It was developed by Karen Simonyan and Andrew Zisserman in 2014. This model achieves *92.7% top-5* test accuracy on ImageNet dataset which contains *14* million images belonging to 1000 classes.

2.4.4.2Architecture

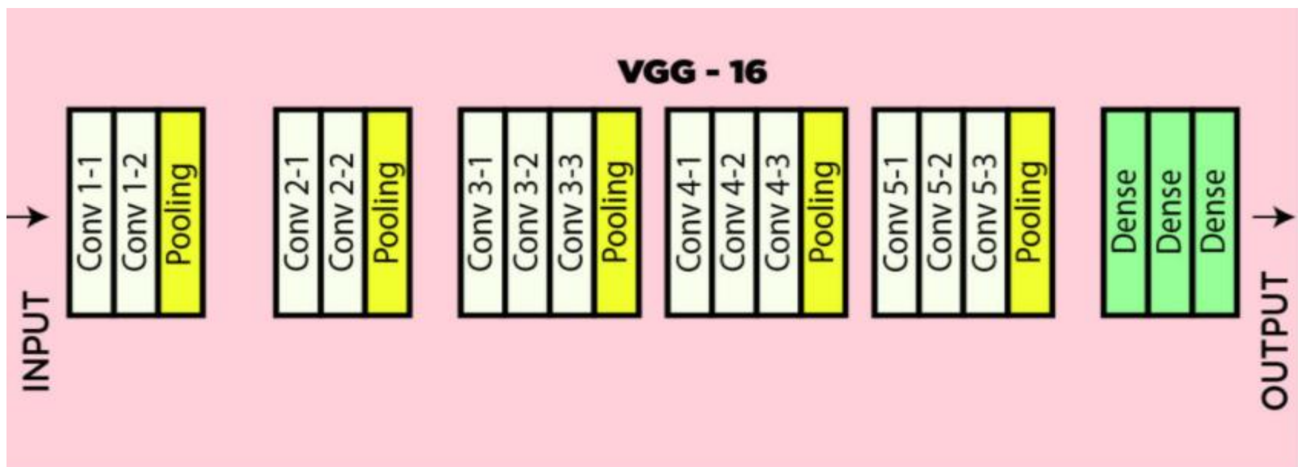

Figure 29: VGG-16 architecture

The input to the network is image of dimensions *(224, 224, 3)*. As shown in biases figure (29), The first two layers have *64* channels of *3*3* filter size and same padding. Then after a max pool layer of stride *(2,2)*, two layers which have convolution layers of 256 filter size and filter size *(3,3)*. This followed by a max pooling layer of stride *(2,2)* which is same as previous layer. Then there are *2* convolution layers of filter size *(3,3)* and *256* filter. After that there are *2* sets of *3* convolution layer and a max pool layer. Each have *512* filters of *(3,3)* size with same padding. This image is then passed to the stack of two convolution layers. In these convolution and max pooling layers, the filters we use is of the size *3*3* instead of *11*11* in AlexNet. In some of the layers, it also uses *1*1* pixel which is used to manipulate the number of input channels. There is a padding of *1-pixel* (same padding) done after each convolution layer to prevent the spatial feature of the image. After the stack of convolution and max-pooling layer, we got a *(7, 7, 512)* feature map. We flatten this output to make it a *(1, 25088)* feature vector. After this there are *3 fully* connected layer, the first layer takes input from the last feature vector and outputs a *(1, 4096)* vector, second layer also outputs a vector of size *(1, 4096)* but the third layer output 1000 channels, then after the output of 3rd fully connected layer is passed to SoftMax layer in order to normalize the classification vector. After the output of classification vector top-5 categories for evaluation. All the hidden layers use RELU as its activation function.

VGG-16 reached a *92.7% top-5* test accuracy. However, the network contains almost 140 million parameters and one forward pass requires nearly 16 billion MAC operations [10].

2.4.5 MobileNet

2.4.5.1 Introduction

MobileNet is an efficient and portable CNN architecture which is used in real world applications. It uses depthwise separable convolution instead of standard convolution in order to build lighter CNN architectures with low latency, low power and reasonable hardware resources which allows us to implement these architectures on programable logic devices such as FPGAs. A standard MobileNet network has 4.2 million parameters which can be further reduced by

tuning the width multiplier (α) hyperparameter appropriately. The size of the input image is 224 × 224 × 3 [11].

2.4.5.2 Depthwise separable convolution

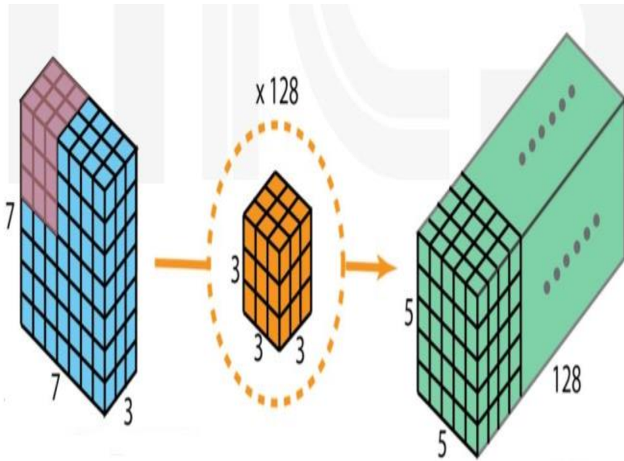

Figure 31: Standard Convolution

Figure 30: Depthwise Convolution

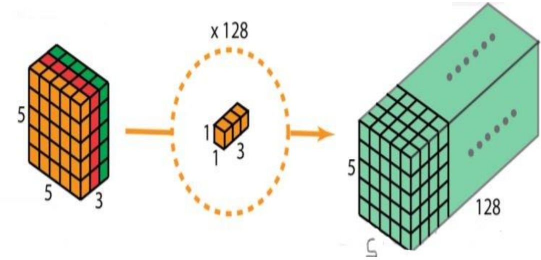

Figure 32: Pointwise Convolution

In order to determine whether CNN architecture is light or not, weights and number of multiplication calculations must be considered. We have an input image of size (7,7,3) and 128 filters of size (3,3,3) from this information we get an output image of size (5,5,128). Considering figure (31), every filter is convolved with a sliding window of input feature map and then all multiplicand numbers are summed, so number of weights of standard convolution is $3 * 3 * 3 * 128 = 3456\ number$, and number of multiplications to generate output feature map in one cycle is $5 * 5 * 3 * 3 * 3 * 128 = 86{,}400\ MAC$.

In depthwise separable convolution, we have a depthwise filter consists of number of channels which treats input feature map channels separately in order to get the right output feature map dimensions, then it is followed by a number of pointwise filters equals to required output feature map channels with dimensions (1,1, input feature map channels).

As shown in figure (30), number of weights of depthwise convolution is $3 * 3 * 3 = 27\ number$, and number of multiplications to generate output feature map in one cycle is $5 * 5 * 3 * 3 * 3 = 675\ MAC$.

As shown in figure (32), number of weights of pointwise convolution is $1 * 1 * 3 * 128 = 384\ number$, and number of multiplications to generate output feature map in one cycle is $5 * 5 * 1 * 1 * 3 * 128 = 9600\ MAC$.

So, total number of weights of depthwise separable convolution is 411 (88% reduction), and total of multiplications is 10275 (88% reduction).

2.4.5.3 Architecture

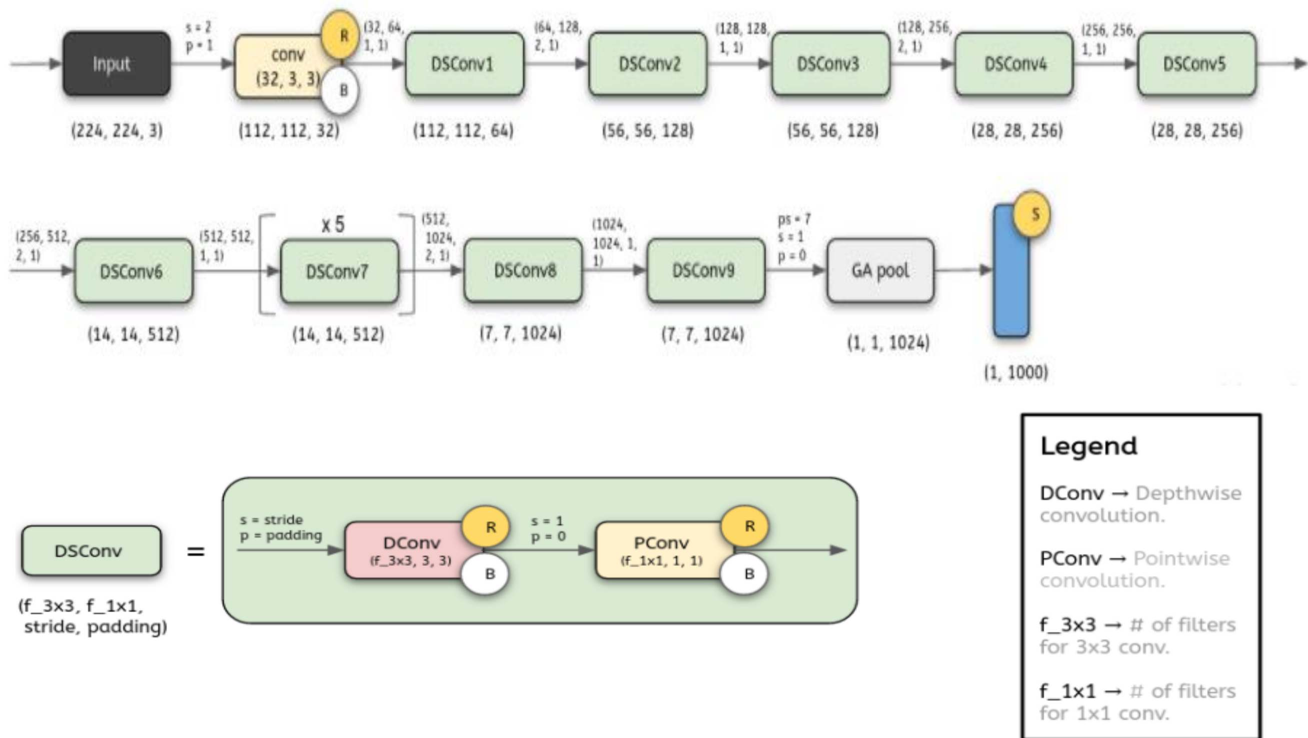

Figure 33: MobileNet architecture

As shown in figure (33), This architecture has only one standard convolution layer to extract a large number of features from the input image and 13 layers of depthwise separable convolution. Every depthwise separable convolution consists of depthwise convolution and pointwise convolution, each of them followed by Relu to reject the negative numbers generated in the feature map and batch normalization layer to ensure that the multiplication results of the architecture chain do not exceed the maximum represented number. It also has an average pooling layer to shrink the size of the input feature map by taking only the important feature and neglecting other secondary features. The last layer is a fully connected layer to translate the results of the architecture into a certain class.

This architecture has 2 fundamental hyperparameters: width multiplier(α) and Resolution Multiplier(ρ). width multiplier(α) is a global hyperparameter that is used to construct smaller and less computationally expensive models as it reduces the number of weights and by extension the number of layers but it increases the number of channels per layer as the number of input channels 'M' becomes $\alpha * M$ and the number of output channels 'N' becomes $\alpha * N$. Its value lies between 0 and 1 but it has commonly used values which are 1, 0.75, 0.5, 0.25. Resolution Multiplier(ρ) is used to decrease the resolution of the input image and this subsequently reduces the input to every layer by the same factor for a given value of ρ the resolution of the input image becomes $224 * \rho$ [11].

Chapter 3: Training

3.1 Training

Training mobile net version one on German Traffic signs which has the following:

- The size of training set is: 34799
- The size of the validation set is: 4410
- The size of test set is: 12630
- The shape of a traffic sign image is: (32, 32 ,3)

The number of unique classes/labels in the data set is: 43

Figure 34: Signs from GTS

We build two models which will be discussed in the following pages:

3.1.1 Model 1

This model has been trained on image net and its open source on internet but we will use it and train the mode on GTS it has 13 layers as shown

```
Model: "mobilenet_0.50_128"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_3 (InputLayer)         [(None, 128, 128, 3)]     0
_________________________________________________________________
conv1 (Conv2D)               (None, 64, 64, 16)        432
_________________________________________________________________
conv1_bn (BatchNormalization (None, 64, 64, 16)        64
_________________________________________________________________
conv1_relu (ReLU)            (None, 64, 64, 16)        0
_________________________________________________________________
conv_dw_1 (DepthwiseConv2D)  (None, 64, 64, 16)        144
_________________________________________________________________
conv_dw_1_bn (BatchNormaliza (None, 64, 64, 16)        64
_________________________________________________________________
conv_dw_1_relu (ReLU)        (None, 64, 64, 16)        0
_________________________________________________________________
conv_pw_1 (Conv2D)           (None, 64, 64, 32)        512
_________________________________________________________________
conv_pw_1_bn (BatchNormaliza (None, 64, 64, 32)        128
_________________________________________________________________
conv_pw_1_relu (ReLU)        (None, 64, 64, 32)        0
_________________________________________________________________
conv_pad_2 (ZeroPadding2D)   (None, 65, 65, 32)        0
_________________________________________________________________
conv_dw_2 (DepthwiseConv2D)  (None, 32, 32, 32)        288
_________________________________________________________________
conv_dw_2_bn (BatchNormaliza (None, 32, 32, 32)        128
_________________________________________________________________
conv_dw_2_relu (ReLU)        (None, 32, 32, 32)        0
_________________________________________________________________
conv_pw_2 (Conv2D)           (None, 32, 32, 64)        2048
_________________________________________________________________
conv_pw_2_bn (BatchNormaliza (None, 32, 32, 64)        256
_________________________________________________________________
conv_pw_2_relu (ReLU)        (None, 32, 32, 64)        0
_________________________________________________________________
conv_dw_3 (DepthwiseConv2D)  (None, 32, 32, 64)        576
```

```
______________________________________________________________
conv_dw_3_bn (BatchNormaliza (None, 32, 32, 64)        256
______________________________________________________________
conv_dw_3_relu (ReLU)        (None, 32, 32, 64)        0
______________________________________________________________
conv_pw_3 (Conv2D)           (None, 32, 32, 64)        4096
______________________________________________________________
conv_pw_3_bn (BatchNormaliza (None, 32, 32, 64)        256
______________________________________________________________
conv_pw_3_relu (ReLU)        (None, 32, 32, 64)        0
______________________________________________________________
conv_pad_4 (ZeroPadding2D)   (None, 33, 33, 64)        0
______________________________________________________________
conv_dw_4 (DepthwiseConv2D)  (None, 16, 16, 64)        576
______________________________________________________________
conv_dw_4_bn (BatchNormaliza (None, 16, 16, 64)        256
______________________________________________________________
conv_dw_4_relu (ReLU)        (None, 16, 16, 64)        0
______________________________________________________________
conv_pw_4 (Conv2D)           (None, 16, 16, 128)       8192
______________________________________________________________
conv_pw_4_bn (BatchNormaliza (None, 16, 16, 128)       512
______________________________________________________________
conv_pw_4_relu (ReLU)        (None, 16, 16, 128)       0
______________________________________________________________
conv_dw_5 (DepthwiseConv2D)  (None, 16, 16, 128)       1152
______________________________________________________________
conv_dw_5_bn (BatchNormaliza (None, 16, 16, 128)       512
______________________________________________________________
conv_dw_5_relu (ReLU)        (None, 16, 16, 128)       0
______________________________________________________________
conv_pw_5 (Conv2D)           (None, 16, 16, 128)       16384
______________________________________________________________
conv_pw_5_bn (BatchNormaliza (None, 16, 16, 128)       512
______________________________________________________________
conv_pw_5_relu (ReLU)        (None, 16, 16, 128)       0
______________________________________________________________
conv_pad_6 (ZeroPadding2D)   (None, 17, 17, 128)       0
______________________________________________________________
conv_dw_6 (DepthwiseConv2D)  (None, 8, 8, 128)         1152
```

```
conv_dw_6 (DepthwiseConv2D)  (None, 8, 8, 128)         1152
_________________________________________________________________
conv_dw_6_bn (BatchNormaliza (None, 8, 8, 128)         512
_________________________________________________________________
conv_dw_6_relu (ReLU)        (None, 8, 8, 128)         0
_________________________________________________________________
conv_pw_6 (Conv2D)           (None, 8, 8, 256)         32768
_________________________________________________________________
conv_pw_6_bn (BatchNormaliza (None, 8, 8, 256)         1024
_________________________________________________________________
conv_pw_6_relu (ReLU)        (None, 8, 8, 256)         0
_________________________________________________________________
conv_dw_7 (DepthwiseConv2D)  (None, 8, 8, 256)         2304
_________________________________________________________________
conv_dw_7_bn (BatchNormaliza (None, 8, 8, 256)         1024
_________________________________________________________________
conv_dw_7_relu (ReLU)        (None, 8, 8, 256)         0
_________________________________________________________________
conv_pw_7 (Conv2D)           (None, 8, 8, 256)         65536
_________________________________________________________________
conv_pw_7_bn (BatchNormaliza (None, 8, 8, 256)         1024
_________________________________________________________________
conv_pw_7_relu (ReLU)        (None, 8, 8, 256)         0
_________________________________________________________________
conv_dw_8 (DepthwiseConv2D)  (None, 8, 8, 256)         2304
_________________________________________________________________
conv_dw_8_bn (BatchNormaliza (None, 8, 8, 256)         1024
_________________________________________________________________
conv_dw_8_relu (ReLU)        (None, 8, 8, 256)         0
_________________________________________________________________
conv_pw_8 (Conv2D)           (None, 8, 8, 256)         65536
_________________________________________________________________
conv_pw_8_bn (BatchNormaliza (None, 8, 8, 256)         1024
_________________________________________________________________
conv_pw_8_relu (ReLU)        (None, 8, 8, 256)         0
_________________________________________________________________
conv_dw_9 (DepthwiseConv2D)  (None, 8, 8, 256)         2304
```

```
________________________________
conv_dw_9 (DepthwiseConv2D)   (None, 8, 8, 256)         2304
________________________________
conv_dw_9_bn (BatchNormaliza (None, 8, 8, 256)         1024
________________________________
conv_dw_9_relu (ReLU)         (None, 8, 8, 256)         0
________________________________
conv_pw_9 (Conv2D)            (None, 8, 8, 256)         65536
________________________________
conv_pw_9_bn (BatchNormaliza (None, 8, 8, 256)         1024
________________________________
conv_pw_9_relu (ReLU)         (None, 8, 8, 256)         0
________________________________
conv_dw_10 (DepthwiseConv2D) (None, 8, 8, 256)         2304
________________________________
conv_dw_10_bn (BatchNormaliz (None, 8, 8, 256)         1024
________________________________
conv_dw_10_relu (ReLU)        (None, 8, 8, 256)         0
________________________________
conv_pw_10 (Conv2D)           (None, 8, 8, 256)         65536
________________________________
conv_pw_10_bn (BatchNormaliz (None, 8, 8, 256)         1024
________________________________
conv_pw_10_relu (ReLU)        (None, 8, 8, 256)         0
________________________________
conv_dw_11 (DepthwiseConv2D) (None, 8, 8, 256)         2304
________________________________
conv_dw_11_bn (BatchNormaliz (None, 8, 8, 256)         1024
________________________________
conv_dw_11_relu (ReLU)        (None, 8, 8, 256)         0
________________________________
conv_pw_11 (Conv2D)           (None, 8, 8, 256)         65536
________________________________
conv_pw_11_bn (BatchNormaliz (None, 8, 8, 256)         1024
________________________________
conv_pw_11_relu (ReLU)        (None, 8, 8, 256)         0
________________________________
conv_pad_12 (ZeroPadding2D)   (None, 9, 9, 256)         0
________________________________
conv_dw_12 (DepthwiseConv2D) (None, 4, 4, 256)         2304
```

;

```
conv_dw_12 (DepthwiseConv2D) (None, 4, 4, 256)        2304

conv_dw_12_bn (BatchNormaliz (None, 4, 4, 256)        1024

conv_dw_12_relu (ReLU)       (None, 4, 4, 256)        0

conv_pw_12 (Conv2D)          (None, 4, 4, 512)        131072

conv_pw_12_bn (BatchNormaliz (None, 4, 4, 512)        2048

conv_pw_12_relu (ReLU)       (None, 4, 4, 512)        0

conv_dw_13 (DepthwiseConv2D) (None, 4, 4, 512)        4608

conv_dw_13_bn (BatchNormaliz (None, 4, 4, 512)        2048

conv_dw_13_relu (ReLU)       (None, 4, 4, 512)        0

conv_pw_13 (Conv2D)          (None, 4, 4, 512)        262144

conv_pw_13_bn (BatchNormaliz (None, 4, 4, 512)        2048

conv_pw_13_relu (ReLU)       (None, 4, 4, 512)        0

global_average_pooling2d_2 ( (None, 512)              0
```

Figure 35: Model 1 Layers

Two Hyperparameters are provided and discussed in the paper [14]:

- Alpha (α): affects weights directly as it affects number of filters [25%,50%,75%,100%].
- Lambda (人): resolution of photos affects multiplications number [128,160,192,224].

Table 6. MobileNet Width Multiplier

Width Multiplier	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
0.75 MobileNet-224	68.4%	325	2.6
0.5 MobileNet-224	63.7%	149	1.3
0.25 MobileNet-224	50.6%	41	0.5

Table 7. MobileNet Resolution

Resolution	ImageNet Accuracy	Million Mult-Adds	Million Parameters
1.0 MobileNet-224	70.6%	569	4.2
1.0 MobileNet-192	69.1%	418	4.2
1.0 MobileNet-160	67.2%	290	4.2
1.0 MobileNet-128	64.4%	186	4.2

Figure 36: Mobile Net trained on image net

As shown in figure (36) as alpha increases, accuracy, required multiplication numbers and weights (million parameters) increase.

As lambda increases, accuracy improves not much and doesn't affect weights but increases the number of multiplications required hugely and doesn't give any advantage or higher improvement in accuracy.

3.1.2 Training of model one on GTS

Using the following steps to training the model:

-GTS is a small training set which will not achieve high accuracy if the training only uses GTS images to train to overcome this problem, we will use a transfer learning approach. Using pre-trained weights of image nets in certain layers then open others layers to train its weights on GTs achieving.

-Merge validation set and test set together and tuning hyperparameters on training set due to small sets.

-Tuning the hyperparameters to get best results taking into consideration the hardware perspective to implement it.

Table 4: Tuning lambda

α	人	Training accuracy	test accuracy	Weights number
0.5	128	97.6%	94.354%	851,595
0.5	160	99.5%	94.5%	851,595
0.5	192	98.16%	95.35%	851,595
0.5	224	98.2%	90.4%	851,595

We will choose 人=128, cause as it increases no huge improvement in accuracy at the same time required a huge number of multiplications which is an effective variable will affect speed and power, so to achieve improvement in test accuracy with 4% will take a high percent of luts and increase power badly, this improvement in test accuracy can achieve with alpha without this tradeoff or mainly it isn't a tradeoff it's like you pay millions of monies to buy an old car to.

Table 5: Tunning alpha

α	p	Training accuracy	test accuracy	Weight
0.25	160	96.86%	84.16%	229,595
0.5	160	99.5%	94.5%	851,595
0.75	160	97.78%	97.66%	1,866,043
1	160	97.92%	98.07%	3,272,939

Increasing alpha increases accuracy and at the same time increases weights which will require more memories and increasing number of multiplications to get benefit from all of these we choose alpha =0.5.

3.1.3 Disadvantage in model 1

Gts image is 32 *32 *3 this model has restriction on input image to be minimum 128 *128*3 which will add overhead in registers mainly and in parallelism in depth wise and pointe wise leading to bad utilization of fpga resources.

-13 Layer will be implemented on fpga by using sharing concept but this will increase delay and decrease flexibility of design, close solutions to others problem.

-this model in 32-bit float which isn't the case in hardware is 16-bit 6 float and 10 integers so there is a huge reduction in accuracy between software and hardware

Let's see how second model can solve all of this

3.2 Model 2 and Quantization

Model consist of 7 layers only which is trained in two steps:

3.2.1 Quantization aware Training

Enable us to make model see integer representation not float by putting a fake layer (Fake Quant nodes) as in figure (3) during training to reduce loss in accuracy between software and hardware actually the software accuracy will be the same as hardware accuracy.[12]

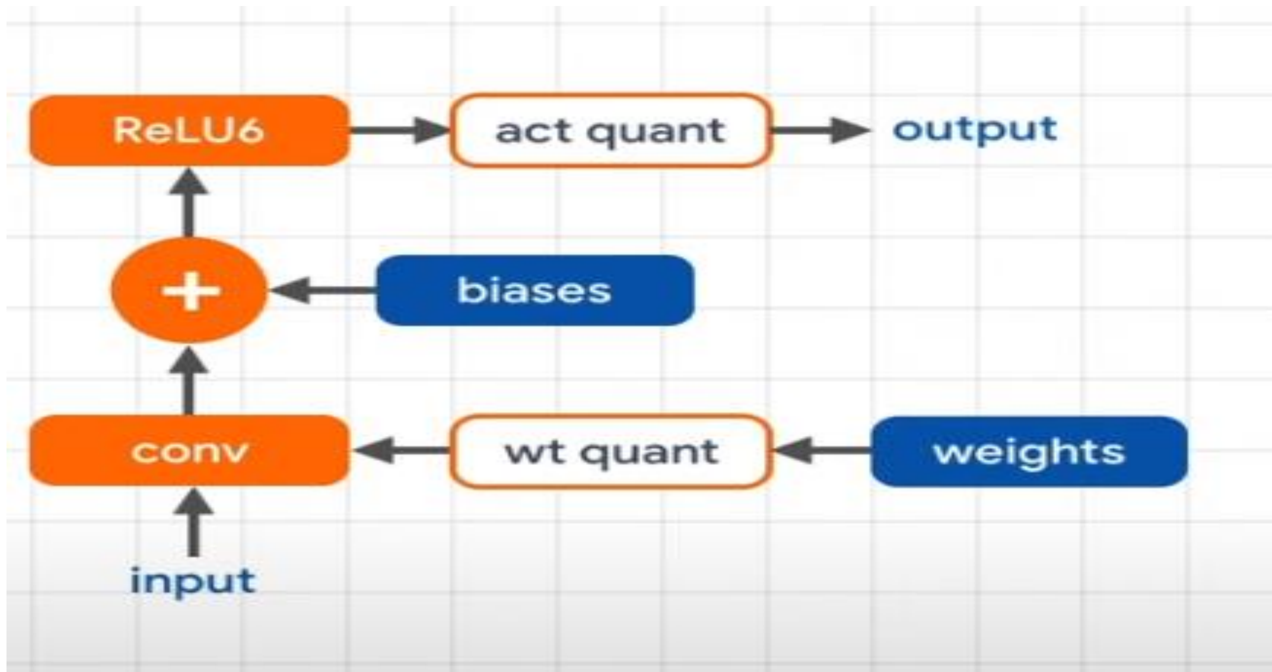

Figure 37: Insert Fake Quant layer during training

This introduces two new parameter scales (s) and zeros (z), scales are used to scale back the low precision values back to the real values (floating), zeros are low precision value that represents the quantized value representing the real value of zero

$$R = S(q - z) \qquad (Eq1)$$

Where: R: real value, Q: quantized value, Z: zero, S: scale

$$S(scale) = \frac{flota \max number(F_{MAX}) - float \min number\ (F_{MIN})}{qunatized \max(Q_{MAX}) - qunatizemin(Q_{MIN})} \qquad (Eq2)$$

To get zero similarly find linear relationship with extremes in two domains:

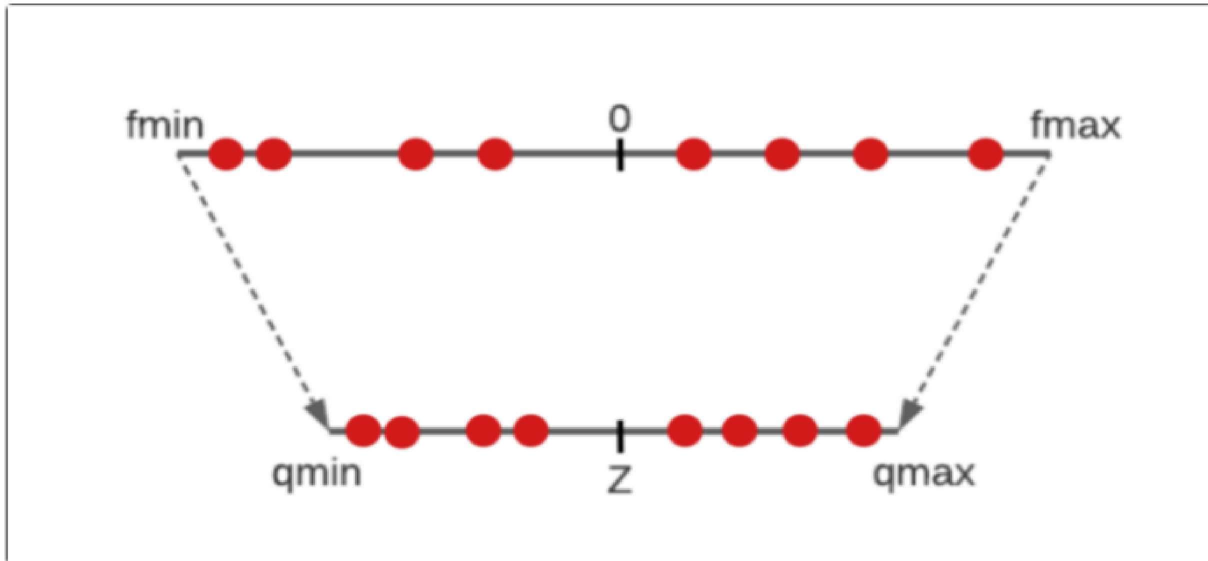

Figure 38: Finding zeros

If fmin and fmax doesn't have real 0 this is problem so to overcome of this

Zero will be chosen to equal Q_{Max} or Q_{Max}which is already mapped value from F_{Max} and F_{Min}

Now after training the model, we will get bias, weights, zeros, scales to test the model in interference time. Note that this approach simulates quantization effects in forward pass of training but backpropagation still happens as usual and all weights and bias are still float to be easily nudged by small values δ.

We get a lot of benefits:

- Improvements in model compression and latency reduction
- Model size shrinks by 4x
- 1.5x-4x improvements in latency

Model	Non-quantized Top-1 Accuracy	8-bit Quantized Accuracy
MobilenetV1 224	71.03%	71.06%
Resnet v1 50	76.3%	76.1%
MobilenetV2 224	70.77%	70.01%

Figure 39: Models are trained on image net

3.2.2 Second Step: Post training quantization

After the training model, we will apply post training quantization to convert all parameters to integers. This will not lead to huge loss in accuracy as qwt is done before this step.

Why don't we make this step only?

this approach works sufficiently well for large models with considerable representational capacity, but leads to significant accuracy drops
for small models which is our case.

3.2.3 How quantization is implemented in hardware

Multiplying 2 real number:

$$r_3 = r_2 r_1$$

Substituting using equation 1 and rearrange terms:
in next equation 3 and equation 4: as previous q is quantized value and z and s is zeros and scales.

$$q_3 = z_3 + \frac{S_1 S_2}{S_3} \sum (q_1 - z_1)(q_2 - z_2) \quad (Eq3)$$

Define M as:

$$M = \frac{S_1 S_2}{S_3} \quad (Eq4)$$

- M we multiplied in SW and stored in memories.
- Choosing $q_1(r_1)$ to be weight and $q_2(r_2)$ to be input then in SW we can subtract weight from its zeros avoiding further addition in HW.
- As every stage input will subtract from its zero which is zero of output in the previous stage which it was added again then no need for it even in the final stage case it's like an offset for all numbers which will be compared to get max between them. For example: in stage 2

$$q_3(output\ stage\ 2) = z_3(zero\ of\ stage\ 2) + number \quad (Eq5)$$

In stage 3 q3 become input which will be subtract from its zero but to get output from stage 2 this zero was added:

$$q_4(output\ stage\ 3) = z_4(zero\ of\ stage\ 3) + M\sum(weight) * (output\ last\ stage - zero\ last\ stage\ ) \quad (Eq6)$$

- Note that zero for input image is equal to =0 this analysis can apply.

3.2.4 Model 2 Layer

It consists of 7 layers only which is approximately half reduction in model compared to model 1. Note that padding layers and quantize layer isn't a physical layer in hardware padding only happens before getting input from stage as will be discussed in hardware implementation.

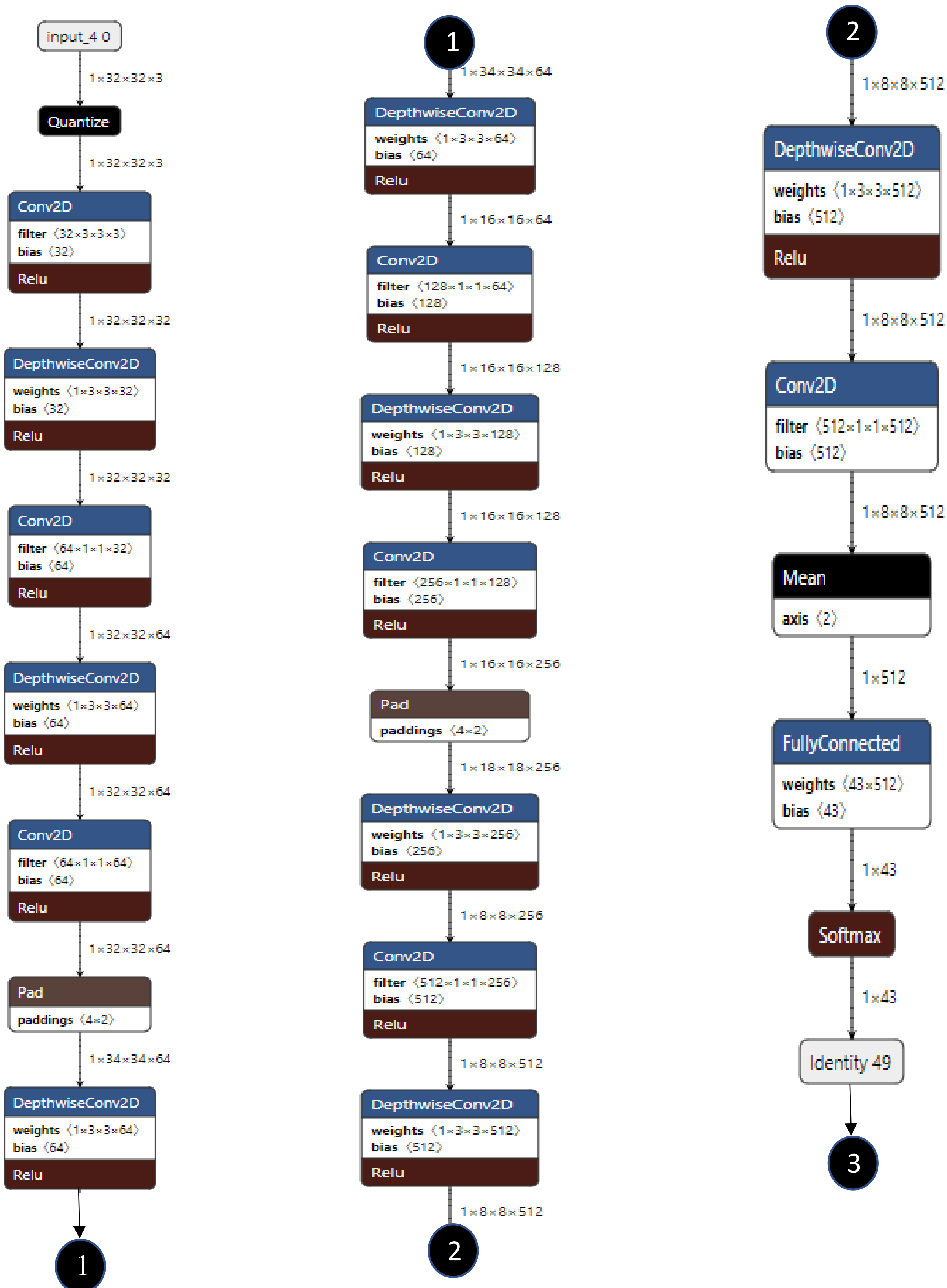

Figure 40: Final mode 2 Layers

Chapter 4: Hardware Design Methodology

4.1 FPGA Introduction and main resources

FPGA (field programmable gate arrays) are semiconductor devices that are based around a matrix of configurable logic blocks (CLB) connected with each other via programmable interconnections. FPGA can be programmed to desired applications.

Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR.

In most FPGAs, logic blocks also include memory elements, which may be simple flip flops or complete blocks of memory. Many FPGAs. many FPGAs can be programmed to implement different logic functions, allowing flexible reconfigurable computing as performed in computer software.

FPGA vs microprocessor in a large wide of digital application related to image processing or communication applications can implemented on a microprocessor but sometimes FPGA is a good choice compared to microprocessor because FPGA has a parallelism in operation, but microprocessor runs its operation sequentially. This makes FPGA in some applications faster in overall operation but still with limited speed of clock. Due to parallelism, flexibility of design and FPGA resources usage, acceleration of some algorithms needs to high papalism became a trend now in the digital design market.

FPGA also has some bulids in DSPs for spatial operations on signal processing like convolutions, and also has IPs like fast adder IPs and FFT IP to reduce usage of LUTs because FPGAs suffer from limited resources.

In the rest of this part the FPGA internal component used in our accelerator will be viewed.

4.2 FPGA 7 series internal components

4.2.1 Configure Logic Block (CLB)

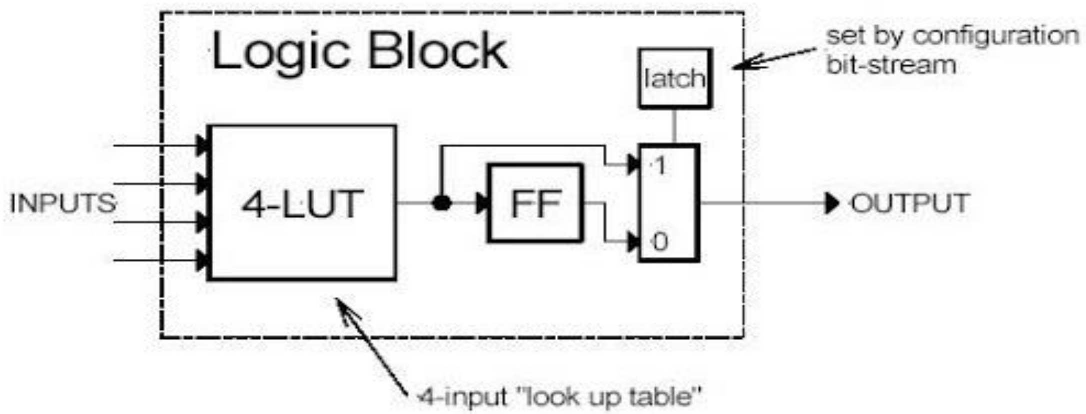

Figure 41: Configurable logic block

A configurable logic block (CLB) is the basic repeating logic resource on an FPGA. When linked together by routing resources, the components in CLBs execute complex logic functions, implement memory functions, and synchronize code on the FPGA, configure logic blocks containing some small components like flip flops, multiplexers and lookup tables (LUT).

LUTs is a collection of gates hardwired on the FPGA. An LUT stores a predefined list of outputs for every combination of inputs. LUTs provide a fast way to retrieve the output of a logic operation because possible results are stored and then referenced rather than calculated as shown in figure (41).

4.2.2 Configurable I/O blocks

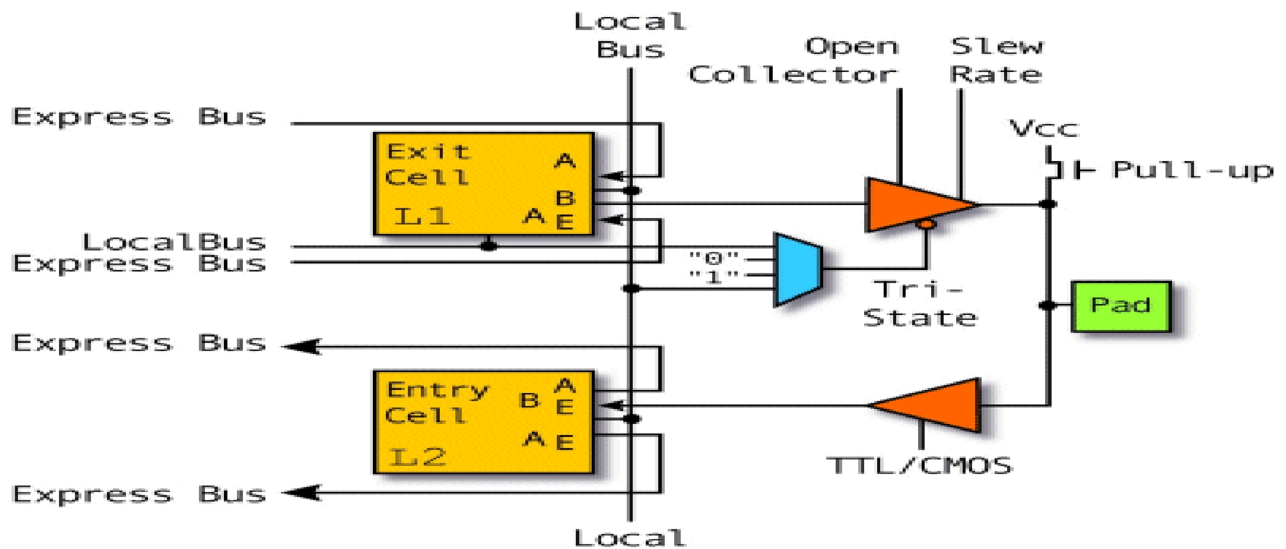

Figure 42: Input output configurable block

A I/o configurable block is designed to be the interface between the FPGA on chip and outer side so I/O ports is used to take signal inside and outside the FPGA this block can be configurable to 3 states input port, output port and (In-out) port that usually uses in the memory paths the internal structure of I/O port as shown in figure (42).

4.2.3 Clock Driver

Clock drivers is set as a pre implemented module in FPGA to solve the problems that appears when all design is connected with one clock signal the first problem is high fanout of this port because it is connected to all flip flops nodes so a strong driver should be implemented to guarantee minimum propagation delay of the clock and puffer tree should also implemented to save the global skew min and constant in all blocks.

4.2.4 DSP Block

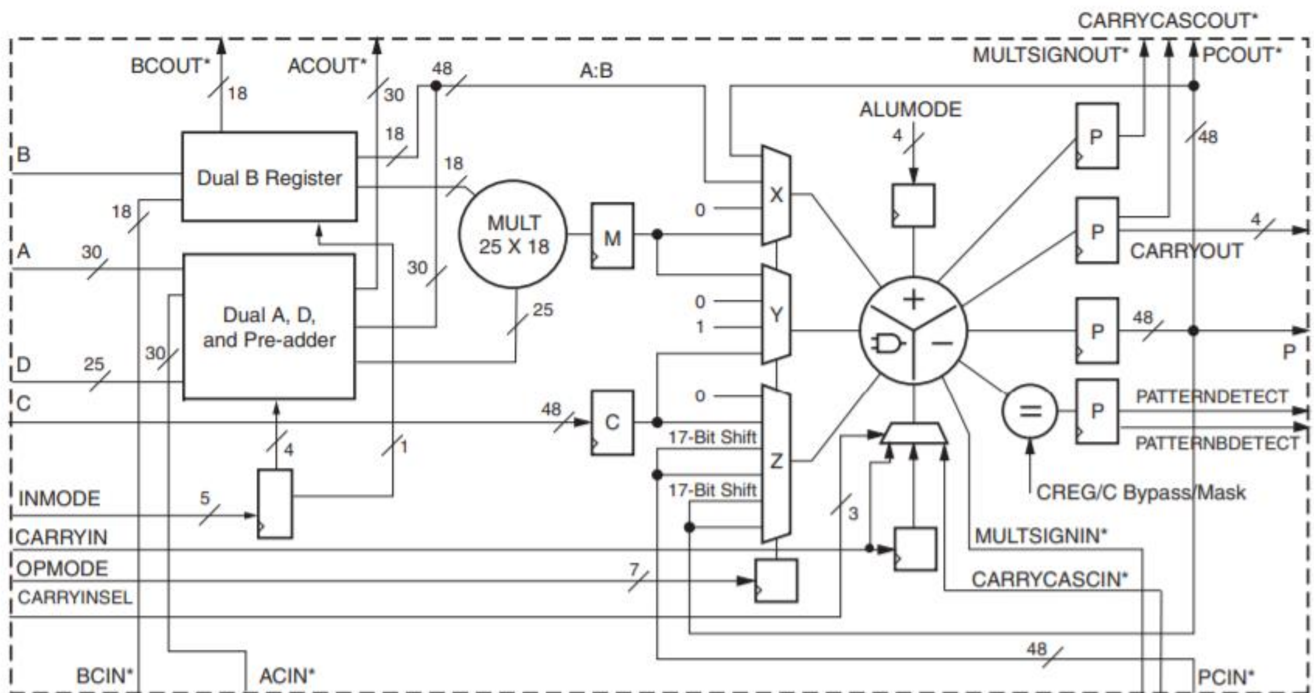

Figure 43: Input output configurable block

Digital Signal Processors (DSPs), are another common type of core that is offered as an IP core or an embedded core. These are essentially specialized processors that are used for manipulating analog signals. They are commonly used for filtering and compression of video or audio signals, Multiply-Accumulate block or MAC is implemented as DSP slice and MAC is mainly used as a building block for complex DSP applications.

4.2.5 Block Ram

Block ram is a Pulk of storage space on an FPGA to save data without using internal LUTs of FF (flip flops). block RAM is slower than the ff based memory but faster than off chip ram but smaller than off chip in size. Block Ram has more than one options to implement your own ram in series 7 one port or 2 ports can be used by different sizes distributions $32Kx1, 16Kx2, 8kx4$,$4kx9$, $2kx18, 1kx36$.

4.3 FPGA digital design flow

1. **Model implementation:** creating a python model to model the function of the accelerators and to use in verification.
2. **System design:** a system design schematic is painted with all signals and blocks we need.
3. **HDL:** hardware description language to specify the functionality of the model as hardware.
4. **Functional verification:** writing a test bench to make sure that output of model = output of accelerator.
5. **Syntheses, implementation and STA check:** running syntheses tool to make sure that the block implemented as designed and STA to make sure that operating frequency met the specifications.
6. **Place and route:** placing cells and routing the interconnection matrices between LUTs.

4.4 MobileNet accelerator design

4.4.1 First Design Approach

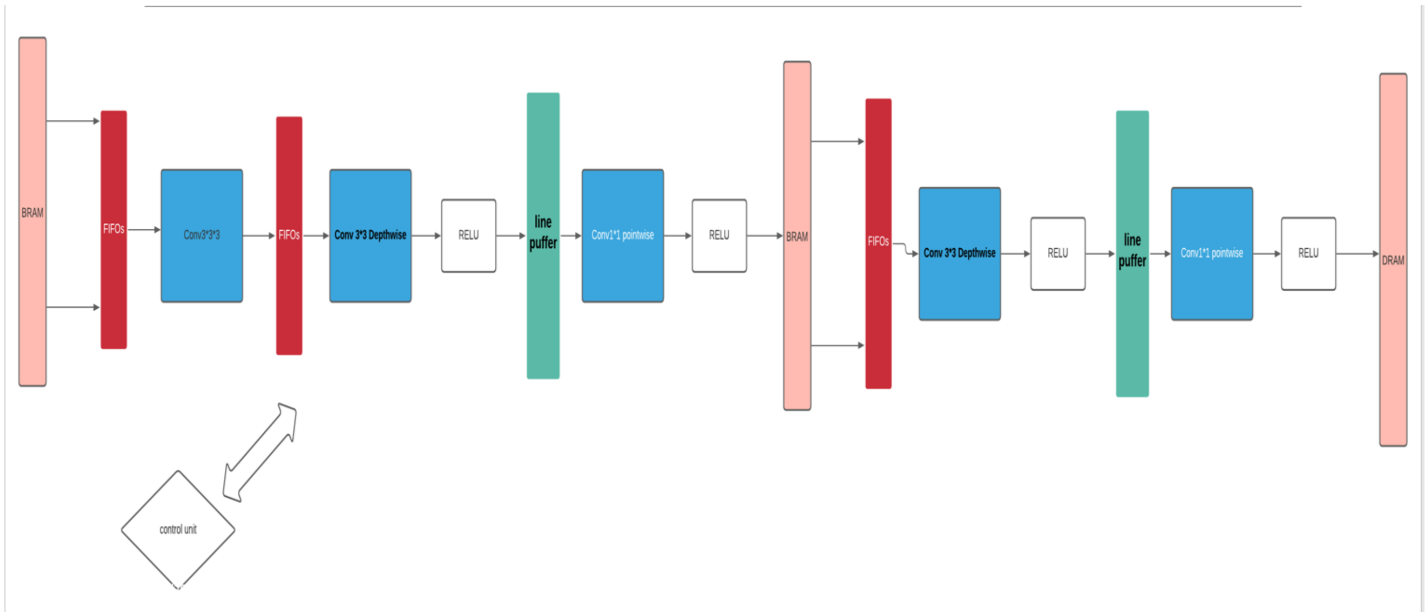

Figure 44: Layer pipeline architecture

Our first approach to design a 13 layer MobileNet is to pipeline the layers with each other's which will lead a high throughput and frames per second but will use a high memory access because 13 layer will access the BRAM in the same time which can make overutilization in the BRAMs in FPGA (1740 BRAM) and from LUT utilization point of view duplication of all modules 13 times will make over utilization in LUTs number shown in figure (45), so this

Part	I/O Pin Count	Available IOBs	LUT Elements	FlipFlops	Block RAMs	Ultra RAMs	DSPs	Gb Transceivers	GTPE2 Transceivers
xc7vx690tffg1926-2	1926	720	433200	866400	1470	0	3600	64	0
xc7vx690tffg1926-2L	1926	720	433200	866400	1470	0	3600	64	0
xc7vx690tffg1926-1	1926	720	433200	866400	1470	0	3600	64	0
xc7vx690tffg1927-3	1927	600	433200	866400	1470	0	3600	80	0
xc7vx690tffg1927-2	1927	600	433200	866400	1470	0	3600	80	0
xc7vx690tffg1927-2L	1927	600	433200	866400	1470	0	3600	80	0
xc7vx690tffg1927-1	1927	600	433200	866400	1470	0	3600	80	0
xc7vx690tffg1930-3	1930	1000	433200	866400	1470	0	3600	24	0
xc7vx690tffg1930-2	1930	1000	433200	866400	1470	0	3600	24	0

Figure 45: Series 7 FPGA resources

architecture may be a good choice in ASIC design or if this design will be prototyped on an emulator (Array of FPGAs) not one FPGA.

4.4.2 Second Design Approach

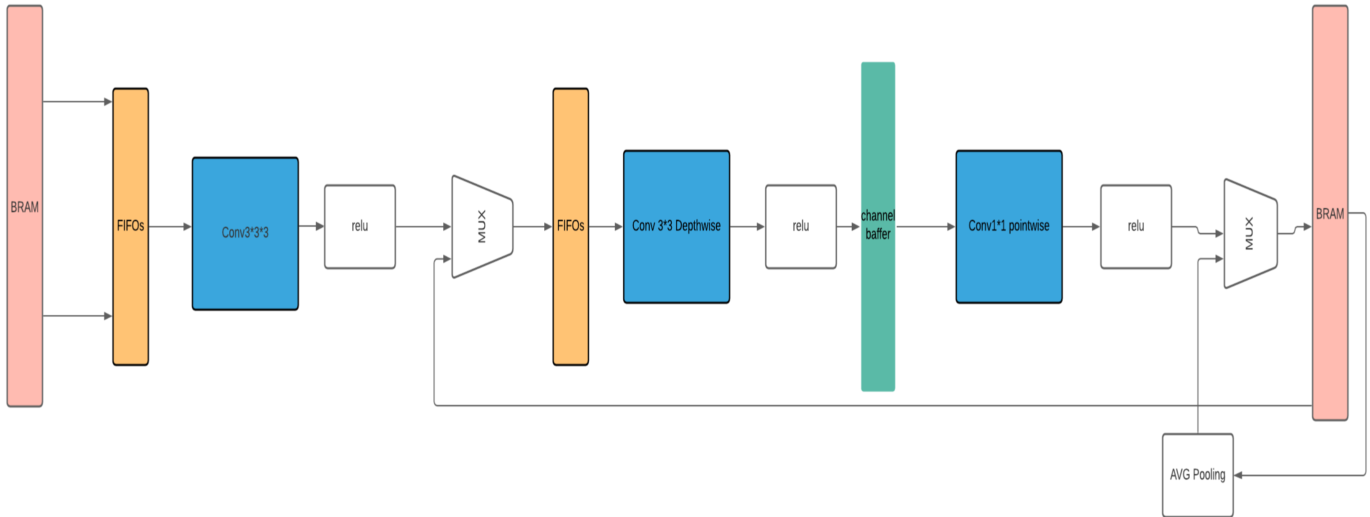

Figure 46: Shared layer architecture

Second approach is to implement one shared layer of convolution with all unique modules 1module stander conv, one module pointwise, one module Depthwise, Relu and average pooling this architecture will be configurable layer so weights and sizes will be changed according to the number current layer the shared layer designed for high-speed target so high parallelism across filters and across channels implemented in the modules. Shifters arrays with some modifications are used to fetch the input window from the input futures correctly without any repetition in the memory data shifters will be discussed in detail in the next parts. After 13 iterations in the accelerator one average pooling will occur.

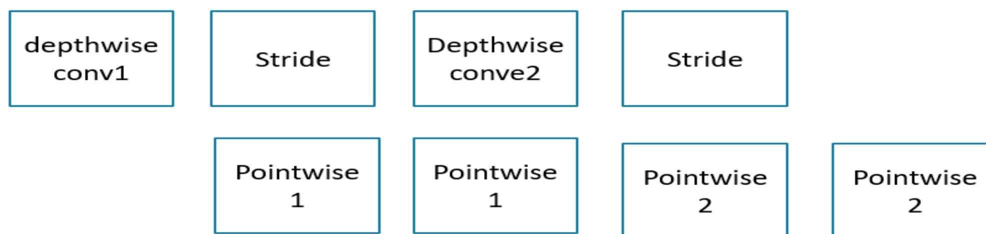

Figure 47: Operation overlapping between depth and point wise modules

An overlap will occur between depthwise block and pointwise block due to point wise taking 1 channel from depthwise so when 1 channel is ready at input of pointwise it will be calculated as shown in figure (47).

4.4.3 Shared Layer approach

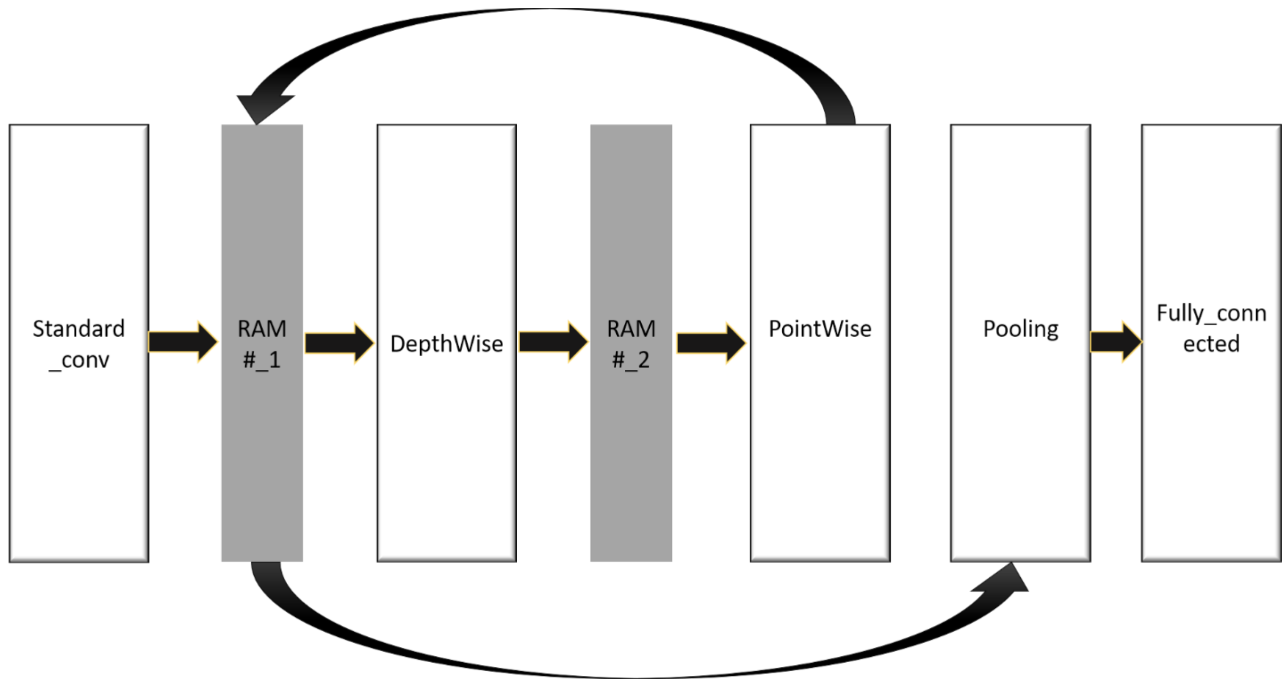

Figure 48: MobileNet accelerator Block diagram

In this approach, one layer of Hardware is implemented and we loop on this layer until the 7 layers are executed. As discussed previously MobileNet has 2 types of convolutions: standard convolution and depthwise separable convolution that includes 2 steps: depthwise convolution and pointwise convolution. So, there are 3 main blocks which execute the two types of convolutions. Firstly, the image entered to the standard convolution block that execute standard convolution and store the output in RAM#1 that consist of 512 block rams but only 32 of them are used in the first layer (standard convolution). Then after standard convolution finishes. depthwise shifter's starts to fetch the input from RAM#1 and stores the result in RAM#2. After Depthwise finishes, the PW buffer starts to fetch the input from RAM#2 and stores its output in RAM#1. Then DW and PW will loop until the 7 layers are executed. then the poling layer fetches the input from RAM#1 and hence the fully connected layer takes the 512 outputs from the pooling layer output buffer and executes FC operation. Finally, the FC layer provides the 43 classes.

4.5 Shifter Block

4.5.1 How multiplication is done in hardware

For simplicity assume we have input 6*6 image how we will fetch values to multiply it with weights in stride one case and stride two case.

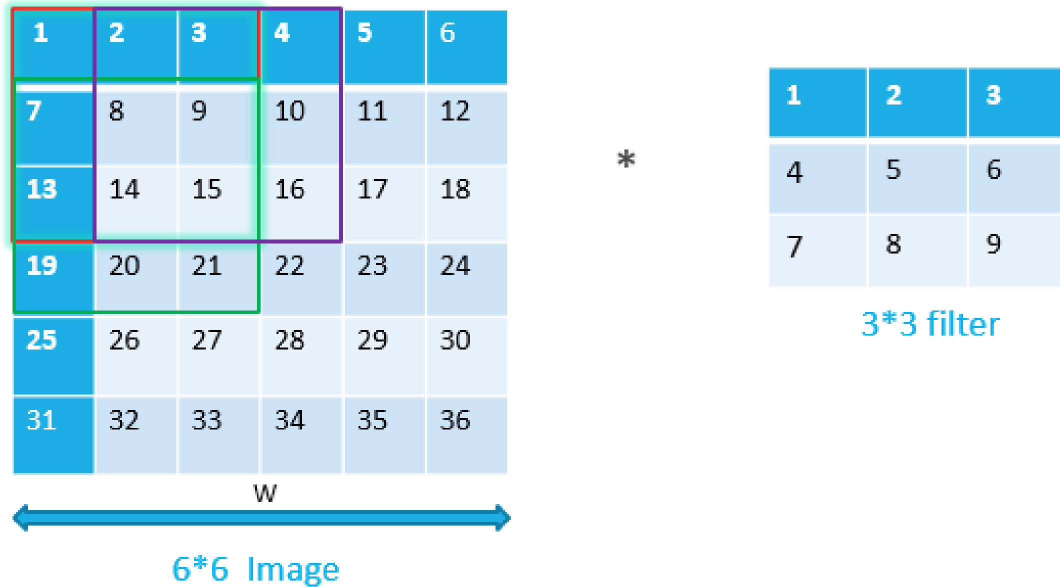

*Figure 49: Image 6*6 and filter 3*3 multiplication*

Fetching values (1 ,2 ,3 ,7 ,8 ,9 ,13 ,14 ,15) to be multiplied with filters will be huge issues cause all hardware depend on this multiplication and what makes it more difficult how we will make stride one for example to get (2,3,4,8,9,10,14,15,16) and following values that will be covered by square as shown in the figure (49) so let's see the following solution.

For first time it's simple solution to duplicate the values in Rams when write it from previous stage or input image but this will duplicate needed memory with factor of 2 with some optimization can be 1.5 but still a huge cost which will affect all the design without any benefit in speed or even

power and real example if image is 32*32*3 as our model this will require ram with more than 1024 location for one channel which mean that we use all the Brams which provided by virtex-7.

```
// BRAM_TDP_MACRO: True Dual Port RAM
//                  Virtex-7
// Xilinx HDL Language Template, version 2018.2

/////////////////////////////////////////////////////////////////////////
// DATA_WIDTH_A/B | BRAM_SIZE | RAM Depth | ADDRA/B Width | WEA/B Width //
// ===============|===========|===========|===============|=============//
//     19-36      |  "36Kb"   |   1024    |    10-bit     |    4-bit    //
//     10-18      |  "36Kb"   |   2048    |    11-bit     |    2-bit    //
//     10-18      |  "18Kb"   |   1024    |    10-bit     |    2-bit    //
//      5-9       |  "36Kb"   |   4096    |    12-bit     |    1-bit    //
//      5-9       |  "18Kb"   |   2048    |    11-bit     |    1-bit    //
//      3-4       |  "36Kb"   |   8192    |    13-bit     |    1-bit    //
//      3-4       |  "18Kb"   |   4096    |    12-bit     |    1-bit    //
//        2       |  "36Kb"   |  16384    |    14-bit     |    1-bit    //
//        2       |  "18Kb"   |   8192    |    13-bit     |    1-bit    //
//        1       |  "36Kb"   |  32768    |    15-bit     |    1-bit    //
//        1       |  "18Kb"   |  16384    |    14-bit     |    1-bit    //
/////////////////////////////////////////////////////////////////////////
```

Figure 50: Bram in virtex- 7

Another solution is to remain location of rams as its but change the address to get the required values to be multiplied which is a complex circuit to implement with disadvantage to make one multiplication with filter every 5 cycles at best cases assuming the additional circuit will not require additional cycles.

One common problem in two solutions discussed how padding will happen in memories which will take an overhead location in memories.

4.5.2 Shift Register with one Controller

This idea will put a number of registers with the following equation

$$\#Regs = 2 * W_{width\ of\ image} + 3 \qquad \text{(Eq7)}$$

- Taking an example of 6*6 image will be 8 *8 cause of padding:

0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	0
0	7	8	9	10	11	12	0
0	13	14	15	16	17	18	0
0	19	20	21	22	23	24	0
0	25	26	27	28	29	30	0
0	31	32	33	34	35	36	0
0	0	0	0	0	0	0	0

Figure 51: Input Image with required Padding

The multiplication is done as following:

- Global Reset will arise to all the design including the register this will help in padding first row of padding and first zero in second row 9 zeros will get in registers so need to wait 9 cycles as was done in memories to fetch these values cause all registers here have an output which will be chosen.

8
7
0
0
6
5
4
3
2
1
0
0
0
0
0
0
0
0
0

Figure 52: Values in Registers

- Then Values will get from Bram and will be shifted until 8 will get in first register the situation now as in figure (52) the red numbers will be multiplied with filters connected to multipliers this registers always have the right values to be multiplied with filters weight.

- Register's index which multiplied values will be in can be calculated from next equations:

$$First\ 3\ Reg : Reg[0], Reg[1], Reg[2] \qquad \text{(Eq8)}$$

$$Second\ 3\ Reg : Reg[W+2], Reg[W+3], Reg[W+4] \qquad \text{(Eq9)}$$

$$Third\ 3\ Reg : Reg[2*(W+2)], Reg[2*(W+2)+1], Reg[2*(W+2)+2] \qquad \text{(Eq10)}$$

- Counter padding is needed to determine location of zero padding between values and the condition on it is easy when counter reach to image width padding will happen so the first register has to mux select between values and zero. Note that counter width is $\log_2 imagewidth$ so the counter will reset after reaching the end of row and start count in the new row. Padding is a signal received from the module controller to free memory address. to avoid skip values padding must come one cycle advance so freeze the address then the controller takes a signal called padding advance.

- Stride one required get one value each cycle but in stride two we can get 2 values in one cycle and make shifts with two in registers not one so stride is a signal coming from the controller depending on the layer.
- Ready signal will get out from the module to tell the main controller that the right values are in registers so multiplication can be done. Logic of it is depend on counter padding and require width counter count to

$$3 * \ (W_{width\ of\ image} + 2_{padding\ in\ each\ row}) \qquad \text{(Eq11)}$$

Which is needed to a lot of reasons:

1- like to differentiate between when the ready signal is up in stride one and strid two.
2- Module takes input signal called End size which means that memory address has reached to its end so the last zero row must get in. the width counter will count this row in stride one. No need for it in stride two.

- Padding counter and width counter in stride one will be incremented with one while in stride 2 will be incremented with two.

- Padding counter and width counter in stride one has Following Reset condition

```
//Stride one Rst condition
if (counter_padding == widthselctioncounter+1)
counter_padding<=0;
else
counter_padding<=counter_padding+1'b1;
if (counter_width3 == 3*(widthselctioncounter+2) )
counter_width3<=(widthselctioncounter+4) +widthselctioncounter;
else
 counter_width3<=counter_width3+1;
```

```
//Stride two Rst condition
if (counter_padding == widthselctioncounter)
counter_padding<=0;
else
 counter_padding<=counter_padding+2'b10;
if (counter_width3 == 3*(widthselctioncounter+2))
counter_width3<=widthselctioncounter+4;
else
 counter_width3<=counter_width3+2'b10;
```

-Note when global Reset come padding counter will start from zero and width counter will start from $W_{image} + 2$ as the first row of zero gets registered.

- Difference in Reset condition comes from in strid two last zeros column and row will not be added to image but due to generality of the module it will be added in register to don't shift the places where we get output from registers to eliminate this effect, we will modify counter start conditions and Ready conditions in stride two as discussed before.

-width selection counter is a signal coming from the main controller to make the module work on different widths like 32 ,16,8. our image 32*32*3 so modules will be added in standard convolution. module be in depth wise also work on different width 32 ,16 ,8 according to layers we operate in determined by controller signal control select.

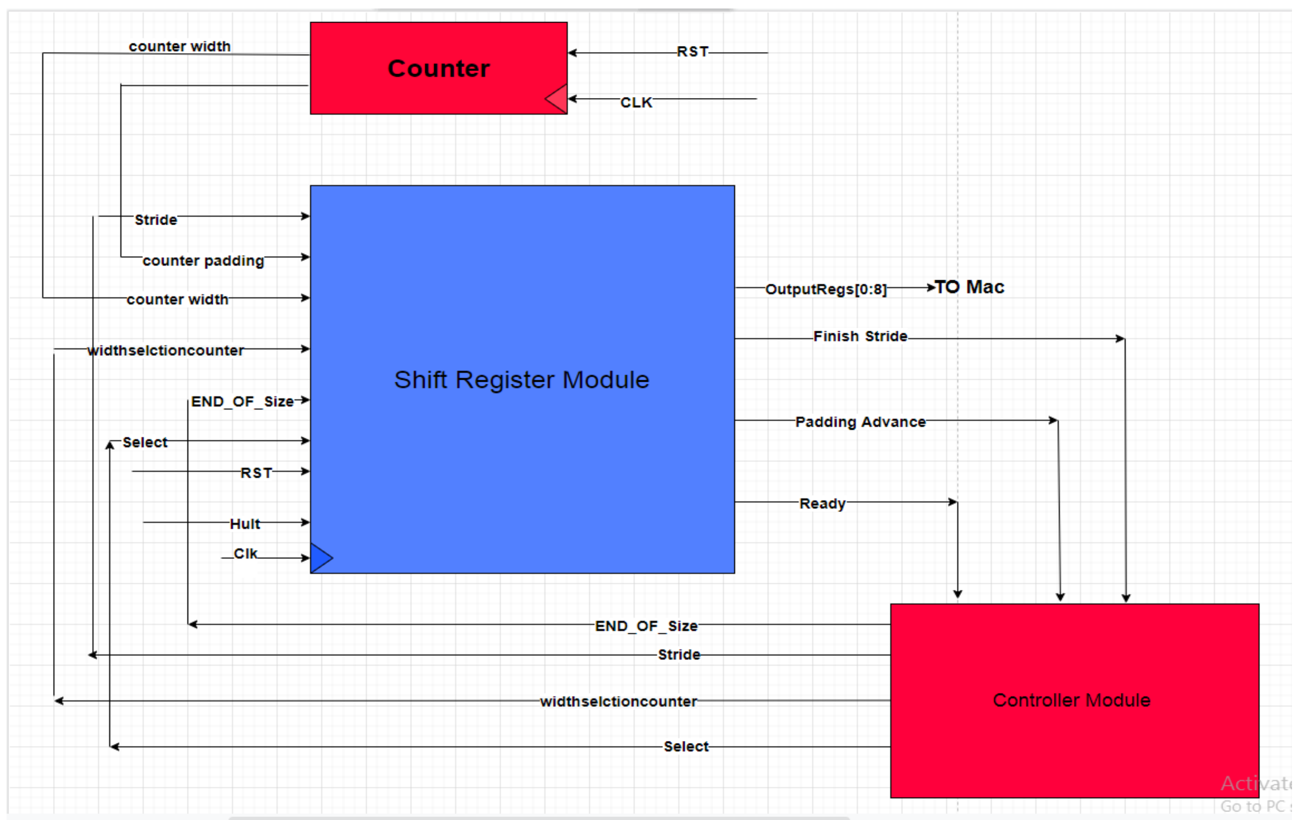

Figure 53: One instance of module

4.5.2.1 Shift Registers Advantages

- One controller can control all the modules of the shift register because all modules will get data synchronized then the 2 counters will be connected to all instances of this shift registers which isn't a huge logic.
- Speed, especially in stride two and in padding time

- Operation Flexibility from layer to layer.

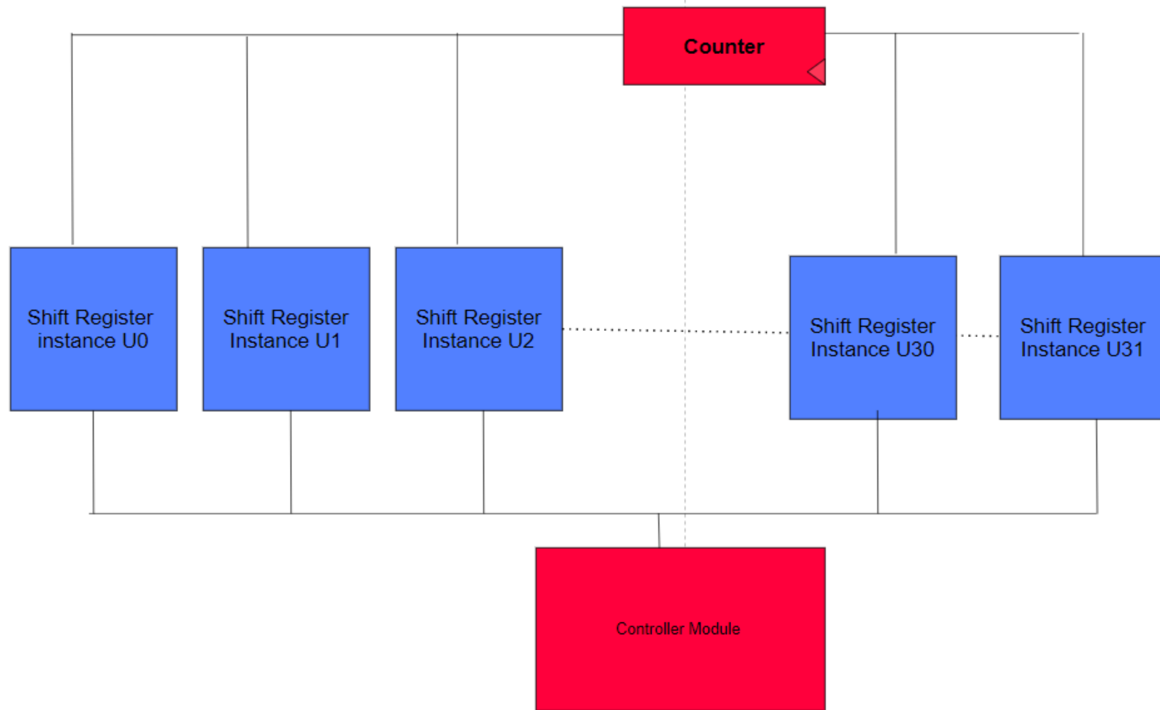

Figure 54: 32 instances controlled by one counter and controller

4.6 Adder Tree Block

4.6.1 Look ahead with carry save adder

Truth table of carry look ahead adder

A	B	Ci	C i +1	Condition
0	0	0	0	No carry generate
0	0	1	0	
0	1	0	0	
0	1	1	1	No carry propagate
1	0	0	0	
1	0	1	1	
1	1	0	1	Carry generate
1	1	1	1	

Figure 56: generation and propagation truth table

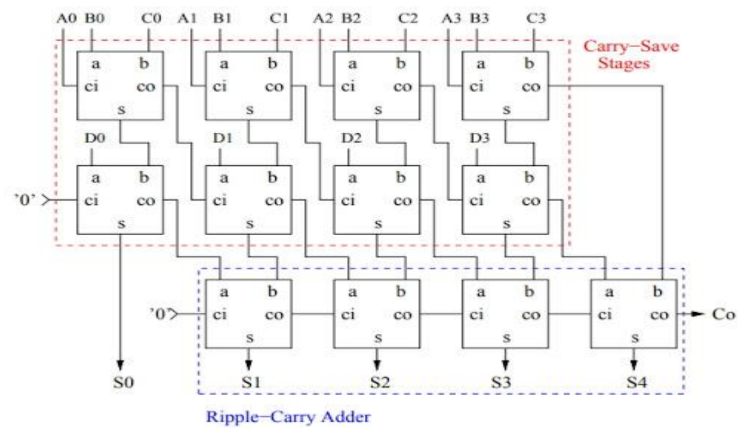

Figure 55: Carry save block

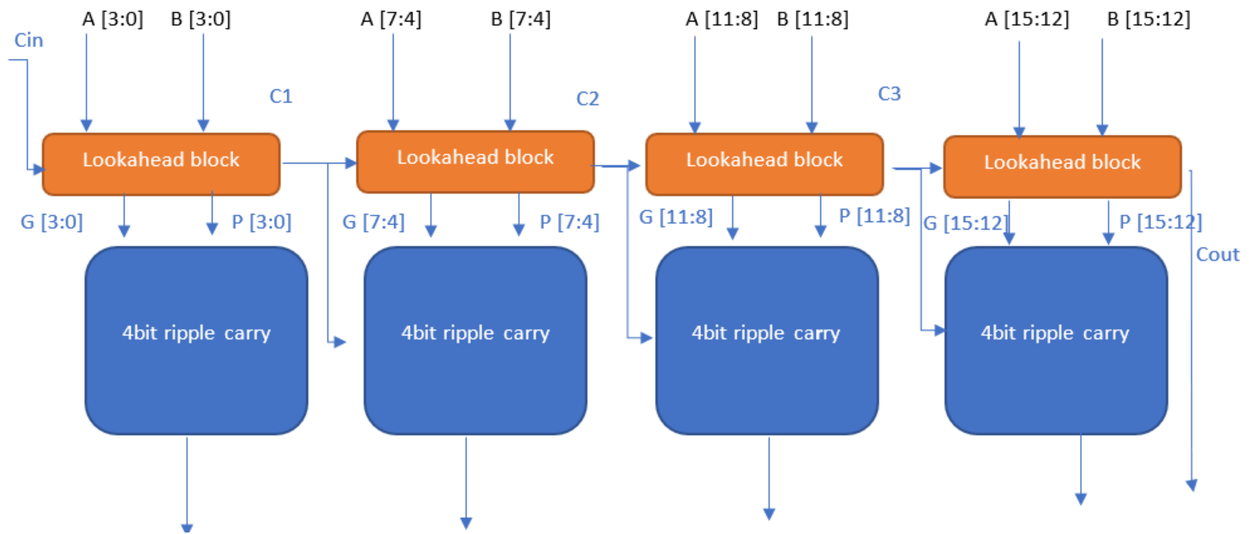

Figure 57: carry lookahead block diagram

The main part in CNN design how to sum wights outs from multiplier, the DSPs modules in FPGA has one adder to accumulate the coming input and add it to data registered in the DSP but if parallelism techniques is applied for enhance the block speed, more than one DSP will use in the same time so the adder in the DSP block will be useless so, fast adder tree is needed to design to guarantee the applied parallelism.

Adder tree is a tree of fast adders cascaded with each other to sum more than 2 numbers. In our design we need to add 9,27,32 numbers with each other at the same time.

Tree is consisting of fast adders and pipeline hierarchy; the pipeline is chosen to make the critical path meet the design constraints. In the next section the fast carry lookahead adder basic unit will be explained.

Look ahead Adder: is based on the propagation and generation techniques to predict the carry output value without waiting for the cascaded chain the propagation (p) and generation (g) truth table shown in figure (55).

the equation of P and G from truth table are

$P_i = A_i \; XOR \; B_i$ (1)

$G_i = A_i \; AND \; B_i$ (2)

So, the full adder can be modified to calculate same and carry from G and P parameters and the equation of sum and carry will be

$SUM = C_{i-1} XOR\ P_i$ (3)

$C_i = G_i + P_i\ AND\ C_{i-1}$ (4)

This truth table need a prediction logic unite to predict “g” & “P” for each adder but the critical path will not be improved if a prediction block added to each full adder this design will be ended by ripple carry adder so the N bits adder is divided to sub adders cascaded and the prediction logic block is added to each group this will improve the critical path and improve speed the block diagram of 16 bits lookahead adder is shown in figure () .G and P carry out equation for 4 bits will be as shown in (5).

$C_4 = G_3 + P_3G_2 + P_3P_2G_1 + P_3P_2P_1G_0 + P_3P_2P_1P_0C_0$ (5).

Carry save adder: carry save adder is important block in adder tree because we need to add more than 2 numbers with each other’s so we need to map 4 number of N bits to tow numbers of N bits to go through look ahead adder so we need carry save adder simply carry save adder is array of full adders with carry save to the next stage without carry probations between full adders like ripple carry adder carry save adder based on using the input carry of full adders as a third input so it can add 3 items in the same time the output of carry save will be 2 numbers of N+1 bits needs to lookahead block to add this numbers and calculate the N+2 bits the correct length of adding 3 numbers of N In the proposed design carry save adder adds 4 inputs at the same time using 2 levels of carry save and carry lookahead adder.

4.6.2 Pipelined adder tree

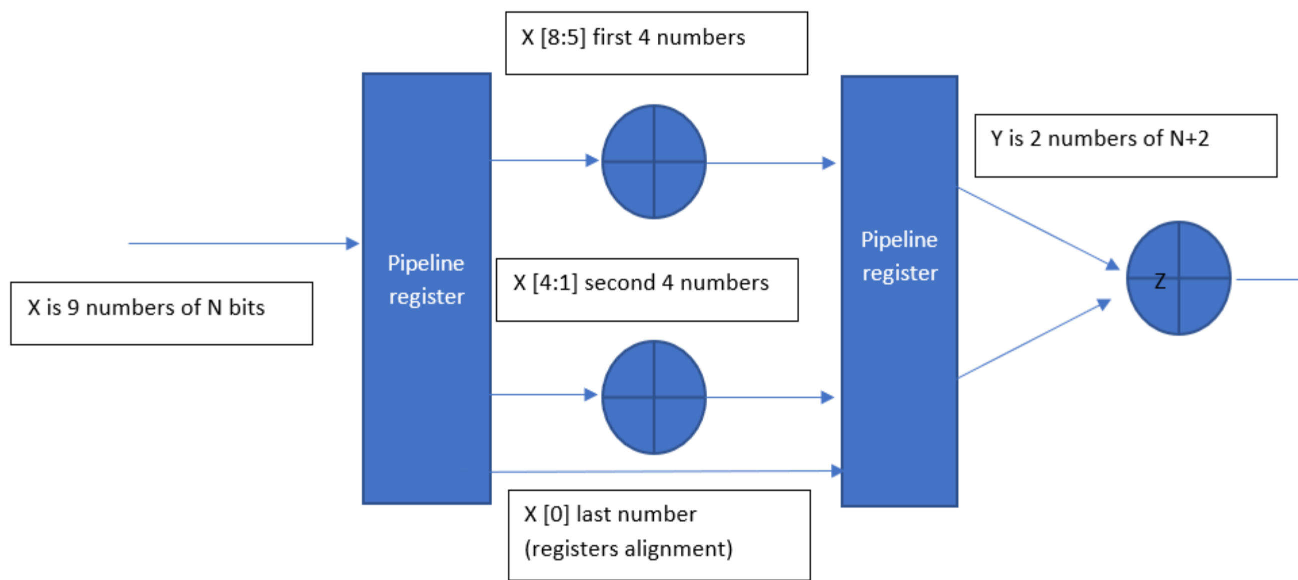

Figure 58: pipelined carry lookahead adder tree adding 9 numbers

Adder tree as shown in figure (58), for example adding 9 numbers in 3 clock cycles using 3 adder blocks explained in the previous section pipeline techniques is applied because the data will change each cycle and to reduce the critical path

4.6.3 Adder tree overflow issue

Due to 8 bits integer representation the proposed based on output of layers must be 16 bits but the adder bit extend equation is $N + \log_2 adder_num$ so if 8 bits are clipped from adder tree output, the output will suffer from overflow error. Overflow error is that the output should be negative but output is positive and so on the solution of overflow issue is to represent data in bits greater than its maximum if numbers can represent in 8 bits if it added using adder of 9 bits adder no overflow will occurs and the most significant bit will be the correct sign bits and sign extension must be added to match the next adding stage in the adder tree this will make overhead in the hardware adder tree but no output error will occurs that means no flow frailer

4.7 Standard Convolution

It is the first layer and only of its type in this architecture, it is used to capture a large number of features of the input image. The input of this layer is (32,32,3) and the output is (32,32,32) with stride 1 and a padding layer.

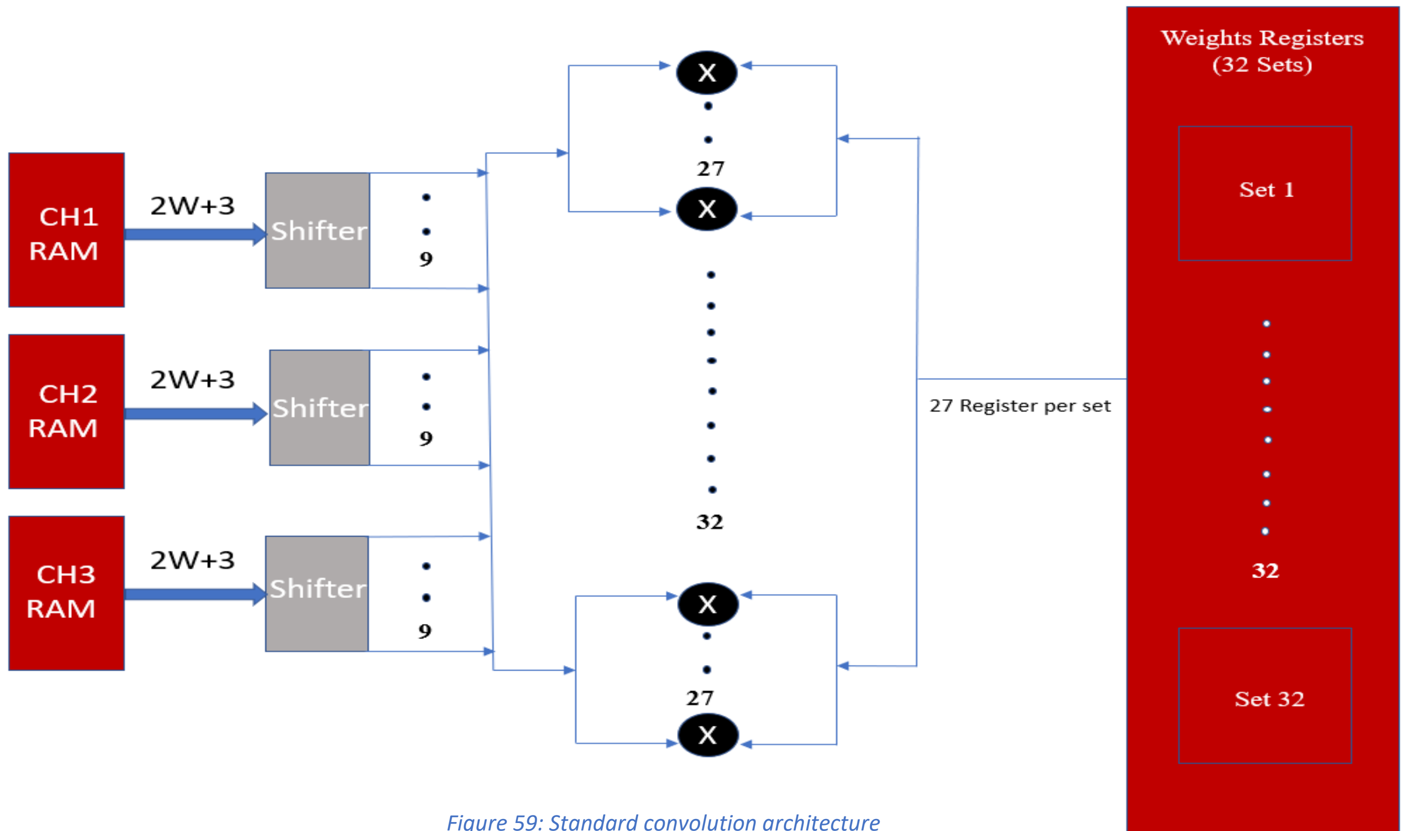

Figure 59: Standard convolution architecture

As shown in figure (59), we need to generate 32 output channels for every pixel, so we will use 32 sets of macs to generate all of these channels in a single clock cycle in order to reduce latency of accessing memory of the input image. We used 27 macs per set to perform the whole input image multiplications in a single cycle to increase layer speed, so we have 864 macs in this layer performing as a multiplier. We have shifters to perform the required stride and zero padding layer, these shifters operate synchronously as every shifter makes a stride every clock cycle after a first ready signal is set to high. Weights are separated into 32 sets; every set contains 27 registers. The width of multiplier output is 16-bits because weights and input feature map are integer numbers represented in 8-bits. We have 3 input rams to access all channels simultaneously to decrease accessing of a single ram and to provide suitable utilization.

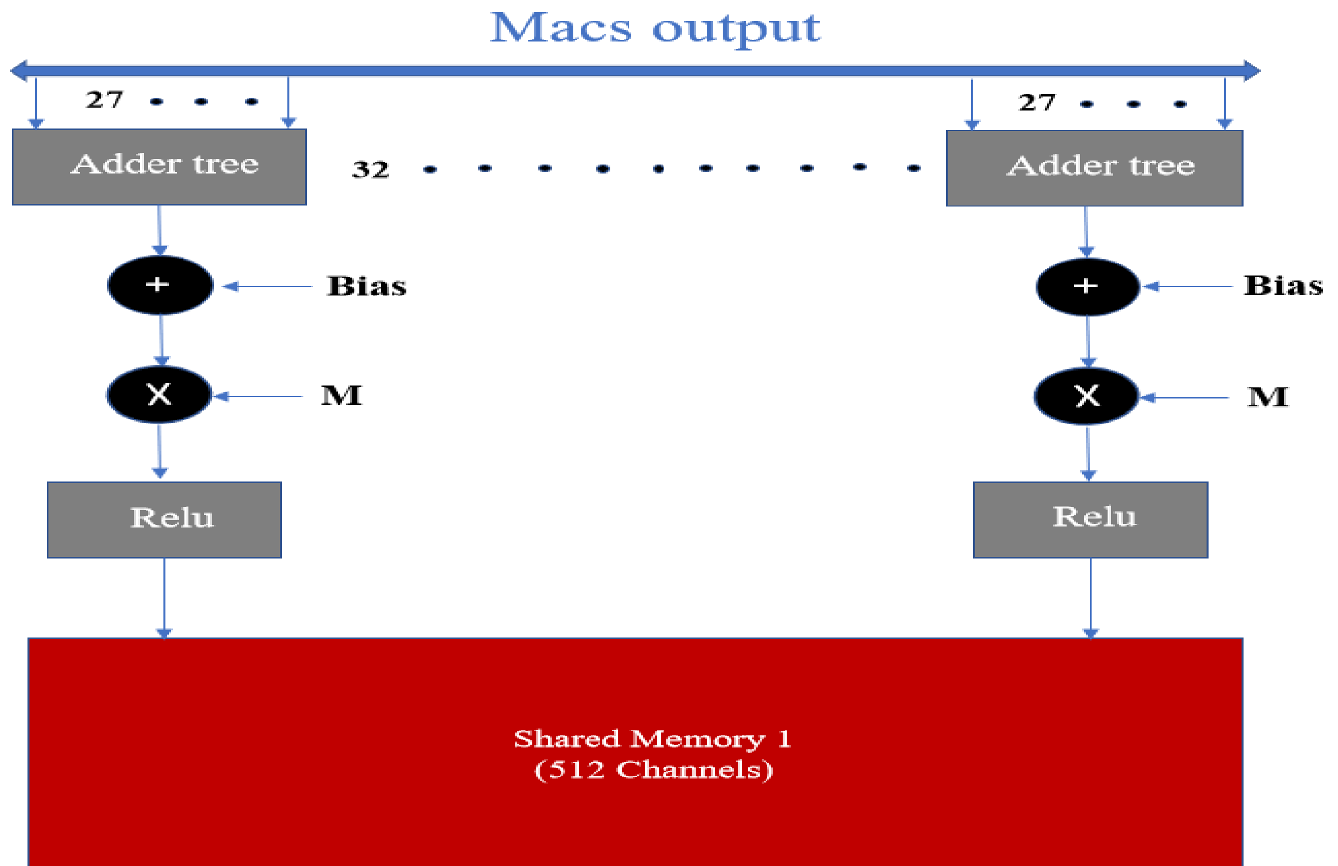

Figure 60:Standard convolution adder tree

As shown in figure (60), the output of mac array is passed to 27 pixels adder tree to be added together then the result is sign extended to be added to 32-bits bias then multiplied by a fraction M represented in 16-bits then stored in shared memory 1. The width of the multiplier is 32-bits and the result should be clipped to be represented in 8-bits, so we will take the integer value bits which are from bit 15 to bit 21 in addition to the sign bit which is bit 31. Relu is used to eliminate negative numbers.

4.8 Depthwise Convolution

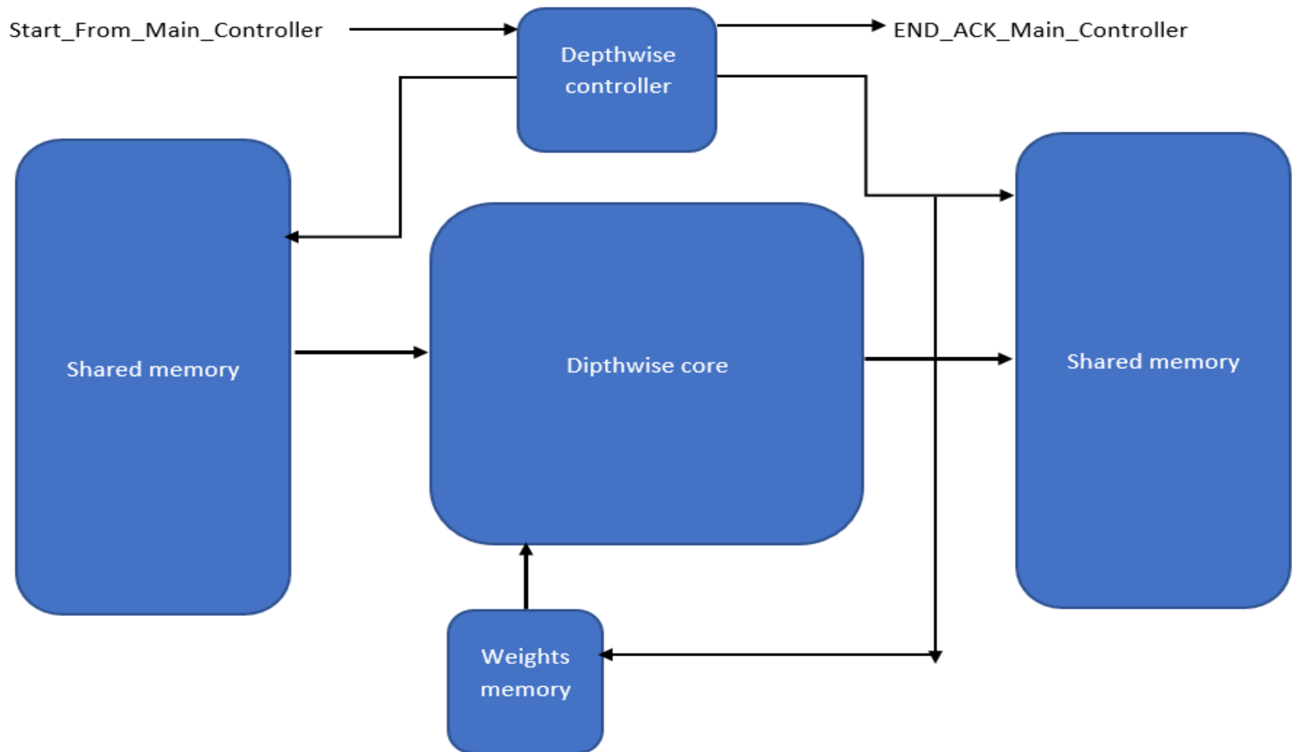

Figure 62: Depthwise block diagram

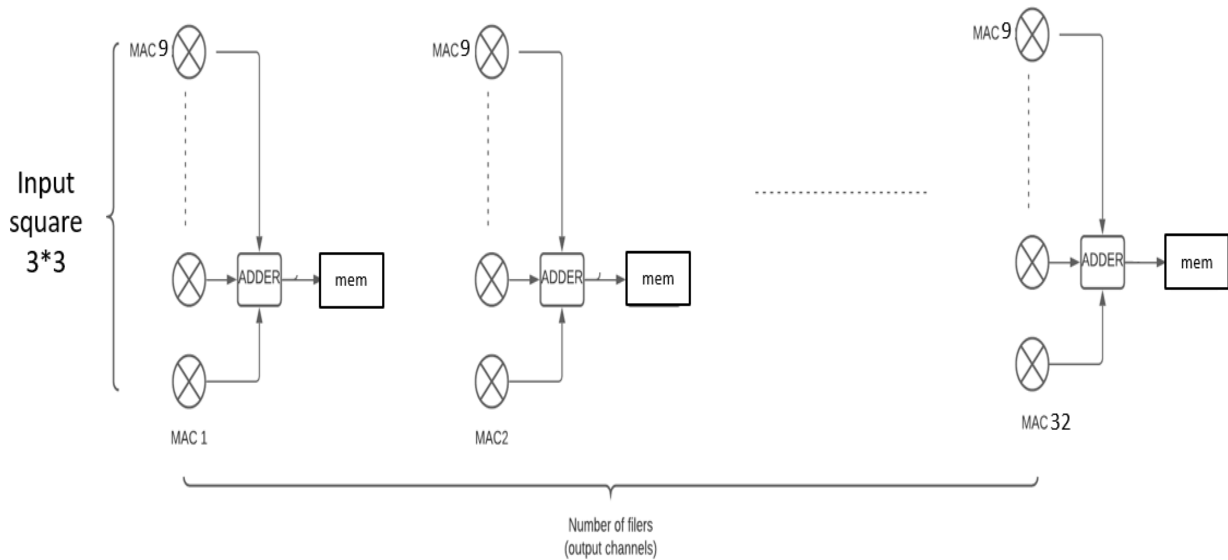

Figure 61: Depthwise core parallelism

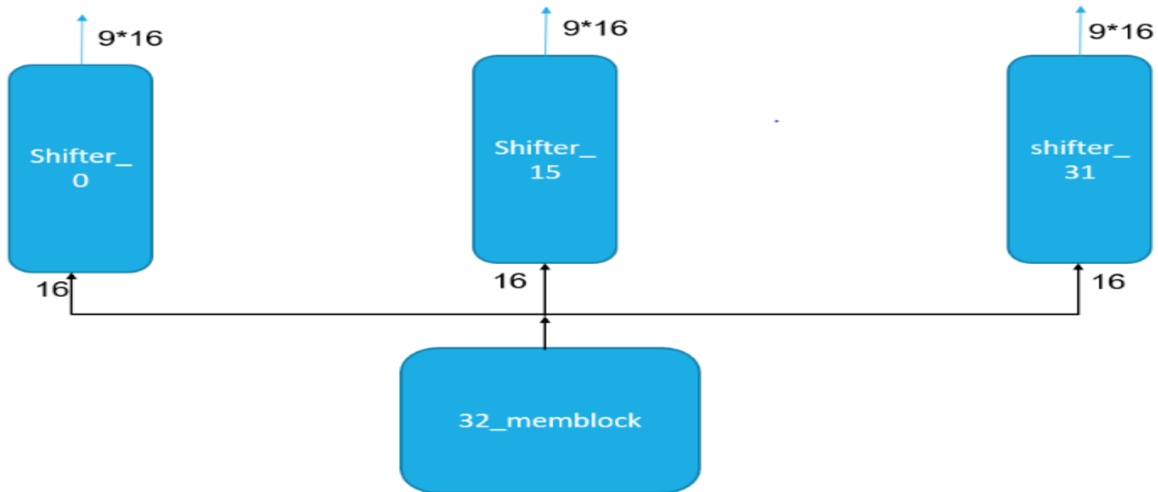

Figure 63: Memory weights fetching

Depthwise convolution the first block in separable convolution, Depthwise conv is 3*3 conv on the input channel only so the output channel from the depth conv layer is equal to the input channel, block parallelism is 9 across filter and 32 across channel as shown in figure (62), this parallelism will lead high throughput, but it can be faster if parallelism across channels in increased but FPGA resources will limit this enhance.

4.8.1 How Depth wise fetch data from memory?

CNN input features map fetching is big challenge in CNN accelerators due to difference in pixels positions during stride so the first solution approach is to make a complex arithmetic block to calculate the next step address which is very complex and will delay the design the second practical approach is a shift registers (FIFOs) to get all pixels which the DW core will use and by shifting pixels each clock means stride this solution will eliminate the complexity but will add some cycles overhead at each down stride operation until the window pixel fetched in the shift register , shift register implementation and control will explained in next section .

4.8.2 How Depthwise fetch weights from memory?

Depthwise conv is 9 parallelism across filter and 32 across channel, this rough parallelism need 9*32 Block Ram to be implemented to get 9*32 weights at the same edge clock so this is very bad utilization in memory ,so in the proposed design the input data shift registers overhead which is 10 cycles at best case weights need only 9 cycles to be fetched if 32 block ram only is implemented ,so the proposed design has 32 block ram for weights each ram has window of 9

weights in 9 address for first 32 channel and 9 for the second channel and so on this will make the utilization better than 32*9 block ram without any overhead added.

each ram input one weight per clock to shift reregister and after 9 clocks the 9 weights will be loaded into the shift register so 32 shift register is needed as shown in figure (63)

4.9 Pointwise Convolution

PW convolution is the combination stage of the DW separable convolution and is used to create a linear combination of the output of the depthwise layer.[13]

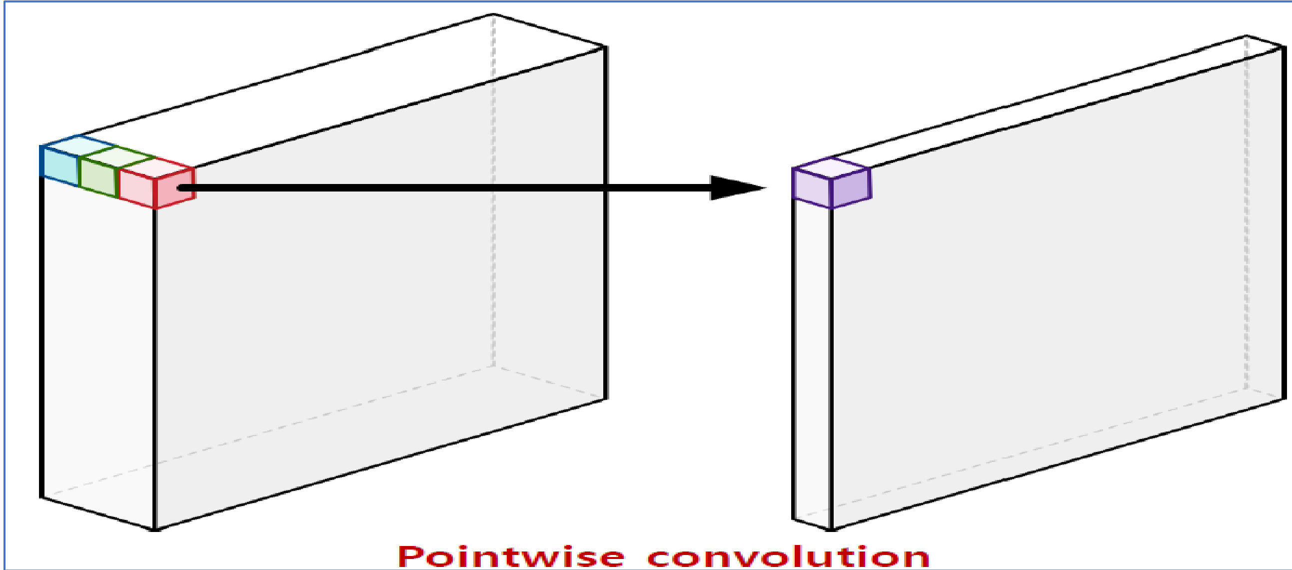

Figure 64: Pointwise convolution

As shown in the figure (64), pointwise takes the depthwise output and applies a linear combination on it. this happens by multiplying each channel by the corresponding filter weight and adding them together.[13]

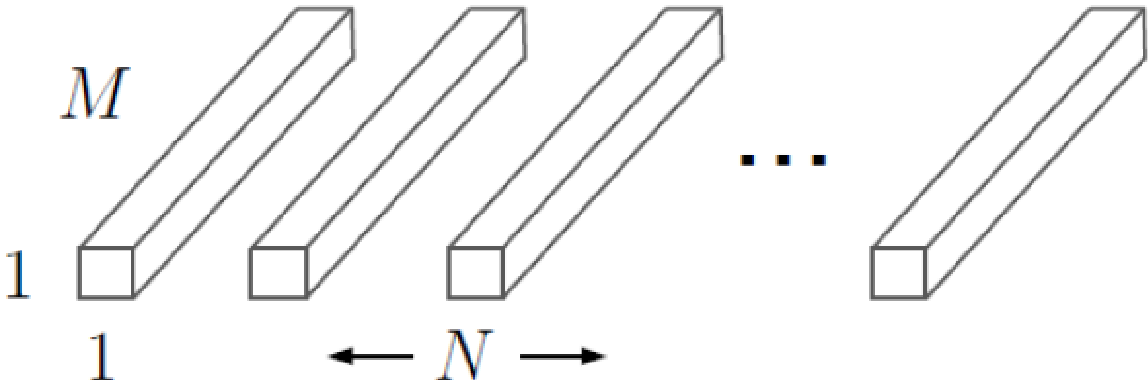

Figure 65: Pointwise filters

Figure (65) shows the pw filters architecture, Where N is the number of filters and also the number of output channels and M is the number of input channels.

4.9.1 Pointwise hardware complexity

Unlike the 3*3 convolution method, the pointwise convolution uses a large number of 1*1 operations So, a large number of multiplication and addition operations are involved in the design.

Y = A0W0 +A1W1 + …………. +AM-1WM-1 +AMWM +YN-1.[14]

Table 6:Resource per layer type

Resource Per Layer Type

Type	Mult-Adds	Parameters
Conv 1×1	94.86%	74.59%
Conv DW 3×3	3.06%	1.06%
Conv 3×3	1.19%	0.02%
Fully Connected	0.18%	24.33%

Table (5) shows that pointwise is the most complex by far from other layers. [13]

4.9.2 Pointwise hardware Structure

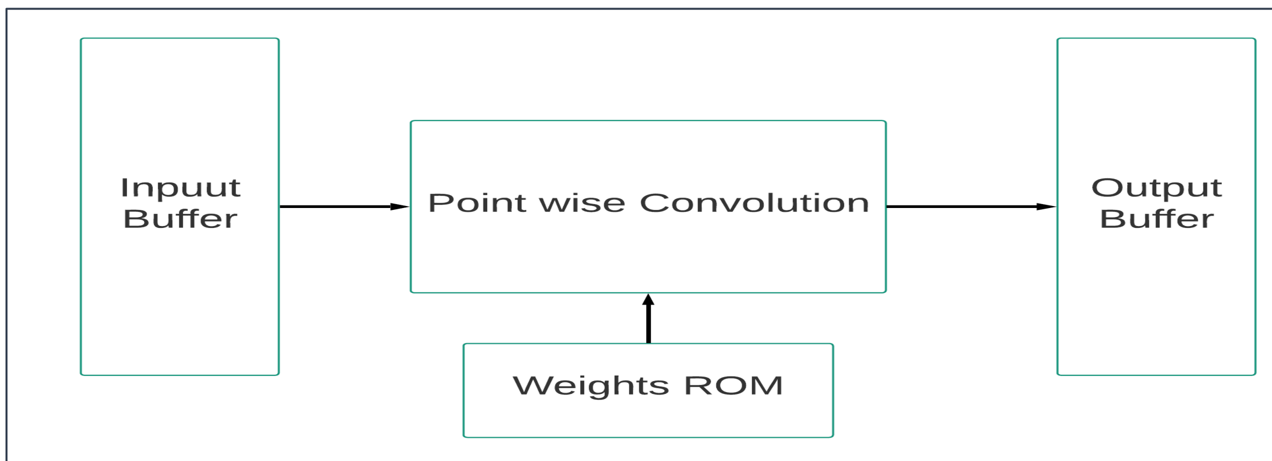

Figure 66: Pointwise hardware structure

The pointwise hardware structure shown in figure (66) consists of 4 main blocks: <u>input buffer</u> which contains depthwise output feature map and is fed as input to pointwise, <u>weights rom</u> that

provides kernels weights which are multiplied by the input channels, pointwise convolution this is the core block that perform the convolution operation through set of MACs (multiply and accumulate block exists in FPGAs) and the output buffer block which stores the output of the pointwise stage. Details of each block are discussed in the next sections.

4.9.3 Input Fetching

Figure 67: Input buffer

32 pixels are fetched from the RAM and stored in the shown buffer that feeds the stored values to each 64 sets in parallel to execute convolution.

4.9.4 Core Block: Pointwise Convolution

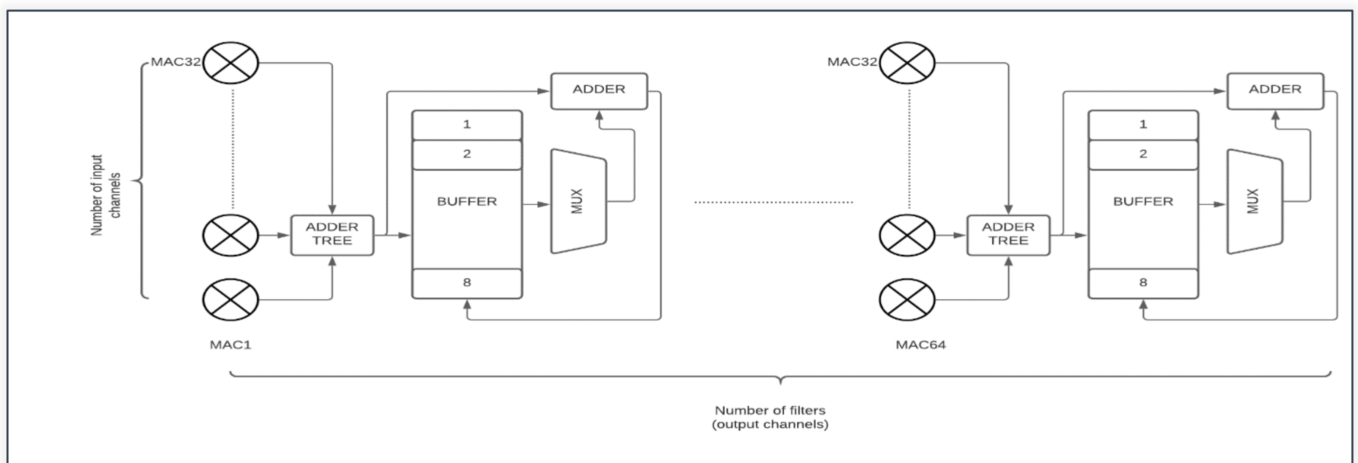

Figure 68: Convolution parallelism

Choosing the parallelism of filters is one of the challenges in MobileNet accelerators, as the MobileNet has an increasing number of filters as we go through the layers. so, if speed is targeted, a large number of MACs are required to achieve the required speed which is not always available specially in FPGA based projects.

In our design ,64 filters operate in parallel with 32 channel depth and the layers with more than 64 filters or more than 32 channel depths, time sharing is applied between each 64 filters and each 32-input channel. These layers (with more than 64 filters or more than 32 channel depths) take more than 1 cycle depending on the number of excesses of the applied resources. In the case of multiple cycles layers the result of the current set is saved in the BUFFERS to be added to the next set, an illustrative example below shows how this operation is done.

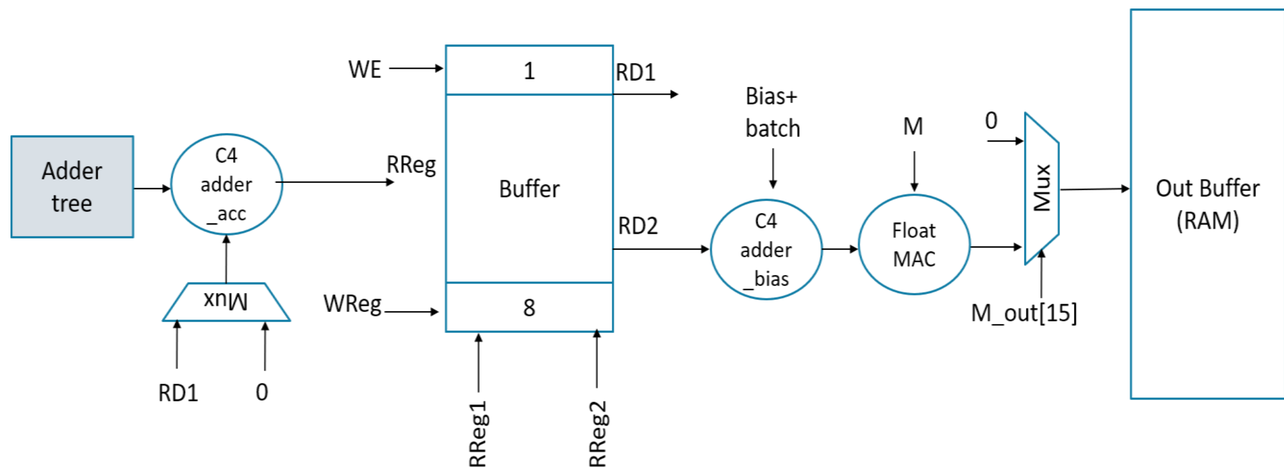

Figure 69: Pointwise sequence of operation

Figure (69) shows the detailed sequence of operation of the pointwise layer. The MACs outputs are added using the 32-adder tree then if accumulation is required -this determined by the point wise controller according to the layer number and the iteration inside the layer- the accumulation adder adds the adder tree output to the stored value in the buffer. Then after iterations are finished the bias adder adds the bias to the stored values in the buffer and finally the float MAC multiplies the result by the M parameter. Then the final values are stored in the RAM.

4.9.5 Weights Fetching

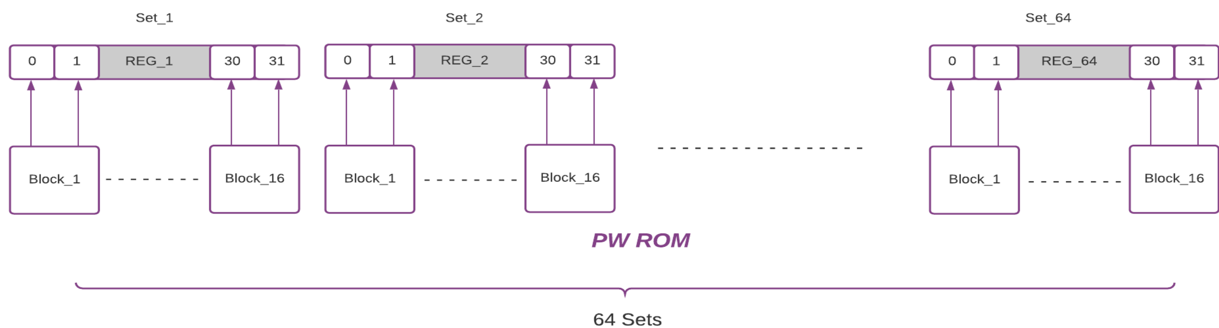

Figure 70: Weights fetching

Weight ROM is divided into 64 sets each containing 16 BRAMs (16 BRAMs? as we read by 2 ports, we can read 32 locations in parallel). Number of weights = 440,320 weights, Each BRAM contains 430 weights. Layers with more than 64 filters or more than 32 channel depths, weights are fetched from ROM in one cycle (due to 16*64 dividing).

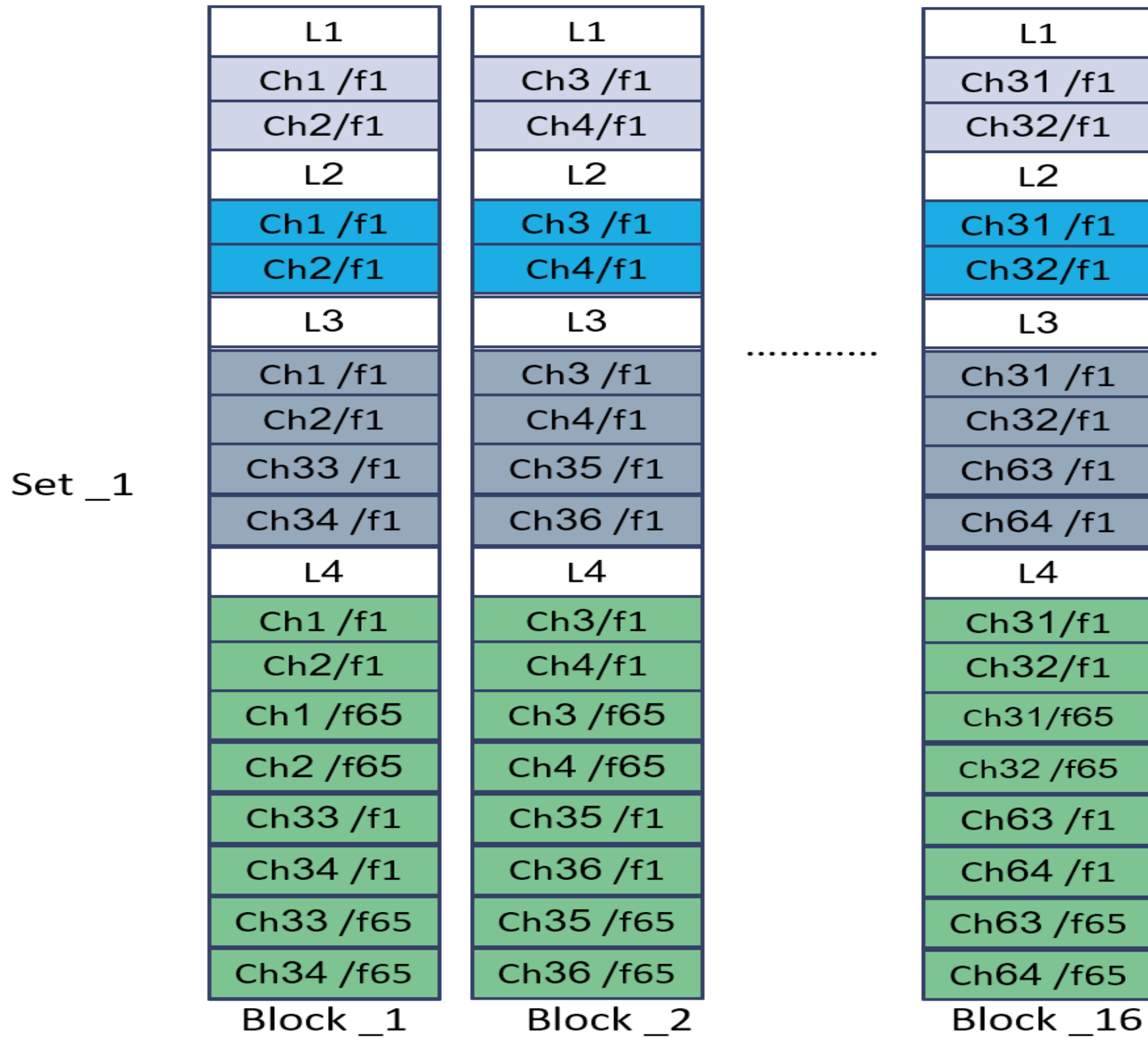

Figure 71: How weights are stored in ROM

Figure (71) shows how weights are stored in ROM.as we have 64 sets (filters in parallel),the first 32 filter's channels stored in 16 block for filter 1 to filter 64,then if a layer has number of input channels more than 32 say 64 we store the filter channels from 64 to 128 in the second 2 rows as shown for layer 3.if a layer has number of input channels more than 32 and number of output channels more than 64 (exceeds the papalism) as occurs in layer 4, firstly the first 32 channels stored in first 2 rows of 16 block for first 64 filters(from 1 to 64) ,then the first 32 channels stored in second 2 rows of 16 block for second 64 filters (from 65 to 128),then the next 32 channels (from 33 to 65) of the first 64 filters (from 1 to 64),are stored in the next 2 rows , and at last the next 32

channels (from 33 to 65) of the next 64 filters(from 65 to 128) are stored in the next 2 rows. And so on till the last layer.

4.9.6 PW Output Storage

To meet the set of FIFO's design discussed previously, we need to store each channel result in a single block. As the maximum dimensions are 512 for channel depth and 32 * 32 locations, we need a 512 block each one contains 32*32 locations as shown in the figure (72) below.

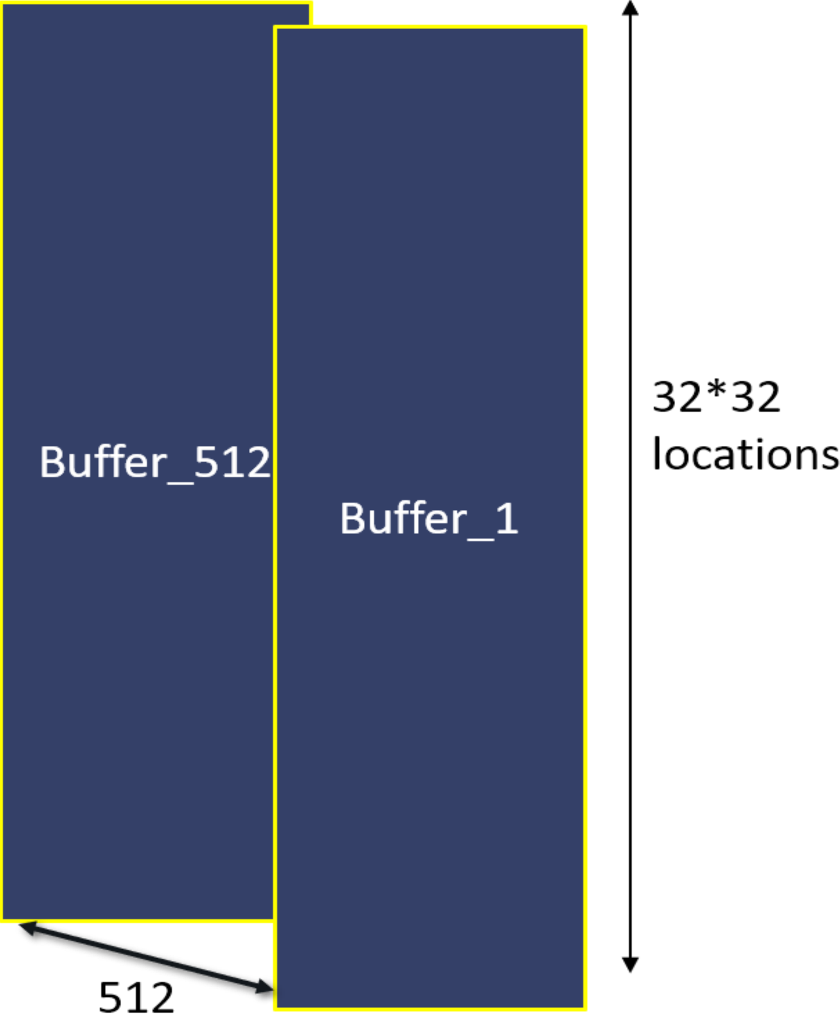

Figure 72: Pointwise output buffer

Layer	O/P size	No of weights
PW_ L1 :	(32, 32, 64)	2048
PW_ L2 :	(32, 32, 64)	4096
PW_ L3 :	(16, 16, 128)	8192
PW_ L4 :	(16, 16, 256)	32768
PW_ L5 :	(8, 8, 512)	131072
PW_ L6 :	(8, 8, 512)	262144
Total		440320

Figure 73: MobileNet PW layers

By analyzing the sizes of layers and output feature map in MobileNet architecture shown in figure (73), we find that not all the layers have 32*32 feature map this number decreases as we go through the layers also, the number of channels increases as we go through the layers. So, in the first layer and the second layer only 32 BRAMS are used for output storing, and as we go through the layers the number of used BRAMS increases but the utilization of BRAMS decreases as the feature map size decreases.

4.9.7 Batch normalization

Batch normalization is used to solve the problem of vanishing gradients that may occur in convolutional neural networks during training. The values of each feature on all samples are normalized into data with mean value of 0 and variance of 1. It makes the convoluted value fall into the center of the effective value region of the nonlinear function, so that vanishing gradient is avoided. The mean μ and variance σ^2 are
as follows: [14]

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i - \mu \ ,$$

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \mu)^2$$

The functions of batch normalization layer are as follows:

$$Y = \frac{\gamma(x_i - \mu)}{\sqrt{\sigma^2 + \varepsilon}} + \beta$$

The parameters of this layer are the scaling factor γ , the translation factor β , the mean μ, the varianceσ^2, and the denominator plus an " is used to prevent the denominator from being 0, and the value of " is 0.001 during training. Arrange them in a form suitable for hardware implementation. [14]

as follows:

$$a = \frac{\gamma}{\sqrt{\sigma^2 + \varepsilon}}$$

$$b = \beta - \frac{\gamma}{\sqrt{\sigma^2 + \varepsilon}}$$

The value is constant for both parameter a and b, so this layer is transformed into the following formula when the FPGA is implemented: [14]

$$Y = ax + b$$

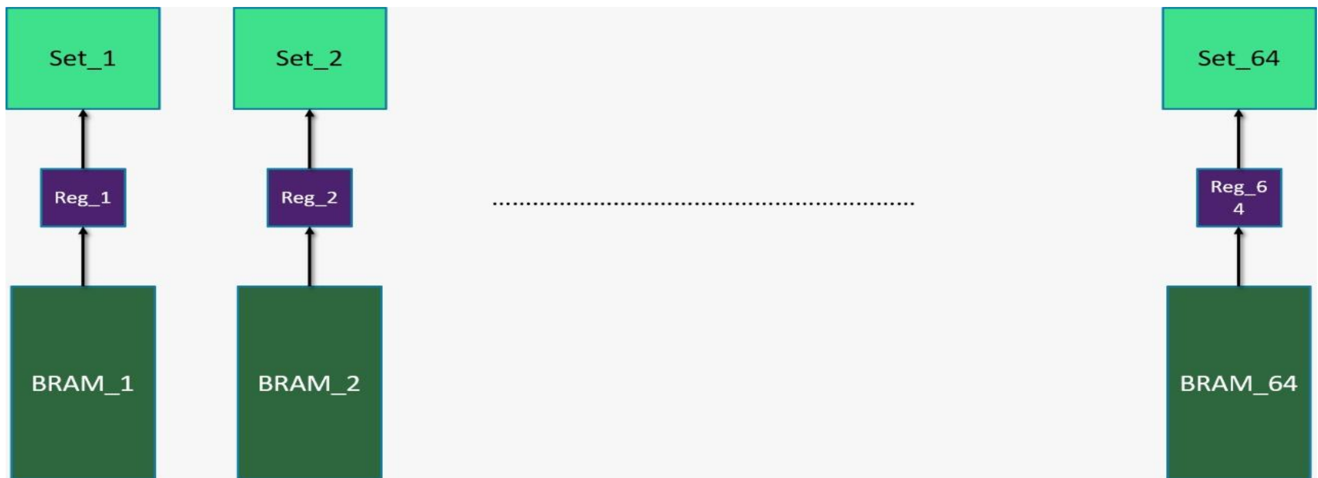

Figure 74: Batch norm weights storing

As a and b are constants, the multiplication of the weight a is done in software and the only the subtracting b is done in the hardware.to reduce the hardware of adding a subtractor, Weights are stored in 2's comp to use the adder "bias adder in figure (74)" in subtracting the bias b.

4.9.8 Illustrative example

The following figures describe how PW works starting from taking depthwise output that is stored in the memory till storing PW output in the memory. The example assumes a 64-channel input to pointwise and 128 kernels is applied.

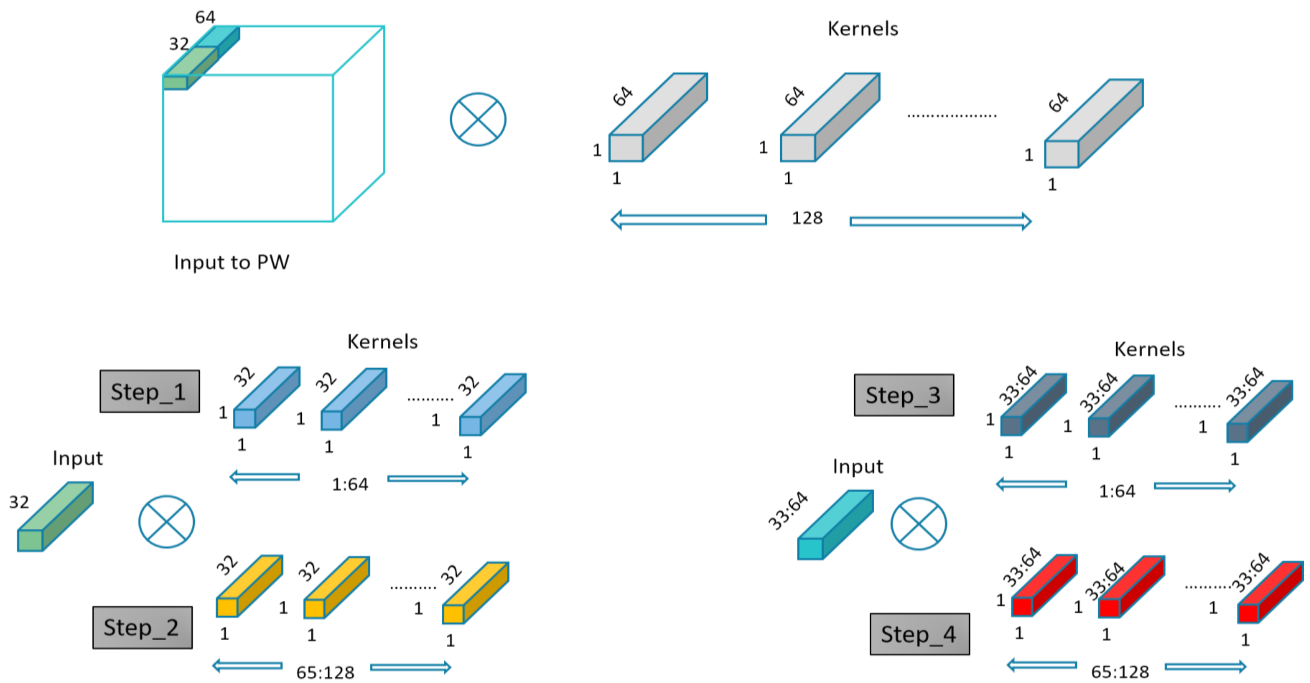

Figure 75: illustrative example

As shown in figure (75), step 1 only 32 channels of input are multiplied by the first 32 channels of the first 64 filters. In step 2 the same 32 channel inputs are multiplied by the first 32 channels of the second 64 filters (from 65 to 128). In step 3, Now we move to the second 32 channels of input and apply the same way that applied in step 1 and step 2 but relative to the second 32 channels (32 to 64).

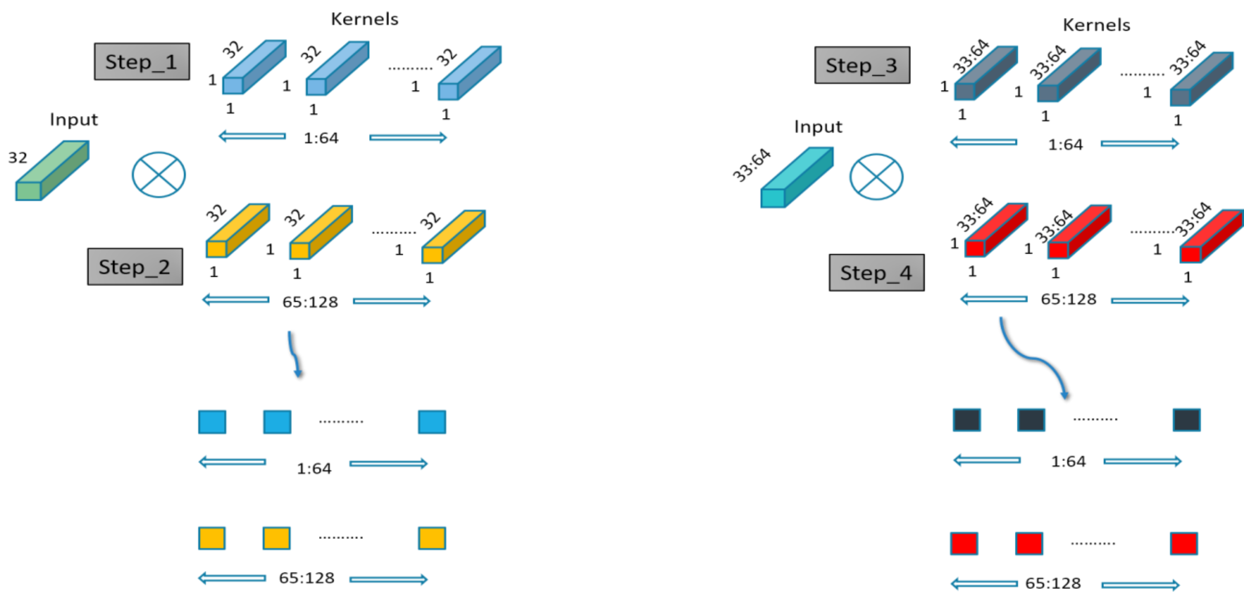

Figure 76: Kernels are applied

Figure (76) shows the result of multiplication after adding by the adder tree each operation results in one pixel, now to complete the operation we need to add blue pixels to dark blue pixels and to add red pixels to yellow pixels this is done using the accumulation adder as shown in figure (77).

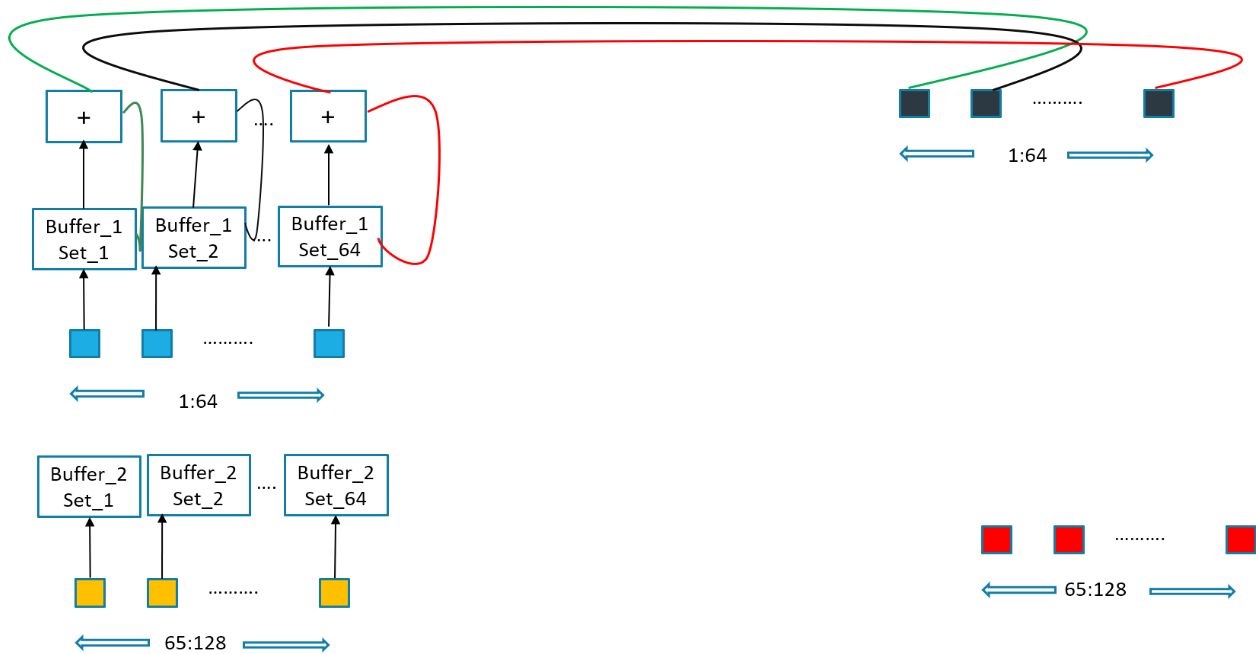

Figure 77: Combing the 2 parts

Figure (77) shows the accumulation operation and storing the result in the buffers one and two. Here only two buffers are used as we have 128 filters in this illustrative example.

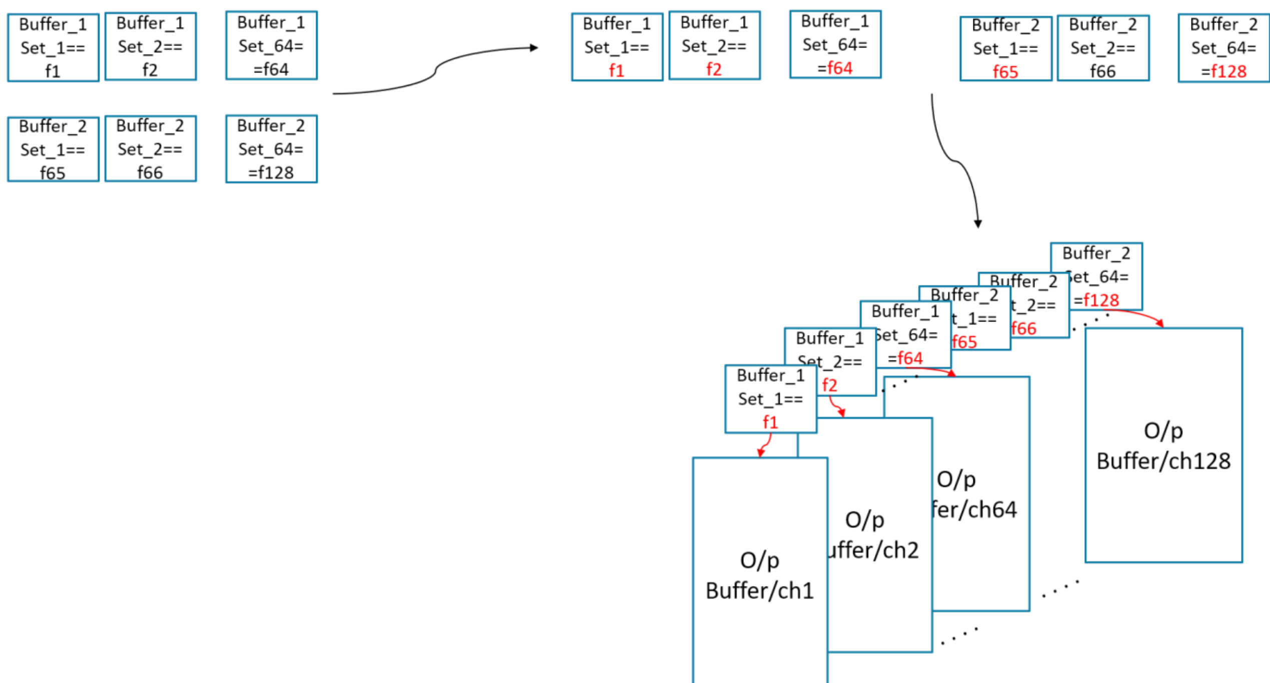

Figure 78: Storing the result in output buffers

As shown in figure (78), Now we have two buffers that have values in each set in the 64 sets , in other words we have the 128 required output pixels needed to be stored in the RAM 1 to be fetched by the depthwise.

4.10 Average Pooling Layer

In CNN architectures average polling is used widely in order to reduce feature map by neglecting secondary features and keeping only important features as shown in figure (79), in mobile net there is only one average pooling layer with input feature map of size (8,8,512) and output size (1,1,512).

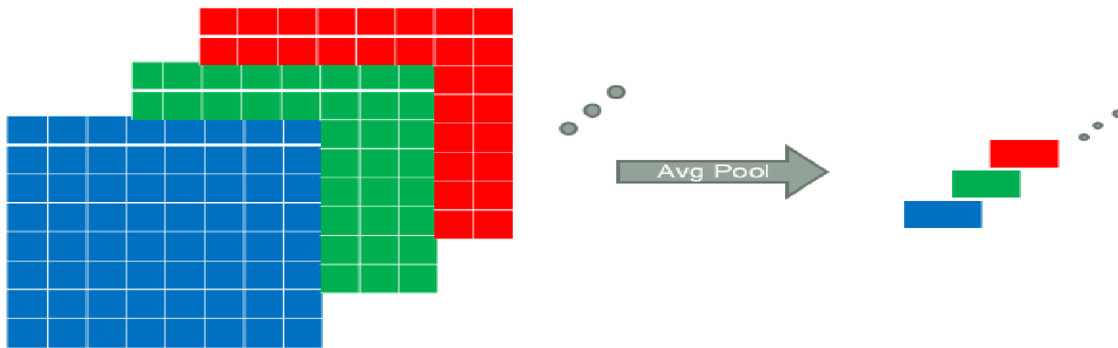

Figure 79: Average Pooling

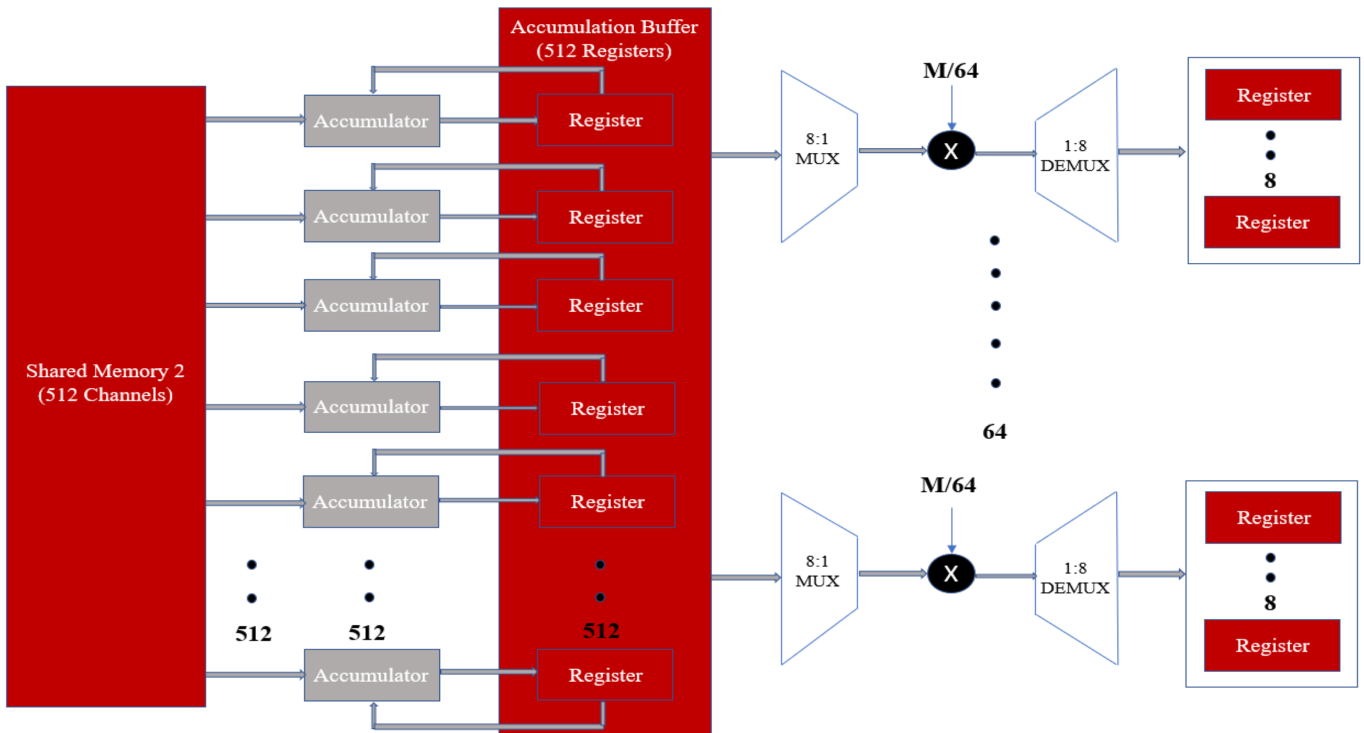

Figure 80: Average Pooling architecture

As shown in figure (80), we take feature map stored in shared memory 2 after each (1*1*512) pixels are generated from the last layer of pointwise convolution then these data are accumulated by 512 adders: one for every channel of the feature map. The output numbers of the last pointwise convolution layer are positive as we have the Relu block which cancels the negative numbers and the maximum represented number is 127 due to the saturation casting block, so the size of the accumulator is only 16 bits. We have 64 macs in order to multiply the values of the accumulation buffer by M value which is taken from the software model then divided by 64. We used 64 macs to have intermediate value of latency compared with 1 mac and to save hardware resources compared with 512 macs. Due to our previous talk, we need an 8:1 multiplexer to choose one register every cycle and a demultiplexer to store mac output in its corresponding register. We made a module of 8:1 multiplexer, 1:8 demultiplexer, mac and 8 registers then 64 instances to cover the whole values of the accumulation buffer.

This operation takes non sequential 64 clock cycles for accumulation and 9 clock cycles for $\frac{M}{64}$ multiplication. Pooling layer works while pointwise convolution is still working and finishes after pointwise convolution terminates by 11 clock cycles. This procedure improves latency as it has only 11 clock cycles delay.

4.11 Fully connected Layer

As discussed in chapter 2, fully connected layer (FC) is required to do classification of the extracted features that are done by the convolution layers.in this layers all the inputs from one layer are connected to every activation unit of the next layer.

4.11.1 FC architecture

The pooling layer provides 512 outputs and we have 43 classes, according to the FC layer algorithm we have 43 neurons each one takes all 512 pooling layer outputs as input and multiply them by different weights then adds the product. There are also 43 biases needed to be added in

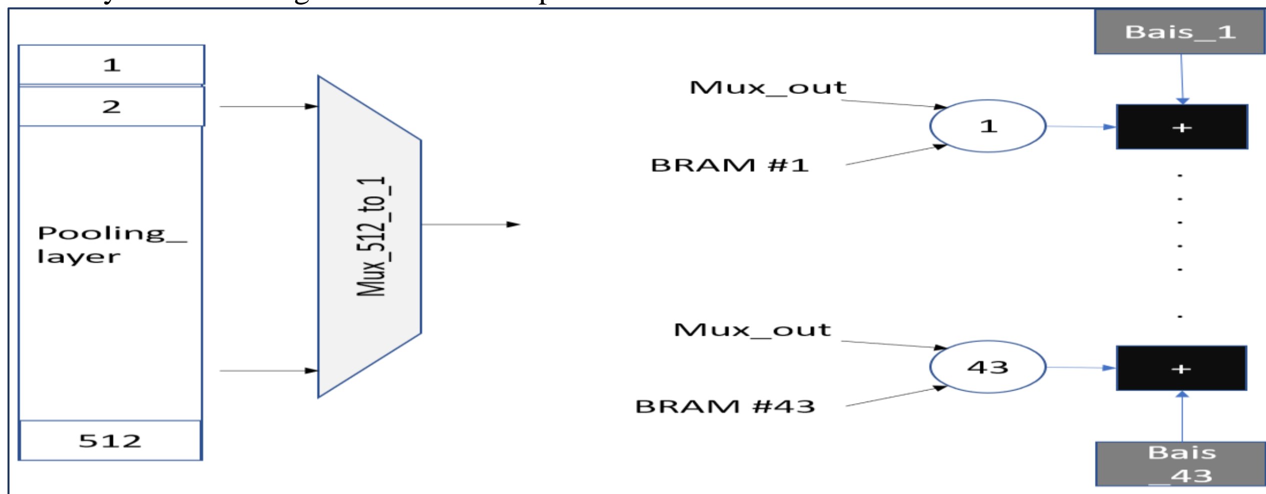

Figure 81: Fully connected layer architecture

each neuron output. Our hardware architecture for this layer utilizes 43 MACs that are initialized by 43 biases then takes one by one input till the full 512 inputs and multiply them by the weights which are provided by 43 BRAMs each BRAM contains 512 weights of each neuron. According

to this architecture the FC layer takes 512 cycles to generate the 43 classes. The following figure (81) and figure (82) shows the FC architecture.

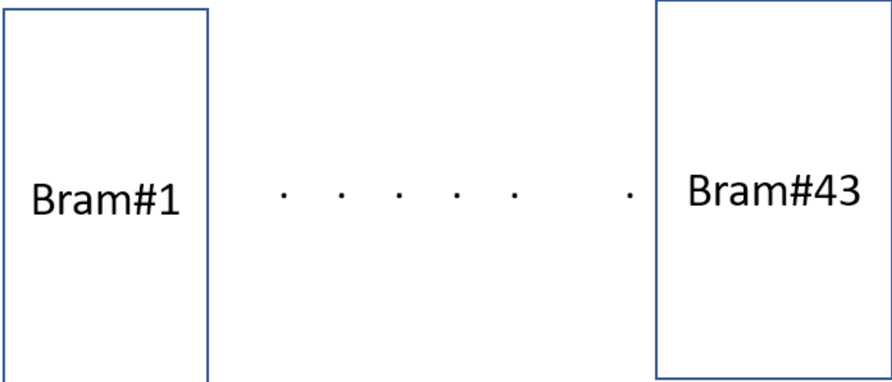

Figure 82: FC weights ROM

Chapter 5: Controllers and Weight distribution

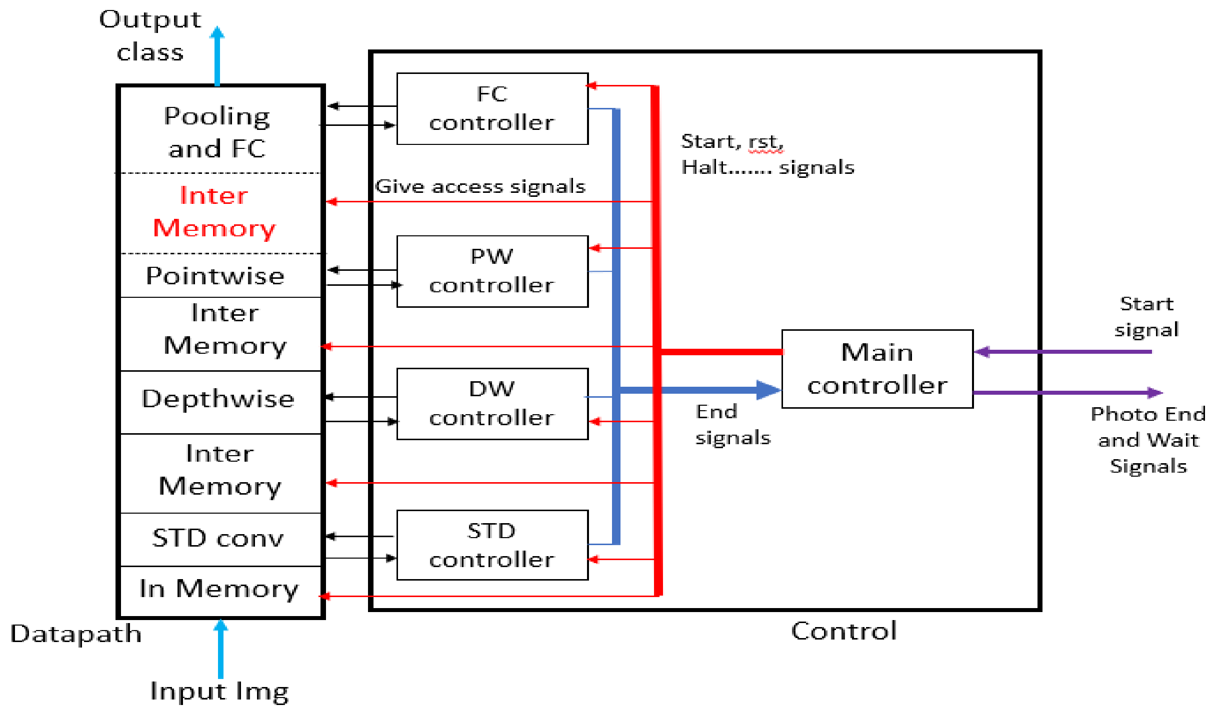

Figure 83: Main parts of digital system

As shown in figure (83), Any digital design system can be divided into two parts the Datapath and the control unit. The control unit in the design is consists of two levels: the main controller and controller to each layer in Datapath.

5.1 Main controller

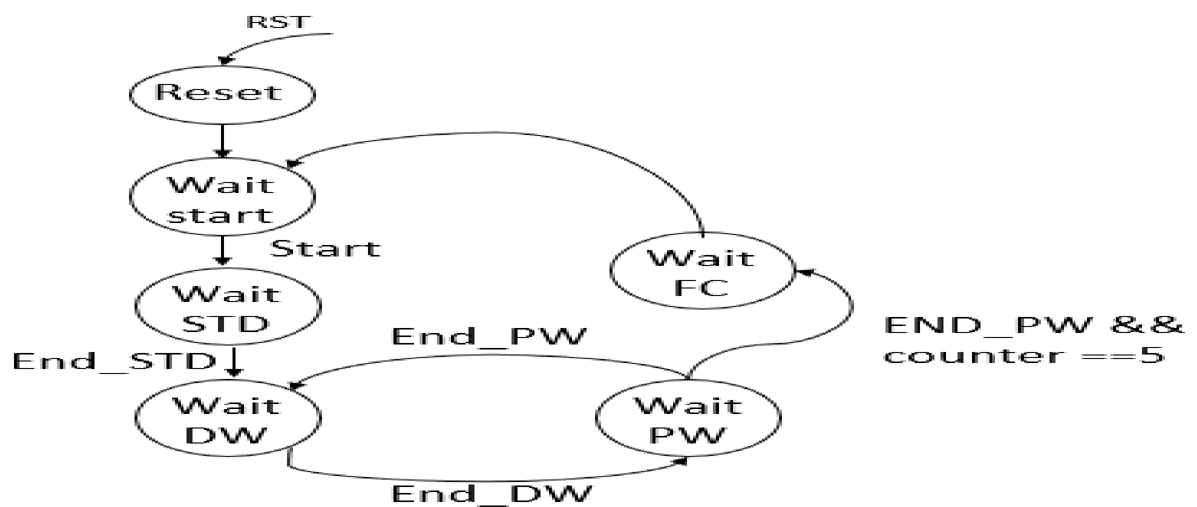

Figure 84: FSM of main controller

As shown in figure (84), the main control unit is Moore finite state machine, it consists of six states as following:

1) Reset state: it represents the state where global reset signal is High and this means that all the system is stopped.

2) Wait start: it represents the state where the outer processor writes the photo in three input memories and waits for the start signal that indicates that the photo is uploaded successfully in memory.

3) Wait STD: it represents the state where the standard convolution layer is processing the input photo and when it finishes the processing it goes to the first depthwise layer.

4) Wait DW: it represents the state where the depthwise convolution layer is processing the input photo and when it finishes the processing it goes to the next pointwise layer.

5) Wait PW: it represents the state where the pointwise convolution layer is processing the input photo and when it finishes the processing it may go to the next depthwise layer or the last layer (Fully connected). This depends on the value of counter that counts the number of finished layers.

The main controller takes start and reset signals from the outer processor and the end signals from inner layer controllers. It gets out the photo end signal to the outer processor and start signals to inner layer controllers and also it gets out the access signals to the memories. The access signals give permission for each layer to master the inter memories so it solves the conflict on memory access.

5.2 Standard convolution and Depthwise controller

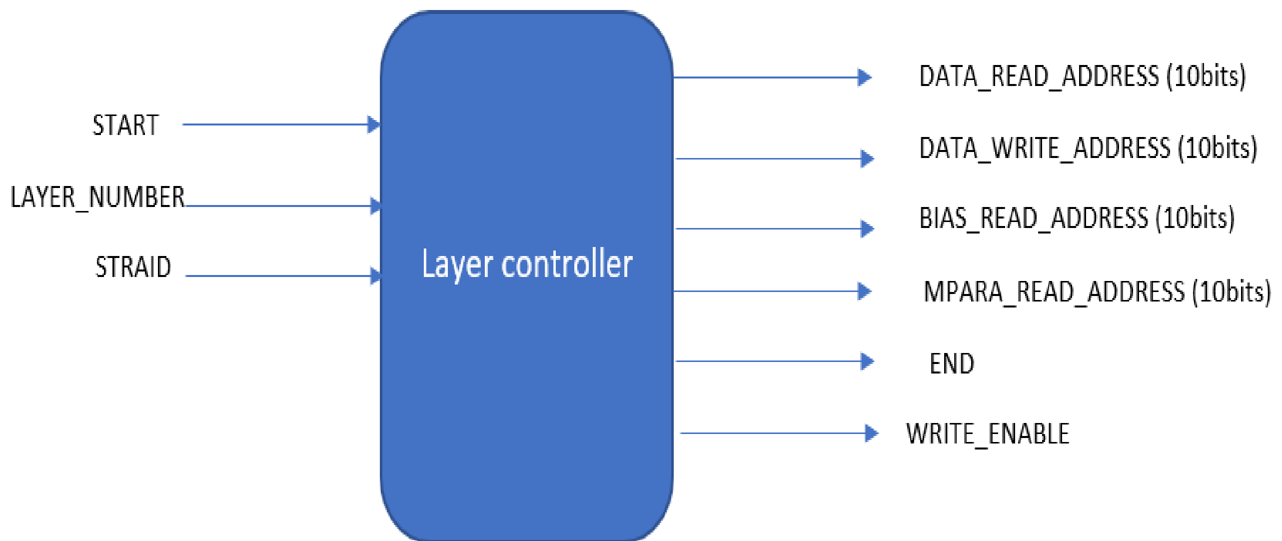

Figure 85: Layer controller entity

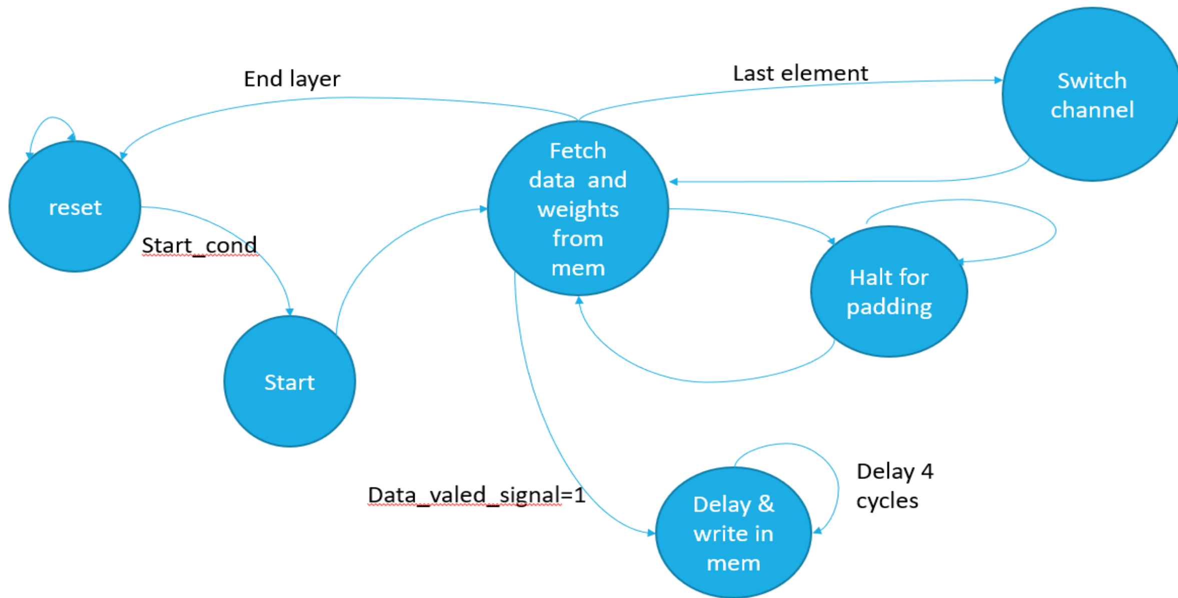

Figure 86: State diagram

Layer controller: design consists of 3 blocks 3*3*3 conv layer, Depthwise and point wise layer each layer has a controller which when a start condition adjusts from the main control unit the layer controller fire some sequence of operations to manage padding operation as shown in figre(86), end of channel, last channel in the layer and how to fitch the input feature map, weights and bias from memory. Also, the layer controller handles the operation of writing the output in memory and design halting conditions during padding operation. the firing sequences is dependent on the layer number and stride of the layer.

5.2.1 Controller operation sequences (Data Flow)

After start request from main controller reset state will start the next state fetching input and weights from memory to shifter, if padding request adjusted from shifter controller will go halt for padding state which halt the address of read and write from memory tell the padding state is end after 2 cycles design complete the data fetching,

As the proposed design is 32 channel parallel after fetching all the current channel elements, the machine is to switch channel state to reset all shifters elements and counters and switch the multiplexer to the next 32 channels to start the next 32 channels from the same layer.

Write state is a state that detects the adjustment of a ready data signal and delay It until the data goes through the pipeline.

5.2.2 Controller operation sequences (Weight, Bias and M parameter)

According to the 32-channel parallelism a 32-block ram is implemented to fetch the layer parameter and according to the filter parallelism 9 macs per filter the memory overhead will be 9 cycles to load in the shift registers but this overhead is less than FIFO overhead in the best-case scenario 8*8 stride 1 which has 10 cycles overhead so 32 the fetch weights state takes 9 cycles in the state fetch data and weights state.

Bias and weights also have a memory address like weights. DW controller manages weights loading at the start of the layer, as shown in section (4.5.2) design parallelism is 32 channels so 9 weights should be loaded from 32 weights Block RAMs so controller handle the address of weights and after 9 clocks cycles should halt and start to count again after the current channels in done the same sequence is fires for bias and M parameters.

5.3 Pointwise controller

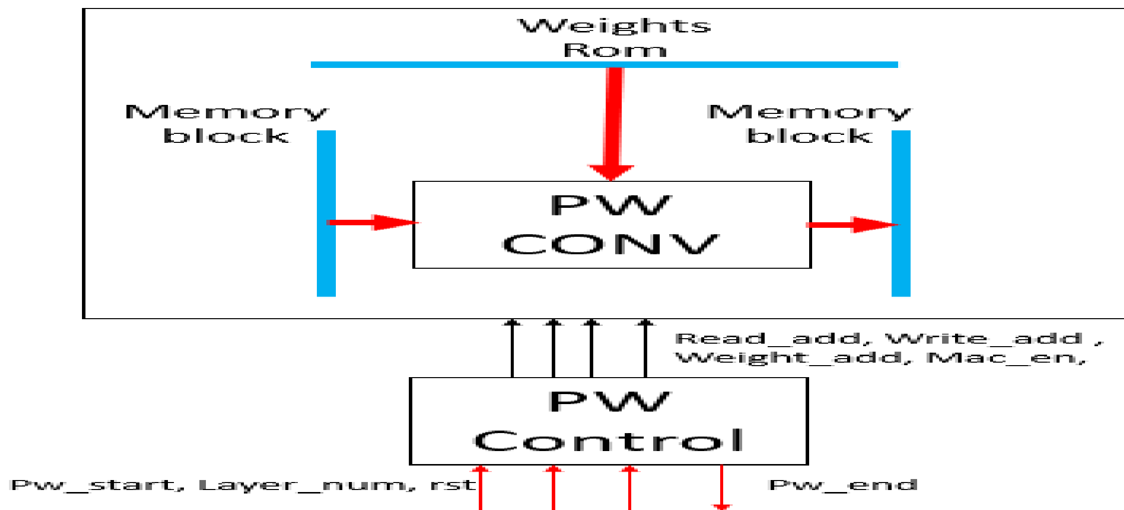

Figure 87: PW controller

As shown in figure (87), the pointwise controller block controls the flow of data through PW convolution between the two memory blocks. The PW controller starts its operation on a certain layer number depending on the signal Layer_num when it detects a pulse on the signal PW starts. The PW controller outputs the addresses and write enable signals required for three memory blocks but it doesn't have the access to memories all the time. The PW controller gets the access to memories from the main control unit. When the PW finches the required layer, it sends an end signal to the main control unit to release the memory resources for other blocks.

5.4 Fully connected controller

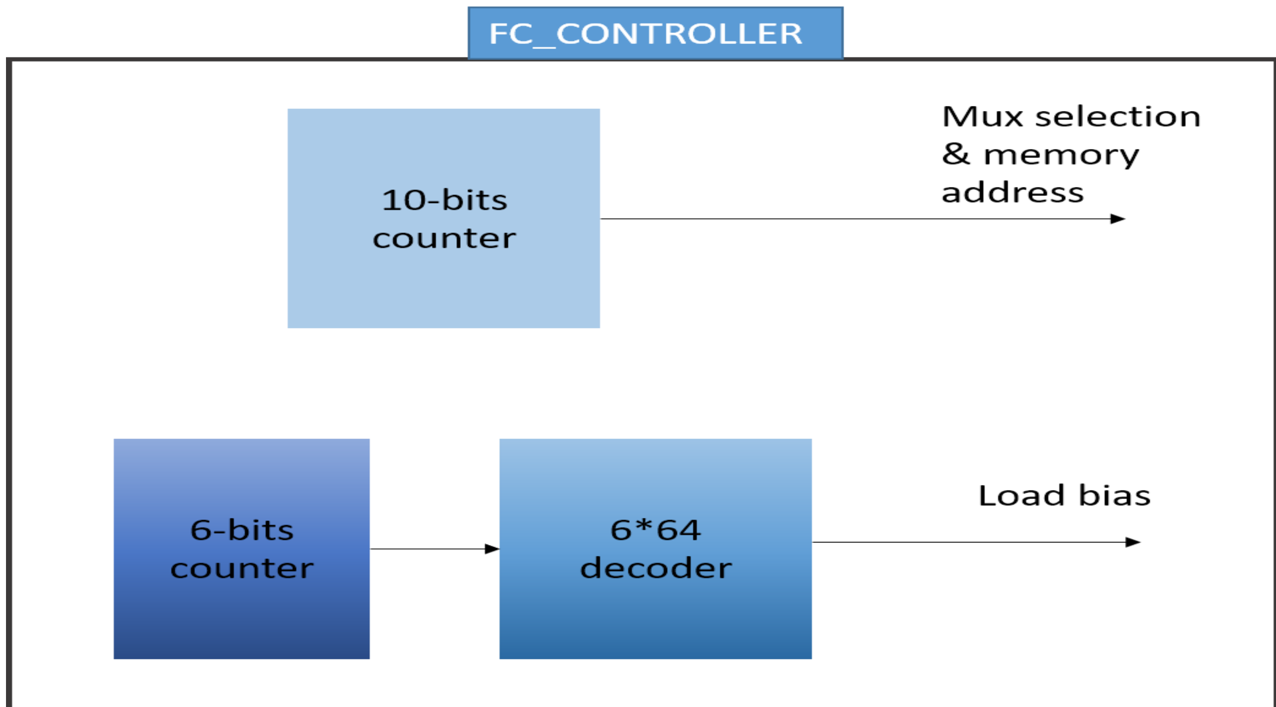

Figure 88: Fully connected controller

Fully connected controller is simple and consist of:

A) 10-bit counter that is used to drive the input multiplexer selection and the weights ROMs addresses that are fed to the MACs to be multiplied.
B) 6-bits counter that connected to 6*64 decoder which drives the load bias signals that enables the MACs registers to store the bias values.

5.5 Weight distribution

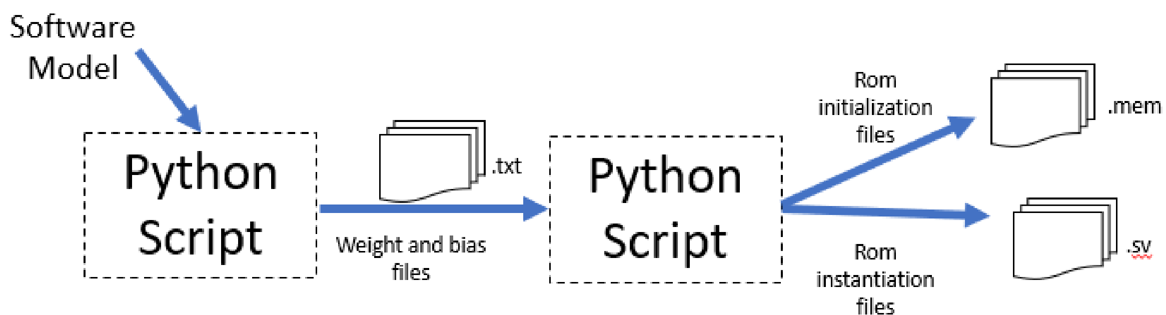

Figure 89: Weight distribution flow

As discussed before, the design contains a huge number of ROMs that save the weights of mobile net models. So, the distribution of weights and making it automated is a critical task.

As shown in figure (89), the weight distribution is done using two python scripts. The first script gets out the software model weights into .txt files each filter in a single file. The second script reads the .txt files then distributes the weights into .mem files in certain order. These files should be used to initialize the ROMs of the design. The second script also writes a .sv file that contains all required instantiation of ROMs using the parameter INIT_FILE to pass the initialization file.

The figure (90) shows an example of all discussed files. The main advantage of this flow is that the weight distribution is automated and doesn't depend on a certain platform to do this task and this makes the file exchange very easy.

```
0.mem
ffff
FF61
009e
00b9
0077
001c
00c5
00cf
00e5
0039
0011
0093
00b2
0083
0095
00f7
0019
00da
006e
00e7
```

```
1.mem
0000
04D7
0060
0060
0028
0024
0051
002c
001c
0021
00ae
00cf
00d8
005f
0079
0026
0078
00c8
000e
00d7
```

```
memory_instances_bias.sv - Notepad
WeightsRom  #( .InitialPath("E:/communication years/GraduationProject/weights/pw/bias/0.mem") ) BiasRom_0 (
        .ADDRA(addrb),
        .ADDRB(addrb+1'b1),
        .rst(rst),
        .clk(clk),
        .DOA(biasW_w[0]),
        .DOB(biasW_w2[0])
);
WeightsRom  #( .InitialPath("E:/communication years/GraduationProject/weights/pw/bias/1.mem") ) BiasRom_1 (
        .ADDRA(addrb),
        .ADDRB(addrb+1'b1),
        .rst(rst),
        .clk(clk),
        .DOA(biasW_w[1]),
        .DOB(biasW_w2[1])
);
```

```
module WeightsPW(
    input clk,rst,
    input [9:0] addrb,
    input [9:0] addrM,
    input [9:0] addrW,
    output [15:0] weights [63:0][31:0], //[31:0]
    output [31:0] biasW [63:0], //[31:0]
    output [15:0] MW [63:0]
    );
    wire [15:0] biasW_w [63:0];
    wire [15:0] biasW_w2 [63:0];
    generate
    genvar i;
    for(i=0;i<64;i=i+1) assign biasW[i] = {biasW_w[i],biasW_w2[i]};
    endgenerate
    //instantiation of bias rom
    `include "E:\\communication years\\GraduationProject\\weights\\pw\\bias\\memory_instances_bias.sv"
   //instantiation of weights rom
   `include "E:\\communication years\\GraduationProject\\weights\\pw\\w\\memory_instances_weights.sv"
   //instantiation of M rom
   `include "E:\\communication years\\GraduationProject\\weights\\pw\\M\\memory_instances_M.sv"

endmodule
```

Activate Windows

Figure 90: Weight distribution files

Chapter 6: Testing methodology and functional simulation result

With the increasing complexity and small time to market of digital systems, it's becoming more difficult to design correct circuits for these systems with respect to function and performance. Through the design process it's necessary to check the function and the performance of each basic unit and then verify the bigger unit and so on.

The functionality of CNN architecture is to predict the class of input image. if there's an error it's very difficult to locate the error in Hardware design, so it's very useful to run each layer alone in HW and compare its output with golden reference coming from software model.

In this section, the testing methodology and functional simulation output of mobile net architecture in each layer will be discussed.

6.1 Testing methodology

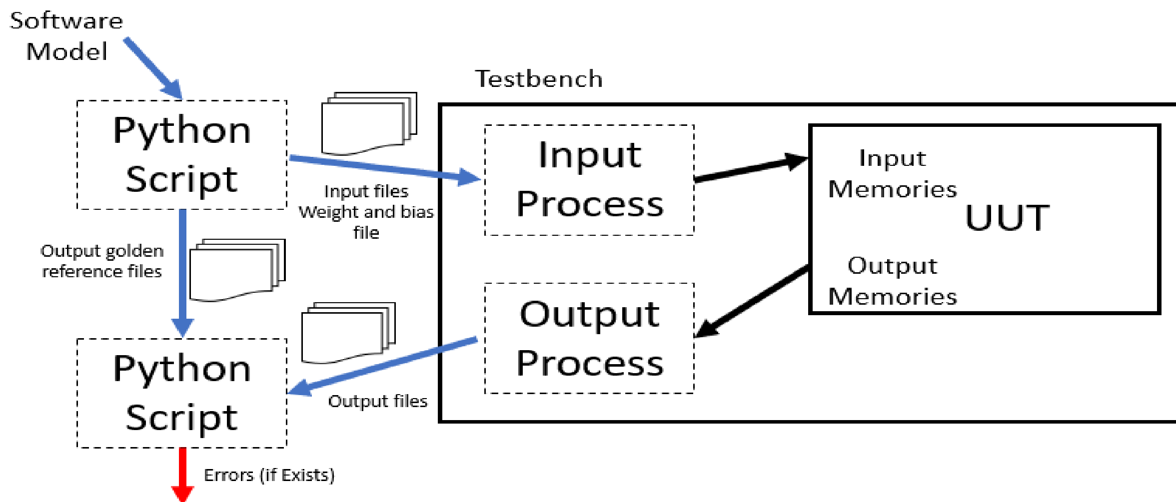

Figure 91: Testing methodology

As shown in figure (91), to test any standalone layer, a testbench is written to accept files of input and weights and upload them in memory in unit under test then the design runs and puts its output in files. Putting output in files makes it easier to compare the output with a golden reference that is generated by a software model.

6.2 Testing Standard convolution layer

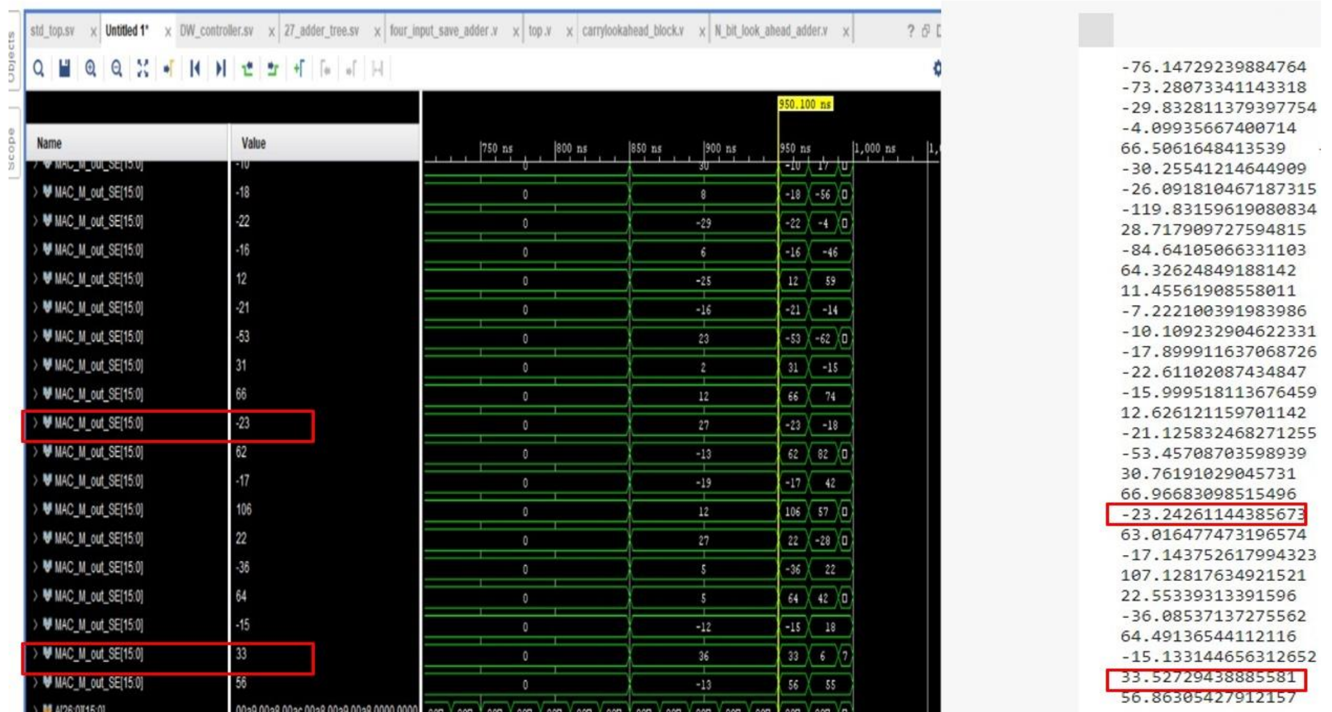

Figure 92: File’s comparison of pooling layer

In figure (92) we see that output from software model is exactly like in hardware error may happen in approximation to integer so may be error equal one decimal between two results.

6.3 Testing Depthwise layer

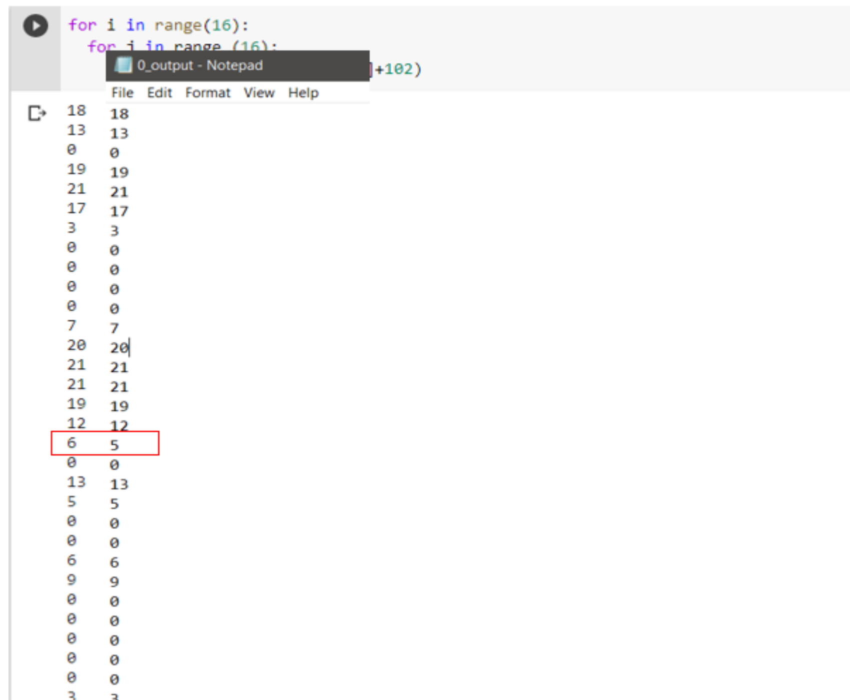

Figure 93 : Depth wise Results

6.4 Testing Pointwise layer

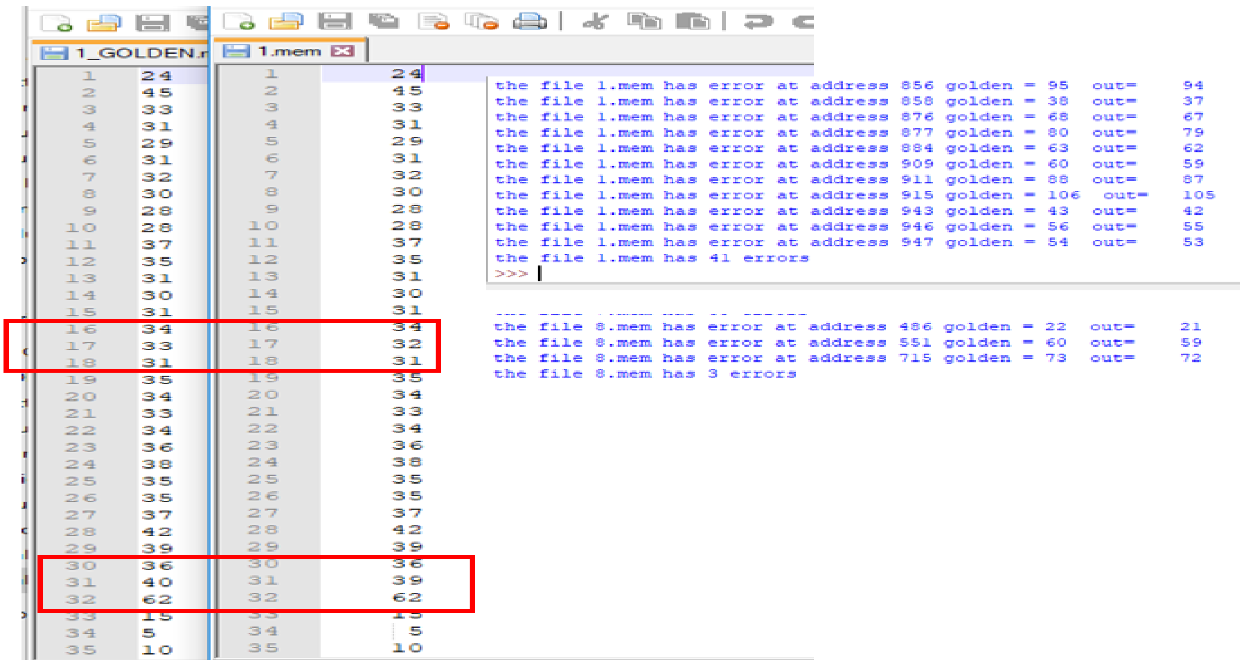

Figure 94: File comparing and expected errors

As shown in figure (94), by comparing the output files from functional simulation and the golden reference files the maximum error in one digit equals 1 decimal and this is due to quantization error in M as discussed before. These errors take place 41 times (4% of pixels) as maximum number in one file while testing the first layer of PW. These errors will not affect the decision at fully connected layers.

6.5 Testing Pooling and Fully connected Layer

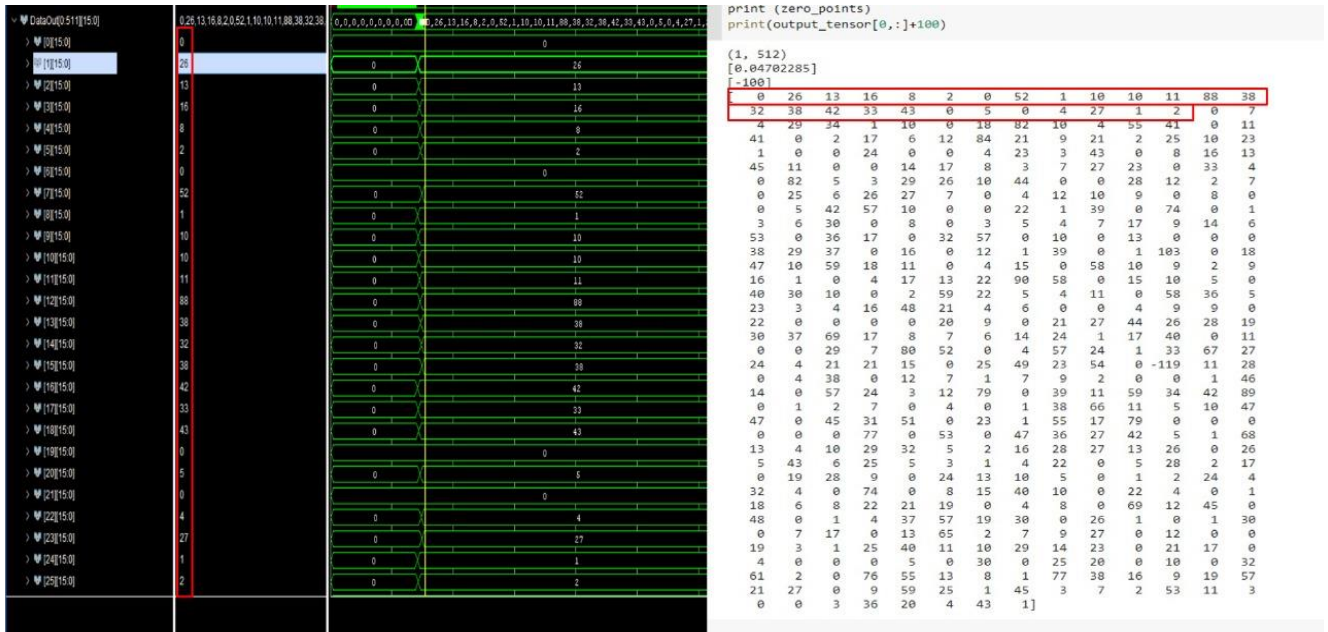

Figure 95 : pooling results

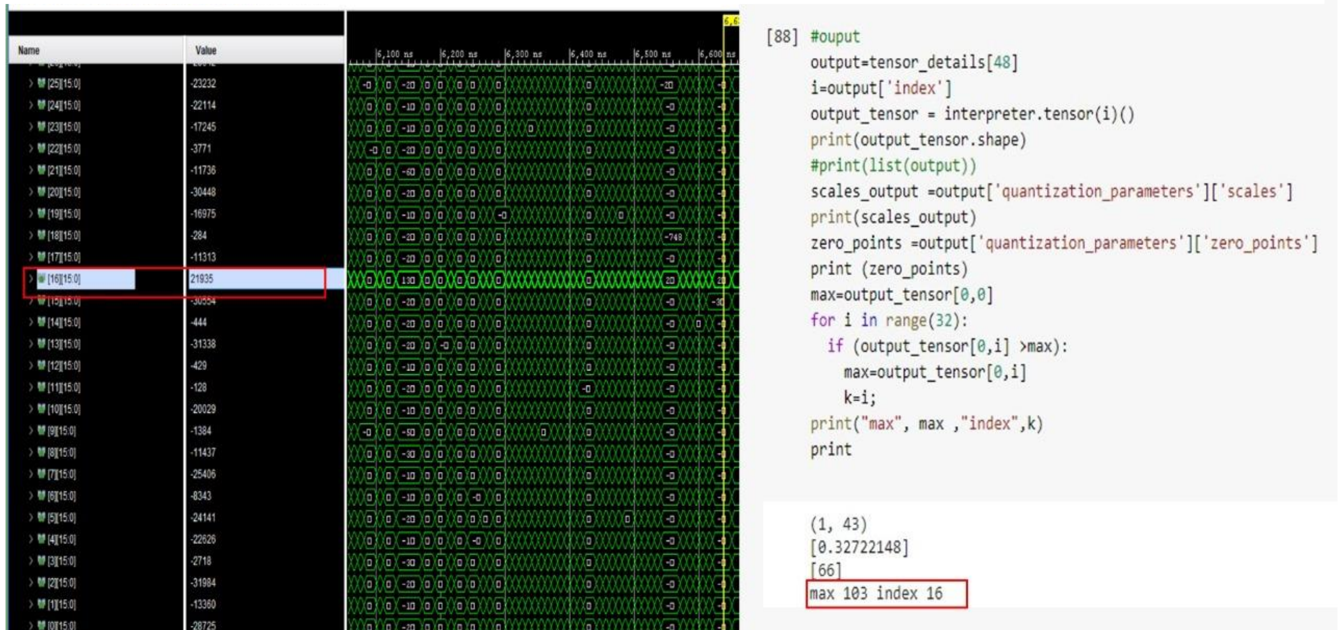

Figure 96 : Classify input image class from fully connected layer

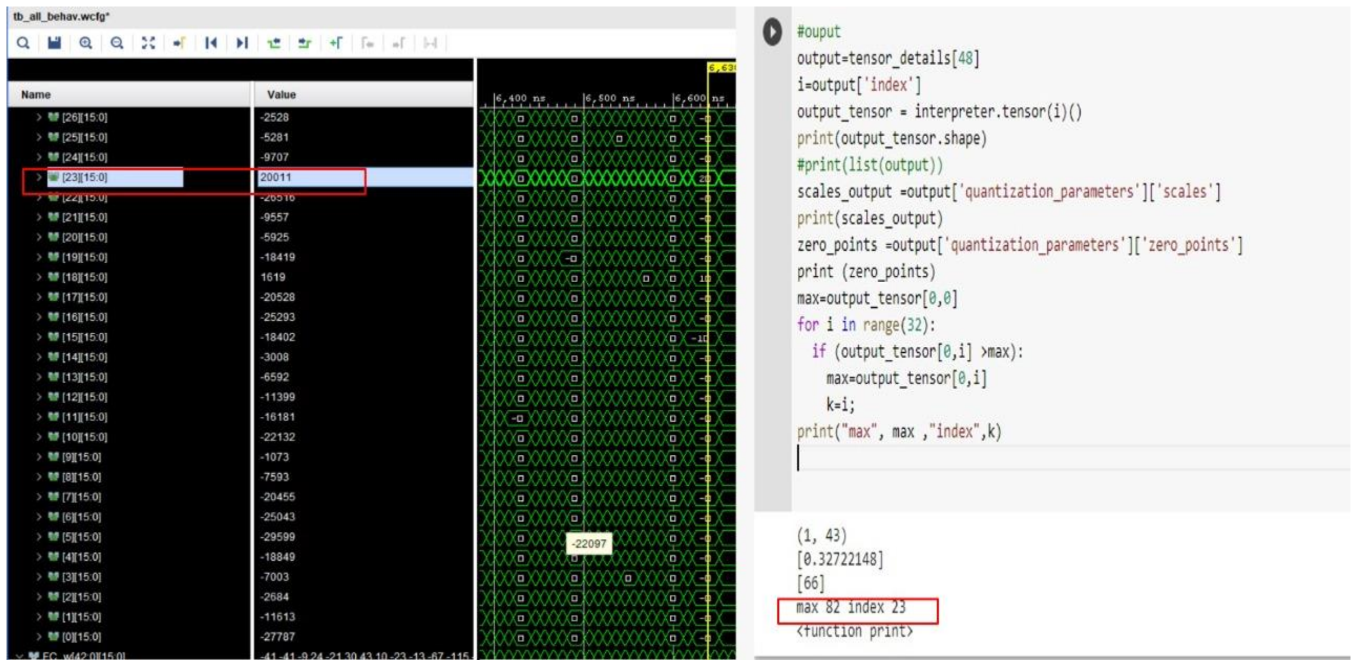

Figure 97 : classify another input image

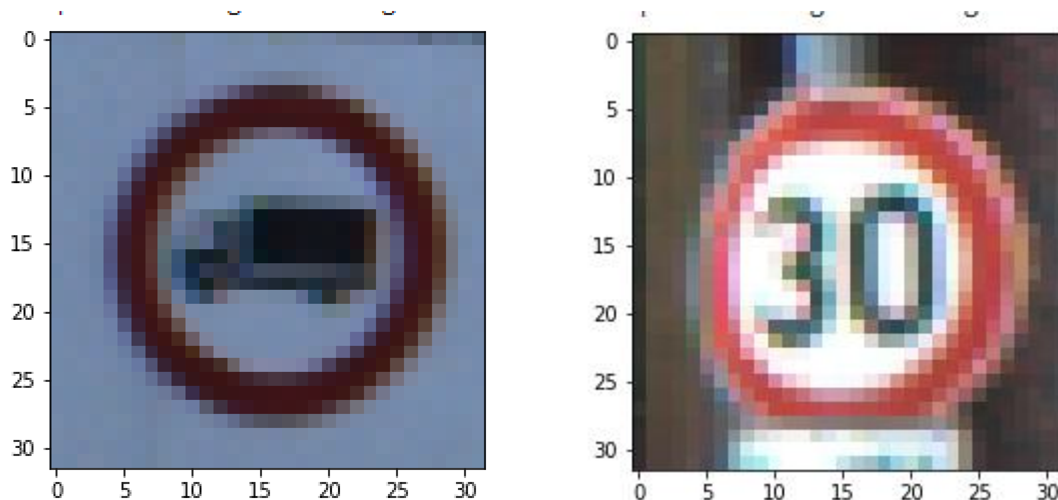

Figure 98 : input image

In figure (95) output from pooling layers gets as input in a fully connected layer to determine the image. In figure (98) the two-input image is classified with a model.

Chapter 7: Results, Future Work and Conclusion

7.1 Results

Figure (99), shows the utilization of the resources after the implementation on Virtex-7 FPGA

Resource	Utilization	Available	Utilization %
LUT	122992	433200	28.39
LUTRAM	99	174200	0.06
FF	101289	866400	11.69
BRAM	1124.50	1470	76.50
DSP	3435	3600	95.42
IO	750	850	88.24
BUFG	3	32	9.38

Figure 99: Resources Utilization

The following Figure (100) shows the power consumption of the design on the Virtex 7 FPGA, total power on chip equals to 13.920 watts. For first second, it seems that it is high power consumption but considering to the high speed achieved by the design it looks that it is normal considering the application requirement for ADAS systems that exist in cars which can afford this high-power requirement.

Summary

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**13.92 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**40.8°C**
Thermal Margin:	44.2°C (36.6 W)
Effective ϑJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Low

Launch Power Constraint Advisor to find and fix invalid switching activity

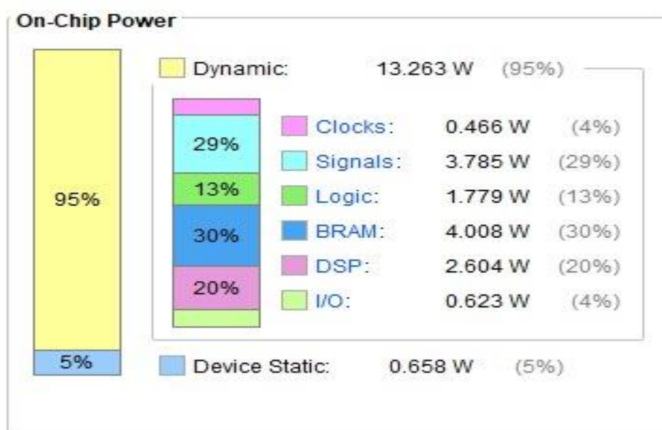

Figure 100: Power Analysis

Figure (101) shows the extracted Timing report which shows that the timing constraints are met at clock frequency of 100 MHz and have positive slack equals to 0.036 ns.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.036 ns	Worst Hold Slack (WHS):	0.043 ns	Worst Pulse Width Slack (WPWS):	4.358 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:	378534	Total Number of Endpoints:	378534	Total Number of Endpoints:	109322

All user specified timing constraints are met.

Figure 101:Timing summary results

Table 7: Stages Results

Layer/parameter	LUTS	Flip flops	DSPs	BRAMs
Standard conv	21.34K	21.465K	896	0
Depth Wise	21.873K	27.604K	320	40
Point Wise	51.948K	18.625K	2112	560
Pooling and fully connected	12K	12K	107	11
Total(Layers +Top) (utilization)	108.7K (24.9%)	80.4K (9.%)	3435 (95.41%)	1124.5 (76%)

Table 8: Timing Results

Layer / Parameter	Worst Negative Slack WNS (ns)	Worst Hold Slack WHS (ns)	Latency
Standard conv (ns)	0.733	0.089	0.01128 us
Depth Wise (ns)	0.734	0.098	0.0836 us
Point Wise (ns)	0.691	0.064	205.1us
Pooling and fully connected (ns)	0.896	0.137	5.23 us
TOP (ns)	0.036	0.043	0.210 ms

7.2 Power optimization

We do on several steps:

-As we seen in design, we don't fetch from memories all time so it isn't smart to make memories enable connect to one all times like in figure (102). if connect enable with logic where we fetch or wright the power will be reduce to 4 Watt in figure (105).

Settings
Summary (12.325 W, Margin: N/A)
Power Supply
Utilization Details
Hierarchical (11.747 W)
Clocks (0.514 W)
Signals (2.006 W)
Logic (1.395 W)
BRAM (4.556 W)
DSP (2.651 W)
I/O (0.624 W)

lame A	Clock A (MHz)	Enable Rate A (%)	Read Width A	Write Width A	Write Mode A	Write Rate A (%)
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000
IF_BUFG	100.000	100.000	18	18	NO_CHANGE	0.000

power_1

Figure 102: Enable Bram's 100%

Utilization	Name	Mode	Signal Rate	Clock Name A	Clock A (MHz)	Enable Rate A (%)	Read Width A	Write Width A	Write Mode A	Write Rate A (%
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	11.899	clk_IBUF_BUFG	100.000	9.211	18	18	NO_CHANGE	1.3
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	11.852	clk_IBUF_BUFG	100.000	9.211	18	18	NO_CHANGE	1.3
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	12.040	clk_IBUF_BUFG	100.000	9.211	18	18	NO_CHANGE	1.3
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	12.087	clk_IBUF_BUFG	100.000	9.211	18	18	NO_CHANGE	1.3
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	11.946	clk_IBUF_BUFG	100.000	9.211	18	18	NO_CHANGE	1.3
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.198	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.198	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.198	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.198	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.198	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.198	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.198	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.111	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.198	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.155	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.184	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.182	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.181	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.138	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.170	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.170	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.112	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.130	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.050	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	5.919	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.050	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.017	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.050	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	5.952	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.050	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	5.952	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0
<0.001 W (<1% of total)	genblk5_0.bram18_tdp_bl.bram18_tdp_bl (RAMB18E1)	RAMB18	6.050	clk_IBUF_BUFG	100.000	9.198	18	18	NO_CHANGE	0.0

Figure 103 : Power optimization for Brams

- clock gating as each stage work exclusively we can disable other stages to save activities which will happen
- Advantage of model that 8 bit so signal power will be reduced.

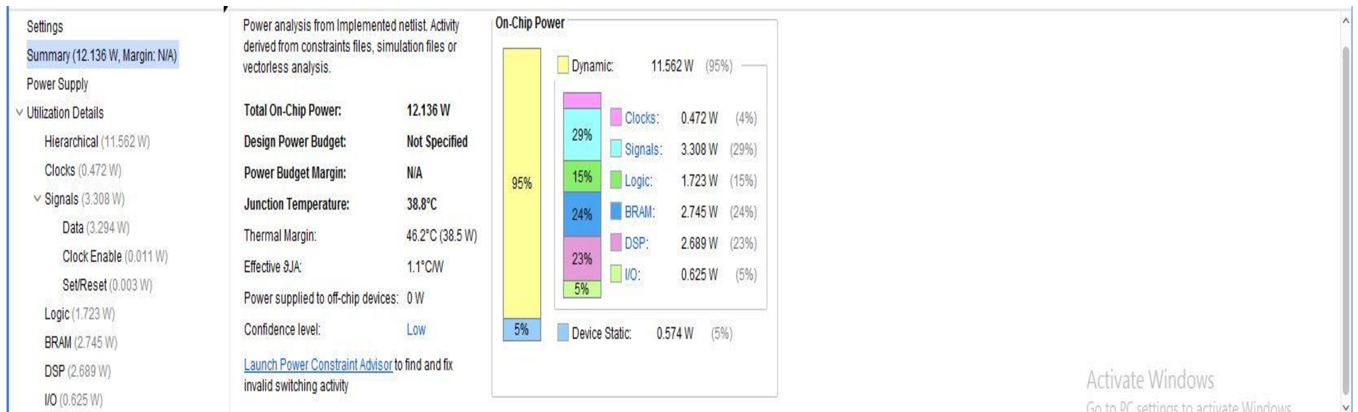

Figure 104 : Signal power if 16 bit is used

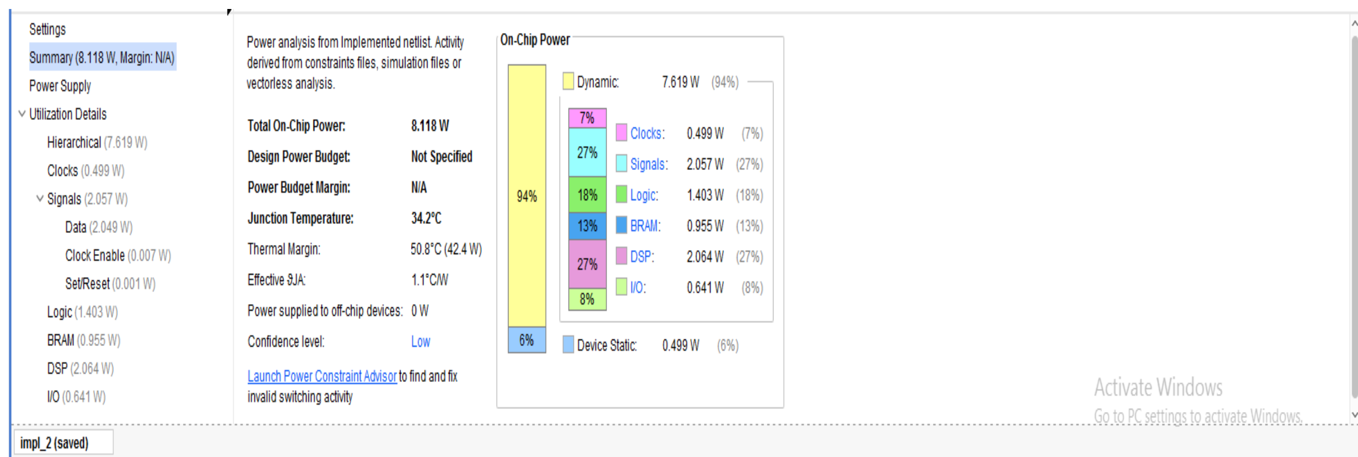

Figure 105 : Signal power if 8 bit is used

7.3 Benchmark

Table 9 : Comparison among different paper results and our results

Parameter\ Design	**Our Work**	**Reference[16]**	**Reference[14](alpha=0.5)**	**Reference[15]**	**Reference[17]**
Target Board	Virtex 7-VC-709	Zynq UltraScale+ ZU104	Zynq7z045	Virtex 7-VC-709	Virtex 7-VC-709
CNN	MobileNet	MobileNet	MobileNet	zynqNet	squeeze-Net
Accuracy (%)	80	43	---------	---------	69.6
No of weights	472K	34K	---------	2.5M	1.2M
No of layers	7	13	---------	10	---------
No of classes	43	6		---------	10
Frequency(MHz)	100	100	100	100	100
LUTs (Utilization)	109K (25%)	161K (69.97%)	9K (4.21%)	345K (79.639 %)	96K (22.39%)
FF(Utilization %)	80.4K (9.3%)	168K (36.57)	16K (3.88%)	184K (21.23%)	159K (18.44%)
DSPs(Utilization %)	3435 (95.4 %)	1104 (63.89%)	109 (12.11%)	3552 (98.67%)	2658 (73.8%)

BRAMs(Utilization %)	1124.5 (7649. %)	159.5 (51.12%)	110.5 (20.28%)	2130 (72.448 %)	992 (64.48%)
Latency (ms)	0.210	0.69119	722.68	80	4.02
Frame rate (fps)	4761	1250	1.38	12.5	248.7
Power(Watt)	8.118	4.075	2.15	10.97	8.9
Energy/image(J)	0.0017	0.002816	1.554	0.88	0.0357

7.4 Future Work

Future work is more techniques and enhancements that can be applied in the design but are left for the future. Some of these techniques the design is ready for them and need little modification to make them true. These new methods can achieve reduction in utilization of FPGA resources which leads to consuming less power and having higher throughput.

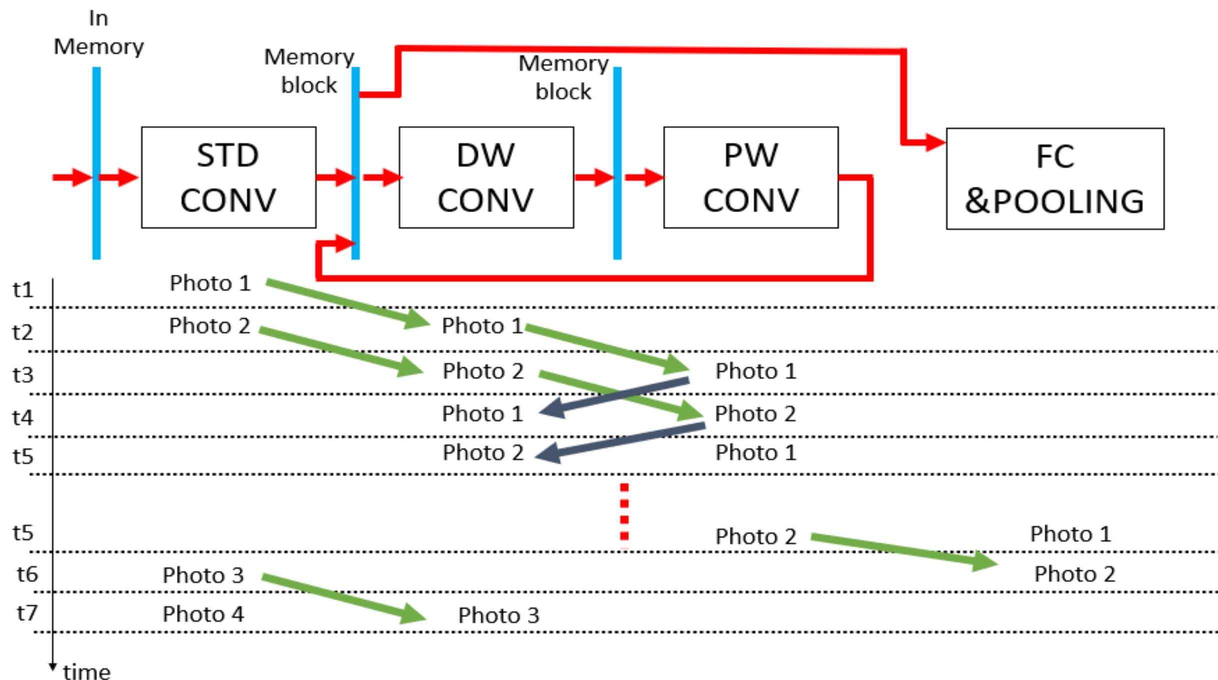

Figure 106:Time diagram of Multiple photos processing

7.4.1 Increase throughput by time sharing between photos

As shown in figure (106), the design is ready to process two images together under the condition that each block memory has one port for reading and one port for writing and to avoid overwriting of required data the write process will not start unless the DW block initializes all its FIFOs.

This modification will require changing only in the main control unit that is responsible of data flow between blocks and the memory access permissions.

7.4.2 Make design ready for ASIC flow

As discussed before, the design is targeting Virtex7 FPGA so it uses an Ips inside FPGA like DSP and BRAM. This is a point of weakness in the design and it can be modified by using more generic syntax to make the design go through ASIC flow. Using BRAMs as a standard memory makes the utilization of memories low due to its minimum size is 16 kbits and can't be smaller. This affects the power consumed by the BRAMs and the area they will consume if the design goes through ASIC flow.

7.4.3 Experimental Work

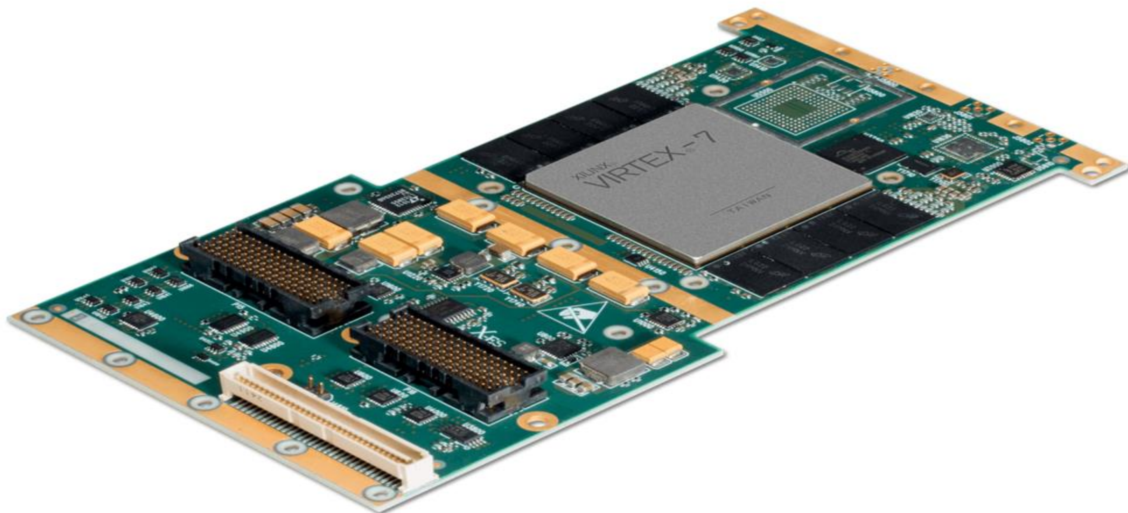

Figure 107: FPGA board

Our design is ready to be tested in real time system, by burning bit stream on virtex-7 FPGA shown in figure (107).

7.5 Conclusion

MobileNet introduces new type of convolution which decreases the parameters and increases performance. MobileNet model achieved 80% accuracy on German Traffic Sign Dataset. Model accuracy can be increased using Transfer learning to 95%. MobileNet Model Consists of 7 layers. Our design consists of 4 Hardware main blocks (STD, DW, PW, Pooling an FC). Our design achieved 0.21ms latency, 4975 fps and the power are 8 watt (0.001705 Joule/Image). The design is ready to make pipeline between Images which will increase the throughput through design.

References

[1] What is ADAS (Advanced Driver Assistance Systems)? – Overview of ADAS Applications. (n.d.). Retrieved from https://www.synopsys.com/automotive/what-is-adas.html .

[2] What's the future of ADAS? (Advanced Driver Assistance Systems). (2020, November 26). Retrieved from https://www.oxts.com/future-of-adas .

[3] Jigang Tang, Songbin Li , Peng Liu ," A Review of Lane Detection Methods based on Deep Learning, Pattern Recognition" ,2020.

[4] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi "You Only Look Once: Unified, Real-Time Object Detection",2016.

[5] Joanna Stanisz, Konrad Lis, Tomasz Kryjak, Marek Gorgon "Optimisation of the PointPillars network for 3D object detection in point clouds",2020.

[6] Jie Peng, Shuai Kang, Zhengyuan,Hangxia Deng4 , Jingxia, Yikai Xu6 , Jing Zhang, Wei Zhao,
Xinling Li, Wuxing Gong, Jinhua Huang, Li Liu1," Residual convolutional neural network for predicting response of transarterial chemoembolization in hepatocellular carcinoma from CT imaging", 2019

[7] Rikiya Yamashita, Mizuho Nishio, Richard Kinh Gian Do2, Kaori Togashi1" Convolutional neural networks: an overview and application in radiology".

[8] Forrest N. Iandola1, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, Kurt Keutzer" SQUEEZENET: ALEXNET-LEVEL ACCURACY WITH50X FEWER PARAMETERS AND <0.5MB MODEL SIZE".

[9] AlexNet: The first CNN to win image net. Retrieved from https://www.mygreatlearning.com/blog/alexnet-the-first-cnn-to-win-image-net/ .

[10] VGG-16 | CNN Model. Retrieved from https://www.geeksforgeeks.org/vgg-16-cnn-model/.

[11] MobileNet architecture. Retrieved from mailto:https://iq.opengenus.org/mobilenet-v1-architecture/

[12] Benoit Jacob,Skirmantas Kligys, Bo Chen,Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, Dmitry Kalenichenko ,"Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference", 2017.

[13] Andrew G. Howard, Menglong Zhu, Bo Chen,Dmitry Kalenichenko, Weijun Wang, Tobias Weyan, Marco Andreetto, Hartwig Adam"MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications",2017.

[14] Jiawen Liao, Liangwei Cai, Yuan Xu, Minya He "Design of Accelerator for MobileNet Convolutional Neural Network Based on FPGA",2019.

[15] Amr Mohamed Gamal Eldin, Aya Hesham Omar, Gamal Saied Fadl, Mennat-Allah Ayman Ahmed, Omnia Essam Ahmed, Sara Mostafa Mohamed” ACCELERATED DEEP NEURAL NETWORKS USING FPGA (ZynqNet Architecture)”,2020.

[16] YULAN SHEN, “Accelerating CNN on FPGA (An Implementation of MobileNet on FPGA)”,2019.

[17] Ahmed Tarek, Abdallah Mohamed, Amr Eid, Fatma Khaled, Farida Khaled “Accelerating Aware Machine Learning for Squeeze-Net Algorithm Design”,2020.