RTL and HLS Implementation of Real-time Machine Learning Hardware Accelerator on FPGA (ShuffleNet)

A Graduation Project Thesis
Submitted in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science
in
Electronics and Electrical Communications Engineering
in the
Faculty of Engineering at Cairo University

By

Ahmed Mohammed El-Sayed
Hassan Mostafa Emam
Abdel-Rahman Ahmed Mahmoud
Mahmoud Maged Nady
Mostafa Mohammed Kassem
Yousof Osama Mostafa

Under the Supervision of
Dr. Hassan Mostafa

Faculty of Engineering, Cairo University
Giza, Egypt
July 2022

Table of Contents

List of Figures

List of Tables

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
BRAM	Block Random Access Memory
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPU	Central Processing Unit
Conv	Convolution
DNN	Deep Neural Network
DSP	Digital Signal Processing
DW	Depth Wise
FC	Fully Connected layer
FF	Flip-Flop
FIFO	First In First Out
FM	Feature Map
FPGA	Field Programmable Gate Arrays
GPU	Graphics Processing Unit
HDL	Hardware Description Language
HLS	High Level Synthesis
I/O	Input/Output
LED	Light-Emitting Diode
LUT	Look Up Table
MAC	Multiply and Accumulate
ML	Machine Learning
MMCM	Mixed-Mode Clock Manager
MUX	Multiplexer
NN	Neural Network
PAR	Place and Route
RAM	Random Access Memory
ReLU	Rectified Linear Unit
ResNet	Residual Neural Network
RGB	Red-Green-Blue
ROM	Read Only Memory
RTL	Register Transfer Level
STA	Static Timing Analysis
TNS	Total Negative Slack
VIO	Virtual Input/Output
WNS	Worst Negative Slack

Acknowledgments

First and foremost, praise is to Allah, the Almighty, the greatest of all, on whom ultimately, we depend for sustenance and guidance. We would like to thank Almighty Allah for giving us the opportunity, determination, and strength to achieve this work.

We want to express our gratitude to our major advisor Dr. Hassan Mostafa for his assistance and dedicated involvement in every step throughout the process providing continuous full and professional support.

We would like to express our gratitude to Mentor Graphics for their sponsorship and especially Eng. Amr Gamal, FPGA Prototyping QA/test Engineer for providing his time and experience to help us overcome some obstacles we faced throughout the project, and for his encouragement throughout the whole year.

Last but not least, we want to thank our families for their support, tolerance, and love during this year, especially during the hard times. We are grateful to our families, colleagues, and friends for always motivating us, without them we wouldn't have come so far.

Abstract

Convolutional Neural Networks (CNNs) make a revolution in machine learning applications. Also, CNNs dominate in all computer vision applications as they give the highest accuracy and performance. However, CNN models have very complex computations, which need powerful hardware to run. So, GPUs are commonly used over generic CPUs. But the main drawbacks of using GPUs are their higher power consumption and generality. Accelerators are the best solution as they can give us the most powerful computations and the best performance with affordable power consumption.

Our project aims at designing and optimizing a hardware accelerator, which is specialized hardware performing a specific CNN model algorithm, which is the ShuffleNet. We have two design approaches to perform. The first is to model the algorithm in a high-level language and synthesize it with a High-Level Synthesis (HLS) tool. The second is to make a hardware architecture block diagram and then use a hardware description language to make the actual hardware in a Register Transfer Logic (RTL) manner. Finally, we will compare the two approaches in different aspects like time to develop, the performance of the accelerator, and power consumption to determine which approach is better.

In this work, the HLS implemented accelerator classifies 24.84 frames per second, the power consumption of the system is 3.828 Watts, the energy per image is 0.154 Joule/image, while the design is run on the Xilinx Virtex-7 FPGA VC709 connectivity kit with an 80 MHz clock frequency. On the other hand, the RTL implemented accelerator classifies 1216 frames per second, the power consumption of the system is 9.156 Watts, the energy per image is 0.007529 Joule/image, while the design is run with a 100 MHz clock frequency.

Keywords: Convolutional Neural Networks (CNNs), ShuffleNet, Hardware Accelerators, Register Transfer Level (RTL), High-Level Synthesis (HLS), Field Programmable Gate Arrays (FPGAs), Dynamic Quantization, Real-Time, High-Speed Processing.

1 Chapter 1: Introduction

1.1 Motivation

Deep learning has been gaining much popularity lately due to its ability to bridge the gap between machines and human capabilities in terms of accuracy when trained with a huge amount of data. One of the domains of deep learning is image classification, which is a very complex task for computers. But advanced computer vision systems can do image classification. These computer vision systems have become important in many applications in robotics, surveillance, and autonomous vehicles. Significant progress has been made regarding the performance of these advanced systems. The availability of powerful computing platforms and the strong market pull have shaped a very fast-paced development. The target of this field is to enable machines to view the world as humans do, and perceive it similarly. The advancements in deep learning have been constructed over the concept of neural networks.

Convolutional Neural Network (CNN) is one of the brain-inspired algorithms that represent the most promising approach to image understanding and classification with significantly higher accuracy than traditional algorithms. The main disadvantage of CNNs is the enormous computational complexity and to achieve accurate results, CNNs need many parameters (some over 100M parameters) and require huge amounts of computational resources and memory, they also offer significant potential for massive parallelization and extensive data reuse. The real-time evaluation of a CNN may need billions or trillions of operations per second to provide image classification on a video stream. The most recent Graphics Processing Units (GPUs) can reach the level of performance that provides the needed effort for image classification. But GPUs are expensive and power-hungry accelerators. Recently, many applications such as self-driving cars need high energy efficiency and real-time performance. So, there is a need to reduce the computational resources to reduce the used power and speed up the calculations.

To speed up running the deep learning algorithms greater than what would be possible with software running on general-purpose computing solutions, hardware acceleration is used as an approach to designing chips to be more convenient to run these algorithms. This kind of acceleration was first introduced nearly 20 years ago. The need for hardware acceleration has been increasing since the discovery of multi-core GPUs to be an alternative to the expensive many-core CPUs. To accurately build the required accelerators for neural networks, we should distinguish between training and inference. Training is an unpredictable process in which the required time to train a model is unknown. Therefore, the computation of this process is intensively high. While inference is a predictable process as it happens after the network is already trained and its time can be estimated. Consequently, there are specific chips dedicated to accelerating training and others for inference. What we are concerned about regarding hardware acceleration is the acceleration of inference time.

To achieve the inference processing constraints, the parallelism concept is introduced. Parallelism is meant to split the required computation into small tasks so that they can be spread among small computational blocks. This approach can be done on-chip level with multi-core processors. There are many products from different companies are introduced to fulfill the tasks of hardware acceleration. These products can be categorized into four different computing devices: Graphical Processing Unit (GPU), Microprocessor, Field Programmable Gate Array (FPGA), and Application Specific Integrated Circuit (ASIC).

Field-Programmable Gate Arrays (FPGAs) are among the most promising platforms that have been considered for efficient high-performance implementations of CNNs with affordable power consumption. FPGAs consist of versatile integrated circuits that provide hundreds of thousands of programmable logic blocks and a configurable interconnect, which enables the implementation of custom-made accelerator architectures in hardware. These have a lot of advantages concerning embedded devices which are providing less computational power to CNNs, high energy efficiency, good performance, and fast development round.

The limitations of the computational resources and memory bandwidth of the FPGA platform must be considered. So, the accelerator structure must be carefully designed to make the computing throughput matches the memory provided by the FPGA. It means that the performance may be degraded due to the bottleneck of the memory. As a result, it’s a must to find ways to increase the speed with efficient memory utilization and energy consumption.

Hardware acceleration can be categorized into two main approaches. The first approach is hardware-independent like reduction in precision using quantization, sharing weights and data reuse. The second approach is customizing the FPGA architecture to be suitable for the algorithm using pipelining, parallelism, and increasing memory utilization.

1.2 Problem Definition

CNN models require a huge number of computations to process a single image due to the convolution operation on the multiple dimensional arrays which represents a computational challenge and results in high power consumption when running on GPUs or CPUs. So, the goal is to create powerful hardware which can be used besides GPUs and CPUs in computers and give us the most powerful computations and the best performance with affordable power consumption. Our project aims at designing and optimizing a hardware accelerator which is specialized hardware performing a specific CNN model algorithm that can be used in real-time applications like autonomous cars.

Most previous work on CNN accelerators mainly focused on computation engine optimization without considering the memory limitation effect on the engine throughput. In this project, we try to match the memory usage with the computation throughput to achieve the optimum solution.

1.3 Solution Approaches

The main objective of this project is to implement CNN architecture on FPGA using two design approaches. The first is to model the algorithm in a high-level language and synthesize it with a High-Level Synthesis (HLS) tool which is not used too much in the literature with large CNN architectures like ours. The second is to make a hardware architecture block diagram and then use a hardware description language to make the actual hardware in a Register Transfer Logic (RTL) manner. Finally, comparing the two approaches in different aspects like the time to develop and the performance of the accelerator in terms of the speed and power consumption to determine which approach is better.

Our performance metric is the amount of power required to complete the computations of a single frame measured in Joule per frame. So, to improve the performance and reduce the number of computations, several techniques of optimization and approximation can be used on FPGAs to accelerate the algorithm like Precision Reduction using fixed-point quantization instead of using 32-bit floating-point number representation as high precision is not always necessary. The energy spent in high precision computations does not lead to more accurate classification by the algorithm. Thus, to reduce the energy consumption of CNN's computations, the main strategy is to quantize its weights and the inputs to its layers. Such quantization leads to a network that is only an approximation of the original network, but with minimal loss in accuracy and without the need to retrain the network which leads to reduced energy consumption. The unique flexibility of the FPGA fabric allows the logic precision to be adjusted to the minimum that a particular network design requires. This allows the accelerator to process more frames per second.

To store all the parameters in the internal memory of the FPGA and eliminate the need for external memory in this design, other techniques of optimization are used in the hardware to increase the memory utilization and minimize the number of block RAMs (BRAMs) required like pipelining to allow the streaming mode of frames and choosing the suitable parallelism in each computation core so that the number of MACs in each layer depends on the number of output filters and the intermediate feature map memories are used as a dual-port and partitioned to support the parallelism used in the computation cores, also for optimization purposes, resource sharing is used in some layers. Some parameters like biases are stored in Look-up Tables (LUTs) due to their low number which will give a bad utilization.

1.4 Methodology

This project has 8 phases as follows and Figure 1.1 shows the flow of steps to do the project.

1. **Software Modelling:** Choosing a suitable CNN model with good accuracy to turn it into hardware and getting the software model which is pre-trained on the ImageNet dataset to run on a CPU and GPU to validate its performance and accuracy using the ImageNet validation set. Then quantizing the model to use fixed-point representation to determine the best data representation to use in the hardware without affecting the accuracy so much.

2. **Surveying on the hardware implementation techniques for CNN:** Getting familiar with how CNN can be implemented in hardware by reading papers and old theses.

3. **Writing the CNN model code in a high-level language** to be used in the High-Level Synthesis (HLS) tool and implementing the hardware.
4. **Defining the hardware architecture block diagram** which will implement the CNN model algorithm.
5. **Writing the Register Transfer Logic (RTL) code** to implement the hardware and checking the timing after the placement and routing.
6. **Testing** the implemented hardware by the HLS and RTL approaches on FPGA to verify their functionality.
7. **Optimizing** the implemented accelerator in the two approaches HLS and RTL.
8. **Comparing the two approaches** and deciding the best approach.

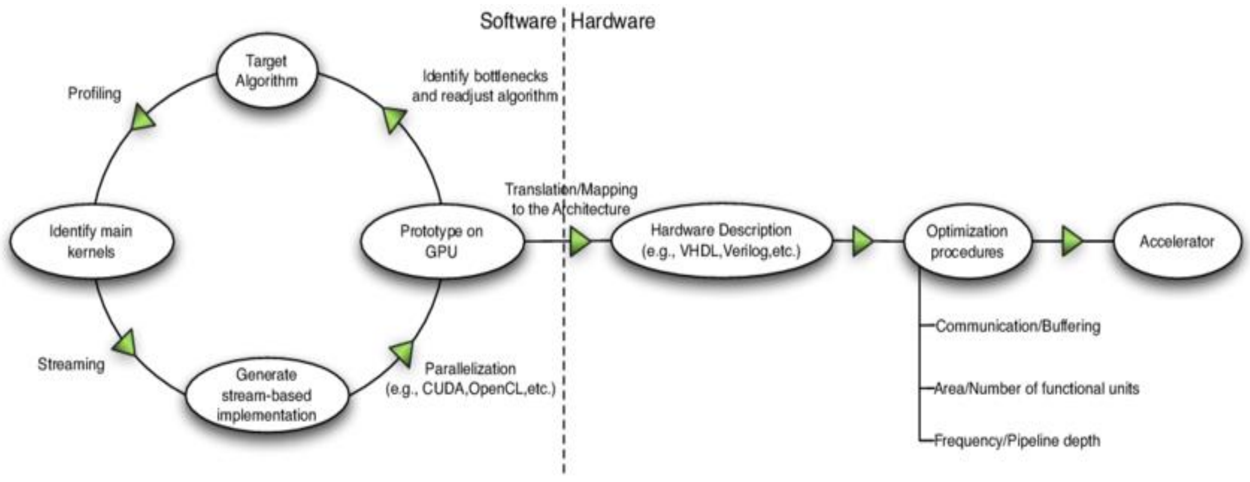

Software
Hardware
Target Algorithm
Profiling
Identify bottlenecks and readjust algorithm
Identify main kernels
Prototype on GPU
Translation/Mapping to the Architecture
Hardware Description (e.g., VHDL,Verilog,etc.)
Optimization procedures
Accelerator
Streaming
Generate stream-based implementation
Parallelization (e.g., CUDA,OpenCL,etc.)
Communication/Buffering
Area/Number of functional units
Frequency/Pipeline depth

Figure 1.1 Project development methodology flow steps

1.5 Thesis Organization

A brief introduction of the contents of each chapter is explained as follows:

Chapter 2 provides background information on neural networks and especially CNN, it discusses the main layers of CNN and their operations and provides information about different CNN architectures used for image classification and the image classification datasets, and also it discusses several methods to improve the efficiency of deep learning implementation. Finally, it discusses the background of FPGAs, including a brief overview of FPGA resources, internal components, and design flow.

Chapter 3 provides information on the chosen CNN for the project which is ShuffleNet V2 and why we chose it among the other architectures then has an overview of the block diagram of the architecture and its layers and parameters. Then each building block is discussed briefly.

Chapter 4 is about the software modeling phase of the project, including the pre-trained model, the chosen fixed-point representation, and the results of the model quantization.

Chapter 5 discusses the RTL design methodology with thorough details about the computation cores, memories, and controllers. Also discusses the verification of the design by showing the testing strategy done to validate the functionality of the design.

Chapter 6 points out the optimizations done in the RTL of the accelerator to reach a better latency, area, and power.

Chapter 7 discusses the HLS approach, the HLS flow, and implementation. It also sheds the light on the limitations of HLS.

Chapter 8 presents the final results obtained after optimizations including utilization, power, and timing, then introduces a benchmark for our design and two different designs implementing a machine learning algorithm on FPGA. Finally comparing the results obtained from RTL with HLS. It also shows the burning of the bit stream of both RTL and HLS on the chosen FPGA.

Chapter 9 concludes the previous work and introduces our ideas for future work.

2 Chapter 2: Background and Related Work

2.1 Neural Networks Overview

The development and advancement of neural networks is the key to training computers to think and understand the world in the manner that the brain does. Essentially, a neural network imitates the human brain. Brains cells which are called neurons are connected via synapses. This is represented as a graph of nodes (neurons) with weighted edges connecting them (synapses).

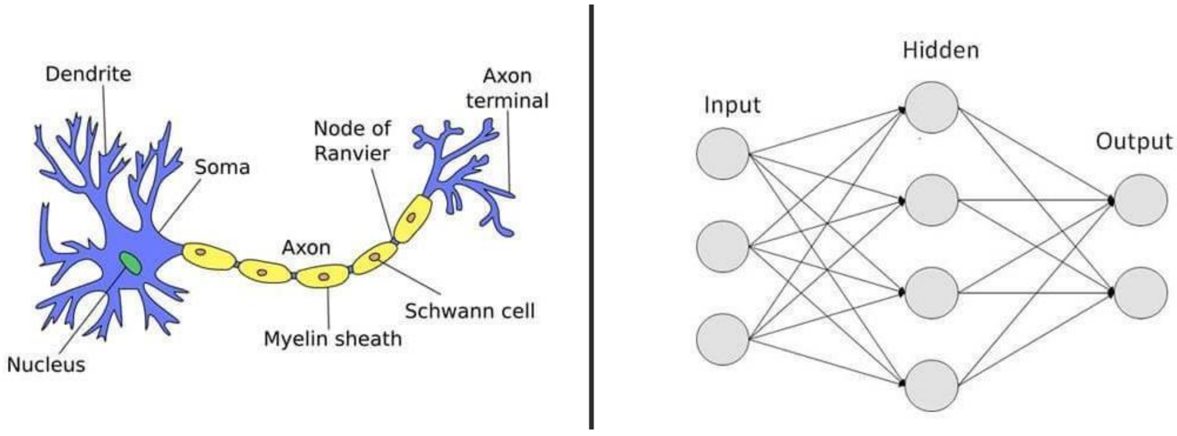


Figure 2.1 The analogy between biological neural system and computer neural network

Looking at Figure 2.1, it’s clear that an artificial neural network (ANN) mimics a biological neuron, and that the brain’s processes can be modeled by building a neural network on a computer. A neural network has input, hidden and output neurons, which are connected by weighted synapses. The amount of forward propagation that passes through the neural network is determined by these weights. They can then be adjusted while the neural network is learning via backpropagation. This forward and backward propagation technique is repeated iteratively on each piece of data in a training data set. The larger the data set and the more diversity in the data set, the more the neural network will learn and the better it will be at predicting outputs.

A neural network is a connected graph with input, hidden and output neuron layers with weighted edges. On the other hand, a Deep Neural Network (DNN) has more layers that might reach 20 or 1,000 layers of neurons.

A neural network is just a core architecture. There are different types of neural networks. For example, Convolutional Neural Networks (CNNs) are very effective for Computer Vision applications and recurrent Neural Networks (RNNs) are also very popular in applications like machine translation and speech recognition.

The network begins with an input layer that receives the input data. Weights are the lines that connect the hidden layers. Each neuron in the hidden layer processes the inputs, which then sends an output to the next hidden layer, and eventually into the output layer.

2.2 CNN Overview

A Convolutional Neural Network (ConvNet/CNN) is a Deep Learning algorithm that can take in an input image, and assign relevance (learnable weights and biases) to multiple objects in the image. The pre-processing required in a ConvNet is much lower compared to other classification algorithms.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlaps to cover the entire visual area.

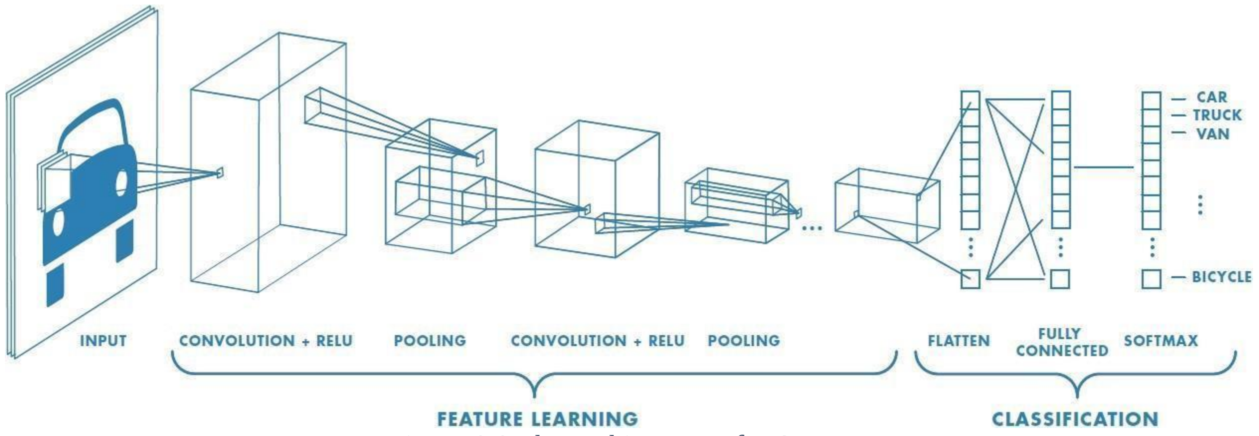


Figure 2.2 The architecture of a ConvNet

CNN is a type of deep learning model for processing data that has a grid pattern, such as images, and is designed to automatically and adaptively learn spatial hierarchies of features, from low- to high-level patterns.

In digital images, pixel values are stored in a two-dimensional (2D) grid, and a small grid of parameters called the kernel, where an optimizable feature extractor is applied at each image position, which makes CNNs highly efficient for image processing, since a feature may occur anywhere in the image. As one layer feeds its output into the next layer, extracted features can hierarchically and progressively become more complex as shown in Figure 2.2. The process of optimizing parameters such as kernels is called training, which is performed to minimize the difference between outputs and ground truth labels through an optimization algorithm called backpropagation and gradient descent.

2.3 General CNN Layers

CNN is a mathematical construct that is typically composed of three types of layers (or building blocks): convolution, pooling, and fully connected layers. The first two, convolution and pooling layers, perform feature extraction, whereas the third, a fully connected layer, maps the extracted features into the final output, such as classification.

Usually, the input to the CNN is an RGB image which is a three-dimensional array that explicitly stores a color value for each pixel with dimensions $n_H * n_W * 3$ as shown in Figure 2.3.

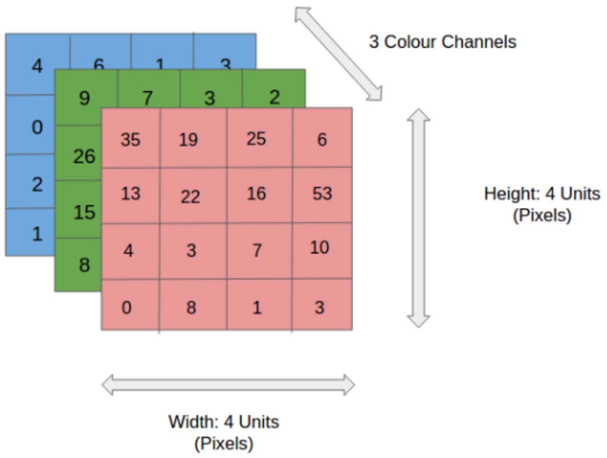
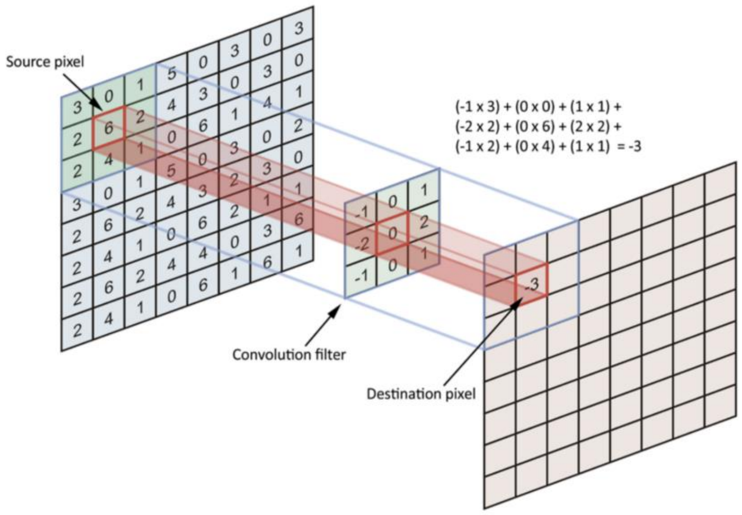


Figure 2.3 RGB input image

2.3.1 Convolution Layer

The Conv layer is the main block of a CNN that does most of the computations. It works by dividing the image into small regions and convolving them with a specific filter (multiplying weights of the filter or kernel (weights) with corresponding receptive field elements), then sliding these filters over the input feature maps as shown in Figure 2.4. Each of these weight filters can be thought of as a feature identifier [1].


Source pixel
(-1 x 3) + (0 x 0) + (1 x 1) +
(-2 x 2) + (0 x 6) + (2 x 2) +
(-1 x 2) + (0 x 4) + (1 x 1) = -3
Convolution filter
Destination pixel
Figure 2.4 Convolving an image with a kernel

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

1. **Filters:**

   The Conv layer's parameters consist of a set of learnable filters which work as feature detectors (edges, simple colors, and curves).

2. **Stride:**

   Stride is the number of pixels by which the filter matrix slides over the input matrix.

2.3.2 Pooling Layer

The Pooling layer (also called sub-sampling) reduces the dimensionality of the input feature maps but retains the most important information. The number of output feature maps is identical to that of input feature maps, while the dimensions of each feature map scale down according to the size of the sub-sampling window (called also kernel). For example, for a pooling layer, there is an average or maximum. Max-pooling being the most popular, this takes a filter P x P and a stride of length S, it then applies it to the input volume and outputs the maximum number in every sub-region that the filter convolves around as shown in Figure 2.5. The pooling layer could be after each convolution layer, after a group of layers, or at the end before the classification layer.

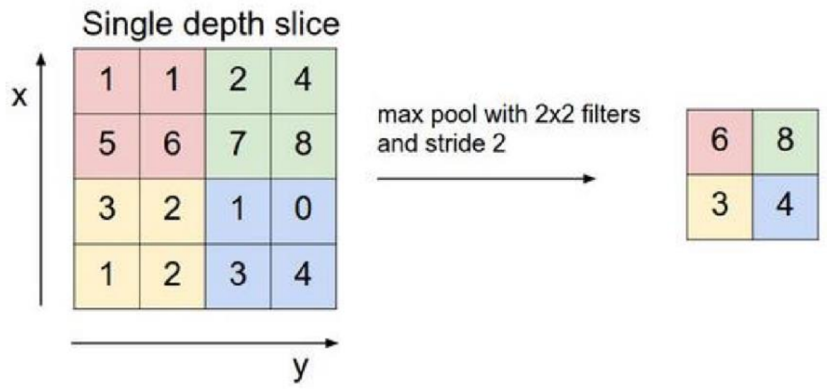


Figure 2.5 Applying Max-pooling to a single depth slice

2.3.3 Fully Connected Layer

The way this fully connected neural network layer (FC) works is that it looks at the output of the previous layer (which represents the activation maps of high-level features) and determines which features most correlate to a particular class by unrolling the input features and the weights and multiply them and outputs an N-dimensional vector where N is the number of classes as shown in Figure 2.6. Also, this layer is followed by SoftMax to show the most correlated class to the input.

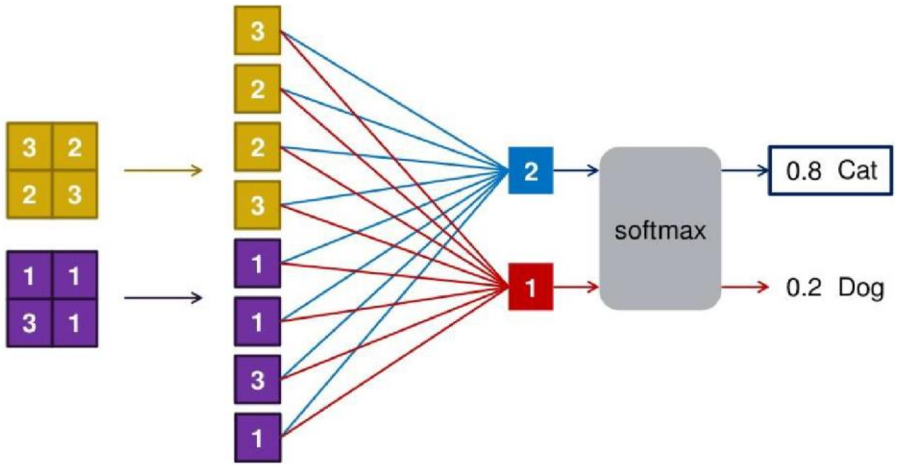


Figure 2.6 The fully connected layer

2.3.4 Activation Layer (ReLU)

After each convolutional layer, it is a convention to apply a nonlinear layer (or activation layer) immediately afterward. The purpose of this layer is to introduce nonlinearity to a system that has just been computing linear operations during the convolutional layers (just element-wise multiplications and summations). In the past, nonlinear functions like "tanh" and "sigmoid" were used, but researchers found out that rectified linear unit (ReLU) layers work far better because the network can train a lot faster (because of the computational efficiency) without making a significant difference to the accuracy [2].

It turns out that one of the problems of using sigmoid functions in machine learning is that there are regions where the slope of the function would gradient to nearly zero and so learning becomes slow because when you implement gradient descent and the gradient is zero the parameters just change very slowly and so learning is very slow whereas by changing the activation function of the neural network to use ReLU function, the gradient is equal to one for all positive values of input and so the gradient is much less likely to gradually shrink to zero and has made the gradient descent algorithm work much faster and this allows us to train bigger neural networks.

The RELU layer applies the function $F(x)$=max (0, x) to all of the values in the input volume as shown in Figure 2.7. In basic terms, this layer just changes all the negative input values to zero and passes the positive values as it is. Thus, increases the nonlinear properties of the model and the overall network without affecting the receptive fields of the convolutional layer.

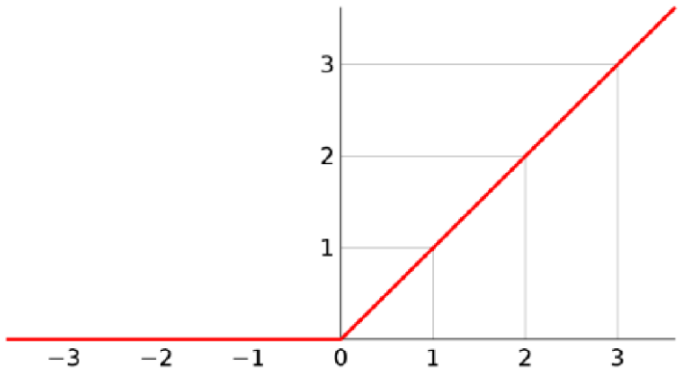


Figure 2.7 The ReLU activation function

2.3.5 Batch Normalization Layer

Neural networks learn slowly if the distribution of their input changes over time. This issue is called the internal covariance shift. Batch normalization addresses this issue by bringing the values to zero mean and unit variance. It then multiplies the normalized results by a learnable parameter (new variance) and adds a learnable parameter to it (new mean) and so giving the network the ability to choose suitable distributions. It also smoothens the flow of gradient and acts as a regulating factor, which improves the generalization of the network without relying on dropout.

2.4 Image Classification Dataset

Image Classification is a fundamental task that attempts to comprehend an entire image as a whole. The goal is to classify the image by assigning it to a specific label. Typically, Image Classification refers to images in which only one object appears and is analyzed. In contrast, object detection involves both classification and localization tasks and is used to analyze more realistic cases in which multiple objects may exist in an image. Figure 2.8 presents a comparison of the datasets' popularity.

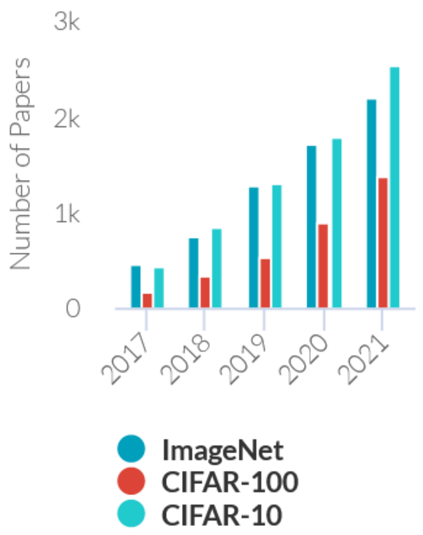


Figure 2.8 Comparison of the popularity of different image classification datasets

2.4.1 ImageNet

The ImageNet dataset contains 14,197,122 annotated images according to the WordNet hierarchy. Since 2010 the dataset has been used in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a benchmark in image classification and object detection. The publicly released dataset contains a set of manually annotated training images. A set of test images is also released, with the manual annotations withheld. ILSVRC annotations fall into one of two categories: (1) image-level annotation of a binary label for the presence or absence of an object class in the image, e.g., "there are cars in this image" but "there are no tigers," and (2) object-level annotation of a tight bounding box and class label around an object instance in the image, e.g., "there is a screwdriver centered at position (20,25) with a width of 50 pixels and height of 30 pixels". The ImageNet project does not own the copyright of the images, therefore only thumbnails and URLs of images are provided.

- Total number of non-empty WordNet synsets: 21841
- Total number of images: 14197122
- Number of images with bounding box annotations: 1,034,908
- Number of synsets with SIFT features: 1000
- Number of images with SIFT features: 1.2 million

2.4.2 CIFAR-10

The CIFAR-10 dataset (Canadian Institute for Advanced Research, 10 classes) is a subset of the Tiny Images dataset and consists of 60000 32x32 color images. The images are labeled with one of 10 mutually exclusive classes: airplane, automobile (but not truck or pickup truck), bird, cat, deer, dog, frog, horse, ship, and truck (but not pickup truck). There are 6000 images per class with 5000 training and 1000 testing images per class.

The criteria for deciding whether an image belongs to a class were as follows:

- The class name should be high on the list of likely answers to the question “What is in this picture?”
- The image should be photo-realistic. Labelers were instructed to reject line drawings.
- The image should contain only one prominent instance of the object to which the class refers. The object may be partially occluded or seen from an unusual viewpoint as long as its identity is still clear to the labeler.

2.4.3 MNIST

The MNIST database (Modified National Institute of Standards and Technology database) is a large collection of handwritten digits. It has a training set of 60,000 examples and a test set of 10,000 examples. It is a subset of a larger NIST Special Database 3 (digits written by employees of the United States Census Bureau) and Special Database 1 (digits written by high school students) which contain monochrome images of handwritten digits. The digits have been size-normalized and centered in a fixed-size image. The original black and white (bi-level) images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of the anti-aliasing technique used by the normalization algorithm. The images were centered in a 28x28 image by computing the center of mass of the pixels and translating the image to position this point at the center of the 28x28 field.

2.5 CNN Architectures for image classification

In recent years, the world witnessed the birth of numerous CNNs. These networks have gotten so deep that it has become extremely difficult to visualize the entire model. We stop keeping track of them and treat them as black-box models. This section is a visualization of 3 small CNN architectures and then ShuffleNet is explained in detail. These illustrations provide a more compact view of the entire model of each architecture. Figure 2.9 shows the accuracy and number of operations of many CNN architectures.

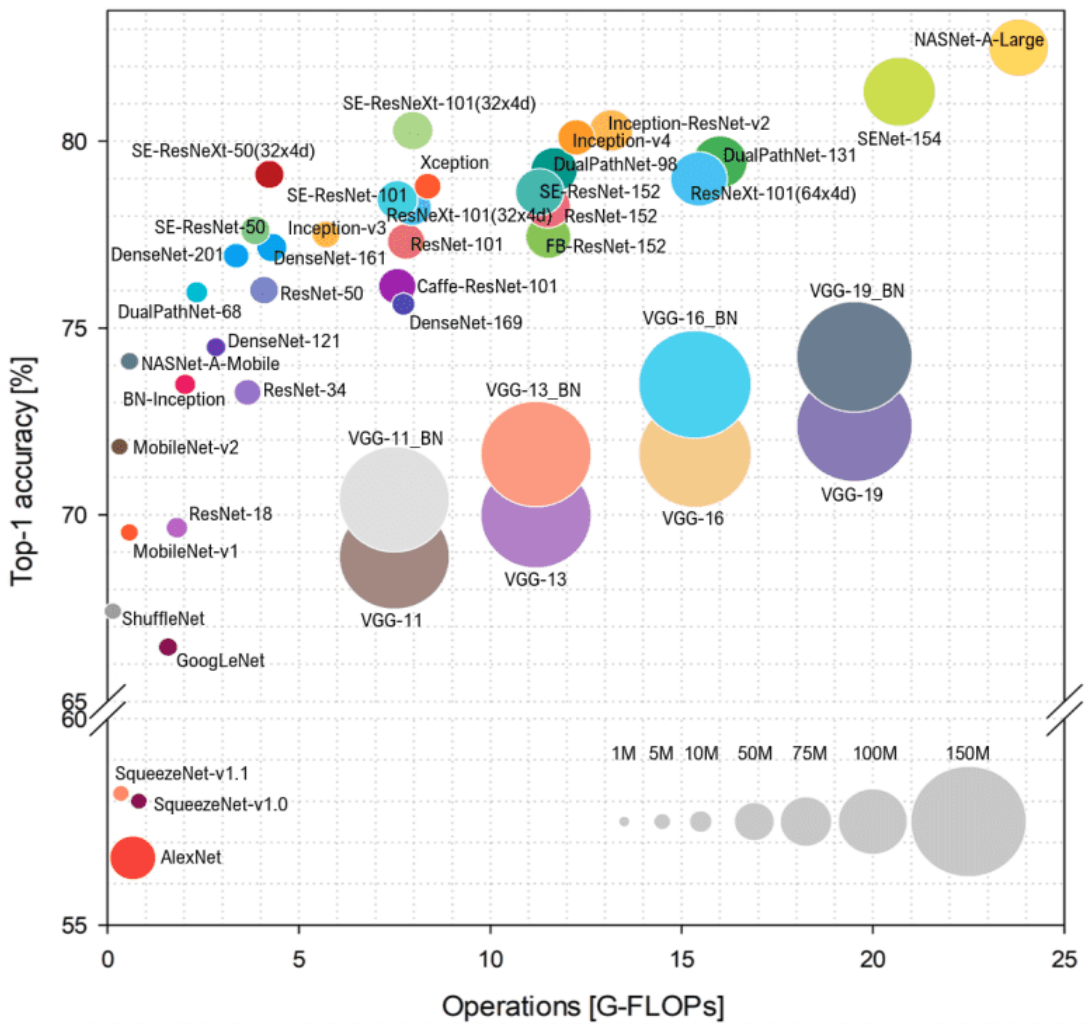


Figure 2.9 Accuracy and number of operations of many CNN architectures

2.5.1 SqueezeNet

SqueezeNet was one of the first small models that performs well. It has ImageNet accuracy similar to AlexNet, the convnet that started the deep learning revolution in 2012, but with 50 times fewer parameters [3].

The idea: Create a neural network using more efficient building blocks, to significantly reduce the number of parameters used by typical CNNs of the time. AlexNet consisted of five convolution layers with large kernels, followed by two massive fully-connected layers. SqueezeNet uses only small Conv layers with 1×1 and 3×3 kernels.

The main building block in SqueezeNet is the Fire module shown in Figure 2.10.

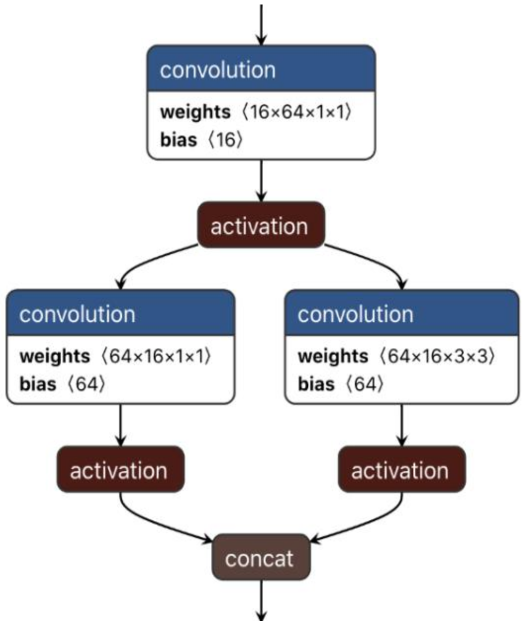


Figure 2.10 The Fire module of SqueezeNet

It first has a squeeze layer. This is a 1×1 convolution that reduces the number of channels, for example from 64 to 16 in the above picture. The purpose of the squeeze layer is to compress the data so that the 3×3 convolution doesn't need to learn so many parameters.

This is followed by an expand block that has two parallel convolution layers: one with a 1×1 kernel, the other with a 3×3 kernel. These Conv layers also increase the number of channels again, from 16 back to 64. Their outputs are concatenated, so the output of this fire module has 128 channels in total.

SqueezeNet has eight of these Fire modules in succession, sometimes with max-pooling layers between them. There are no fully-connected layers. At the very end is a convolution layer that performs the classification, followed by global average pooling. (Interestingly, the classification layer both has ReLU and SoftMax applied to it.).

To measure the accuracy of the model, we calculate two metrics; top-1 and top-5 accuracies as shown in Table 2.1, where the top-k metric computes the number of times where the correct label is among the top-k labels predicted (ranked by predicted scores).

ImageNet classification accuracy:

Table 2.1 Accuracy and number of parameters of SqueezeNet v1.1

	parameters	**top-1**	**top-5**
SqueezeNet v1.1	1.25 M	57.5%	80.3%

2.5.2 SqueezeNext

SqueezeNext is based on SqueezeNet but with some architectural improvements. The new SqueezeNext block is shown in Figure 2.11.

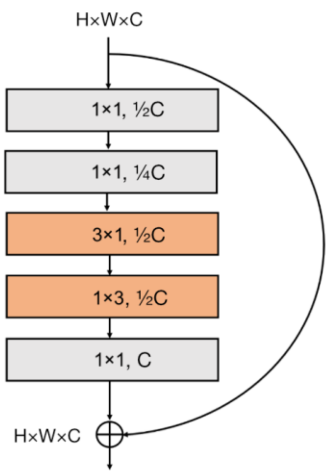


Figure 2.11 The SqueezeNext building block

There are five Conv layers in each block, all with the batch norm and ReLU [4]:

- The first two layers are bottleneck layers, i.e., 1×1 convolutions that reduce the number of channels. If `C` is the number of input channels, the first bottleneck layer reduces this to `C/2` and the second one to `C/4`.

  As with the original SqueezeNet, these bottlenecks are used to cut back the number of parameters needed by the convolution layers that do the actual filtering.

- Unlike the original SqueezeNet, there is no longer a 3×3 convolution. Instead, this has been split up into two smaller convolutions: 3×1 and 1×3. These both have `C/2` filters. Splitting it into two smaller layers decreases the number of parameters needed. At the same time, it increases the depth of the network, which generally improves the model.

  The order of these two convolutions alternates. If this block has 3×1 followed by 1×3, the next block does 1×3 first and 3×1 second, and so on.

  The fifth and final layer is an expansion layer, a 1×1 convolution that brings the number of channels back to `C`.

As is common with modern architectures, there is also a residual connection that is used to allow gradients to flow through a network directly, without passing through non-linear activation functions. Non-linear activation functions, by the nature of being non-linear, cause the gradients to explode or vanish (depending on the weights). Adding this connection forms conceptually a 'bus' that flows right the way through the network, and in reverse, the gradients can flow backward along with it too. It also helps the model learn more features, which increases the model's accuracy.

Recall that the original SqueezeNet had convolutions in both branches, but that doesn't happen here: the residual branch has no convolution in it. However, there is a small exception to this rule.

The blocks in SqueezeNext are organized into four sections. In each new section, the spatial dimensions of the feature maps are halved. To achieve this, the very first bottleneck convolution has stride 2. Now the residual branch for that block must also have a stride 2 convolution, otherwise, the outputs of both branches cannot be summed up.

Also, in this first block, the number of output channels can be different than the number of input channels, so the branch with the residual connection must match that too.

The very first layer in SqueezeNext is a 7×7 convolution with 64 filters and stride 2. This layer does not use zero padding. It is immediately followed by max-pooling, so the input size of 227×227 pixels is very quickly reduced to 55×55 pixels.

Because of the pooling layer, the first convolution in the very first section has a stride of 1 instead of 2. Not sure why they use a max-pooling layer there and not anywhere else.

At the end of the model is a single fully-connected layer that performs the actual classification. Right before this layer is a bottleneck layer that reduces the number of channels, which saves a lot of parameters in the FC layer, and a global average pooling to reduce the spatial dimensions. (SqueezeNet used a Conv layer to do the classification, and did global pooling last.)

SqueezeNext, like MobileNet, is not just a single fixed design but a family of possible architectures:

- A width multiplier determines the number of filters in each block. The paper examines width multipliers of 1.0, 1.5, and 2.0.
- You can also change the number of building blocks used. The paper shows models with 23, 34, and 44 blocks. As you might expect, more blocks give better results.
- You can vary how many blocks there are in each section. For example, the "v5" variant has relatively few blocks in the first two sections (which work on feature maps of sizes 55×55 and 28×28), many blocks in the third section (feature map size 14×14), and only one block in the last section (size 7×7).
- In other variants of SqueezeNext, the first bottleneck and the 1×3 and 3×1 layers use group convolution with a group size of two, which cuts the number of parameters in half for those layers. (This is a little bit like making them depth-wise convolutions, which is a grouped convolution where the number of groups equals the number of channels.)

The authors of SqueezeNext did not use depth-wise separable convolutions on purpose, because these "do not give good performance on some embedded systems due to its low arithmetic intensity (ratio of computing to bandwidth)."

Interesting, because depth-wise separable convolutions work very well on the iPhone GPU (see the success of MobileNet); the paper does not go into details about what sort of embedded systems they intended SqueezeNext for.

To try out different design approaches, the authors simulated the performance of possible architectures on a hypothetical neural network accelerator chip. This is different from MnasNet, which tries out the architectures on actual hardware. Because of this, I'm not sure how well SqueezeNext's design decisions translate to the real world — or at least to iPhone hardware.

The paper also shows results for models that use Iterative Deep Aggregation (IDA). Instead of just having a linear sequence of blocks that only classifies at the very end, with IDA we already predict each block, and these predictions are then combined with those of the final classification layer. This idea improves accuracy (a little) but at the cost of more parameters. (It doesn't appear to be a major feature of SqueezeNext).

ImageNet classification accuracy:

Table 2.2 Accuracy and number of parameters of SquezeNext

	parameters	top-1	top-5
1.0-SqNxt-23	0.7 M	59.05%	82.60%
1.0-SqNxt-44	1.2 M	62.64%	85.15%
1.0-SqNxt-44-IDA	1.5 M	63.75%	85.97%
2.0-SqNxt-23v5	3.2 M	67.44%	88.20%

The accuracy scores for a few different variations of the architecture are shown in Table 2.2. The 1.0-SqNxt-44 model can be compared to the old SqueezeNet. While it scores better than the original, its accuracy is still lower than MobileNet v1, even if we compare it with a version of MobileNet that has a similar number of parameters (0.5 depth multiplier, top-1 score 63.3%).

The SqueezeNext paper claims better top-5 accuracy than MobileNet v1 with 1.3× fewer parameters. To get this result, they compare their 2.0-SqNext23v5 model with MobileNet-1.0-224. However, the version of MobileNet they're using is not as good as the official one — they trained it using the same hyperparameters as their models, for getting a fairer comparison.

2.5.3 MobileNet V2

MobileNet V2 uses depth-wise separable convolutions, and its main building block is shown in Figure 2.12.

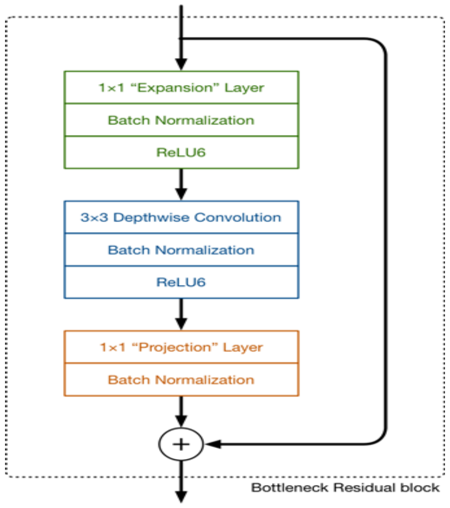


Figure 2.12 The MobileNet V2 building block

This time there are three convolutional layers in the block. The last two are the ones we already know: a depth-wise convolution that filters the inputs, followed by a 1×1 pointwise convolution layer. However, this 1×1 layer now has a different job.

In V1 the pointwise convolution either kept the number of channels the same or doubled them. In V2 it does the opposite: it makes the number of channels smaller. This is why this layer is now known as the projection layer — it projects data with a high number of dimensions (channels) into a tensor with a much lower number of dimensions [5].

For example, the depth-wise layer may work on a tensor with 144 channels, which the projection layer will then shrink down to only 24 channels. This kind of layer is also called a bottleneck layer because it reduces the amount of data that flows through the network. (This is where the "bottleneck residual block" gets its name from the output of each block is a bottleneck.)

The first layer is the new kid in the block. This is also a 1×1 convolution. Its purpose is to expand the number of channels in the data before it goes into the depth-wise convolution. Hence, this expansion layer always has more output channels than input channels as shown in Figure 2.13, and does the opposite of the projection layer.

Exactly how much the data gets expanded is given by the expansion factor. This is one of those hyperparameters for experimenting with different architecture tradeoffs. The default expansion factor is 6.

For example, if there is a tensor with 24 channels going into a block, the expansion layer first converts this into a new tensor with 24 * 6 = 144 channels. Next, the depth-wise convolution applies its filters to that 144-channel tensor. And finally, the projection layer projects the 144 filtered channels back to a smaller number, say 24 again.

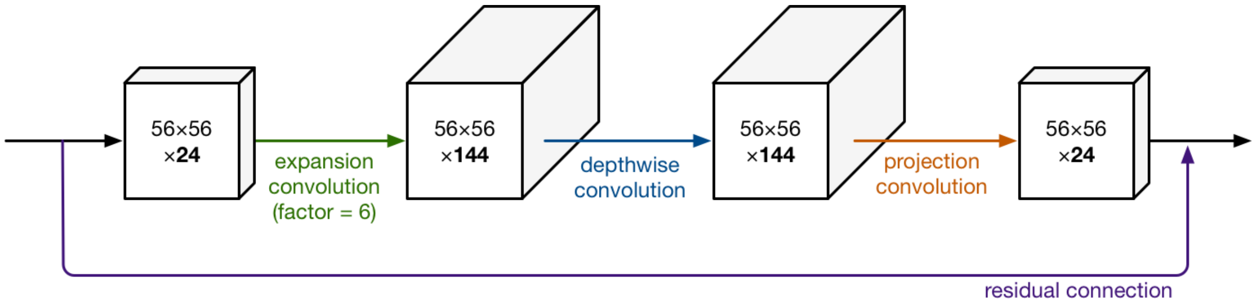


Figure 2.13 Expanding the data before depth-wise convolution

So, the input and the output of the block are low-dimensional tensors, while the filtering step that happens inside the block is done on a high-dimensional tensor.

The second new thing in MobileNet V2's building block is the residual connection. This works just like in ResNet and exists to help with the flow of gradients through the network. (The residual connection is only used when the number of channels going into the block is the same as the number of channels coming out of it, which is not always the case as every few blocks the output channels are increased.)

As usual, each layer has batch normalization and the activation function is ReLU6. However, the output of the projection layer does not have an activation function applied to it. Since this layer produces low-dimensional data, the authors of the paper found that using a non-linearity after this layer destroyed useful information.

The full MobileNet V2 architecture, then, consists of 17 of these building blocks in a row. This is followed by a regular 1×1 convolution, a global average pooling layer, and a classification layer. (Small detail: the very first block is slightly different, it uses a regular 3×3 convolution with 32 channels instead of the expansion layer.)

The authors of MobileNet v2 call this an inverted residual because it goes between the bottleneck layers, which have only a small number of channels. Whereas a normal residual connection from ResNet goes between layers that have many channels.

The full MobileNet v2 architecture consists of 17 of these building blocks in a row. This is followed by a regular 1×1 convolution, a global average pooling layer, and a classification layer. ImageNet classification accuracy is shown in Table 2.3.

Table 2.3 Accuracy and number of parameters of MobileNet V2

	parameters	**top-1**	**top-5**
MobileNet v2	3.47 M	71.8%	91.0%

2.6 Hardware design Methodology

2.6.1 FPGA Introduction and main resources

FPGAs (Field Programmable Gate Arrays) are programmable integrated circuits that have been designed to be configured by a designer after manufacturing. FPGAs consist of a matrix of configurable logic blocks (CLB) which are connected by interconnections that can be programmed using an HDL (hardware description language), Verilog, or VHDL, to implement different functions and applications.

The matrix of CLBs allows the FPGA to perform very complex combinational and sequential logic or logic as simple as logic gates. This can’t be done without the reconfigurable interconnects, in addition, FPGA logic blocks contain simple memory blocks like flip flops or complete memory blocks. This makes FPGAs implement different logic functions with high flexibility.

Microprocessors can be used to implement most digital applications that are related to image processing. But microprocessors have a great hold back in that they execute functions sequentially. But on the other hand, FPGAs are faster for some applications due to their parallelism in operations with flexibility in design and usage of resources. So, where speed is critical, FPGA can be much faster even with limited clock speed. This makes using FPGAs in hardware accelerators that need parallel processing a trend in the digital market.

Graphic processing units (GPUs) have the potential of running with a throughput higher than FPGA can ever reach. But only for algorithms that are especially suited for that. If the algorithm is not optimal, the GPU will lose a lot of performance. FPGA on the other hand runs slower, but you can implement problem-specific hardware that will be very efficient and get stuff done faster. FPGAs consume less power compared to GPUs, where the main reason for GPUs being power-hungry is that they require additional complexity around their compute resources to facilitate software programmability. The decreased power consumption in FPGAs might have a big impact on a lot of applications where great performance isn't the most important factor. However, FPGAs have a higher time to market when compared to GPUs.

In computer vision and other applications, GPUs are used to implement deep learning algorithms. But FPGAs are very flexible hardware with a variety of interfaces that allow designers to configure them according to their specifications and constraints in a very optimal way, which allows them to obtain specifications like area, speed, and power consumption by managing their limited resources. A comparison between FPGAs and GPUs is shown in Figure 2.14 and the comparison between FPGAs, CPUs, and GPUs is summarized in the table below, which demonstrates how FPGAs excel in resource optimization in terms of power and cost.

Therefore, FPGA is a perfect choice as a design style to implement our project (compromises between speed, area, and power) instead of using a CPU or GPU. Table 2.4 shows a comparison between CPU, FPGA, and GPU.

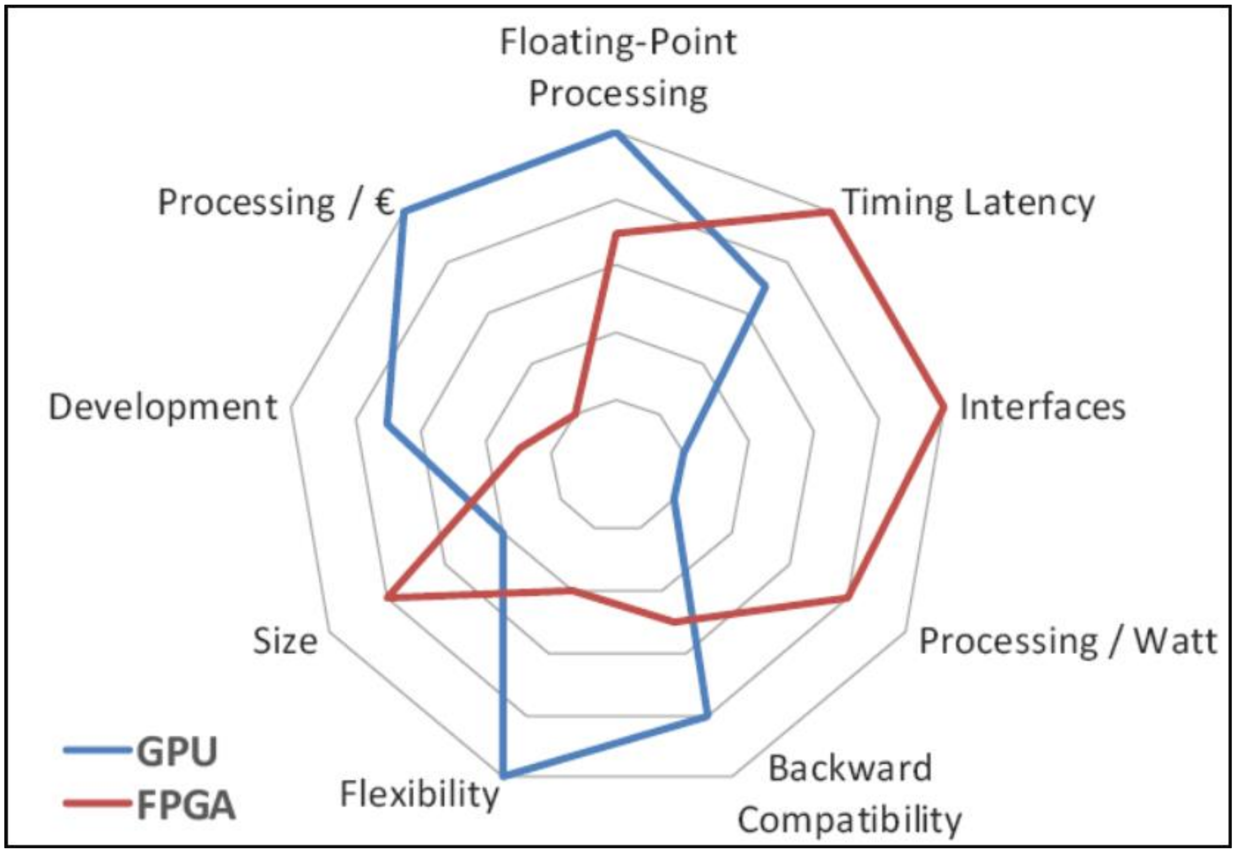


Figure 2.14 FPGA Vs GPU

Table 2.4 Comparison between CPU, FPGA, and GPU

	CPU	FPGA	GPU
Throughput	Low	Intermediate	High
Power consumption	Low	Intermediate	High
Time to market	Low	Intermediate	Low

2.6.2 FPGA Internal Components

FPGAs are popular among digital designers because they provide a flexible design platform. They're made up of a variety of logic blocks that may be used to implement a variety of functions and boost the system's flexibility. As shown in Figure 2.15.

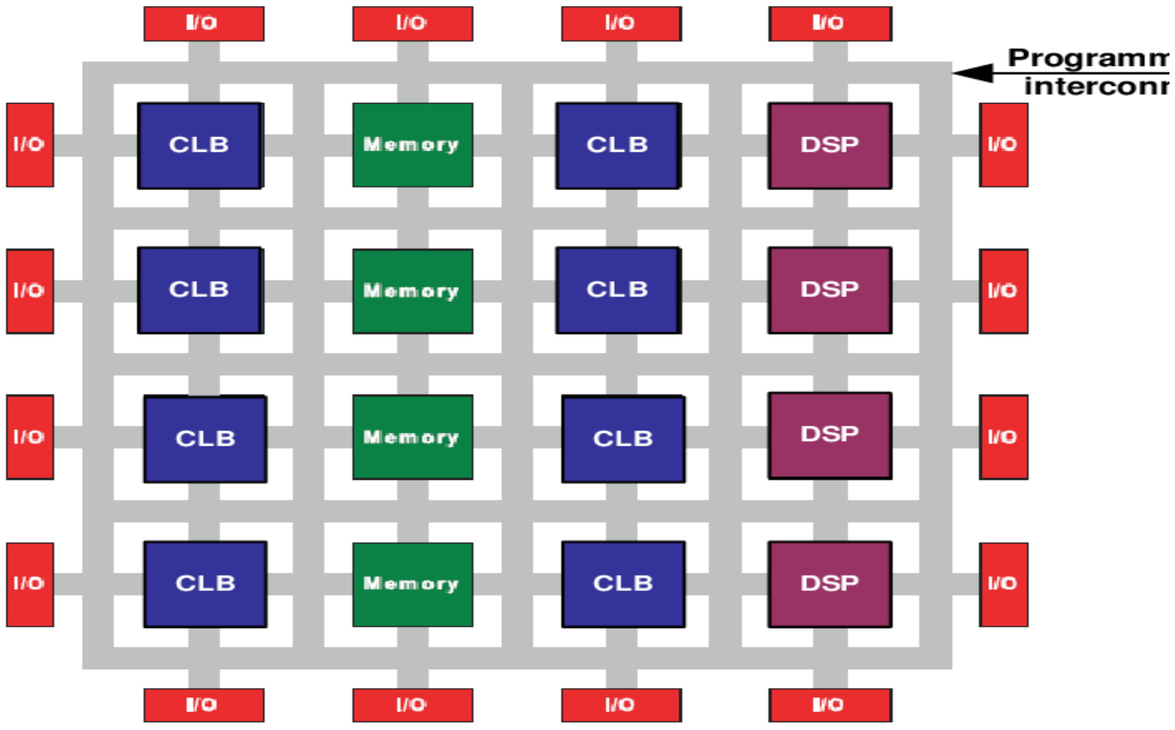


Figure 2.15 FPGA components

2.6.2.1 Configurable Logic Blocks (CLBs)

FPGA logic resources are the fundamental computational components that implement and store the target circuit's functionality. A CLB is still the most basic element of an FPGA, allowing the user to design nearly any logical function into the chip [6]. CLB components perform complex logic, implement memory functions, and synchronize code on the FPGA when connected by routing resources. Each CLB is made up of several slices that are further decomposed into look-up tables (LUTs), flip-flops (FFs), and multiplexers (MUXs) as shown in Figure 2.16.

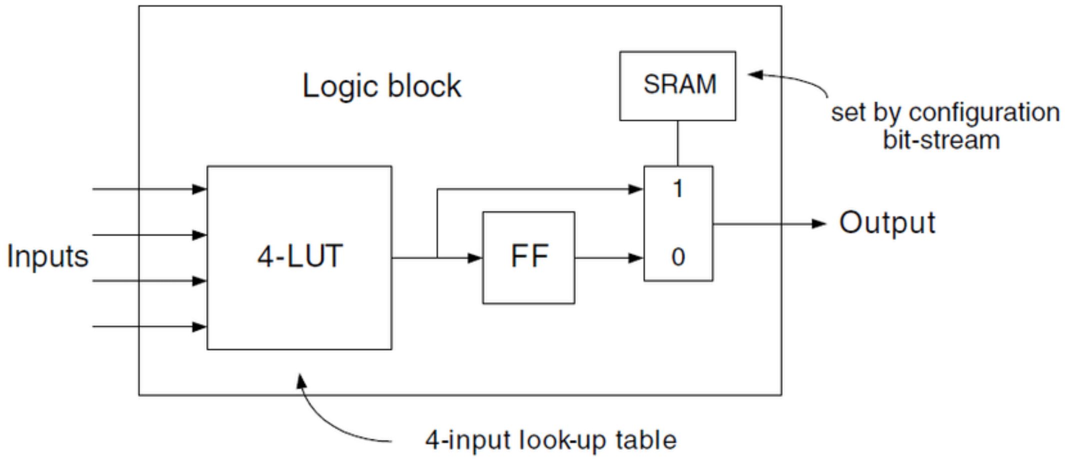


Figure 2.16 FPGA CLB

2.6.2.2 Configurable I/O Blocks

To get signals onto the chip and send them off again, a Configurable input/output (I/O) Block is used, as illustrated in Figure 2.17. It has three-state and open-collector output controls, as well as an input buffer and an output buffer. Pull-up and pull-down resistors are commonly found on the chip's outputs and can be used to terminate signals and buses without the need for discrete resistors external to the device.

The output polarity can normally be configured for active high or active low output, and the slew rate can usually be programmed for fast or slow rise and fall times. Flip-flops are typically found on outputs, allowing clocked signals to be delivered straight to pins without substantial delay, making it easier to satisfy the setup time required for external devices. Flip-flops on the inputs, meanwhile, lessen the delay on a signal before it reaches a flip-flop, lowering the hold time needed.

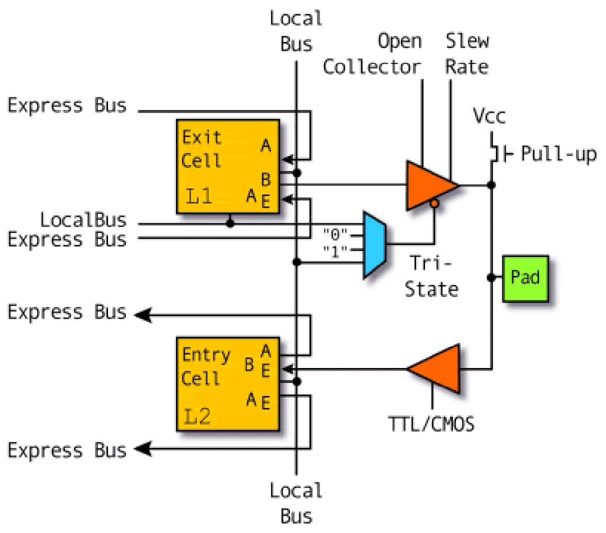


Figure 2.17 FPGA Configurable I/O block

2.6.2.3 Programmable Interconnect

A hierarchy of interconnect resources can be shown in Figure 2.18, long lines can be mainly used to connect critical CLBs on the chip that are physically separated from one another without causing a significant delay. Within the chip, these long lines can also be used as buses.

Short lines are also used to link separate CLBs that are physically near to one another. Transistors are used to turn connections between lines on and off. The FPGA also has multiple programmable switch matrices for connecting these long and short lines in unique, flexible combinations.

Global clock lines are unique long lines that are designed for low impedance and consequently fast propagation times. The clock buffers and each clocked element in each CLB are connected to them. This ensures that the clock signals arrive at different flip-flops inside the device that has minimal skew.

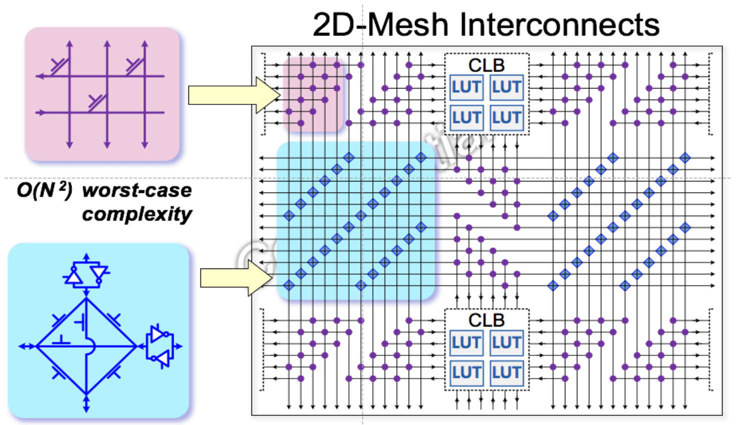


Figure 2.18 FPGA interconnect

2.6.2.4 Clock driver

Clock drivers are special I/O blocks with specific high-drive clock buffers that are distributed across the chip. These buffers connect to the clock input pads and drive the clock signals to the above-mentioned global clock lines. Low skew times and short propagation times are the goals of these clock lines. With FPGAs, synchronous design is required since absolute skew and delay can only be ensured on the global clock lines.

Clock drivers are configured as a pre-implemented module in FPGA to solve questions that occur when all designs are connected to a single clock signal. The first issue is that this port has a significant fanout because it is connected to all flip-flop nodes; thus, a powerful driver should be implemented to ensure that the clock propagation delay is kept to a minimum. In addition, the buffer tree should be used to save the global skew minimum and constant across all blocks.

2.6.2.5 Block RAM

It's a specialized RAM block that stores data on the FPGA without using any extra LUTs, whereas distributed RAM uses LUTs. Block RAM is slower than FF-based memory but quicker than off-chip memory. It is also smaller in size than off-chip memory. It operates as a relatively large memory structure.

2.6.2.6 DSP Cores

DSPs (Digital Signal Processors), as shown in Figure 2.20, are another common type of core available as an IP or embedded core. These are digital signal manipulators that are specialized processors. They're typically used for video or audio signal filtering and compression. The Multiply-Accumulate block, or MAC, is implemented as a DSP slice and is primarily used as a building block for complex DSP applications.

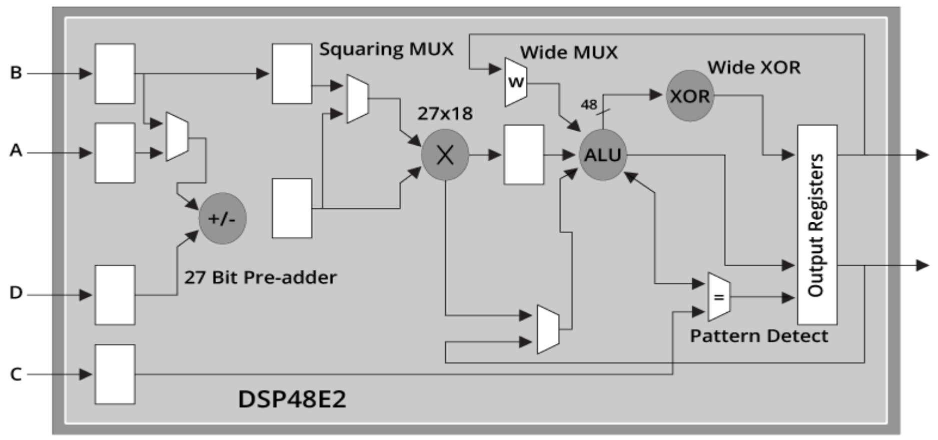


Figure 2.19 DSP Core

2.6.3 FPGA Design Flow

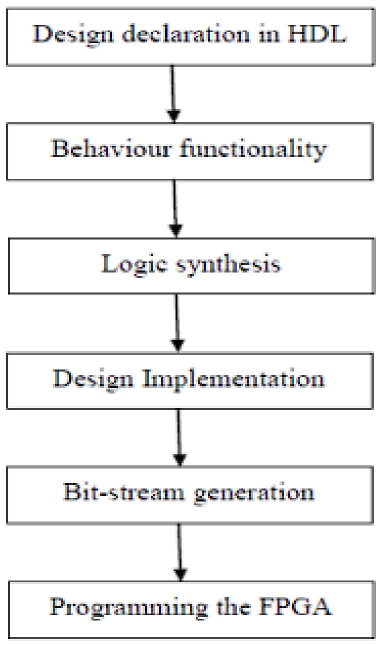


Figure 2.20 FPGA design flow

1) **System design:** This process determines all application specifications as well as a deep understanding of the application's function.
2) **Modeling:** building a python code to model the accelerators' function and use it in verification.

3) **RTL:** The HDL code for the model is written, and then Behavioral Simulation is performed to ensure that the HDL accurately describes the function required and to express the functionality of the model as hardware.
4) **Synthesis:** creates a netlist from the provided HDL source files. It's split into three steps:
   a. **Syntax check**.
   b. **Design association to logic cells.**
   c. **Optimization:** involves reducing logic and eliminating unnecessary logic to make the design smaller and quicker.
   d. **Technology Mapping:** Connecting design to logic, predicting and adding time estimates, generating output reports, and generating a netlist file including all the design and constraints.

Then, static timing analysis is performed to ensure that the operating frequency fulfills the specifications.

5) **Implementation:** By mapping synthesized netlists to the target FPGA's structure and connecting design resources to the FPGA's internal and I/O logic, the physical design layout can be determined. It is divided into three sub-processes:
   a. **Translate:** Takes the pin assignment & time requirements (e.g., input clock period, maximum delay, etc.) provided by a User Constraints File and combines them with the netlists into one large netlist.
   b. **Map:** Creates a Native Circuit Description (NCD) by comparing the resources specified in the input netlist file to the available resources of the target FPGA and dividing the netlist circuit into sub-blocks to fit into the FPGA logic blocks.
   c. **Place & Route (PnR):** The NCD sub-blocks are physically placed into FPGA logic blocks, and signals are routed between logic blocks such that time requirements are fulfilled, resulting in a fully routed NCD file.
6) **Programming the FPGA:** Converts the final NCD file into an FPGA-compatible format, then programs the FPGA using the generated bitstream file.

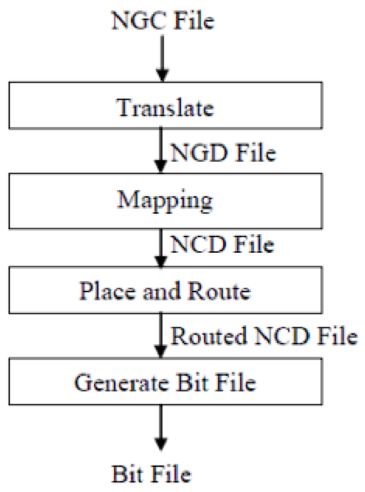


Figure 2.21 FPGA implementation steps

3 Chapter 3: ShuffleNet V2

3.1 Criteria for choosing ShuffleNet V2

Mainly, the robust and efficient model among CNN models was determined depending on the following parameters: Top-1 and Top-5 accuracy, Number of parameters, and Number of MACs (Multiply and Accumulate). Due to limited resources on FPGA, we put criteria to choose among these architectures which are Minimum Top-1 accuracy equals 65% and a maximum number of parameters equals 2.5 million. We found three architectures that achieve these criteria which are SqueezeNext-23, MobileNet V2 0.5x, and ShuffleNet V2 1x. According to Table 3.1, ShuffleNet V2 1x is the least in terms of the number of MACs in comparison to SqueezeNext and MobileNet V2 so the computational complexity of ShuffleNet is small also Top1 accuracy is better than SqueezeNext and MobileNet V2 [7].

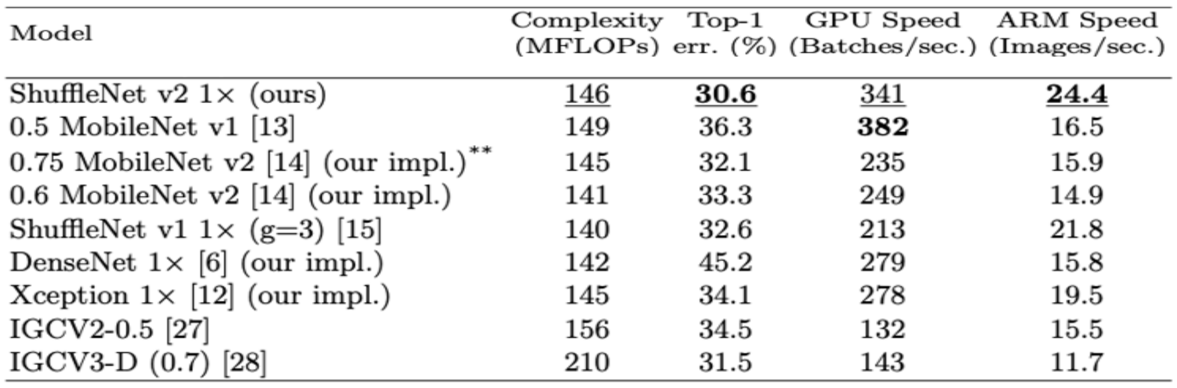
Also, by comparing ShuffleNet V2 1x with other architectures in terms of complexity, error rate, GPU speed, and ARM speed [9]. We found that despite its lack of GPU speed, ShuffleNet v2 has the lowest top-1 error rate, which balances the other drawbacks as shown in Table 3.2.

Finally, we chose the ShuffleNet V2 1x model which gives the best accuracy in very limited computational budgets on FPGA, focusing on common mobile platforms such as drones, robots, and phones.

Table 3.1 ShuffleNet V2 Vs SqueezeNext and MobileNet V2

	# MACs	# Parameters	Top-5 accuracy	Top-1 accuracy
SqueezeNext-23	749 M	2.4 M	88.2%	67.2%
MobileNet V2 0.5x	97 M	1.95 M	86.4%	65.4%
ShuffleNet V2 1x	73 M	2.3 M	88.9%	69.4%

Table 3.2 ShuffleNet Vs other Convnets



Model	Complexity (MFLOPs)	Top-1 err. (%)	GPU Speed (Batches/sec.)	ARM Speed (Images/sec.)
ShuffleNet v2 1× (ours)	146	**30.6**	341	**24.4**
0.5 MobileNet v1 [13]	149	36.3	**382**	16.5
0.75 MobileNet v2 [14] (our impl.)**	145	32.1	235	15.9
0.6 MobileNet v2 [14] (our impl.)	141	33.3	249	14.9
ShuffleNet v1 1× (g=3) [15]	140	32.6	213	21.8
DenseNet 1× [6] (our impl.)	142	45.2	279	15.8
Xception 1× [12] (our impl.)	145	34.1	278	19.5
IGCV2-0.5 [27]	156	34.5	132	15.5
IGCV3-D (0.7) [28]	210	31.5	143	11.7

3.2 The Network Architecture

The complete design as shown in Figure 3.1 and Table 3.3, begins with a standard 3x3 convolution followed by a max-pooling, both are stride 2. There are three stages after that, each with four or eight ShuffleNet blocks. The first shuffle block in each stage is stride 2, and the others are stride1. Finally, there are 1x1 convolution, global average pooling, and fully-connected layers. Table 3.3 shows the detailed description of all ShuffleNet CNN Layers and their Parameters. we can notice that parameters within a stage stay the same, and for the next stage, the output channels are doubled [8].

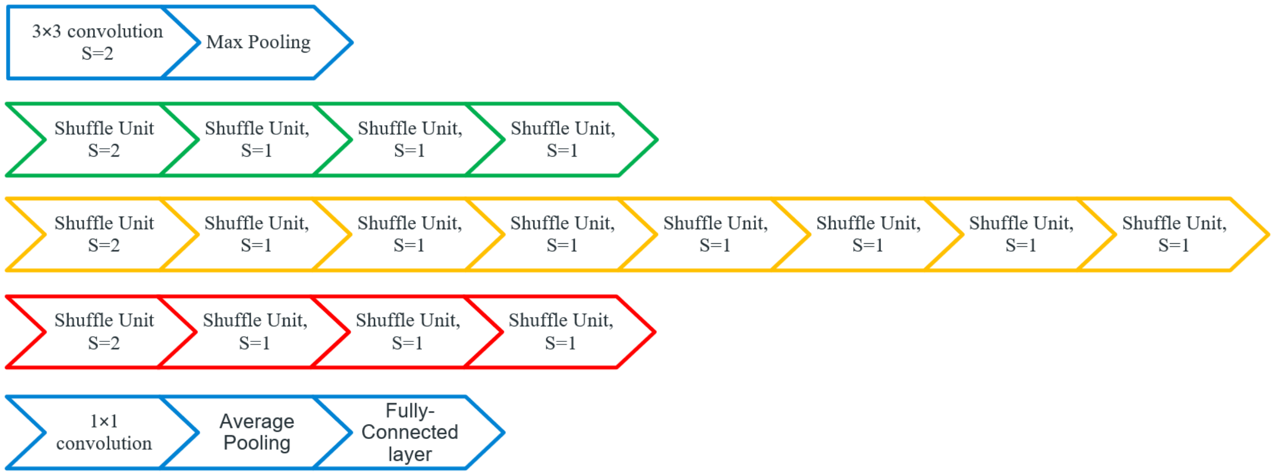
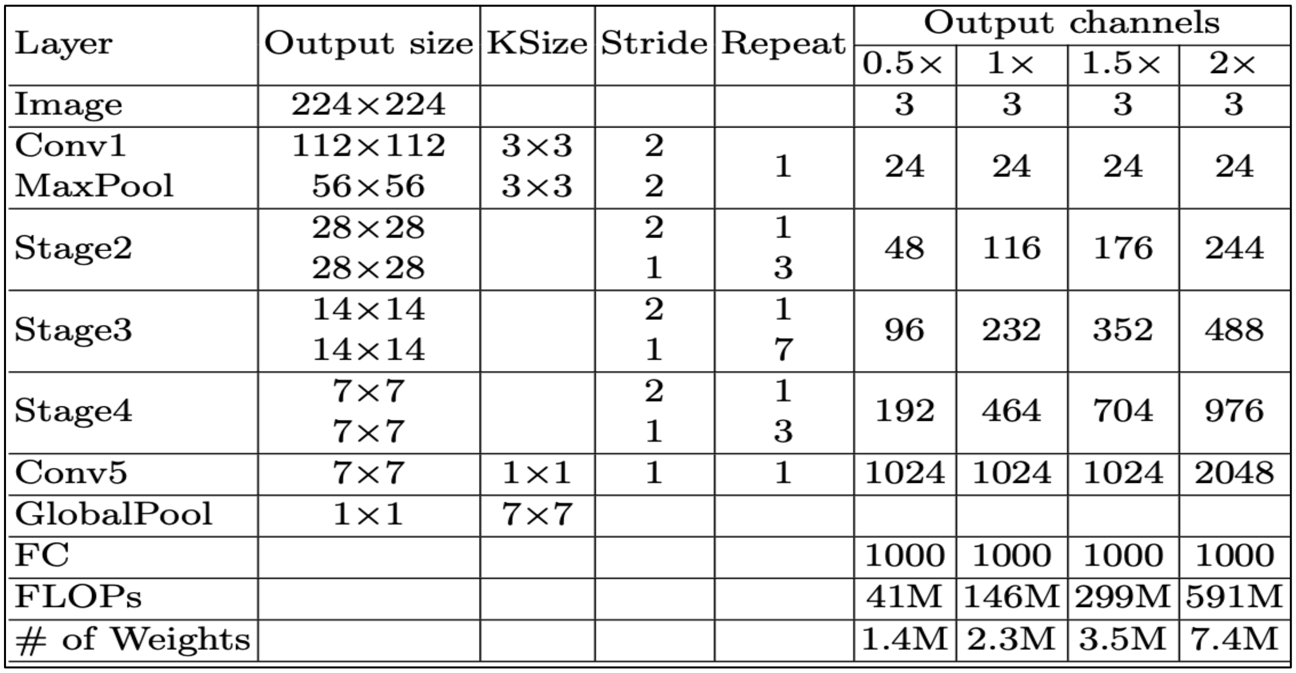


Figure 3.1 ShuffleNet architecture

Table 3.3 Detailed Description of all ShuffleNet CNN Layers and their Parameters



Layer	Output size	KSize	Stride	Repeat	Output channels			
					0.5×	1×	1.5×	2×
Image	224×224				3	3	3	3
Conv1 MaxPool	112×112 56×56	3×3 3×3	2 2	1	24	24	24	24
Stage2	28×28 28×28		2 1	1 3	48	116	176	244
Stage3	14×14 14×14		2 1	1 7	96	232	352	488
Stage4	7×7 7×7		2 1	1 3	192	464	704	976
Conv5	7×7	1×1	1	1	1024	1024	1024	2048
GlobalPool	1×1	7×7						
FC					1000	1000	1000	1000
FLOPs					41M	146M	299M	591M
# of Weights					1.4M	2.3M	3.5M	7.4M

3.2.1 3by3 Convolution and Max-pooling

- **3x3 Convolution**

Convolutions in Neural Networks apply filters to extract features from actual data. A filter could be related to anything, for pictures of humans, one filter could be associated with seeing noses, another with eyes, and so on. Each feature extracted from input data will be in the form of activation maps.

Convolution, like a typical neural network, is a linear operation that includes multiplying a set of weights with the input. The multiplication is done between an array of input data and a three-dimensional array of weights, called a filter or a kernel. The output from multiplying the filter with the input array one time is a single value. As the filter is applied multiple times to the input array, the result is a two-dimensional array of output values that represent the filtering of the input.

In our model, we perform the convolution operation on the input image matrix of size 224x224x3 with 24 filters each of size 3x3x3 that produce an output of size 112x112x24 with several channels equal to the number of filters applied [10].

Looking at Figure 3.2, the dot product of the filter and the first 27 elements of the image matrix produce one single pixel of the output matrix. Then, from left to right, top to bottom, slide the filter by one square across the image and repeat the computation. Finally, generate a two-dimensional feature map. The number of output channels will be equal to the number of filters. Then apply the ReLU activation function to all of the values in the input volume which takes the max between the value and zero.

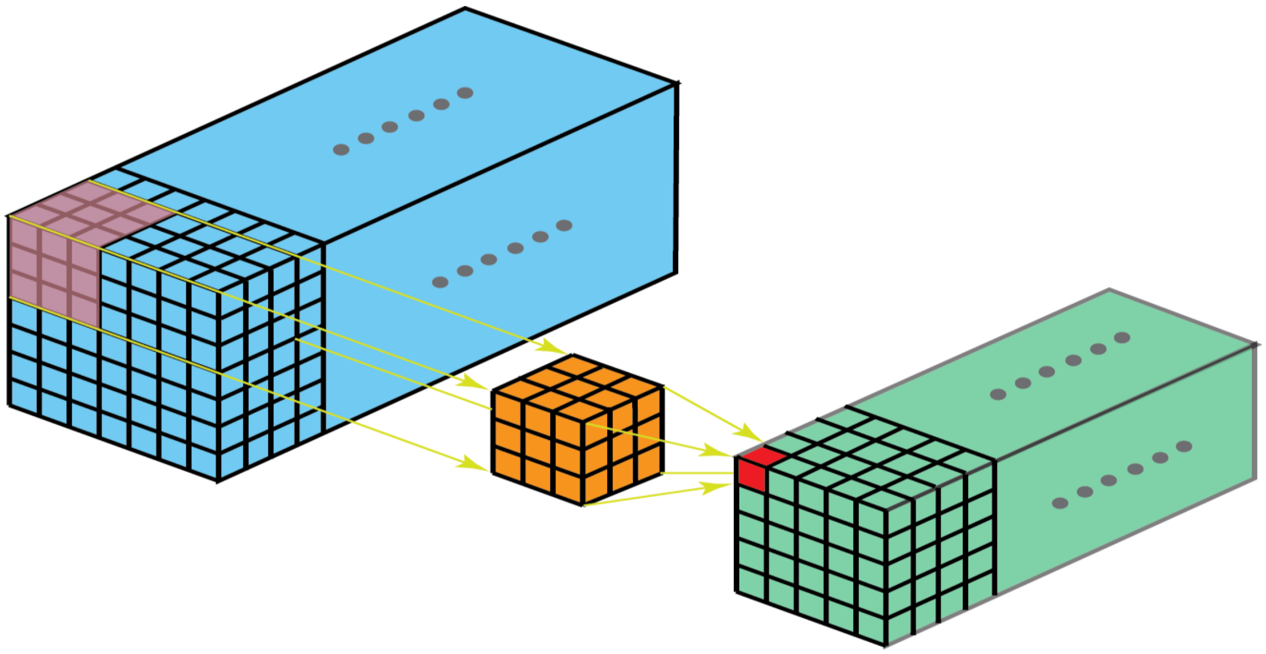


Figure 3.2 3x3 convolution operation in an input image

- **Max-pooling**

It is a pooling operation that determines the maximum value for patches of a feature map and uses it to build a down-sampled (pooled) feature map. It's commonly used after a convolutional layer. The pooling layer works on each feature map separately to build a new set of pooled feature maps with the same number of channels.

In our model, the input matrix to max-pooling is of size 112x112x24 using a 3x3 window with stride 2 that produces an output matrix of size 56x56x24. Figure 3.3 shows an example of a Max Pooling operation on a small image.

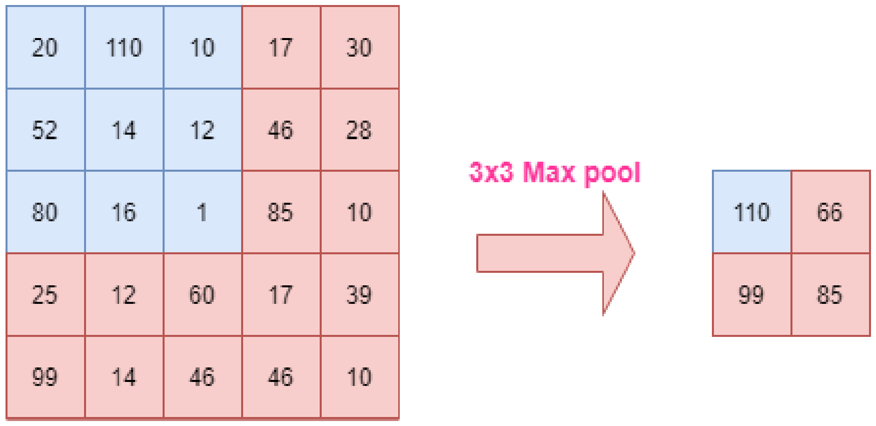


Figure 3.3 Max-pooling operation with window 3x3

3.2.2 Shuffle Units

The Shuffle unit is the building block of ShuffleNet v2. There are three stages in the model, each with 4 or 8 shuffle units. The Shuffle unit can be one of the following two types: stride 2 - shuffle unit and stride 1 - shuffle unit. The first unit in each stage is stride 2, and the others are stride 1. The first unit with a stride of 2 is used to reduce the dimensions. Figure 3.4 shows the block diagram of the two types of shuffle units.

In the shuffle unit with stride 2, the left branch is a little different from the shuffle unit with stride 1. It performs a 3×3 depth-wise convolution with stride 2, followed by a 1×1 convolution to make sure the outputs of both branches have the same spatial dimensions (or they can be concatenated at the end). Also, there is no channel split operation in such a block, so both branches work on the same data and concatenate effectively double the number of channels.

DW convolution followed by 1x1 convolution can be designed to have the same inputs and output dimensions as the normal convolutional operation, but it can be done at a much lower computational cost.

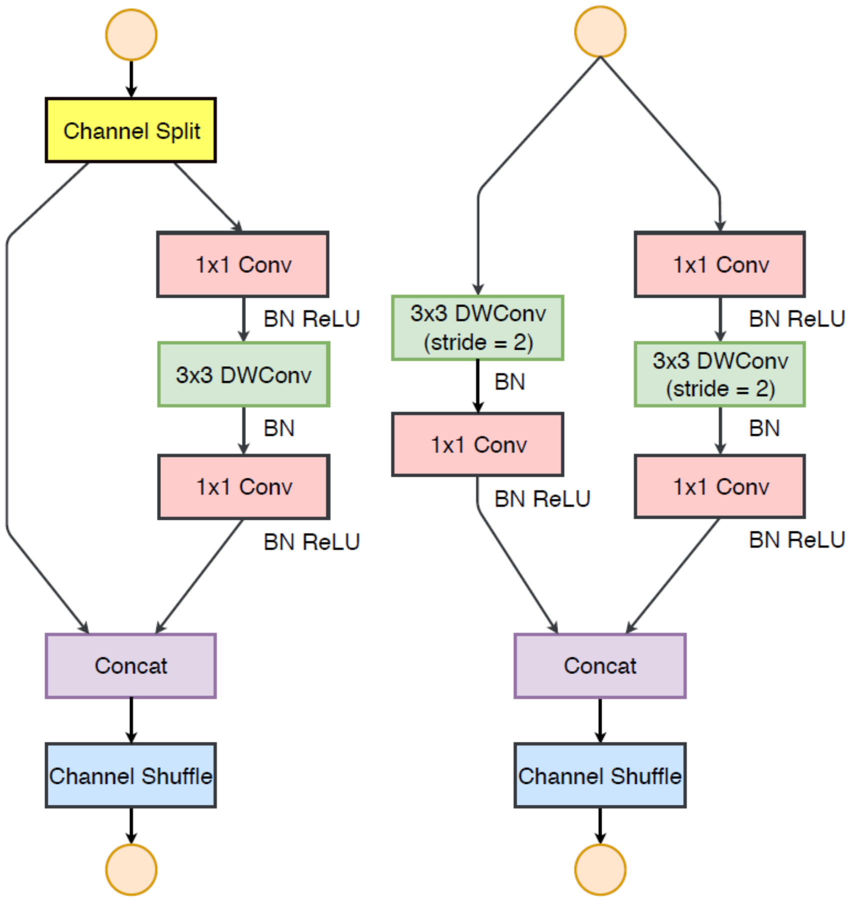


Figure 3.4 ShuffleNet unit with stride 1, ShuffleNet unit with stride 2

Explanation of each block in the Shuffle unit:

- **Channel split and channel shuffle**

A channel split operation sends half of the channels through the left branch and the other half through the right branch, thus splitting the channels into two groups. One of which is kept as the identity. The other branch has an equal number of input and output channels along the three convolutions. Because each convolution filter now works on half the input channels and creates half the output channels, you only need half the parameters.

As a result, each output is only derived from a fraction of the inputs, dividing the output channels into the same number of groups. This is inefficient since there is no information transfer between groups, limiting the network's ability to learn new things. So, channel shuffle is used to shuffle the input channels.

The ShuffleNet V2 model utilizes these two new operations, channel split and channel shuffle, to greatly reduce computation costs while maintaining accuracy.

The feature map is shuffled along the channels dimension using the channel shuffle procedure, as follows as shown in Figure 3.5.

1. Splitting the input channels into 2 halves.
2. Perform Convolution on one half.
3. Shuffle the channels to eliminate the side-effect of splitting.

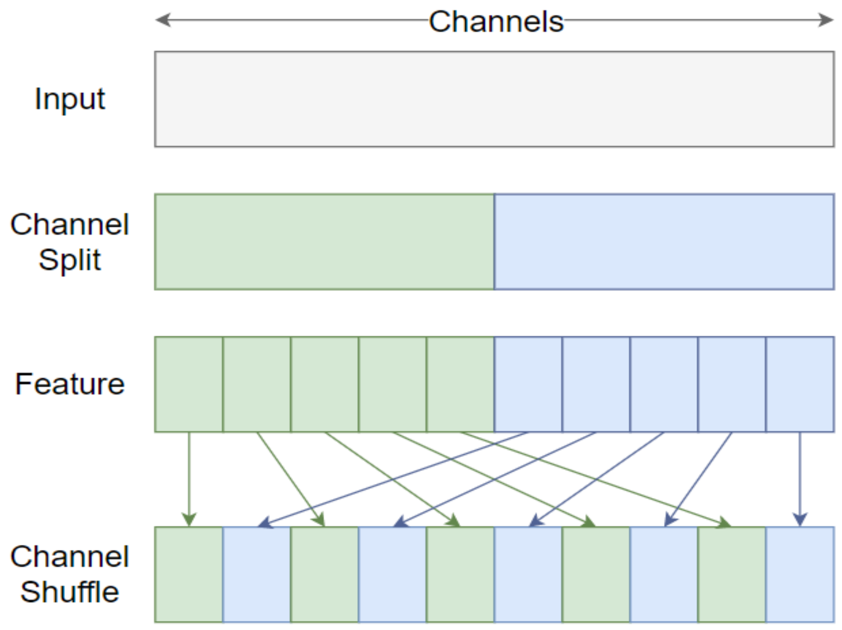


Figure 3.5 Channel shuffle operation for output channels

- **DW convolution**

Depth wise Convolution is a sort of convolution in which each input channel receives a single convolutional filter. The filter is as deep as the input in normal 2D convolution performed over multiple input channels to mix channels to generate each element in the output. Depth-wise convolutions, on the other hand, maintain each channel separately. This is done by applying each filter to the corresponding input channel as shown in Figure 3.6 [11].

To summarize the steps:

1. Divide the input into channels and apply the corresponding filter.
2. Each input channel is convolved with the appropriate filter.
3. The convolved outputs are stacked together.

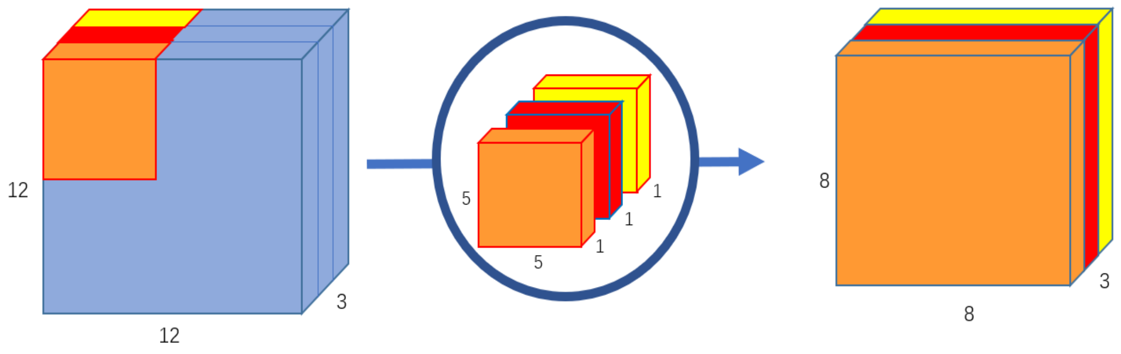


Figure 3.6 DW convolution operation

3.2.3 1x1 Convolution, Avg pooling, and Fully Connected (FC)

- **1x1 convolution**

A convolutional layer with a 1×1 filter can be used at any point in a convolutional neural network to control the number of feature maps and to mix up features. It's also known as a projection layer, a feature map, or a channel pooling layer. It maps an input pixel with all its channels to an output pixel and each filter will produce an output channel.

Here the input to this stage is the output of shuffle unit stages which is of size 7x7x464 with 1024 filters each of size 1x1x464 that produces an output of size 7x7x1024 with several channels equal to the filters used [12].

Figure 3.7 shows an example of a 1x1 convolution operation on an input matrix of size 64x64x192 using one filter of size 1x1x192 to produce an output of size 64x64x1.

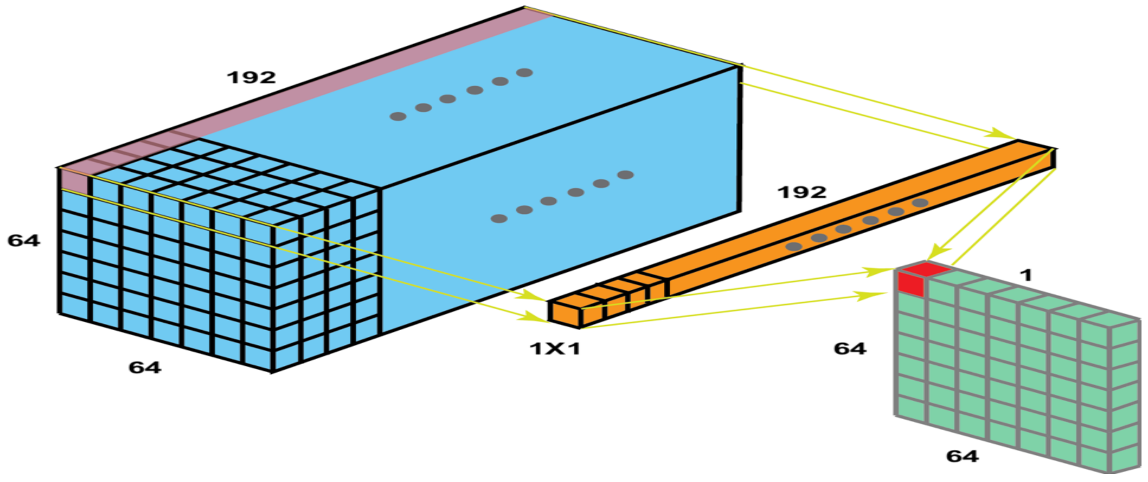


Figure 3.7 1x1 convolution using one filter of size 1x1x192

- **Avg-pooling**

It is similar to the max-pooling explained at the beginning of the model, but it calculates the average value for a sliding window on the input feature map and creates a down-sampled (pooled) feature map. It extracts features more smoothly than max-pooling, whereas max-pooling extracts more pronounced features like edges.

It takes the output of 1x1 convolution down-sampled its size to produce an output of size 1x1x1024. Figure 3.8 shows an example of Avg pooling on an input matrix using a 3x3 window.

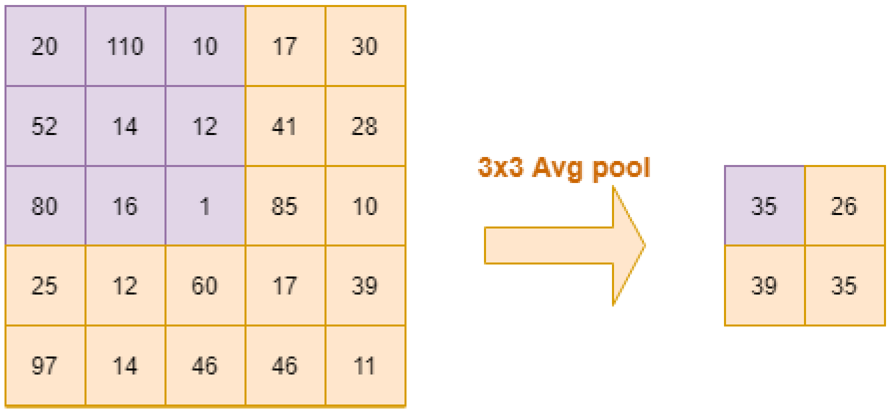


Figure 3.8 Avg pooling operation with a 3x3 window

- **FC layer**

The fully connected layer (FC) works with a flattened input, where all of the inputs from one layer are connected to every activation unit of the next layer as shown in Figure 3.9. FC layers are commonly found near the end of CNN architectures and can be utilized to optimize goals like class scores.

It is the last layer in the model taking an input of the previous stage which is of size 1x1x1024 and it will generate the final output to classify data into ImageNet 1000 class.

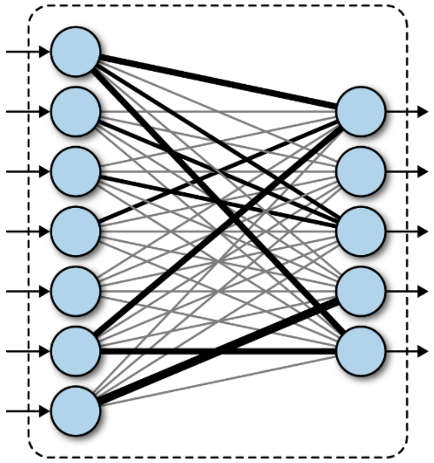


Figure 3.9 FC layer classifying data into various classes

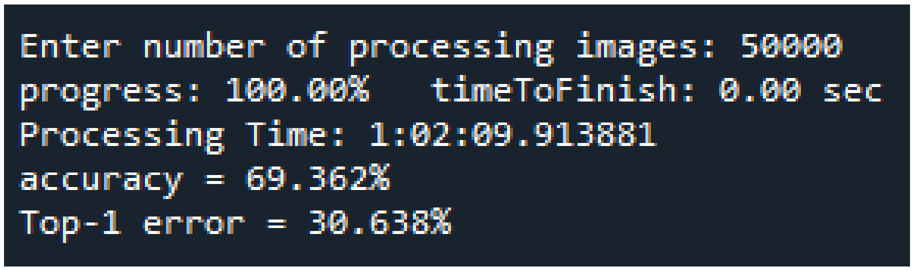
4 Chapter 4: Software Modeling

4.1 Pretrained Model

Our software modeling phase depends on a pertained model for ShuffleNet V2 which is available on PyTorch Hub for researchers [13]. The software model is developed by the PyTorch team using Python. It is open-source and free to use. This model is pre-trained on ImageNet and ready to use. The pre-trained model is based on the original paper that proposes the architecture of the ShuffleNet V2 [14]. We can run this model on Google Collab or we can get it for GitHub and use it locally.

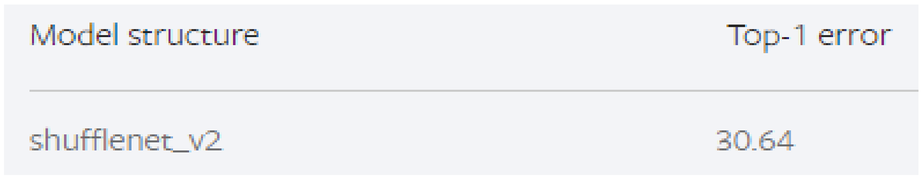
4.2 Model Validation

The first step we need to take is to validate the pre-trained model and get the accuracy on the ImageNet validation set to be sure that there is no something wrong with our software model. So, we edited the script which runs the ShuffleNet V2 software model. Then, we ran the model on the 50000 photos of the ImageNet validation set and got the accuracy as shown in Figure 4.1. Our accuracy is the same as mentioned in PyTorch Hub for researchers [13] as shown in Figure 4.2.



```
Enter number of processing images: 50000
progress: 100.00%   timeToFinish: 0.00 sec
Processing Time: 1:02:09.913881
accuracy = 69.362%
Top-1 error = 30.638%
```

Figure 4.1 Our software model accuracy



Model structure	Top-1 error
shufflenet_v2	30.64

Figure 4.2 Software model accuracy from PyTorch Hub

4.3 Fixed Point Representation

4.3.1 Fixed Point vs Floating-Point Representation

One of the most important aspects of the design is how to represent the data. The most common, important, and useful are Integer, Floating point, and Fixed-point representations. We will focus on Floating point and Fixed-point representations as they what will be better for our design. In Table 4.1, a full comparison between Floating point and Fixed-point representations is done. The comparison spans many points of view to choose the best representation for our hardware design.

Table 4.1 Fixed point vs Floating-point representation

	Fixed point representation	Floating-point representation
Hardware Complexity	Low	High
Hardware Area	Small	Large
Delay	Low	High
Power	Low	High
Memory Resources	Small	Large
Accuracy	Lower	High

Our PyTorch software model uses a single-precision floating-point format which is 32 bits. For a software model, it will be a good choice to use a 32-bit floating-point representation to get high accuracy. Also, software models can run on powerful hardware like GPUs which will provide the complex computation resources needed to make operations on floating-point data and memory recourses to store this amount of data which will be larger.

In our hardware design, we will use the fixed-point representation because it will be more suitable for the hardware design because of the following points:

- Accelerator targets high speed.
- Power is to be minimized.
- The area is to be minimized
- Hardware design will be simpler.
- The memory resources in the FPGA are limited.
- Accuracy will be acceptable and very close to the floating point.

So Fixed-point representation gives the best tradeoff between accuracy and hardware complexity and this is the biggest advantage of using the Fixed-point which decreases the area of hardware enabling us to place more hardware to make parallelism and pipelining which improves the throughput of the accelerator with a small and acceptable loss in the accuracy.

4.3.2 Static and Dynamic Fixed-Point Representation

Fixed-point representation is quite similar to the way used in decimal numbers to represent fractional numbers by using a point that is divided number into 2 parts. Let’s look at an example: $a = 01010111$ in binary (base 2) so we can interpret it as $a = 87$ in decimal (base 10) however, we can consider a binary point and interpret the final number as fractional $a = 0101.0111$ in binary (base 2) so we can interpret it as $a = 5.375$ in decimal (base 10).

$$a = 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 + 0 * 2^{-1} + 1 * 2^{-2} + 1 * 2^{-3} + 1 * 2^{-4} = 5.40625$$

here we use a 4 bit as integer and 4 bits as a fraction.

By generalizing this way, we can get the Fixed-point representation which consists of two parts, the first is the integer part and the second is the fractional part. We can add 1-bit to the most significant bit called a sign bit which determines whether it is a positive or negative number. As shown in Figure 4.3, there the total number of bits is called bit width which represents how many bits are needed to store a specific fixed-point representation.

Q format notation is used to easily represent the integer and fractional parts [15], if we use 3-bits for integer and 4-bit for fraction we can simply write this as “Q3.4”. The range of numbers that can be held with Fixed-point representation is $-2^{Bit\ Width} \rightarrow\ 2^{Bit\ Width}\ -\ 2^{Fractional\ Length}$ and $2^{Fractional\ Length}$ is called the step size or resolution. Resolution is the smallest fraction number that can be represented by a specific fixed-point representation.

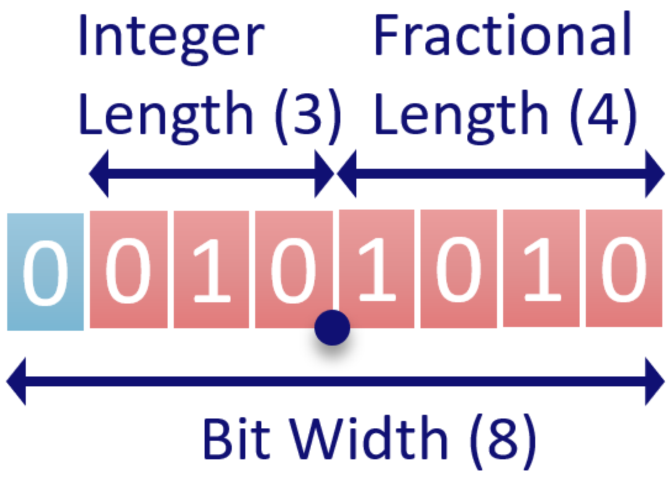


Figure 4.3 Fixed point representation illustration

The fixed-point representation will suffer from a saturation issue if the data to be represented is a positive number that is larger than the positive limit of the fixed point. The same issue will exist when the data is negative and is smaller than the negative limit of the fixed point. The solution for such an issue is to make the fixed-point representation dynamic, not static.

Static fixed-point representation is to use the fixed point with constant bit width, integer length and fractional length are constant along with the whole design which means that all numbers across all the design will have the same upper limit for positive values, the same bottom limit for negative value and the same resolution. The static fixed point will suffer from the saturation problem mentioned above.

Dynamic fixed-point representation is the solution for the saturation issue. Because CNN has a wide dynamic range of values as any layer output is an accumulation for all the previous layers the saturation issue will exist so a dynamic fixed point might be used to increase the accuracy. The dynamic fixed point is based on the idea that will make each layer has its fixed-point representation by varying the fraction length and also bit-width between layers. In dynamic fixed point, each number is represented as follows:

$$(-1)^s \cdot 2^{-FL} \sum_{i=0}^{B-2} 2^i \cdot x_i$$

Where B denotes the bit width, s is the sign bit, FL is the fractional length and x is the mantissa bits [16]. This allows a better coverage for a wide dynamic range of data when the data range is small, we can use small bit width and when the data range is large, we can use larger bit width. This is beneficial not only for accuracy but also for memory resources as to get high accuracy with a static fixed point we will use a large fixed point to be suitable for the biggest range of values but it is useless for smaller ranges so with the dynamic range we use only the suitable fixed point for each range of numbers which result in smaller data to be stored in fewer memory resources.

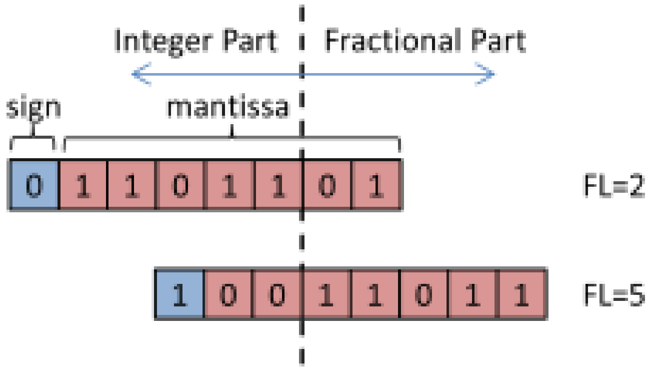


Figure 4.4 Dynamic fixed-point example

As shown in Figure 4.4, the dynamic fixed point is illustrated where two numbers have the same bit width which is 8-bits but because each number belongs to a different layer they have different fractional lengths.

4.3.3 Fixed-Point Addition

Addition in Fixed-point representation is like the addition of binary integer numbers. The first step to adding two Fixed-point numbers is to align the binary point of the two numbers we make a sign extension for the number which has a shorter integer number if needed. If the fractional part is shorter in one of the numbers, we put zero on the right side to align the two numbers. Let's look at an example as shown in Figure 4.5 [15].

$$\begin{array}{rr} 1\,1\,0\,.\,1\,1 & -1.25 \\ +\;0\,1\,1\,.\,0\,1\,0 & +3.25 \\ \hline \\ 1\,0\,1\,0\,.\,0\,0\,0 & +2 \end{array}$$

Figure 4.5 Fixed point addition

We should be careful when adding two numbers due to the overflow issue which might happen to result that the output is not correct because adding two N-bit numbers can lead to an (N+1)-bit as a result. In the previous example, there is no overflow so we should read the output in the same number of bits as inputs to take the output correctly. Let's look at another example in which an overflow issue exists as shown in Figure 4.6 [15].

$$\begin{array}{rrr} & 110.11 & -1.25 \\ + & 100.001 & -3.875 \\ \hline & 1010.111 & -5.125 \end{array}$$

Figure 4.6 Overflow in Fixed-point addition

In this example, the output must be taken in several bits larger than inputs by one bit as mentioned before to avoid errors in the result. One solution for the overflow issue is to make a sign extension before adding the two numbers and then take the output of the same size as inputs so that the result will always be correct not only when an overflow happens but also when there is no overflow. To make it clear, let's look at an example as shown in Figure 4.7 [15]. In this example, sign extension is done before adding the two numbers to make the range of numbers larger so that after the output result will fit in the size as inputs and it is clear for example that if we take the same number of bits as inputs at the output the result will be corrected so overflow issue is solved.

$$\begin{array}{rrr} & 1110.11 & -1.25 \\ + & 1100.001 & -3.875 \\ \hline & 11010.111 & -5.125 \end{array}$$

Figure 4.7 Overflow solution in Fixed-point addition

In many digital signal processors (DSPs), the register at the output of an accumulator has several bits more than the inputs by several bits which are called the guard bits [15]. Guard bits let us avoid the overflow issue as we accumulate some numbers and put the result at the same register so it should be large enough to hold the final result of accumulation.

4.3.4 Fixed-Point Multiplication

Multiplication in Fixed point is quite similar to binary integer numbers multiplication. The first step is to make the two numbers in the same size of bits with a sign extension on the left and zeros on the right if needed. Secondly, we will ignore the binary point of the two numbers which are multiplied, and treat the numbers as two's complements. The third step is to do the multiplication by generating the partial products and then adding them together as explained before in the Fixed-point addition to obtain the final result. Finally, we will determine the position of the binary point that we ignored before. To illustrate the multiplication methodology, let's look at an example as shown in Figure 4.8 and Figure 4.9 [17].

$$a = (101001)_2 \times (2^{-3})_{10}$$

$$b = (100010)_2 \times (2^{-3})_{10}$$

$$a \times b = \left((101001)_2 \times (2^{-3})_{10}\right) \times \left((100010)_2 \times (2^{-3})_{10}\right)$$

$$a \times b = \left((101001)_2 \times (100010)_2\right) \times (2^{-6})_{10}$$

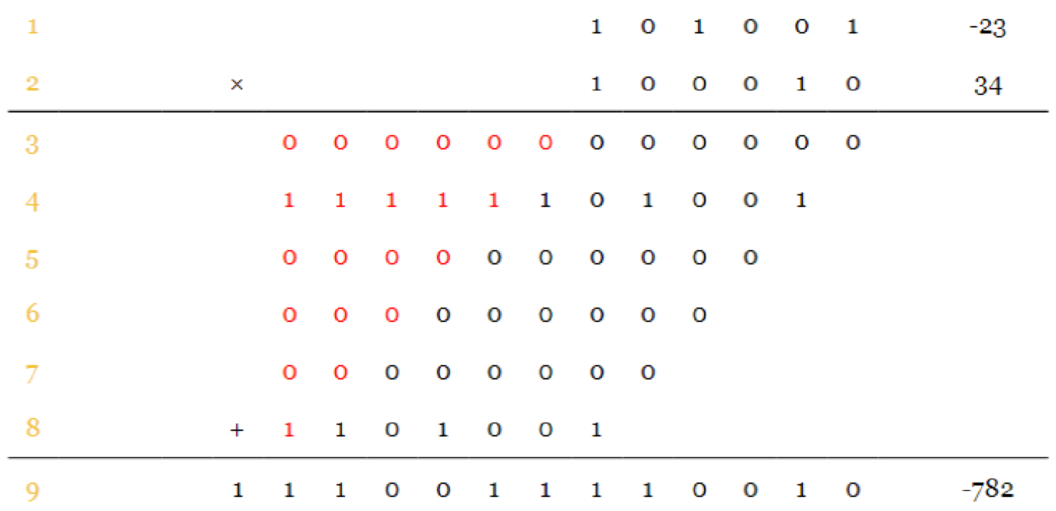
Figure 4.8 Preparing numbers for multiplication

1							1	0	1	0	0	1	41
2	×						1	0	0	0	1	0	34
3							0	0	0	0	0	0	
4						1	0	1	0	0	1		
5					0	0	0	0	0	0			
6				0	0	0	0	0	0				
7			0	0	0	0	0	0					
8	+	1	0	1	0	0	1						
9		1	0	1	0	1	1	1	0	0	1	0	1394

Figure 4.9 Unsigned multiplication fixed point

In this example, two unsigned numbers are multiplied. We can generalize the rule by which the binary point position of the output is determined by saying it will be in the position which is the sum of the two binary point positions of the input.

In multiplication, we also should care about the overflow issue as the output of the multiplication is much larger than the input. If the inputs have sizes that are Na-bits and Nb-bits the output will have a size of (Na+Nb)-bits to avoid the overflow and get the result correctly.

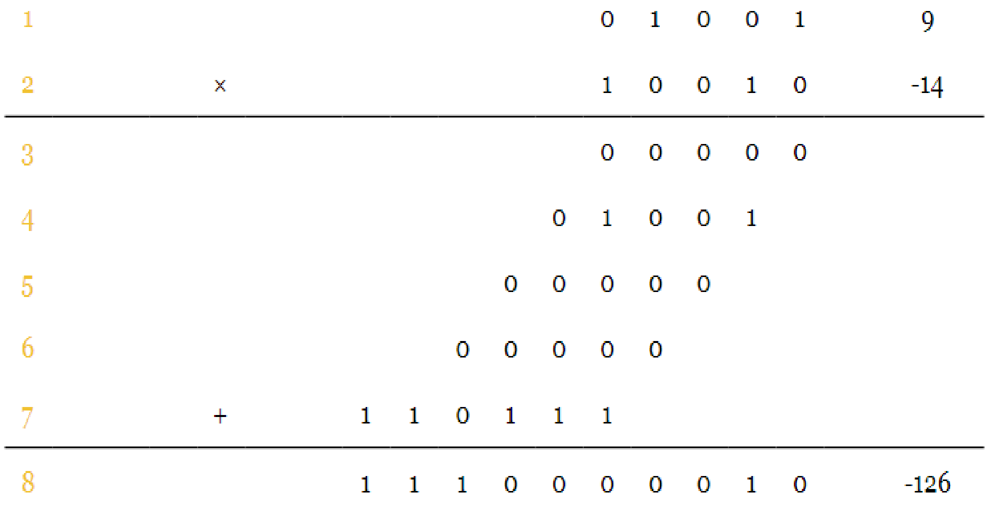


1								1	0	1	0	0	1	-23
2	×							1	0	0	0	1	0	34
3		0	0	0	0	0	0	0	0	0	0	0	0	
4		1	1	1	1	1	1	0	1	0	0	1		
5		0	0	0	0	0	0	0	0	0	0			
6		0	0	0	0	0	0	0	0	0				
7		0	0	0	0	0	0	0	0					
8	+	1	1	0	1	0	0	1						
9	1	1	1	0	0	1	1	1	1	0	0	1	0	-782

Figure 4.10 Signed by unsigned multiplication

In the case of multiplying a signed number by an unsigned number, the steps are the same as multiplying unsigned by unsigned and it is illustrated in an example that multiplies signed by unsigned number as shown in Figure 4.10 [17]. Only one difference exists which is to sign extend the partial products as one of the operands is a signed number. In this example, the decimal point is ignored while doing multiplication and will be determined in the output result as stated before.

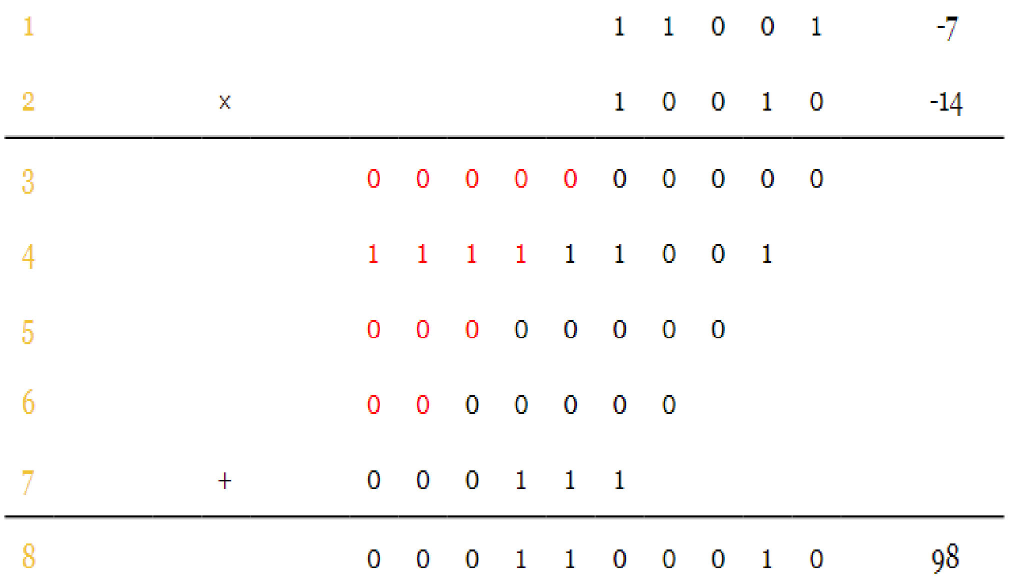
There is a little bit of difference while multiplying by a signed number. The difference is that the last partial product will be represented in a longer size by one bit which will be determined by sign extension then take the two's complement. This is because the last partial product is resulted from multiplying with the sign bit which gives it a negative sign. After that, we can add this partial product to the rest of the partial products. Let's look at an example to illustrate unsigned by signed multiplication as shown in Figure 4.11 [17].



1							0	1	0	0	1	9
2	×						1	0	0	1	0	-14
3							0	0	0	0	0	
4						0	1	0	0	1		
5					0	0	0	0	0			
6				0	0	0	0	0				
7	+	1	1	0	1	1	1					
8		1	1	1	0	0	0	0	0	1	0	-126

Figure 4.11 Unsigned by signed multiplication

In this example, the last partial product is the two's complement of the multiplicand. In the case of multiplying a signed number by a signed number same procedure as multiplying unsigned by signed will be happened but with a sign extension for the partial products, the final partial product will be treated in the same way. This case is illustrated as shown in Figure 4.12 [17].



```
1                               1   1   0   0   1       -7
2      x                        1   0   0   1   0      -14
-----------------------------------------------------------
3               0   0   0   0   0   0   0   0   0   0
4               1   1   1   1   1   1   0   0   1
5               0   0   0   0   0   0   0   0
6               0   0   0   0   0   0   0
7      +        0   0   0   1   1   1
-----------------------------------------------------------
8               0   0   0   1   1   0   0   0   1   0       98
```

Figure 4.12 Signed by signed multiplication

4.4 Quantization

As mentioned before our software model for the ShuffleNet V2 uses a single-precision floating-point format which is 32 bits. So, we should quantize the software model to use a fixed-point representation and run it on the ImageNet validation set to choose the best (smallest) Fixed point bit width and fractional length to minimize the size of data (in bits) which will be stored in the FPGA and to minimize the loss in the accuracy. Quantization is done through two steps which are model quantization and weights quantization.

4.4.1 Model Quantization

Model quantization is the step in which we quantize the model itself by editing the python code that models the ShuffleNet V2. This is achieved by using a python library called Fxpmath which is a library for fractional fixed-point (base 2) arithmetic and binary manipulation [18]. This library can take an input of a NumPy array [19]. In our PyTorch software model data is stored in a data holder called tensors [20]. So, we convert these tensors which hold the data into NumPy arrays so that we can use the functions in Fxpmath library to quantize these data into Fixed-point representation. Then we bring NumPy arrays again to PyTorch tensors so the software model can process the data in the next stages. Fxpmath library gives us the freedom to choose the bit width and fractional length of the Fixed-point so we can get the optimum Fixed-point representation by try and error.

4.4.2 Weights Quantization

Weights quantization is the step in which we quantize the weights of the ShuffleNet V2 to Fixed-point representation. The reason we quantize the model and the weights in different steps is that we use a pre-trained model. So, in model quantization, we actually quantize only the forward path of the model without backward propagation because we don’t use it and we don’t even need it. As a result of using a pre-trained model, weights of the model are stored in a file so that the model run can read the weights from this file. This file is in the “pth” format. So, we developed a script in which we first read the content of the weights file. Then we quantize the weights in the same way we quantize the model using the Fxpmath library. The last step is to write the quantized weights in a “pth” file then replace it with the original weights file so that the model reads the quantized weights when it is run instead of the original weights.

4.4.3 Quantization Results

After quantizing the model and its weights, we had to verify that the Fixed-point representation still gives us a small loss in accuracy. Also, we need to get the optimum bit width and fractional length for a Fixed-point representation to use in the hardware. We will go with the try-and-error method. To get a starting point, we looked at the data word which BRAM of the FPGA can hold. We found that BRAM can have words with widths of 1, 2, 4, 9, 18, 36, and 72 bits [21]. So, we chose to start with 16 bits as a bit width for the Fixed-point representation then we will decrease the bit width and fractional length till getting the optimum size for bit width and fractional lengths.

Firstly, we ran the model only on 100 photos of the ImageNet validation set to know where the breaking point is. The breaking point is the bit-width with fractional length in which the accuracy has a big loss so we shouldn’t use these widths or any smaller bit width or fractional length. The results of this trial are shown in Table 4.2.

Table 4.2 Quantization results on 100 Photos

Floating Point 32-bit Accuracy		74.00%	
Bit Width	**Integer length**	**Fractional Length**	**Accuracy**
16	6	9	72.00%
16	7	8	71.00%
14	6	7	46.00%
16	8	7	71.00%
15	6	8	70.00%
14	5	8	64.00%

From Table 4.2, we can notice that 16-bit and 15-bit Fixed-point representations have acceptable accuracy concerning the original accuracy of the Floating-point 32-bits. Another important conclusion from this result is that in 14-bit fixed-point representation when the fractional length is 7 bits and when integer length is 5 bits, the accuracy decreased significantly making it clear that the minimum acceptable size for fractional length and integer length is 8 bits and 6 bits respectively. So, we can guess that a 15-bit Fixed-point representation with an 8-bit fractional length will be the most suitable and smallest representation for our design but we will try on a bigger number of photos to be sure and to have the final accuracy of the quantized model and weights.

In the second try, we will run the best-fixed point representation from the first try on 1000 photos from the ImageNet validation set and the results are shown in Table 4.3.

Table 4.3 Quantization results on 1000 Photos

Floating Point 32-bit Accuracy		70.10%	
Bit Width	**Integer length**	**Fractional Length**	**Accuracy**
16	6	9	70.30%
16	7	8	70.50%
15	6	8	70.30%
14	5	8	61.70%
14	6	7	48.90%

From Table 4.3, we can get the same conclusion as before which is 16-bit and 15-bit Fixed point is the most suitable for us and 15-bit is the smallest size we can get. We try again 14-bit to make sure that it is the breaking point. The last try will run on the whole 50000 photos of the ImageNet validation set on the quantized model and weights to decide finally which representation to go with and the results are shown in Table 4.4.

Table 4.4 Quantization results on 50000 photos

Floating Point 32-bit Accuracy		69.362%	
Bit Width	**Integer length**	**Fractional Length**	**Accuracy**
16	7	8	68.340%
16	6	9	69.070%
16	5	10	60.496%
15	6	8	68.300%
14	5	8	60.922%
14	6	7	45.202%

Finally, from Table 4.4, as we go down to 14 bit fixed-point, the accuracy dropped about 8% which is not acceptable. So, the 14-bit is considered the breakpoint. We will go with a 16-bit or 15-bit fixed-point representation. If we need to decrease the data size to fit in the memory resources, then a 15-bit representation is better. If memory is sufficient, the 16-bit will give higher accuracy. In the two cases, the loss in accuracy doesn’t exceed 1% which is very acceptable and the difference between these two cases is small.

5 Chapter 5: RTL Design Methodology and Verification

5.1 Introduction

This chapter illustrates our hardware design approach for the ShuffleNet accelerator. The architecture we propose here is used for the RTL design flow. First, we must define our target to be achieved by the design approach we used, which will be evident in our choices in the architecture design. We aim to achieve high throughput (real-time) with reasonable power consumption. Also, we designed the architecture to work on real-time images, not only one. Now, we can talk about the details of our design approach. It is based on the following:

1. Controller – Datapath architecture.
2. Dividing the architecture into three stages (3 pipeline stages on frames).
3. Designing a computation core for each type of convolution, pooling…. etc.
4. Pipelining computation cores to reduce the critical path delay.
5. Parallelism in filters, kernel, and channels in the computation cores (using adder trees).
6. Using an enhanced type of adders in the adder trees instead of two input adders.
7. Using DSP cores of the FPGA to perform multiplication operations in computation cores.
8. Weights and biases are stored in internal memories (ROM) on the FPGA.
9. Using cache memories between some layers to store feature maps (single port, dual port, ping-pong).
10. Using minimum area algorithm of memory IPs in Vivado design suite to reduce the number of BRAMs.
11. Reusing of computation cores and memories.
12. Applying the effect of batch normalization without division units or any extra units by changing the weights of convolution layers before batch normalization.
13. Distributed controllers instead of one big controller.
14. Using fixed-point operations and quantization.

Through this chapter, we will explain each one of them in more detail and how they are applied in the complete architecture.

Our proposed architecture based on the above design approach divides the layers of the ShuffleNet CNN into 3 Groups. This allows the architecture to process three images simultaneously; we have chosen three stages for the frame pipeline to compromise between throughput and the number of memories needed to store weights, biases, and feature maps and enhance BRAMs utilization. Also, the number of operations in each group is close to the others to avoid overhead. As shown in Figure 5.1, Group 1 contains 3*3 convolution and max-pool layers, Group 2 contains all ShuffleNet building block layers in the CNN, and Group 3 contains 1*1 convolution, average-pool, FC layers, and a classification unit.

Also, our proposed architecture contains distributed controllers to perform hierarchal control for the accelerator; each stage of the three stages has its controllers and the master controller (the accelerator controller) at the top level to supervise and control all other controllers. Group 1 and Group 2 have sub-controllers for some blocks. This approach is better than using one centralized controller as it makes designing and debugging the control part simple, easy, and specialized. The only drawback of this approach is managing the interaction between these controllers, but in our architecture, most controllers interact with one controller (almost the accelerator controller).

Our proposed architecture can deal with multiple images, not only one as we designed a photo memory. It is a ping-pong memory to store the next image and allow the accelerator to read the current image simultaneously with the writing operation. Moreover, the architecture has three pipeline stages.

Finally, the design of the three groups (Controller-Datapath architecture), intermediate memories between groups, and the accelerator controller is illustrated throughout the other sections of this chapter. And at the last section, we show how our proposed architecture is functionally verified on images from the ImageNet dataset and compare its accuracy with that of the software model.

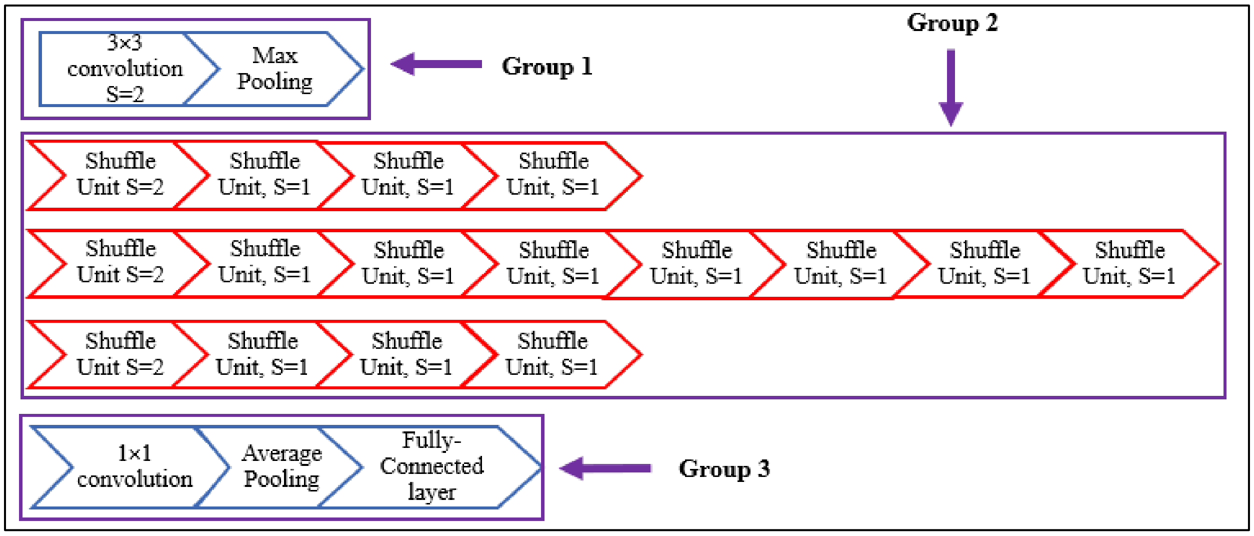
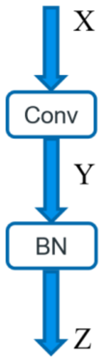


Figure 5.1 CNN layers are divided into 3 Groups

5.2 Performing Batch Normalization on Software

ShuffleNet CNN includes batch normalization operation after some convolution layers, especially after Group 3 convolution layers. The formula of the batch normalization is, as shown in Figure 5.2. Thus, it is clear that batch normalization needs fixed-point division, which is complex in hardware implementation. We tried to find an alternative approach for batch normalization. So, the optimization we explain in this section is not to help the design meet timing; it is a way of simply implementing the batch normalization without any division unit or extra hardware. Our proposed optimization for the batch normalization is as follows; we can apply the same effect of the batch normalization by updating the weights and biases of the convolution before it in the model with batch normalization parameters, as shown in Figure 5.3, and this optimization is done by software. Thus, this optimization will avoid implementing complex hardware for division or any extra hardware for batch normalization. Moreover, it will save power and reduce the number of calculations as updating weights is only done once in software.



$$Y = wX$$

$$Z = \frac{Y - mean}{\sqrt{Var + \epsilon}} \times \gamma + \beta$$

$$Z = \frac{wX - mean}{\sqrt{Var + \epsilon}} \times \gamma + \beta$$

Figure 5.2 Before BN optimization

$$Z = \frac{wX - mean}{\sqrt{Var + \epsilon}} \times \gamma + \beta$$

$$Z = \frac{w\gamma X}{\sqrt{Var + \epsilon}} - \frac{mean \times \gamma}{\sqrt{Var + \epsilon}} + \beta$$

$$new\ weight = \frac{w\gamma}{\sqrt{Var + \epsilon}}$$
$$new\ bias = -\frac{mean \times \gamma}{\sqrt{Var + \epsilon}} + \beta$$

Figure 5.3 New weights and bias after implementing batch normalization

5.3 Group 1 Design

As shown in the introduction group 1 take care of the 3x3 Convolution and max pooling in the start of the shuffle model. So Group1 contains of input memory (photo memory), 3x3convolution block, max pooling block, output memory (max pool memory) and the group 1 controller and this is the hierarchical of the design for group 1 as shown in Figure 5.4.

- Memories
  - Data memories
    - Photo memory
    - Max pooling memory
  - Filter memories
    - Weights memory
    - Bias memory
  - Computation Cores

    - ♦ 3x3 convolution & RELU Core
    - ♦ Max Pooling Core
- FIFOs
    - ♦ 3x3 convolution FIFOs
    - ♦ Max pooling FIFOs
- Controllers
    - ♦ FIFO controllers
    - ♦ Group 1 controller

And is this section we will discuss each component in detail.

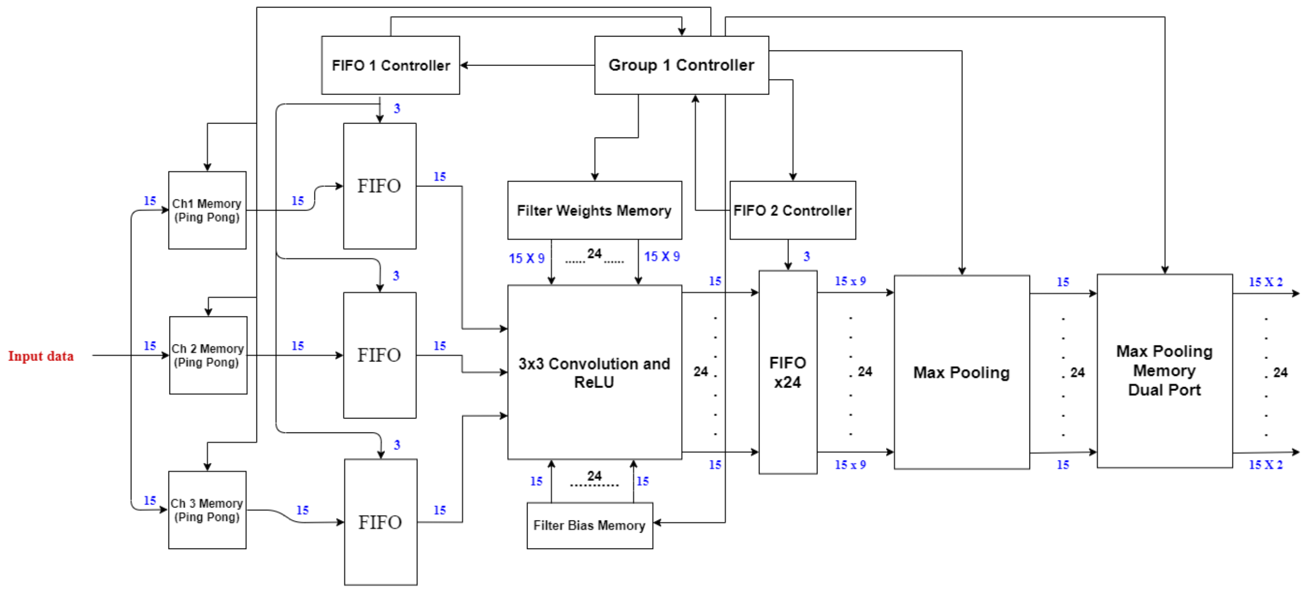


Figure 5.4 Group1 Block diagram

5.3.1 Memories

In the accelerator is two types of memory, Data memories and weights memories, the data memories are used to story the data and the input and output of each convolution block. The weights memories are ROMs used to store the filters weights for all the convolution blocks inside the model.

<u>Data Memories</u>

5.3.1.1 Photo Memory

The photo memory works as the input memory of the accelerator and the interface between the accelerator and the system that will contain the accelerator.

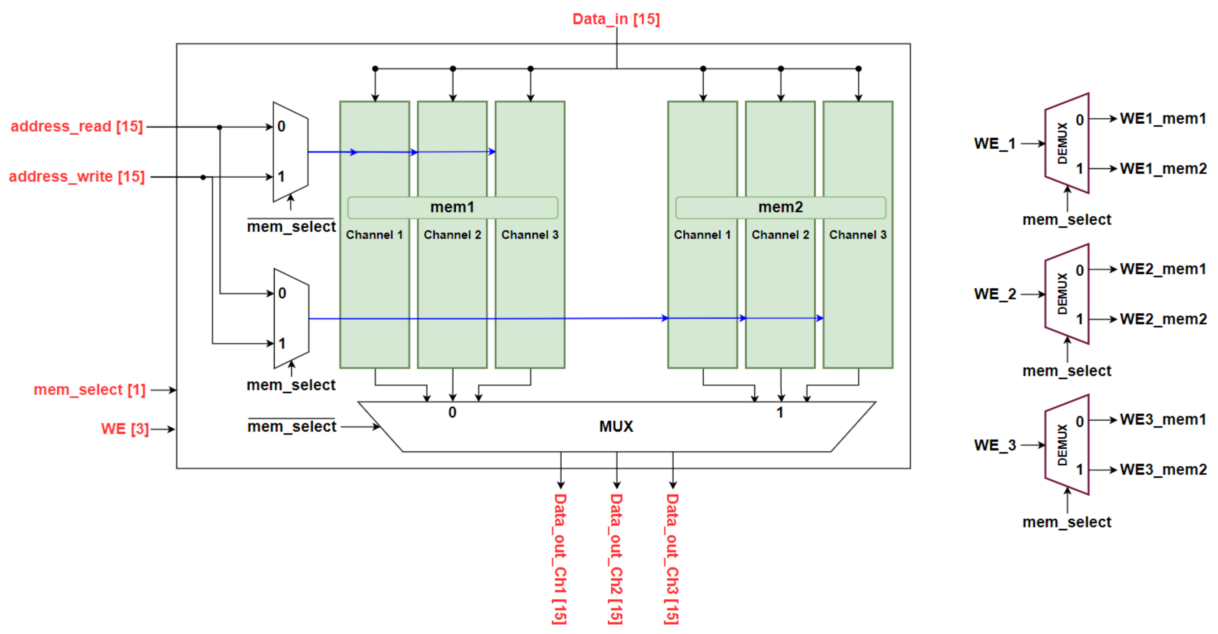


Figure 5.5 Photo memory block diagram

Since the input photo size expected by the model is 224*224*3 where 3 is number of channels as photos in RGB has 3 channels for the red, green and blue so we decided to partition the memory into 3 instance one for each input channel and to achieve high speed and the pipeline of the model we should manage to read new photo while group 1 processing the photo so we use a ping-pong structure for the photo memory as shown in the Figure 5.5.

5.3.1.2 Max Pooling Memory

This memory takes the output of the Group1 (Max Pooling Block) to be ready to be processed by Group2. We have full parallelism in filters in 3x3 core and max pooling core so the max pool is generating 24 outputs at the same time, one for each output channel, (number of the output channels from Group1 is 24) so we need to store them at the same time so we must partition the memory to 24 instance each one has one of the channels.

FILTER MEMORIES

5.3.1.3 Weights Memory

We need to store the filters of the 3x3 convolution into ROMs so we can read them to process the data, max pooling has no weights, so the organization of the memory dependent on the parallelism used in the computation core (how many weights must be read each clock cycle). and the number of weights per filter (to get better memory utilization possible).

As Group1 parallelism is 24 filter, 3 in the window and 3 parallel channels this mean we need 3*3*24 = 216 weight each clock cycle this mean we must partition the memory to 216 instances but if we look at the filter size is 3x3x3 this mean that each instance will have only 3 weights stored (very bad utilization) so we implement it my LUTs and its output goes to mux as shown in Figure 5.6.

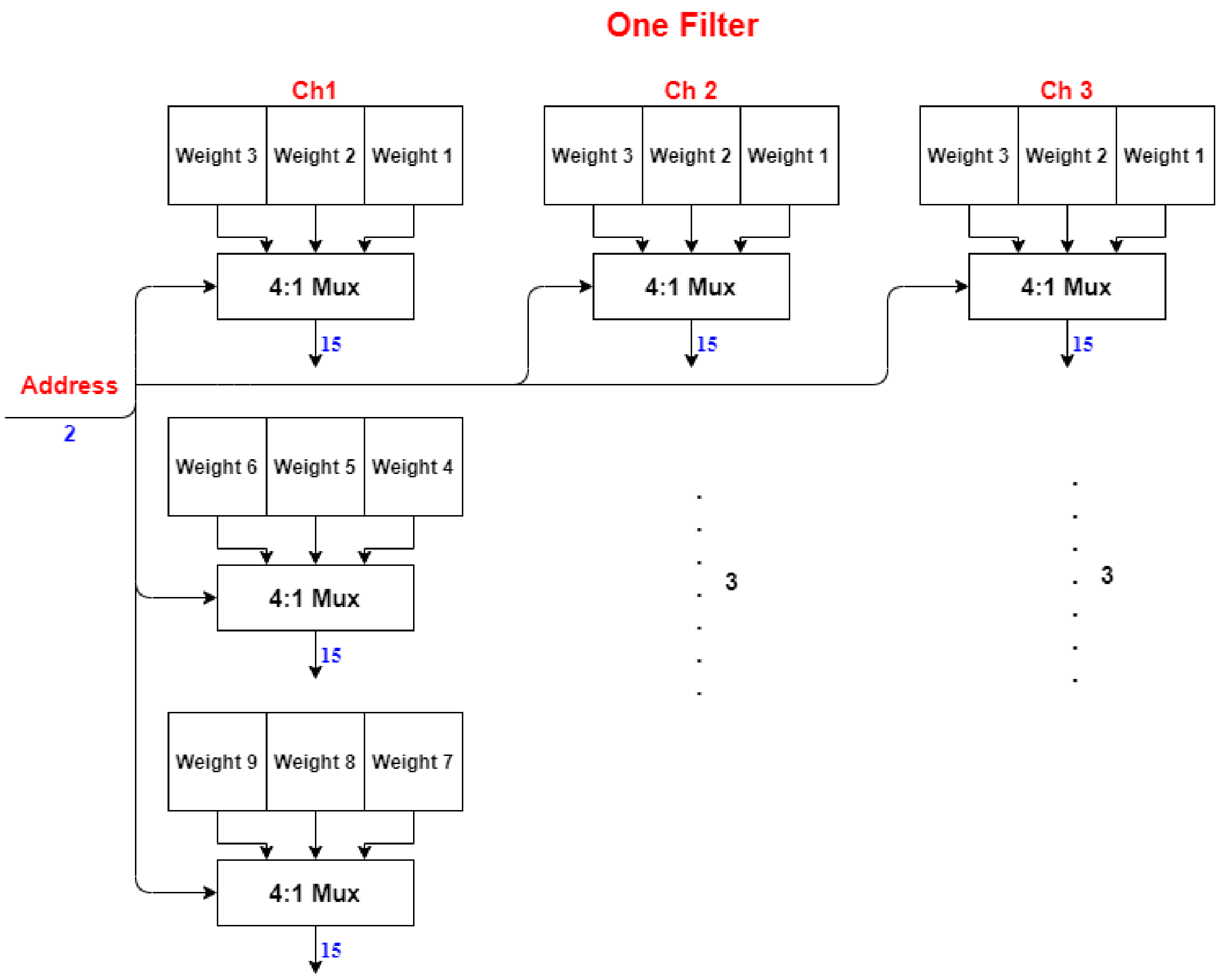


Figure 5.6 3x3 Filter memory implementation

5.3.1.4 Bias Memory

As known each filter has a bias weight this mean we have 24 bias that should be read in parallel, so we also implement this memory as LUTs and get the output to the 3x3 core as shown in Figure 5.7.

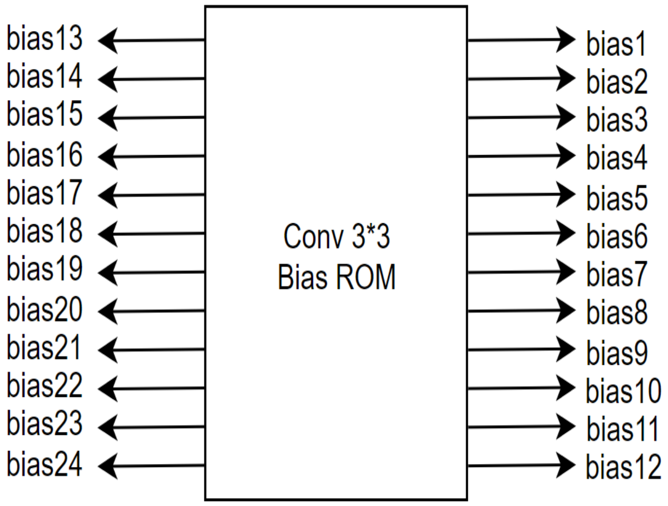


Figure 5.7 3x3 Conv Bias memory

5.3.2 Computation Cores

The 3x3 convolution and the max pooling are the two computing cores for this group. We will go through how quantization is used to have a 15-bit fixed point output from the core without overflow in each of them, as well as the parallelism used in filters, channels, and window, the pipelining registers used to break the long timing path.

5.3.2.1 3x3 Convolution Core

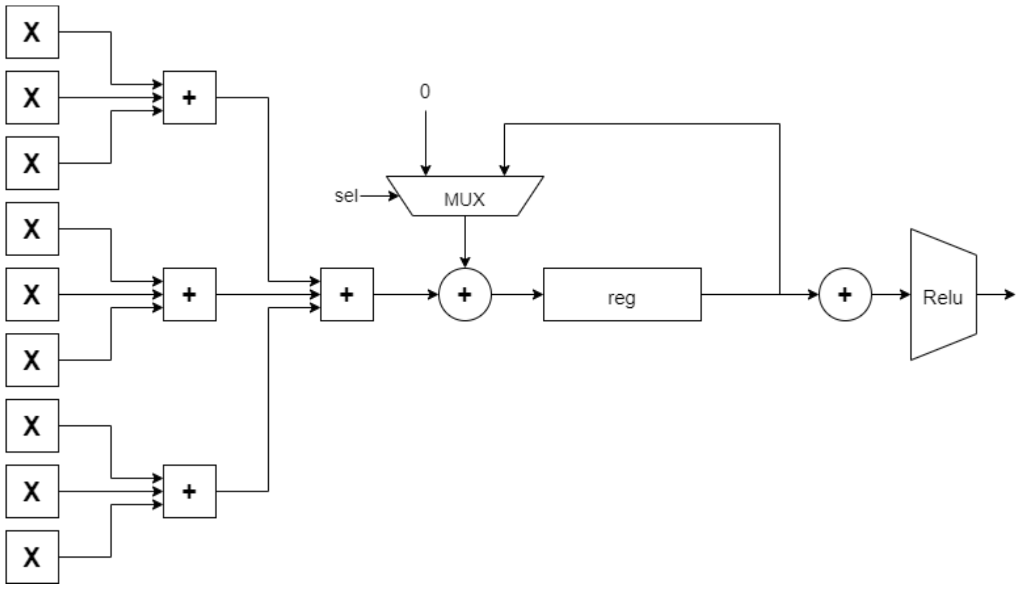


Figure 5.8 3x3 Conv Computation core

Parallelism

Each filter has three channels each one contains 9 parameters, and the number of filters is 24 filters, so to speed the operation we used full parallelism if filter this mean that the previous core is repeated 24 times. And us use 3 parallel in channel and 3 parallel in window as summarized in Table 5.1. Using this parallelism, we need to accumulate 3 times to get one pixel in the output (each output takes 3 clock cycle to be calculated) as shown in Figure 5.8.

Table 5.1 Parallelism in the 3x3 Convolution

Parallelism in Filters	Parallelism in Channels	Parallelism in window
24	3	3

Inputs

The inputs to the 3x3 convolution core are as follows:

1. Data: 9 parallel inputs coming from the 3 FIFOs (1 FIFO for each channel).
2. Weights: (24*) 9 parallel inputs from filter memory.
3. Bias: (24*) 1 input from bias memory.

Output

The output of the 3x3 convolution core is 24 output every 3 clock cycles. The output goes to 24 FIFOs the get the data ready to the Max Pooling core. We don't store the output in intermediate memory to use less memory and to allow both cores to work in the same time.

Methodology

The input 9-elements (3 from each input channels) from three FIFOs and the same data is processed in the 24 core to apply to them the all 24 filter in parallel. The first step in convolution operation is multiplying each input to the corresponding pixel in the filter by using DSPs, then adding the nine multiplayer outputs using an adder tree, adding each three of outputs using the Adder3 block. And takes the next 9 inputs and so one until we accumulate 3 times after which adds the bias to the addition's outcome. After the multipliers, the pipelining registers are employed to reduce the timing path.

To avoid overflow after each multiplier we double the number of bits and after each 3-input adders we increase the number of bits by 2 and make sure that the accumulation register had enough bits to avoid overflow, the number of total added bit due addition is log2(M), where M is number of addition operation then a RELU step is done to get the output to be positive only then we add a quantized to get the output of the core to be 15 bits and if the number is bigger than the maximum number that should be stored in 15 bit we output the maximum number that can be represented in 15 bits.

The accumulator register should be reset after each calculated output to reset the accumulation process. But instead of used a reset signal we added a MUX to choose adding the stored value or add a zero as another way of resetting the register. And this allowed us to save the wasted clock cycle in resetting the register.

3-Input Adder:

In our design we use 3-input adder instead of normal 2-input adder to implement out adder trees. That have a lot of advantages.

1) The time of one 3-input adder is less than two 2-input adders.
2) The area of one 3-input adder is better than two 2-input adders.
3) Use less registers in the tree.

The 3-input adder is stage of full adders (FA) but instead of the carry input we use the third input. This allows them to work in parallel ant the FAs generate 2 victors the SUM vector and the CARRY vector this to vectors are added using 2-input adder.

5.3.2.2 Max Pooling Core

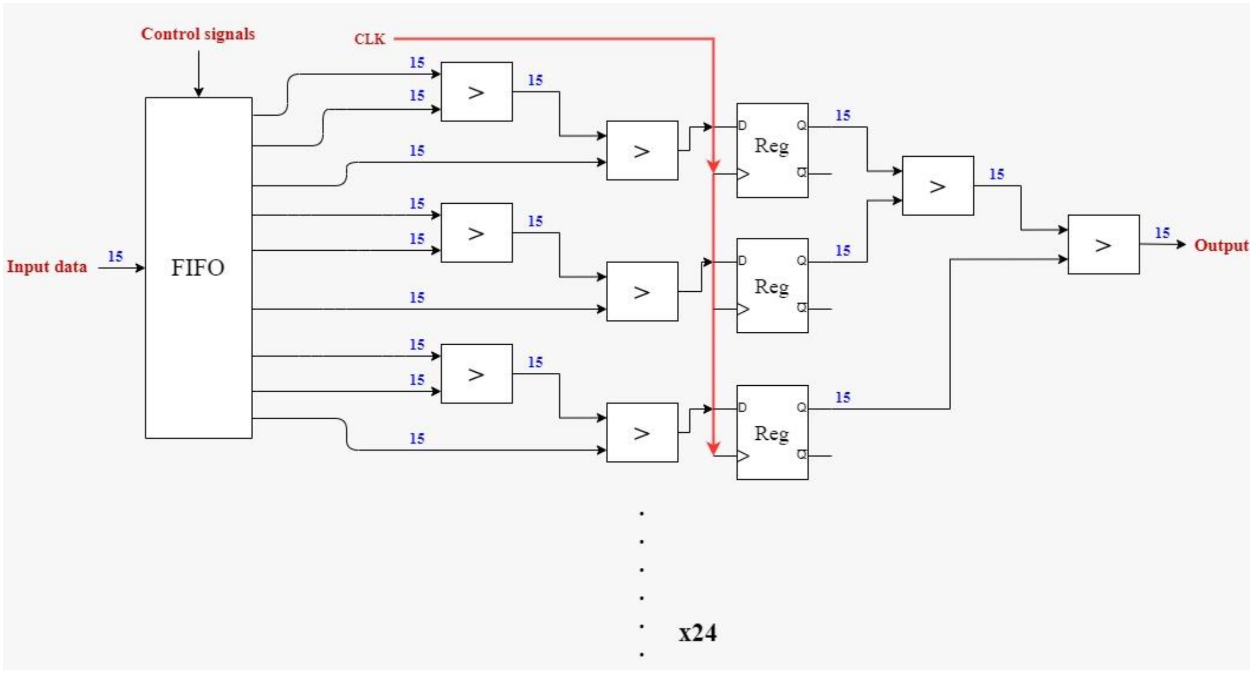


Figure 5.9 Maxpooling computation core

Parallelism

Since the input is 24 channels and the average pooling window is 9 elements, we used a full parallel in channels and window because the output of the 3x3 convolution is 24 channels in parallel as summarized in Table 5.2 and the max pool work the output of the 3x3 conv in the same time the output is generated.

Table 5.2 Parallelism in the Maxpooling core

Parallelism in Filters	Parallelism in Channels	Parallelism in window
NA	24	9

Input

The inputs to the Max Pooling core are (24*) 9 parallel input data coming from the 24 FIFOs.

Output

The output of the Max Pooling core is (24*) 1 output to be written in the max pool memory.

Methodology

This core takes 3x3 window from the output of the 3x3 convolution core and outputs the maximum of them. Using a tree of comparators. The sizes don't change along the tree because comparing does not increase the size. We break the critical path with registers after the second stage of comparing as shown in Figure 5.9.

5.3.3 FIFOs

In this section will discuss the FIFOs used in the group and why we need them. As we know that in convolution process, we take a window from the feature map and multiply it by the filter and add all the result to get one output pixel and the window keep moving until we scan the hole feature map. During this operation we need to read from the feature map memory from different location with complex equation and between two different windows from the input there are overlapping in data that mean we will have to read the data more than one which will waste and consume time and power.

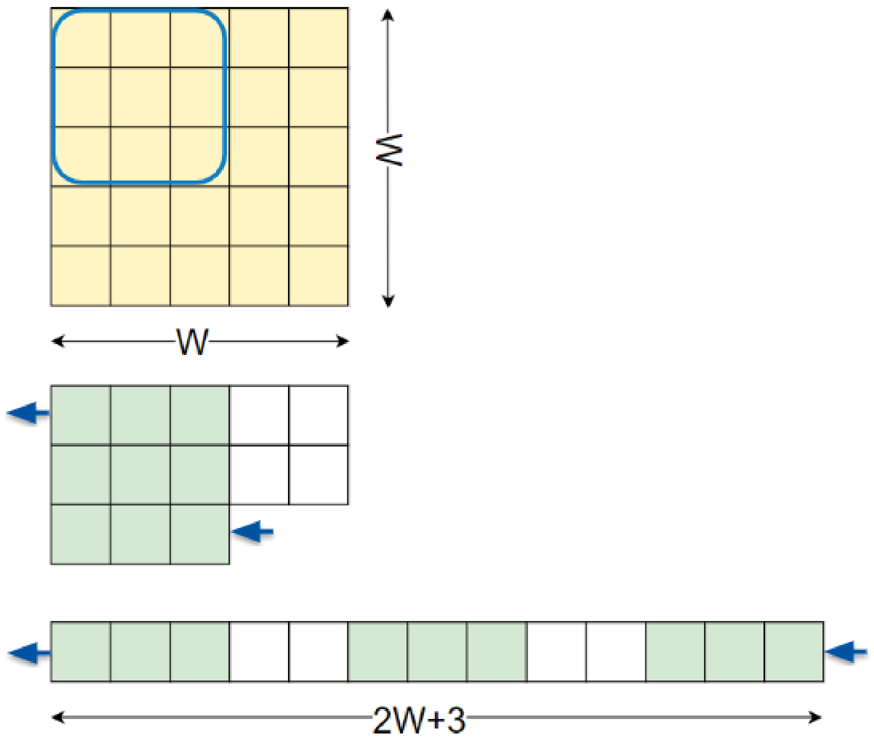


Figure 5.10 FIFO mechanism

So how to solve or optimize this operation. We use FIFO with the size of (2W+3), where W is feature map number of columns, the FIFO is simply a shift register and we read the data from the feature map 1 by 1 and store it in the FIFO as shown in Figure 5.10. By the time you fill the FIFO with data then the window will be in fixed place and each new data loaded the new window will be in the same place by using FIFOs that allowed us to read the content of the memory one by one in order without repeating any read operation instead of using a complex equation and complex logic to access the memory by the right address.

FIFOs in Group1:

- For 3x3 convolution: we use 3 FIFOs of size (2*226+3) as the size of the input photo is 224 and we have 2 bits for the padding. And 3 FIFOs to store each input channel in a separate FIFO so we can use full parallelism in channels.
- For the Max Pooling: we use a 24 FIFOs of size (2*114+3) the store the output of the 3x3 conv block and get each window ready for the max pooling core to process it without store to memory and then read from it again.

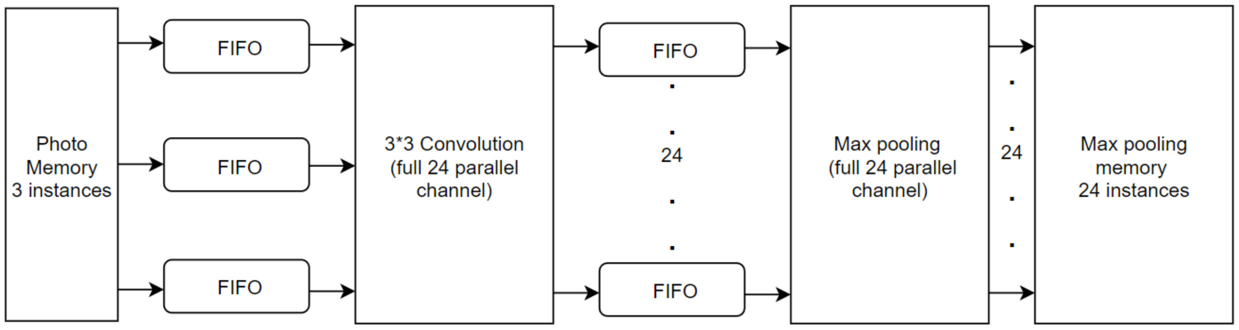


Photo Memory 3 instances
FIFO
FIFO
FIFO
3*3 Convolution (full 24 parallel channel)
FIFO
24
FIFO
Max pooling (full 24 parallel channel)
24
Max pooling memory 24 instances

Figure 5.11 FIFOs for the 3x3 Conv and Maxpooling cores

5.3.4 Controllers

In this section will discuss the controllers implemented in Group1 and explain each one role in the Group1. The group has three controllers.

1) 3x3 convolution FIFOs controller
2) Max Pooling FIFOs controller
3) Group1 Controller

The 3x3 convolution FIFOs controller and the Max Pooling FIFOs controller are almost having the same operation and functions so will discuss only one of them.

5.3.4.1 3x3 Convolution FIFOs Controller

The main purpose of this controller is to manage the storage data in the FIFO to solve the padding problem. But why is padding a problem and how the controller solves it.

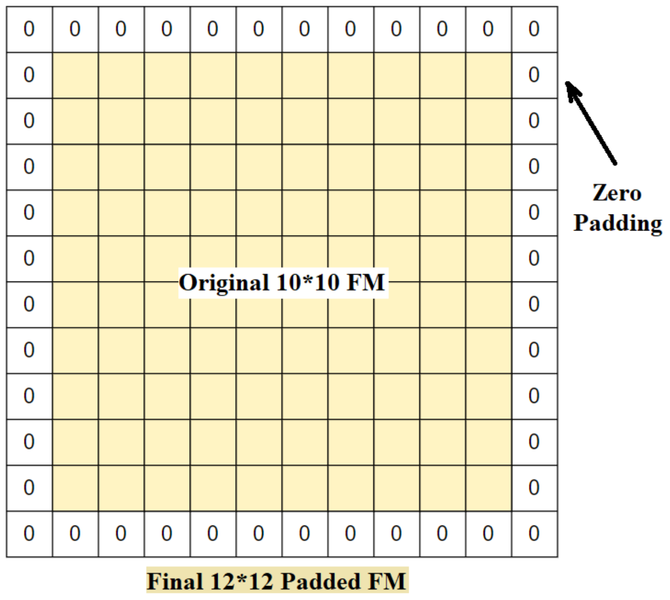


Figure 5.12 Padding a feature map

To pad a channel of the input you need to add zeros around the feature map as shown in Figure 5.12. A way to do that in hardware is to store zeros in certain position in the data memory and store your data around those zeros in calculated order so when you read the data you read it in the right order with the padded zeros. But anyone can see how much this method waste and how much complexity it adds to the design. Because you need to store the zeros in no order and then read them again. So, haw the FIFO controller manage to solve that padding problem.

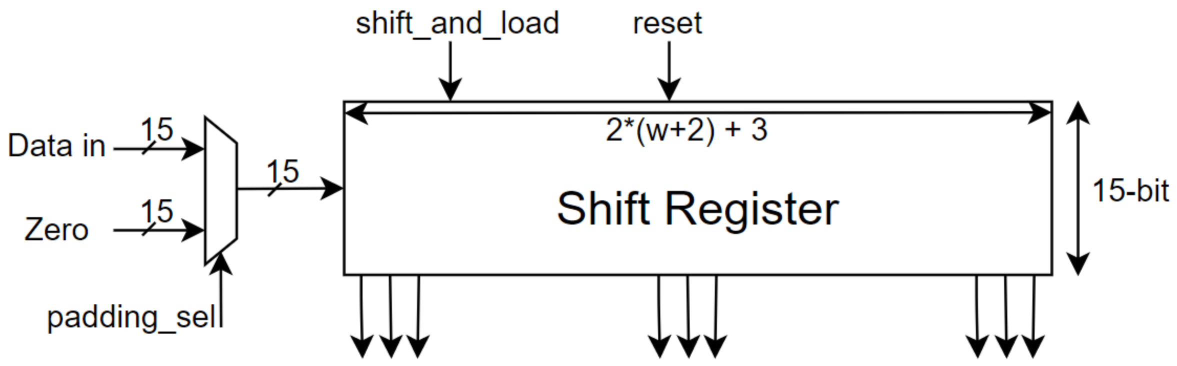


Figure 5.13 FIFO architecture

By adding a mux in the input of the FIFO to choose from load a data from the memory or to load a zero as shown in Figure 5.13. And simply the controller job is to let the FIFO load zeros in certain positions passed on counters in the controller to keep tracking of the data. The Controller is implemented as a finite state machine with four states as shown in Figure 5.14.

1) Idle
2) Load row
3) Load window
4) Process

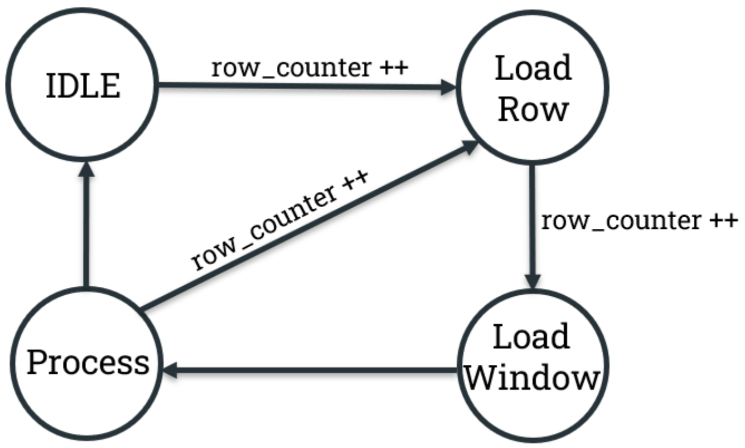


Figure 5.14 State diagram of the FIFO controller FSM

Idle: in that state we reset the FIFOs content and all signals are set to them default values waiting for start signal so start the reading process. Important note that in that state effectively we stored the first row of any padded feature map is it all zeros.

Load Row: in that state we load a row of data which is zero then W data then zero (where W is feature map size). After loading the row of data, the counter reaches its maximum and then go to load window state.

Load Window: if you notch that after loading a row of data in order to get a valid window for processing, we need to load 3 data before starting processing and this is the job of that state. It loads a zero for padding thin loads 2 data from memory and then we go to the process state.

Process: in that state we give signals to the computation core that there is a valid window of data which can be processed. And keep loading data to get the new window and so on until we process all possible windows in that row. From this point we customize the process according to some factors.

- If the convolution was stride 2 so we need to skip a row, we go to the load row state to load a row without processing it.
- If the convolution is stride 1, we go to load window then process and so on until we reach the last row.
- If we at the last row, we go to the idle state and end the operation.

5.3.4.2 Group1 Controller

This controller purpose is to synchronize all the blocks operations together. To make sure that the data is read from the memory is correct and at the correct clock cycle to be stored in the FIFOs. And to give the starting signals to the FIFOs and the resets of the registers in the computation cores. And the addresses for the Filter memory.

This controller is very simple that it is not even a state machine it uses only counters and logic based on the input signals from the FIFOs controllers.

5.4 Group 2 Design

The building block of Group 2 is the Shuffle unit which has two forms, the stride 1 block and the stride 2 block as discussed before in chapter 3. Group 2 consists of a sequence of 16 Shuffle unit blocks split into 3 stages as shown in Figure 5.15. Each stage starts with one stride 2 block and then continues with stride 1 blocks. The input to Group 2 is a 56*56*24 feature map coming from the Maxpool memory. Each stride 2 block decreases the width of the feature map to its half and increases the number of channels till it reaches 7*7*464 at the Group 2 output which is stored in the Extra memory.

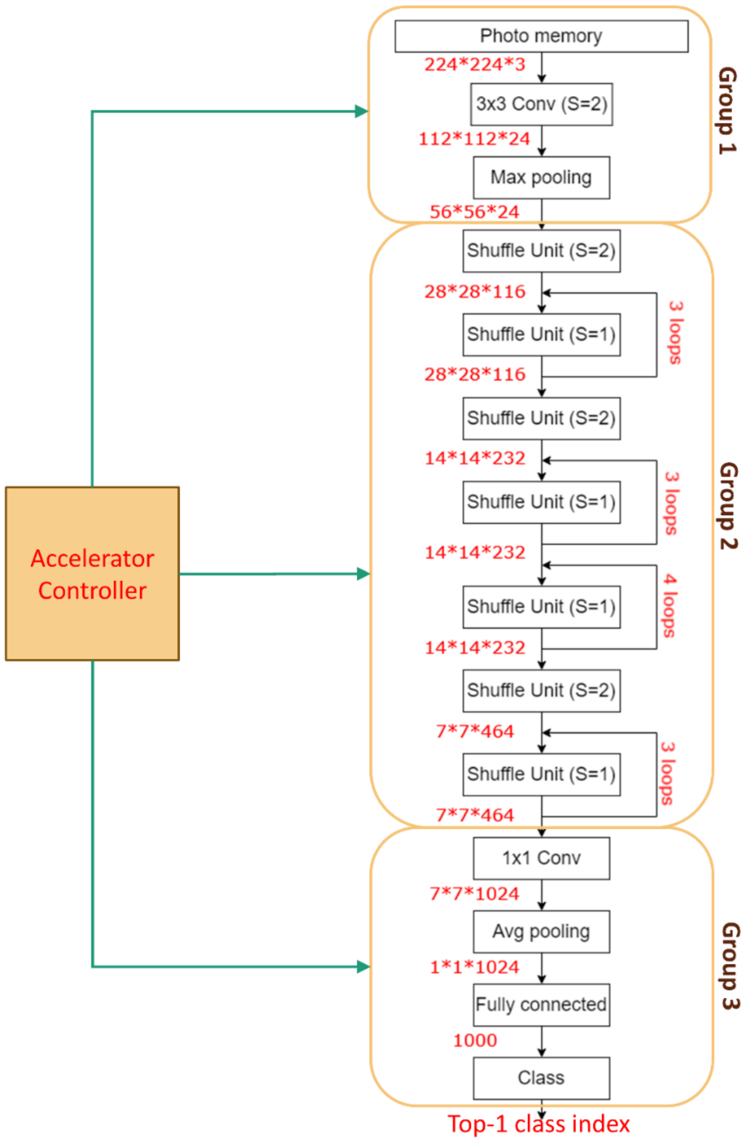


Figure 5.15 Group 2 consists of a sequence of 16 Shuffle unit

In stride 2 block, we have 5 convolutions to be done, but since the 1by1 convolution and the 3by3 DW convolution are done simultaneously in an alternating way, we can use only 2 convolution cores, one for the 1by1 convolution and the other for the 3by3 DW convolution and reuse them. In stride 1 block, we have 3 convolutions to be done, but since the 1by1 convolution and the 3by3 DW convolution are done in an alternating way as shown in Figure 5.16, we can use only 2 convolution cores, one for the 1by1 convolution and the other for the 3by3 DW convolution and reuse them.

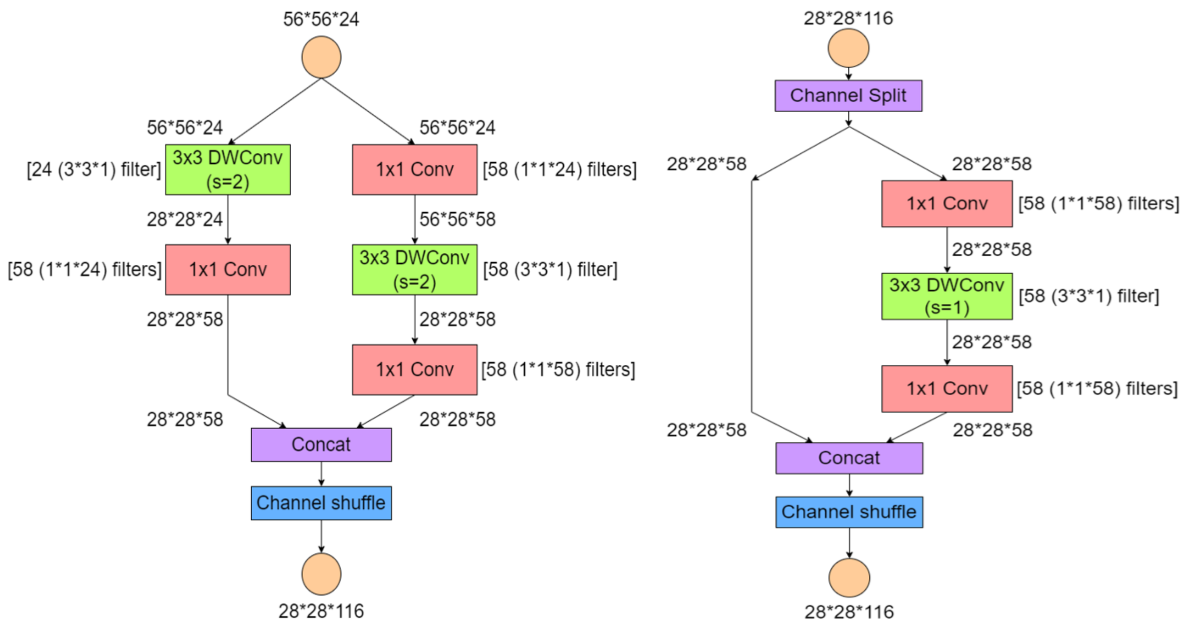


Figure 5.16 Stride 2 and stride 1 blocks

Since the stride 1 and stride 2 blocks are mutually exclusive and never done in parallel, then we can use the same 2 convolution blocks in the whole architecture and all the stages can share the same 2 convolution cores.

The hierarchy of Group 2 consists of computational cores, feature map memories, filter and bias memories, controllers and other special blocks like Shuffling Unit and FIFO Registers. This hierarchical system can be structured like follows:

(1) 3by3 DWconv block
  a. 3by3 DWconv Computational core
  b. 3by3 DWconv filter memories
  c. 3by3 DWconv bias memories
  d. FIFO and FIFO Controller
  e. 3by3 DWconv Controller
(2) 1by1 Conv block
  a. 1by1 Conv Computational core
  b. 1by1 Conv filter memories
  c. 1by1 Conv bias memories
  d. 1by1 Conv Controller
(3) Shuffling Unit
(4) Feature Map Memories (Shuffle, X_Left, X_Right and Y memories)
(5) Group 2 Controller

Throughout this section we will explain how these blocks are designed and assembled together.

5.4.1 Computational Cores

This group has two computational cores which are the 3by3 DW convolution and the 1by1 Convolution. In each of them we will discuss the parallelism used in filters, channels and window, The pipelining registers used to break the long timing path and how quantization is done to have a 15-bit fixed point output from the core without overflow.

5.4.1.1 3by3 DW Convolution

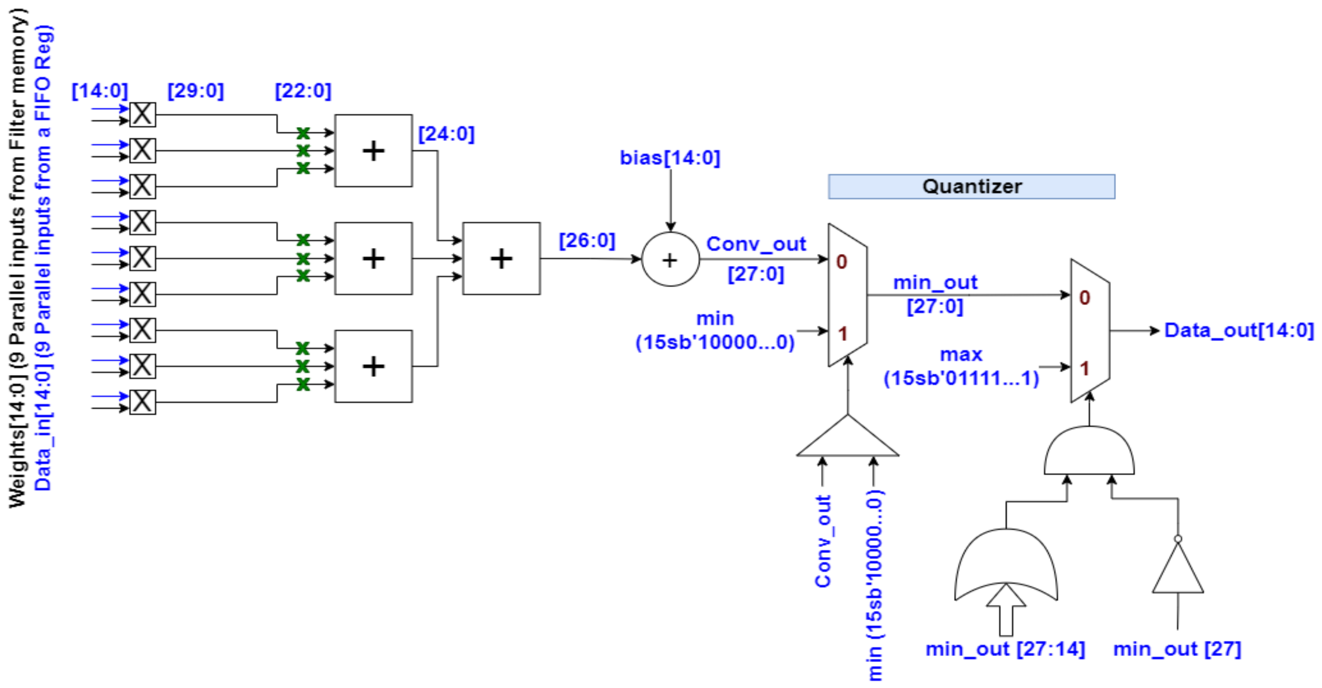


Figure 5.17 3by3 DWconv core

Parallelism

Since each filter has one channel consisting of 9 elements which is a small number, then convolution is performed to the whole window in parallel. And to speed up the convolution operation, 58 filters are executed in parallel for a window of parameters as summarized in Table 5.3 then in the next cycle the next 58 filters are convolved and so on till the filters of convolution are done.

Table 5.3 Parallelism in the 3by3 DWconv core

Parallelism in Filters	Parallelism in Channels	Parallelism in window
58	NA	9

Inputs

The inputs to the 3by3 DW convolution core are as follows:

4. Data: 9 parallel inputs coming from the FIFO registers.
5. Weights: (58*) 9 parallel inputs from filter memory.
6. Bias: (58*) 1 input from bias memory.

Output

The output of the 3by3 DW convolution core is (58*) 1 output to be written in a feature map memory.

Methodology

The input 9-elements window from the FIFO enters this layer and the operation is held by convolving it with 58 filters in parallel. The convolution operation is executed by multiplying each parameter pixel to the corresponding pixel in the filter by using DSPs, then adding the 9 products together using an adder tree, where the Adder3 block is used to add each 3 products. Then adding the bias to the result of addition as shown in Figure 5.17. The pipelining registers are used after the multipliers to break the long timing path and to make sure the inputs to the adder tree are constant when not using them so as to save dynamic power.

The multipliers double the number of bits from 15 to 30 then the 7 LSBs are thrown away as quantization noise, the 3-input adder increments the number of bits by 2 and the 2-input adder increments the number of bits by 1. Thus, the output of convolution is 27-bits not 15-bits and overflow may occur. To make sure the output is the correct 15-bits without overflow, a quantizer is placed at the end of each computation core to compare the output of convolution to the maximum and minimum possible values of the 15-bit fixed point output. The quantizer consists of two multiplexers, one for maximum checking and the other for minimum checking. Firstly, we check for the minimum value using a comparator and if the number is larger than the minimum value, the number is passed as it is. Otherwise, the minimum value is passed (saturation case to avoid overflow). Secondly, we check for the maximum value using a simple logic circuit instead of a comparator and if the number is smaller than the maximum value, the number is passed as it is. Otherwise, the maximum value is passed (saturation case to avoid overflow).

5.4.1.2 1by1 Convolution

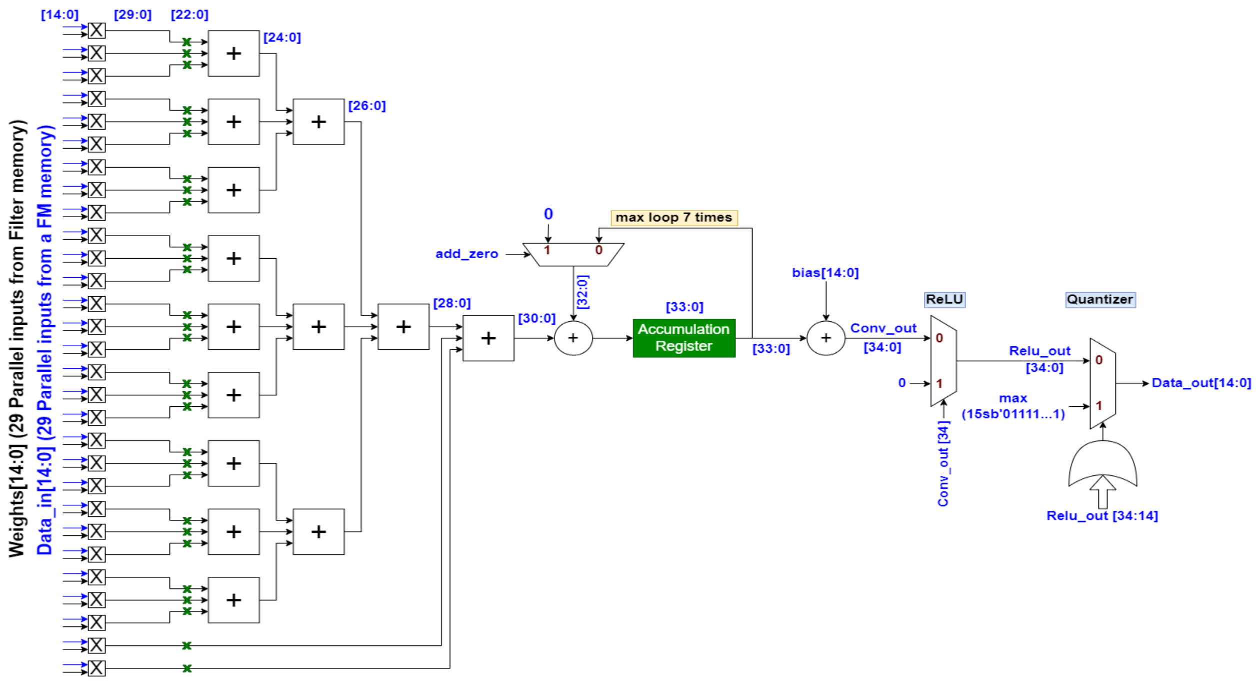


Figure 5.18 1by1 Conv core

Parallelism

Each filter has multiple channels that can be 24, 58, 116 or 232 channels depending on the layer and each channel has one element only. Thus, to speed up the convolution operation, 58 filters are executed in parallel for 29 channels of each filter as summarized in Table 5.4 then in the next cycle the next 29 channels are convolved and so on till all the channels are done. That's why an accumulator is placed to sum up all the products of all channels.

Table 5.4 Parallelism in the 1by1 Conv core

Parallelism in Filters	Parallelism in Channels	Parallelism in window
58	29	NA

Inputs

The inputs to the 1by1 Convolution core are as follows:

1. Data: 29 parallel inputs coming from a feature map memory.
2. Weights: (58*) 29 parallel inputs from filter memory.
3. Bias: (58*) 1 input from bias memory.
7. Control signals like the reset and add_zero signals.

Output

The output of the 1by1 Convolution core is (58*) 1 output to be written in a feature map memory.

Methodology

The input 29 elements from the feature map memory enters this layer and the operation is held by convolving it with 58 filters in parallel. The convolution operation is executed by multiplying each parameter pixel to the corresponding pixel in the filter by using DSPs, then adding the 29 products together using an adder tree, where the Adder3 block is used to add each 3 products. 27 products are added in a 3-level adder tree then the remaining two products are added using an extra 3-input adder level. Then accumulating the result to the result of the next 29 channels until all the channels are convolved. The accumulator loops a maximum of 7 times, this maximum happens when the filter has 232 channels. In the feedback of the accumulator, we either add the accumulated value or add zero using a MUX. This is used to add zero instead of accumulation in the beginning of each pixel to save the clock cycle wasted to reset the accumulator register as shown in Figure 5.18.

After the accumulation of all channels' convolution, the bias is added to the result. After any 1by1 convolution in this architecture there is a ReLU that rectifies the output to only positive values and force the negative values to be zero. The ReLU checks if the number is negative using the MSB of the convolution output instead of using comparators which leads to high power consumption. The pipelining registers are used after the multipliers to break the long timing path and to make sure the inputs to the adder tree are constant when not using them so as to save dynamic power.

The multipliers double the number of bits from 15 to 30 then the 7 LSBs are thrown away as quantization noise, the 3-input adder increments the number of bits by 2, the 2-input adder increments the number of bits by 1 and the accumulator increments the number of bits by $\lceil \log_2 7 \rceil = 3$. Thus, the output of convolution is 34-bits not 15-bits and overflow may occur. To make sure the output is the correct 15-bits without overflow, a quantizer is placed at the end of each computation core to compare the output of convolution after the ReLU to the maximum possible value of the 15-bit fixed point output. The quantizer consists of a multiplexer and a simple logic for maximum checking. We check for the maximum value using a simple logic circuit instead of a comparator to save power and area. If the number is smaller than the maximum value, the number is passed as it is. Otherwise, the maximum value is passed (saturation case to avoid overflow). Checking for the minimum case is not required here as the ReLU output will never be a negative value.

5.4.2 Filter and Bias Memories

ShuffleNet V2 holds about 2.3M parameters, which are the architecture's weights and biases. The relatively low parameters make it feasible to implement the architecture on FPGAs for acceleration approach. However, this architecture deployment requires more attention to weights and biases rather than all other building blocks, as they consume a huge number of resources and, of course, they affect speed directly. Usually, fixed parameters are stored into Read Only Memories (ROMs), with a controller to take over fetches, flags and more.

Some FPGAs have off-chip ROMs, which have great utility in account of high latency, however, ShuffleNet implementation targets speed as its first priority. So, off-chip memories would not be an option. We can put the parameters on one of two types of memories on the FPGA.

First type of memories is the block RAM. FPGAs normally have an on-chip BRAM matrix, which could be configured as FIFO, RAM or ROM. Targeted device (Virtex-7) contains a sum of 2940 BRAM 18Kb instances. BRAMs can have dual ports for the same instances, allowing performance of half the latency. Second type of memories is the distributed ROM (LUT), which can be configured to hold design parameters. Normally, distributed ROMs grant more speed than BRAMs, which makes them a good approach to ShuffleNet implementation.

In this section, the weight and bias memories of both the 1by1 convolution and 3by3 DW convolution will be explained including their organization, number of memory instances and utilization of BRAMs or LUTs used.

5.4.2.1 1by1 Conv Weights Memories

As we have a single computation core for the 1by1 Convolution used by all layers in group 2, so it's better to have one memory to hold all the 1by1 weights. This will save a lot of BRAMs. The number of instances in the memory is indicated by the parallelism used in the 1by1 convolution core, which is 58 in filters and 29 in channels, so we need 29*58 weights simultaneously. We will use 29*29 = 841 instances of dual port BRAM memories to provide the 29*58 parallel weights. Each instance is 880 weights as explained in Table 5.5 stored in 0.5 BRAM, so total BRAMs for the 1by1 weights is 420.5 BRAMs.

<u>Number of weights per memory instance</u>

Table 5.5 Number of weights per memory instance of the 1by1 Conv weights memory

1st & 2nd 1by1 Conv layer	2*(2 weights)
3rd to 9th 1by1 Conv layer	7*(4 weights)
10th to 26th 1by1 Conv layer	17*(16 weights)

27th to 35th 1by1 Conv layer	9*(64 weights)
Total weights per instance	880 weights

Filter memories organization

One approach to put weights in the memory instances is to put all weights of the first convolution filter then the second filter in the row above and so on as shown in Figure 5.19, but this approach is not practical as it will require multiplexers to route the correct 29 weights to the computation core which has a 29 parallelism in channels. Thus, another approach will be used to organize the weights in a way that will not require any extra hardware with the memory.

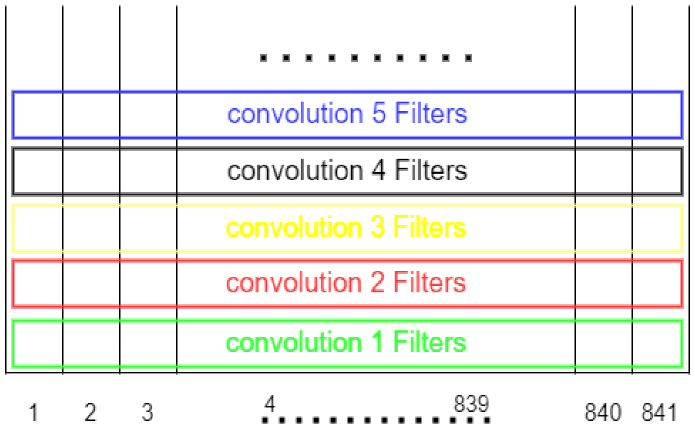


Figure 5.19 First approach of the filter memories organization for the 1by1 Conv

To make all the weight inputs of all cores connected to same ports of memories all the time, each filter will have its channels as a multiple of 29, except for the first two 1by1 convolution layers that have 24 channels only will be extended with zeros to be 29 channels. As the filter is calculated on the same core, so channels of it must come to the same inputs of the core. Hence, each 29 channels of the filter are put above each other on the same 29 memory instances to be routed directly to the computation core just by incrementing the filters memory address counter.

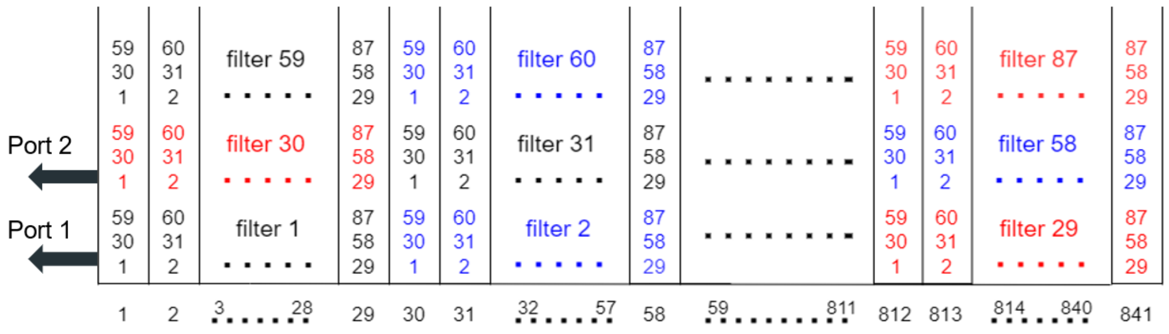


Figure 5.20 Second approach of the filter memories organization for the 1by1 Conv

Figure 5.20 shows the organization of the 1by1 filters memories. Assuming each filter has 3*29=87 channels placed above each other, each filter will be placed in 3 rows, where each row carries 29 filters. Using the 2 ports of the memory we can output all the 58 filters, 29 channels per filter, where the offset between the filter memory address of port 1 and port2 is obviously the number of channels as a multiple of 29, which means that the beginning of filter (x+29) is above the beginning of filter (x) by the number of channels. So, accessing the filter memory is straight forward without any means of multiplexing.

5.4.2.2 3by3 DWconv Weights Memories

As we have a single computation core for the 3by3 DW Convolution used by all layers in group 2, so it’s better to have one memory to hold all the 3by3 weights. This will save a lot of BRAMs. The number of instances in the memory is indicated by the parallelism used in the 3by3 convolution core, which is 58 in filters and 9 in window, so we need 9*58 weights simultaneously. We will use 9*58 = 522 instances of LUT memories to provide the 9*58 parallel weights. We will use LUTs instead of dual port BRAM as each instance is 43 weights only that will give a very poor utilization of the BRAMs as explained in Table 5.6. This 43 weights instance will be stored in 15 LUTs, so total LUTs used for the 3by3 DWconv weights is 7830 LUTs.

Number of weights per memory instance

Table 5.6 Number of weights per memory instance of the 3by3 DWconv weights memory

1st to 5th 3by3 DWconv layer	5*(1 weight)
6th to 14th 3by3 DWconv layer	9*(2 weights)
15th to 19th 3by3 DWconv layer	5*(4 weights)
Total weights per instance	43 weights

Filter memories organization

One approach to put weights in the memory instances is to put all weights of the first convolution filter then the second filter in the row above and so on as shown in Figure 5.21, but this approach is not practical as it will require multiplexers to route the correct 9 weights to the computation core which has a 9 parallelism in window. Thus, another approach will be used to organize the weights in a way that will not require any extra hardware with the memory.

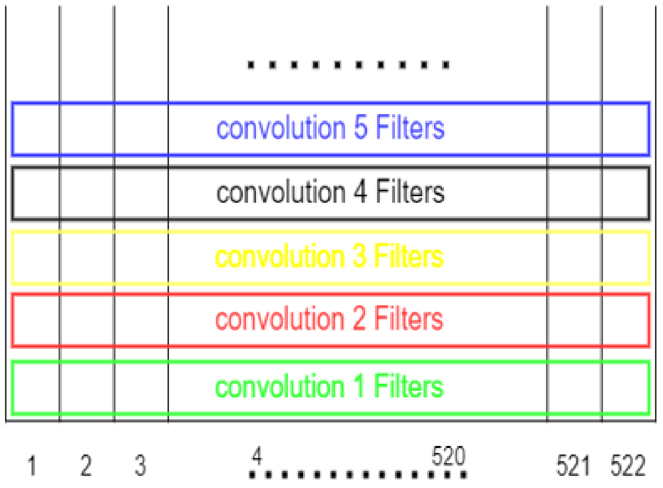


Figure 5.21 First approach of the filter memories organization for the 3by3 DWconv

To make all the weight inputs of all cores connected to same ports of memories all the time, each filter will have its 9 weights put in 9 instances of memory. As the filter is calculated on the same core, so all its weights must come to the same inputs of the core. Hence, after finishing each

58 filters in parallel we just increment the filters memory address counter to move forward to the next 58 filters.

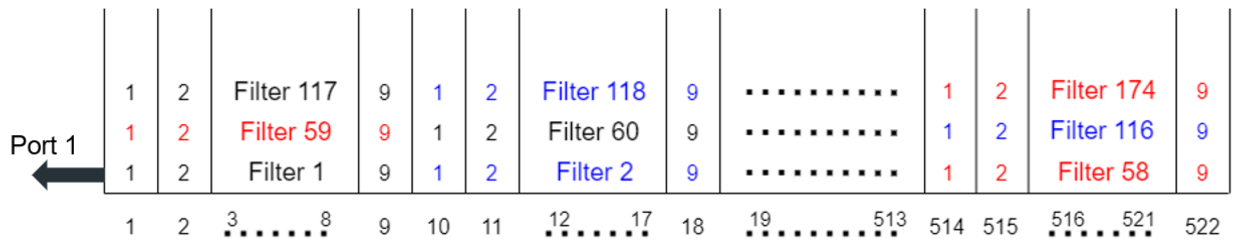


Figure 5.22 Second approach of the filter memories organization for the 3by3 DWconv

Figure 5.22 shows the organization of the 3by3 DWconv filter memories. Each filter has a 9 elements window, where each row carries 58 filters. Using only a single port of the memory we can output all the 58 filters, 9 weights per filter, where the filter (x+58) is put above filter x on the same 9 instances. So, accessing the filter memory is straight forward without any means of multiplexing.

5.4.2.3 1by1 Conv Bias Memories

Since we have 58 parallel filters for the 1by1 Conv core, so we need the memory to provide 58 biases simultaneously. We will not just connect the biases to VDD and GND as we have a huge number of biases, so it's better to store them in LUTs. We will use 58 instances of LUT memories to provide the 58 parallel biases. We will use LUTs instead of BRAMs as each instance is 79 biases only that will give a very poor utilization of the BRAMs as explained in Table 5.7. This 79 biases instance will be stored in 25 LUTs, so total LUTs used for the 1by1 Conv weights is 1450 LUTs.

Number of biases per memory instance

Table 5.7 Number of biases per memory instance of the 1by1 Conv bias memory

1st & 9th 1by1 Conv layer	9*(1 biases)
10th to 26th 1by1 Conv layer	17*(2 biases)
27th to 35th 1by1 Conv layer	9*(4 biases)
Total bias per instance	79 biases

Bias memories organization

To make all the bias inputs of all cores connected to same ports of memories all the time, the 58 memory instances are connected to the 58 bias inputs of the 58 parallel 1by1 cores. Hence, after finishing each 58 filters in parallel we just increment the bias memory address counter to get the next 58 biases.

5.4.2.4 3by3 DWconv Bias Memories

Since we have 58 parallel filters for the 3by3 DWconv core, so we need the memory to provide 58 biases simultaneously. We will not just connect the biases to VDD and GND as we have a huge number of biases, so it's better to store them in LUTs. We will use 58 instances of LUT memories to provide the 58 parallel biases. We will use LUTs instead of BRAMs as each instance is 43 biases only that will give a very poor utilization of the BRAMs as explained in Table

5.8. These 43 biases instance will be stored in 15 LUTs, so total LUTs used for the 3by3 DWconv weights is 870 LUTs.

Number of biases per memory instance

Table 5.8 Number of biases per memory instance of the 3by3 DWconv bias memory

1st 3by3 DWconv layer	1*(1 bias)
2nd to 5th 3by3 DWconv layer	4*(1 biases)
6th to 14th 3by3 DWconv layer	9*(2 biases)
15th to 19th 3by3 DWconv layer	5*(4 biases)
Total weights per instance	43 biases

Bias memories organization

To make all the bias inputs of all cores connected to same ports of memories all the time, the 58 memory instances are connected to the 58 bias inputs of the 58 parallel 3by3 DWconv cores. Hence, after finishing each 58 filters in parallel we just increment the bias memory address counter to get the next 58 biases as shown in Figure 5.23.

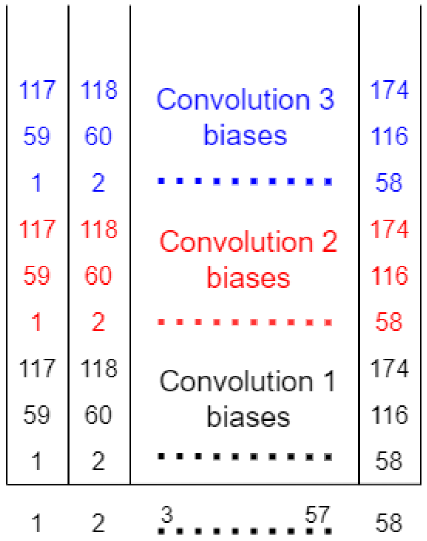


Figure 5.23 Bias memory organization

5.4.3 Feature Map Memories

Each layer has output feature maps that have to be stored in order to be passed to the next layer, storing can be done in memory or cache using different resources in FPGA. The number of words of output feature maps of the first 1by1 Conv layer is 56x56x58=181,888 words, so 181,888 flops are needed in order to store these feature maps in a cache using flip flops which is a massive number for storing only one layer so this implementation can't be used. But if the output feature maps of 1by1 Conv layer gets stored in memories using BRAMs, about 100 BRAMs will be needed which is considered acceptable utilization of resources, but by calculating the total number of feature maps that need to be stored, a very huge number of BRAMs will be needed, in addition to that, BRAMs has a queuing problem as it’s needed to store multiple feature maps in the same clock cycle while BRAMs can’t store more than 2 inputs in one clock cycle, so this implementation can’t be used either.

The used implementation is a modification of the previous implementation using BRAMs to avoid the problems caused, 4 shared memories are used for all layers so that they can store and write from the same memory, and they are 4 to avoid the conflict between storing and reading in the same memory. Another modification to solve the queuing problem is using BRAM memories of multiple instances, and as feature maps consist of multiple channels so elements of each channel can be stored independently in the same clock cycle in different BRAM. So, this modification solved the queuing problem and allowed the BRAMs to be used as storage for feature maps between layers.

5.4.3.1 Reading and Writing Mechanisms

To determine the FM memory organization, we must study writing and reading mechanisms of 3*3 DW conv and 1*1 conv. Let's assume we have a 2*2*116 FM to clarify the writing and reading mechanisms.

3by3 DWconv reading mechanism

The 3by3 DWconv has a parallelism of 58 filters, so it will read 58 channels from the first pixel in the FM, then the 58 channels from the second pixel in the FM and so on till the fourth pixel. After reading from memory 58 input channels of all pixels, start reading the next 58 input channels as shown in Figure 5.24, where each color represents a pixel.

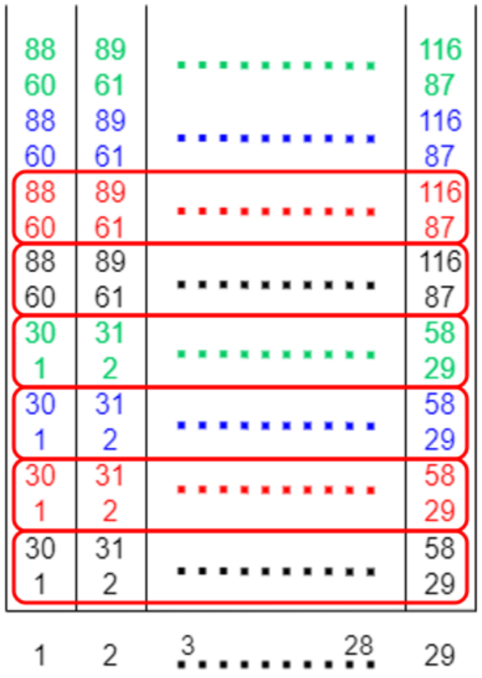


Figure 5.24 3by3 DWconv reading mechanism

1by1 Conv writing mechanism

The 1by1 Conv has a parallelism of 58 filters, so it will write 58 channels of the first pixel in the FM, then the 58 channels of the second pixel in the FM and so on till the fourth pixel. After writing in memory 58 channels of all pixels, start writing the next 58 channels as shown in Figure 5.25. This means that 1by1 writing mechanism is compatible with the 3by3 reading mechanism.

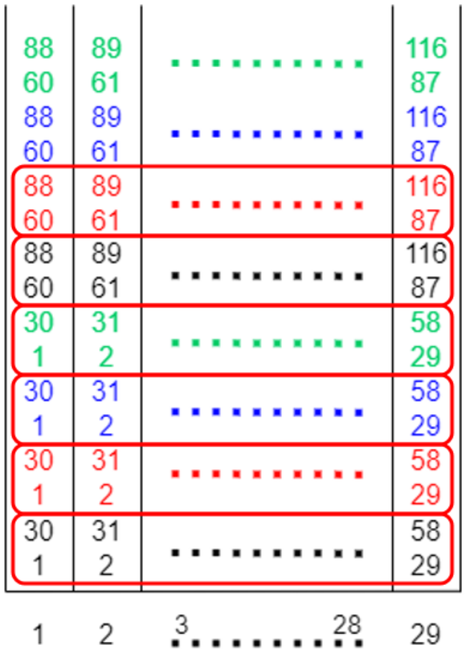


Figure 5.25 1by1 Conv writing mechanism

1by1 Conv reading mechanism

The 1by1 Conv has a parallelism of 29 channels, so it will read the first 29 channels from the first pixel in the FM, then the next 29 channels from the same pixel in the FM and so on till the fourth 29 channels from the same pixel. After reading from memory all channels (29 by 29) of the same pixel, start reading the next pixel in the same way as shown in Figure 5.26, where each color represents a pixel.

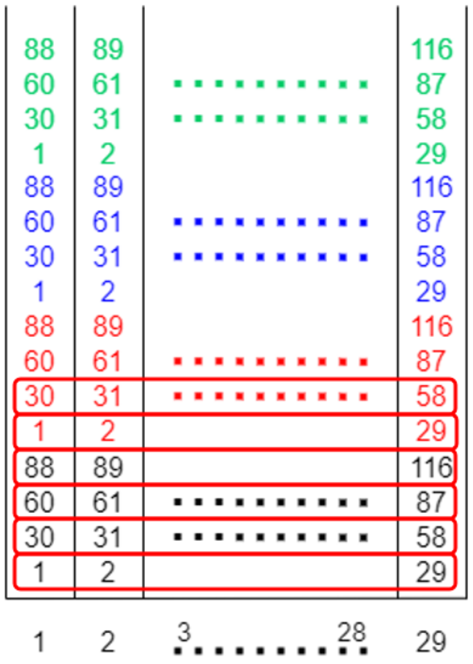


Figure 5.26 1by1 Conv reading mechanism

3by3 DWconv writing mechanism

The 3by3 Conv has a parallelism of 58 filters, so it will write 58 channels of the first pixel in the FM, then the 58 channels of the second pixel in the FM and so on till the fourth pixel. After writing in memory 58 channels of all pixels, start writing the next 58 channels in the same way as shown in Figure 5.27, where each color represents a pixel. This means that 3by3 writing mechanism is not compatible with the 1by1 reading mechanism.

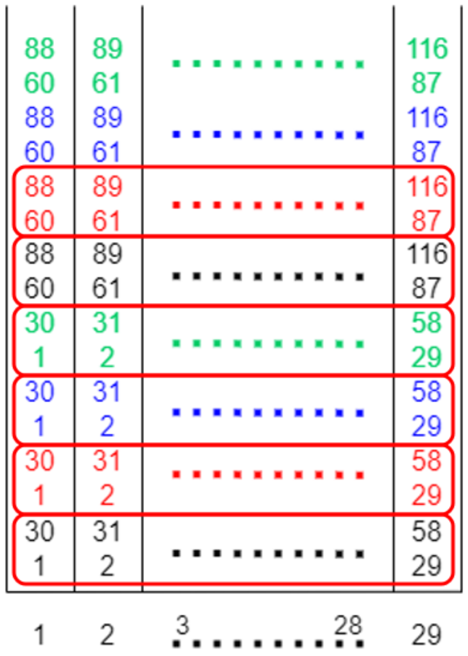


Figure 5.27 3by3 DWconv writing mechanism

5.4.3.2 Feature Map Organization in Memories

As we saw, all writing and reading mechanisms are the same except for the 1by1 Conv reading mechanism. Also, At the beginning of each stride 2 layer, the 1by1 Conv and the 3by3 DWconv will read from the same memory. Then organization in this memory must be suitable for both types of reading mechanisms. To do this we will follow one of two possible approaches.

First approach of memory organization

We will make organization in memory compatible with most mechanisms and make the 1by1 conv reading counter jumps to get the correct channels. But this approach has many drawbacks as follows:

- Jumping is very hard as every jump depends on feature map size and channels.
- The counter is initialized to different values after every pixel. Also, these different values depend on feature map size and number of channels.
- Maxpool memory is organized as all channels of same pixel because Maxpooling of group 1 is writing like this.
- Extra memory must be organized as all channels of same pixel because last 1by1 Conv of group 3 is reading like this.

Second approach of memory organization

Since the first approach is hard to implement and has many drawbacks, we will follow this second approach, which is making the organization in memory compatible with the 1by1 Conv reading mechanism, where all channels of each pixel are placed above each other, then the channels of the next pixel and so on. Thus, the 3by3 DWconv reading and writing mechanisms and the 1by1 Conv writing mechanism will be the same by making the counter jumps to get the correct channels as shown in Figure 5.28. The counter jumps by the number of channels each time and initialized to half the number of channels.

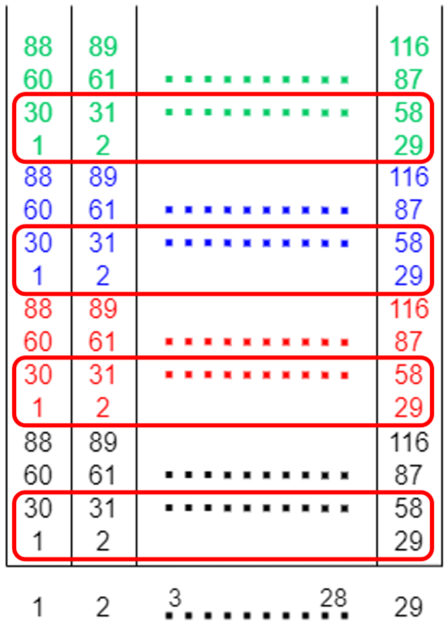


Figure 5.28 Second approach of memory organization

5.4.3.3 Technicalities of the Feature Map Memories

The intermediate memories between the different layers of convolution are only 4 memories named X_Left, X_Right, Y and Shuffle memories connected to the cores as shown in Figure 5.29.

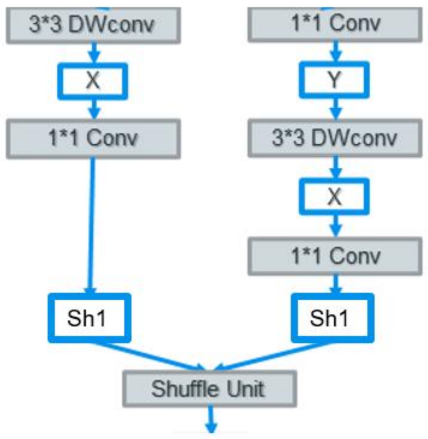


Figure 5.29 The intermediate feature map memories in Group 2

X_Left Memory

- 29 instances of 784 words (0.5 BRAM).
- Write in it: 3by3 DWconv core.
- Read from it: 1by1 Conv core.

X_Right Memory

- 29 instances of 1568 words (1 BRAM).
- Write in it: 3by3 DWconv core.
- Read from it: 1by1 Conv core.

Y Memory

- 29 instances of 6272 words (3.5 BRAM).
- Write in it: 1by1 Conv core.
- Read from it: 3by3 DWconv core.

Shuffle Memory

- 58 instances of 1568 words (1 BRAM).
- Write in it: 1by1 Conv core.
- Read from it: 1by1 Conv and 3by3 DWconv cores across the Shuffling unit.

Extra Memory

- 29 instances of 784 words (0.5 BRAM).
- Write in it: 1by1 Conv core across Shuffling unit.
- Read from it: Last 1by1 Conv core in group 2.

5.4.4 Shuffling Unit

In ShuffleNet V2 architecture, a channel shuffling is done between any stride 1 or stride 2 blocks to eliminate the side-effect of channel splitting by allowing information transfer, so the accuracy is maintained. To do the shuffling effect taking into consideration the organization of feature map memories, let’s assume we have two memories, shuffling from Shuffle 1 memory to Shuffle 2 memory as shown in Figure 5.30.

In stride 2, we need to read all channels, so we will read all channels from shuffle 2 memory. But the 1by1 Conv of Stride 2 needs to read all channels from 58 instances and we have only 29 parallel channels, so we need MUXs to choose between first 29 channels and second 29 channels (1 or 30, 2 or 31 and so on) as shown in Figure 5.31.

In stride 1, we need to read first half of channels only to the 1by1 Conv, so we will read half channels from shuffle 2 memory by making the address counter jump by half the number of channels.

To avoid this complexity of having two modes of reading from the Shuffle memory, one for stride 1 and the other for stride 2, we can organize first half of channels only in the left half of shuffle 2 memory to avoid this case.

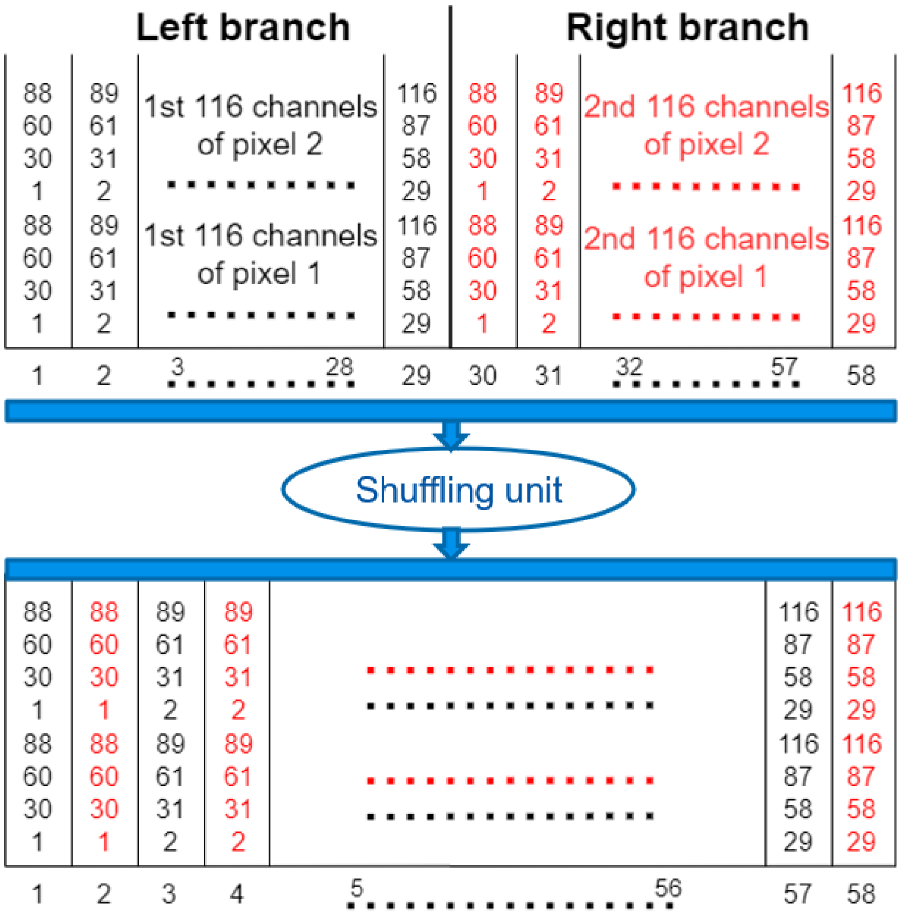

Left branch
Right branch
1st 116 channels of pixel 2
1st 116 channels of pixel 1
2nd 116 channels of pixel 2
2nd 116 channels of pixel 1
Shuffling unit

Figure 5.30 Shuffling the data from Shuffle 1 to Shuffle 2 memory

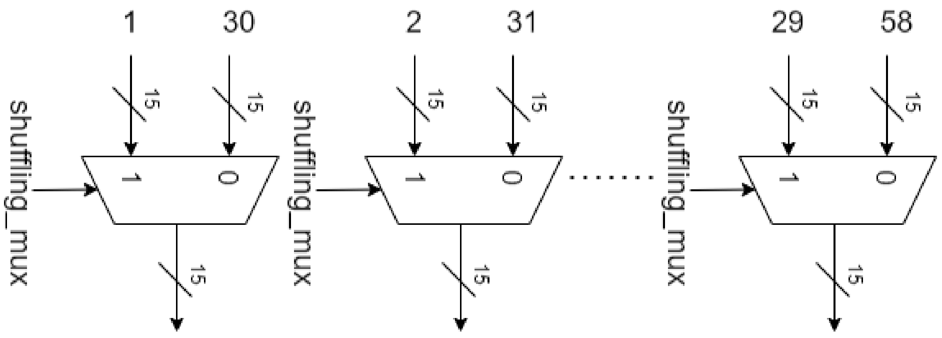


Figure 5.31 MUXs to choose between first 29 channels and second 29 channels

When we look on the impact of shuffling unit, it shuffles channels but also it can be seen as shuffling the instances of memory without affecting values stored in them. So, this leads us to just shuffle the ports of memory to have the effect of shuffling the channels. Thus, instead of reading data unshuffled then shuffling it then writing it in another memory, we can just read data from the Shuffle memory directly shuffled by choosing the correct instances.

The Shuffle unit will be the multiplexers that shuffle instances of memory by using their ports without moving any data. These MUXs are for the 1by1 Conv to have the 29 input channels either from the first half or the second half of memory channels. But, the 3by3 DWconv needs all the 58 ports to be connected. The connections inside the Shuffling unit are shown in Figure 5.32, where the black wires are connected to port 1 and blue wires are connected to port 2 of the Shuffle memory.

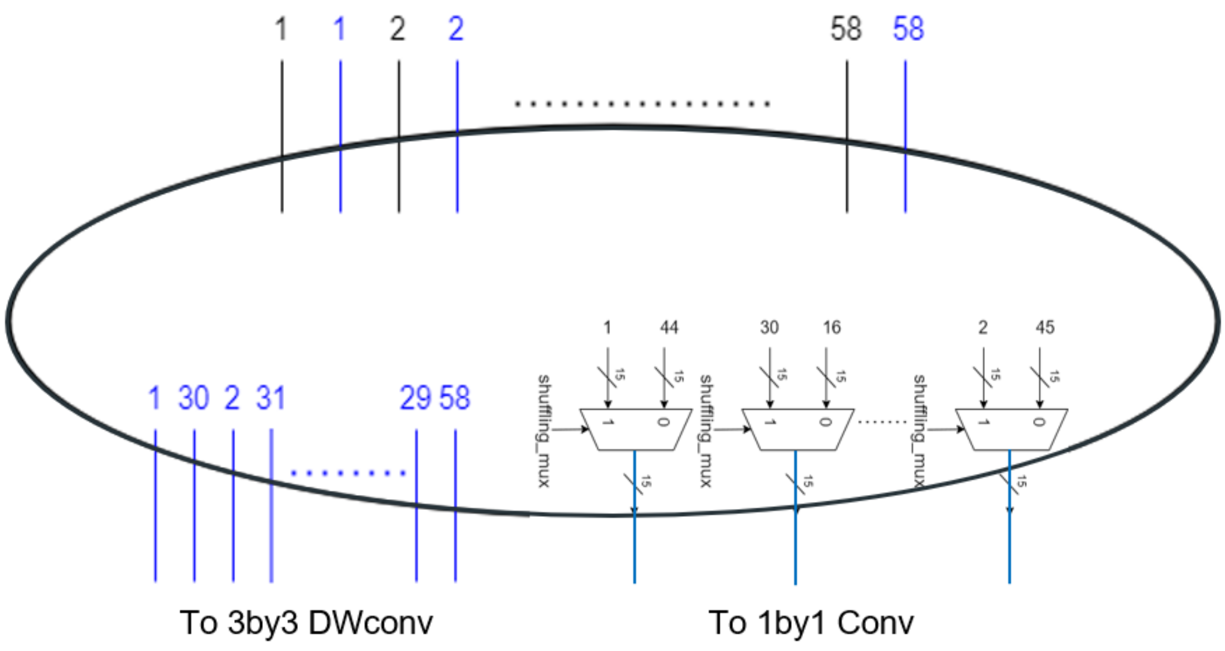


Figure 5.32 Shuffling unit block

We need to shuffle the ports of the Shuffle memory to have the same effect of channel shuffling. Figure 5.33 and Figure 5.34 shows which of the 58 instances of the Shuffle memory are connected to which inputs of the MUXs in the Shuffling unit. Each 29 instances are split into 15 and 14 groups, then by shuffling them and observing the colors of shuffled instances we can find that the inputs to the first MUX are the ports of the instances x and $x + 29 + \left\lceil \frac{29}{2} \right\rceil$ and the inputs to the second MUX are the ports of the instances $x + 29$ and $x + \left\lceil \frac{29}{2} \right\rceil$ and so on for all the 29 MUXs. The outputs of MUXs go to the 1by1 Conv core. This way of shuffling saves a lot of power and delay compared to shuffling the data words from one memory to another.

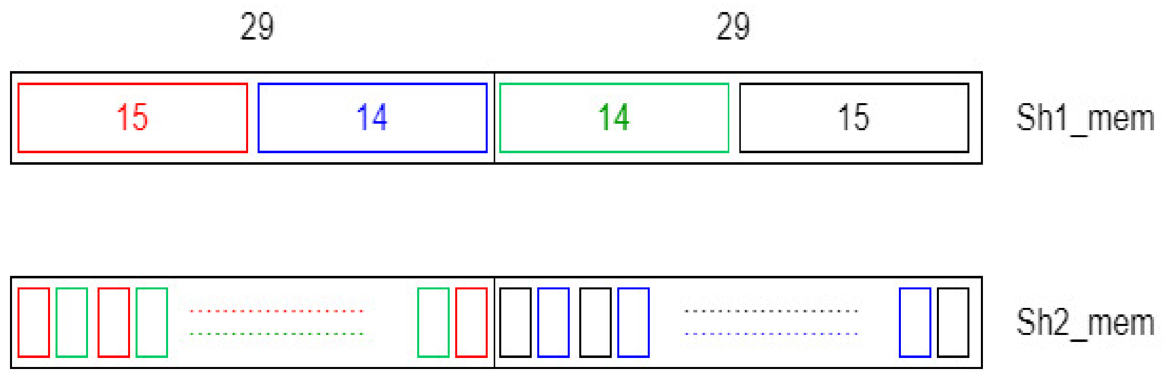


Figure 5.33 The effect of shuffling

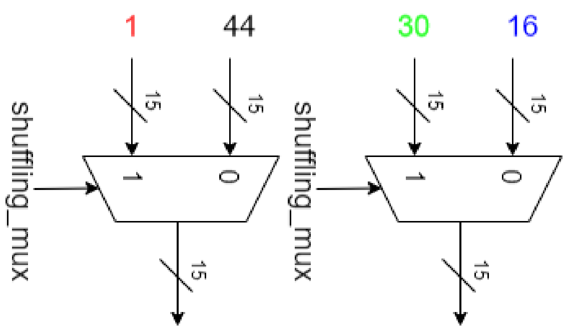


Figure 5.34 MUXs to read shuffled data from the correct ports

<u>Shuffling for stride 1 blocks, problem and solution</u>

When shuffling data between two stride 1 blocks a problem will occur because we read the data shuffled but the data is still in the memory unshuffled, so the empty branch in stride 1 block will pass the data unshuffled, however, it's supposed to pass shuffled data. Thus, we need to shuffle that half of data that is passed without shuffling. One approach is to shuffle the data in the shuffle memory to another memory in the empty path and read the shuffled data from this new memory, but this adds an overhead in the used resources and if we used one of the existing FM memories this will add extra delay to rewrite the shuffled data to the shuffle memory to begin a new stride block. Another approach is to use two memories for shuffling, shuffle 1 and shuffle 2 memories, but this is a waste of resources and increases latency.

Our solution to this problem is to write the shuffled data in the Shuffle memory again as we only will write half the data, we can write it the half that we don't need anymore. We will read the data and write it shuffled starting from the last address in the memory. This is done by reading one row of 58 instances using single port and writing it shuffled in 2 rows of 29 instances using dual port in the half that's not needed. After wring in address X, we write in address X-2, then X-4 and so on till address 0 as we write using 2 ports in two rows. This way we will never overwrite any needed data. The half that is needed is whole rows of 58 instances and the half that we want to write the shuffled data in is the whole 29 instances on the left part of memory. So, there's an intersection between the needed part and the part that we will write in, so writing will have a special technique. We also didn't dedicate a controller for this action, we made the 1by1 Conv controller handle it during the operation of 3by3 DWconv. This solution saves time, power and area.

5.4.5 Connections of Cores and Memories Block Diagram

When connecting the cores and memories we will find that the computation cores read from different memories depending on the layer of convolution as shown in Figure 5.35. For example, the 1by1 Conv core reads from Shuffle Memory through the Shuffling unit in the first layer of any stride 1 or stride 2 block except for the first stride 2 block it will read from Maxpool memory, and reads from X_Left memory in the middle layer of any stride 2 block, and reads from X_Right memory in the last layer of any stride 1 or stride 2 block.

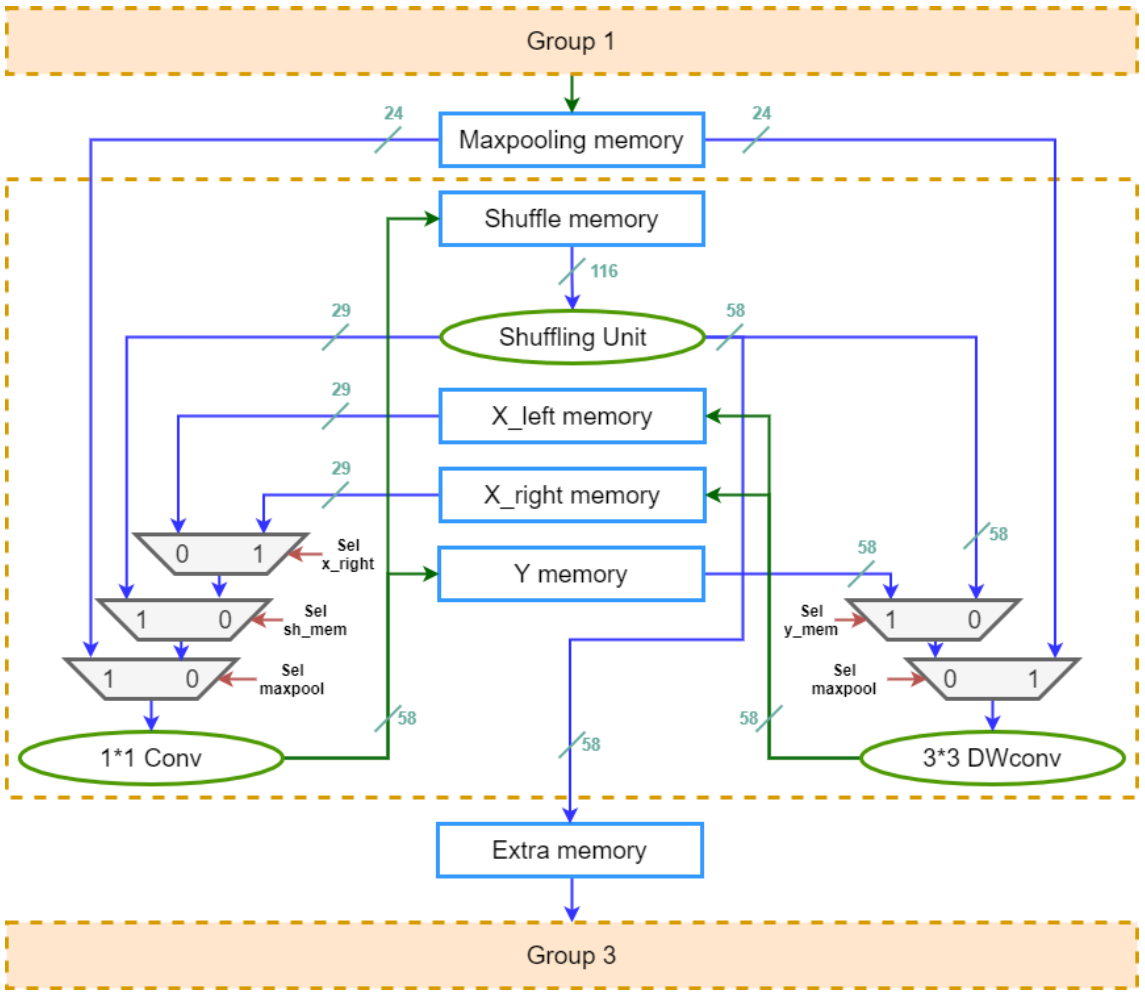


Figure 5.35 Connections of Cores and Memories Block Diagram

The select lines of these integration MUXs come from different controllers. For example, the sel_maxpool line comes from Group 2 controller and other selects come from either the 1by1 or 3by3 Controllers depending on the associated computation core. The used MUXs are cascaded instead of using one big MUX as we found this gives a better delay and to separate the control signals depending on the controllers' hierarchy.

In the end of Group 2, the data is taken from Shuffle memory and shuffled to the Extra memory to let Group 3 start its process immediately, so Group 2 can start using Shuffle memory again and this takes 342 clock cycles to be finished.

5.4.6 FIFO Architecture

As explained before in Group 1 design, a FIFO Register is used before the 3by3 DW convolutional layer to fetch the 9 elements window from the feature map that are going to be convolved, then with a simple shift in the FIFO register, the window is moved by one stride. The size of the FIFO should be $(2 * W + 3)$ where W is the size of the feature map.

In Group 2, we have 4 possible widths of the feature maps in group, so FIFO must support all the 4 widths (56, 28, 14, 7) or we should have four FIFOs, one for each width. Using multiple

FIFOs is inefficient as it will require a lot of registers. So, we will use one FIFO of the largest required size to be a one-fit-all FIFO. The size of the FIFO register will be $(2 * W_{max} + 3) = 119$ registers as shown in Figure 5.36.

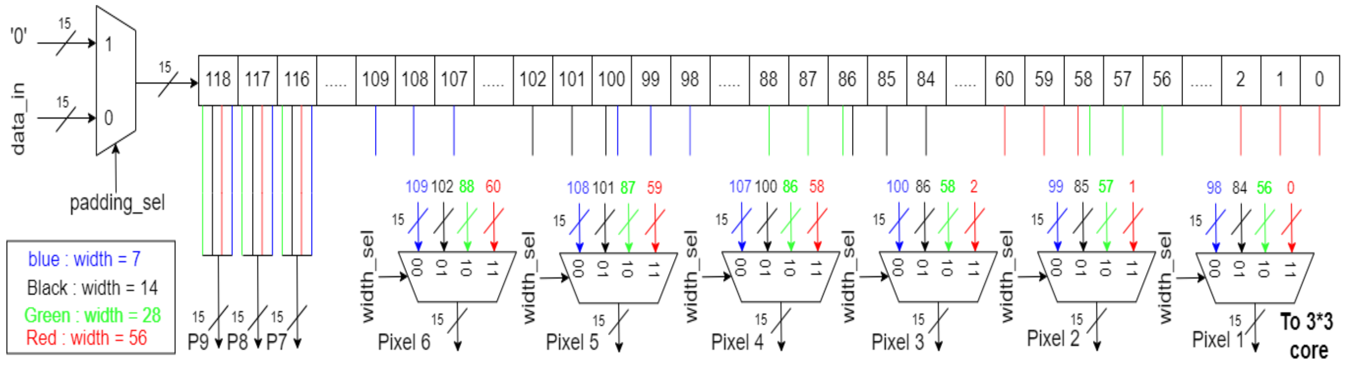


Figure 5.36 The FIFO register and the MUXs to select according to the width

The FIFO register, has input MUX to select zeros to add padding when required or input data from the feature map memory. It has output MUXs to select between outputs of different widths, where the first three outputs are always taken from the same registers, so we only need six 4:1 MUXs with select line coming from the FIFO controller according to the current width. The 9 outputs of the FIFO register go to the 3by3 DWconv computation core, that’s why we need 58 FIFOs to serve the parallelism used in the 3by3 DWconv.

5.4.7 Controllers

We have a hierarchy of 4 controllers for group 2, where each part of the group has its own control unit that ensures higher accuracy, modularity, easiness of the debugging and testing of each block as well as the whole design and to do any changes to any part of the group easily without affecting the other parts. Every controller unit is controlled by the main controller of the group. Thus, the highest level of control in group 2 is the Group 2 controller, then the second level has the 1by1 Conv controller and the 3by3 DWconv controller, and finally the FIFO controller is under the 3by3 DWconv controller. In this section we will discuss how these controllers are designed using finite state machines (FSMs) and the other logic used to do the control operation.

5.4.7.1 FIFO Controller

The FIFO controller controls the FIFO operation from fetching the data, determining the width_sel of the MUXs from the width, processing for stride 1 or stride 2, choosing between shifting data or padding zeros, reset the FIFO registers and reset the registers used in the 3by3 DWconv core. This controller interfaces between the FIFO and an upper controller which is the 3by3 DWconv controller. Figure 5.37 shows the block diagram of the FIFO controller and its inputs and outputs. The controller is implemented using a finite state machine and a simple logic to determine the width_sel from the width.

Block diagram of the FIFO controller

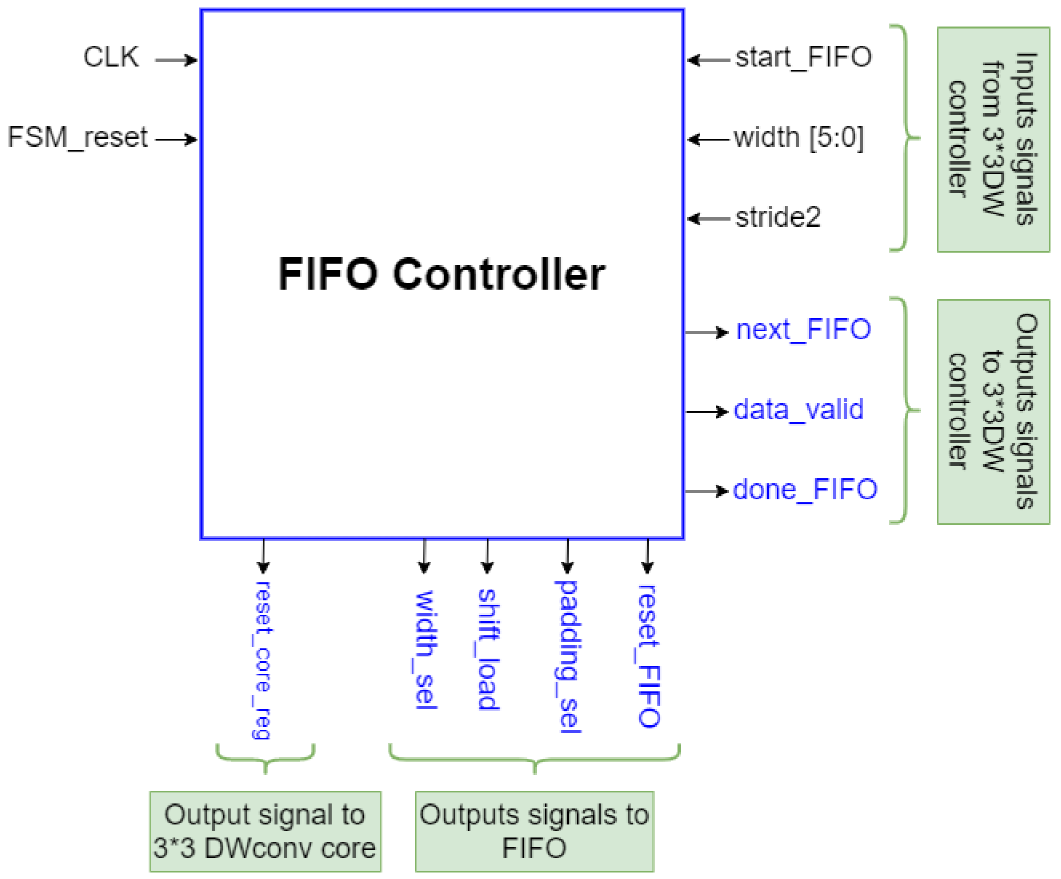


Figure 5.37 Block diagram of the FIFO controller

Width_sel logic

The width of FM in Group 2 starts from 56 and decreases gradually till 7 with possible width values {56, 28, 14, 7}. The FIFO output depends on the width, so the width_sel signal is generated in the FIFO controller and output to the FIFO using the logic shown in Figure 5.38.

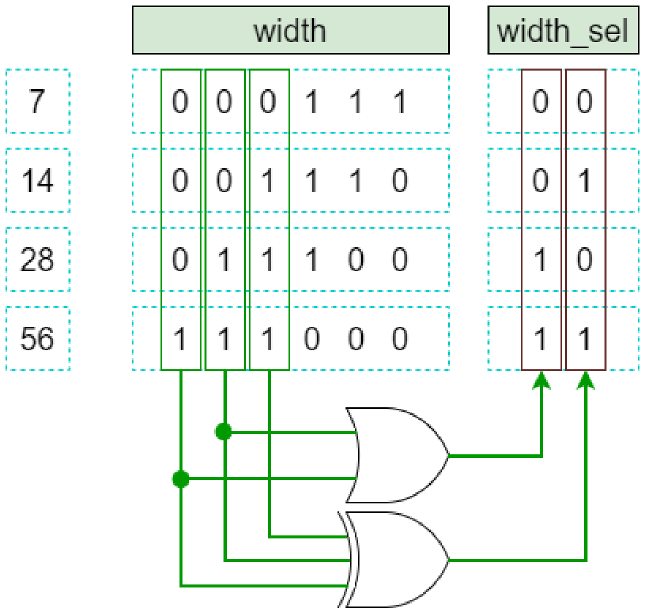


Figure 5.38 Width select logic

FSM state diagram of the FIFO controller

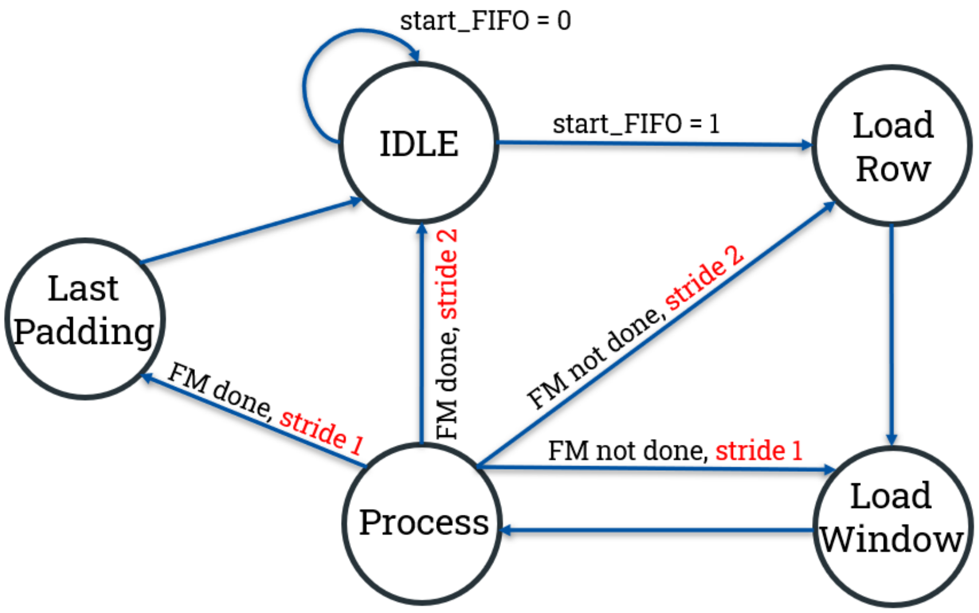


Figure 5.39 FSM state diagram of the FIFO controller

The FSM has 5 states as shown in Figure 5.39, it’s normally in the IDLE state till the start signal comes from the 3by3 DWconv controller to start the FIFO operation. It starts by shifting and loading the first row in the FM element by element. The row of padding is not loaded as the FIFO register is already reset in the IDLE state. Then loading 3 more elements to complete loading the window and finally processing that window and all the windows in that row till it finishes. Then continue scanning the feature map by Load Row if it’s a stride 2 layer or by Load Window if it’s a stride 1 layer. When the FM is all scanned except the last row of padding, if it’s stride 2, the last row is not taken, if it's stride 1, this padding row is processed in special state then it goes to IDLE. Figure 5.40 shows this flow where (1) is IDLE, (2) is Load Row, (3) is Load Window, and (4,5,6) are Processing with stride 2.

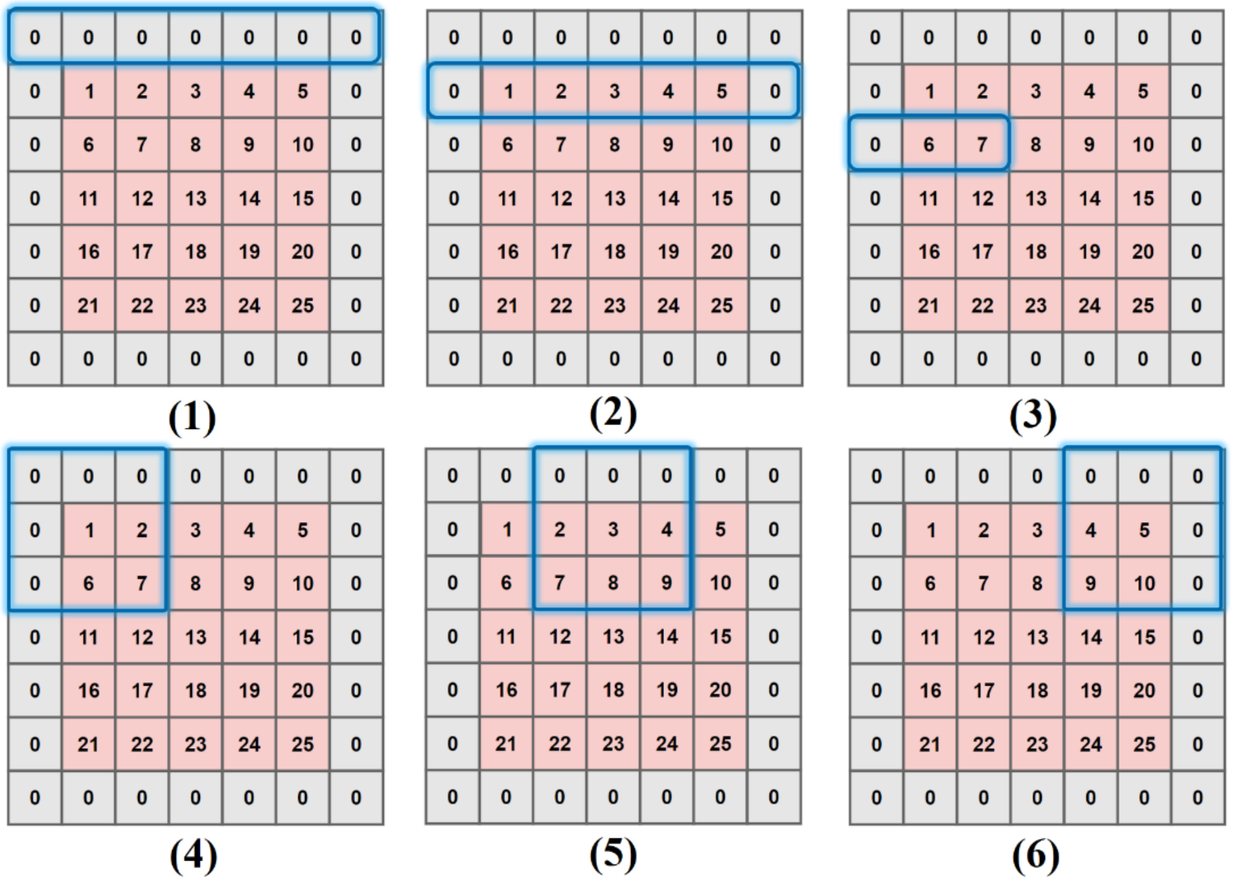


Figure 5.40 The flow of the FIFO FSM

<u>Counters used in the FIFO controller</u>

- **Row counter**: counts the rows of FM including padding.
- **Lr counter**: counts the elements in a row of FM including padding.
- **Lw counter**: counts the 3*3 windows.

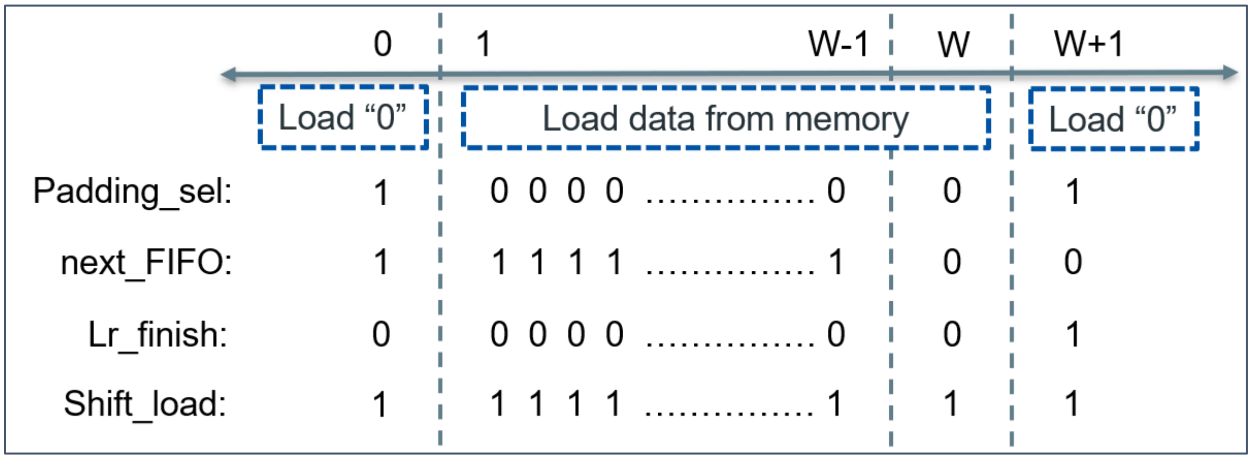
<u>Explanation of each state in the FIFO controller</u>

(1) IDLE state

- As long as start_FIFO is low, it keeps in the in the IDLE state.
- Inside the state:
    i. All output signals are low by default.
    ii. The FIFO register is reset to store zeros (first row of padding).
    iii. All counters are reset.
- When start _FIFO becomes high, it goes to Load Row state and increment the row counter.

(2) Load Row state

- The core registers are reset to save dynamic power, as the data_valid is always 0 in this state.
- Lr counter counts from 0 to W+1 with the control signals as shown in Figure 5.41.



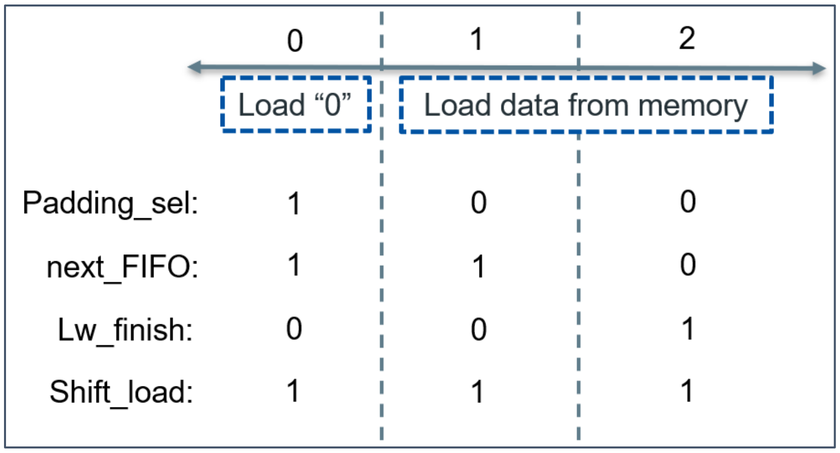
	0	1		W-1	W	W+1
	Load “0”	Load data from memory				Load “0”
Padding_sel:	1	0 0 0 0	....	0	0	1
next_FIFO:	1	1 1 1 1	....	1	0	0
Lr_finish:	0	0 0 0 0	....	0	0	1
Shift_load:	1	1 1 1 1	....	1	1	1

Figure 5.41 The control signals in the Load Row state

- When Lr counter reaches W+2, it goes to Load Window state, and increments the row counter.

(3) Load Window state

- Lw counter counts from 0 to 2 with the control signals as shown in Figure 5.42.
- When Lw counter reaches 3, it goes to Process state and the next_FIFO signal is raised to get data at beginning of process



	0	1	2
	Load “0”	Load data from memory	
Padding_sel:	1	0	0
next_FIFO:	1	1	0
Lw_finish:	0	0	1
Shift_load:	1	1	1

Figure 5.42 The control signals in the Load Window state

(4) Process state

- In the first window, the data_valid signal is high in both stride 1 and stride 2. Then, data_valid keeps high in stride 1 and toggles in stride 2.
- The Lr counter is reused in process state instead of using a new counter.
- Lr counter counts from 0 to W-2 with the control signals as shown in Figure 5.43.

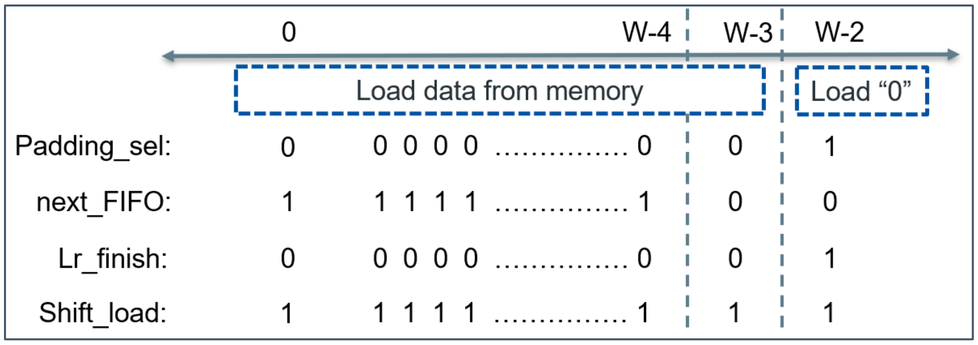

0
W-4
W-3
W-2
Load data from memory
Load "0"
Padding_sel: 0 0 0 0 0 .... 0 0 1
next_FIFO: 1 1 1 1 1 .... 1 0 0
Lr_finish: 0 0 0 0 0 .... 0 0 1
Shift_load: 1 1 1 1 1 .... 1 1 1

Figure 5.43 The control signals in the Process state

- When Lr counter reaches W-2, it goes to Load Row state if stride 2 or to Load Window state if stride 1 and increments the row counter.
- If it's stride 2, the row counter is checked in this state if it reached W, it goes to IDLE state and the done_FIFO signal is raised high to indicate that the FIFO operation is done in this feature map.
- If it's stride 1, it goes to Last padding state to finish processing the last row of padding in the feature map.

(5) Last Padding state

- This state happens in stride 1 only
- The Lr counter is reused in Last Padding state instead of using a new counter.
- Lr counter counts from 0 to W+1.
- Padding select is high all the time to load all zeros for the padding row.
- When Lr counter reaches W+2, it goes to IDLE state.
- The done_FIFO signal is raised high to indicate that the FIFO operation is done in this feature map.

5.4.7.2 3by3 DWconv Controller

The 3by3 DWconv controller controls the operation of the 3by3 DWconv core and its connected read and write memories by determining the write enable signals and addresses of the memories at the correct clock cycle to write the processed data either in the X_Left or X_Right memories and read from the Maxpool memory or the Shuffle memory or the Y memory by choosing between them using the MUX select signals. It also outputs the address of the filter and bias memory which is the same signal. It interfaces with the FIFO controller to tell it when to start its operation and tell it the current width and whether it's stride 1 or stride 2.

It also interfaces with the 1by1 Conv controller to organize the stages when only one of the 1by1 Conv and 3by3 DWconv are working solely. This is demonstrated using the done1_1 and the done3_3 signals to let the other entity know it can start.

It also interfaces with the Group 2 controller to know which layer of 3by3 DWconv is it and to know the width, number of channels and when to reset the address filter counter and whether this is the stage where the 3by3 DWconv reads from the Maxpool memory using the signal stage2_case24, and to organize the stages when both the 1by1 Conv and 3by3 DWconv are working simultaneously in parallel. This is demonstrated using the done3_3_G signals to let the Group2 controller know when both cores are done to start the next layer.

Figure 5.44 shows the block diagram of the 3by3 DWconv controller and its inputs and outputs. The controller is implemented using a finite state machine and a simple logic to generate the Start_FIFO pulse.

Block diagram of the 3by3 DWconv controller

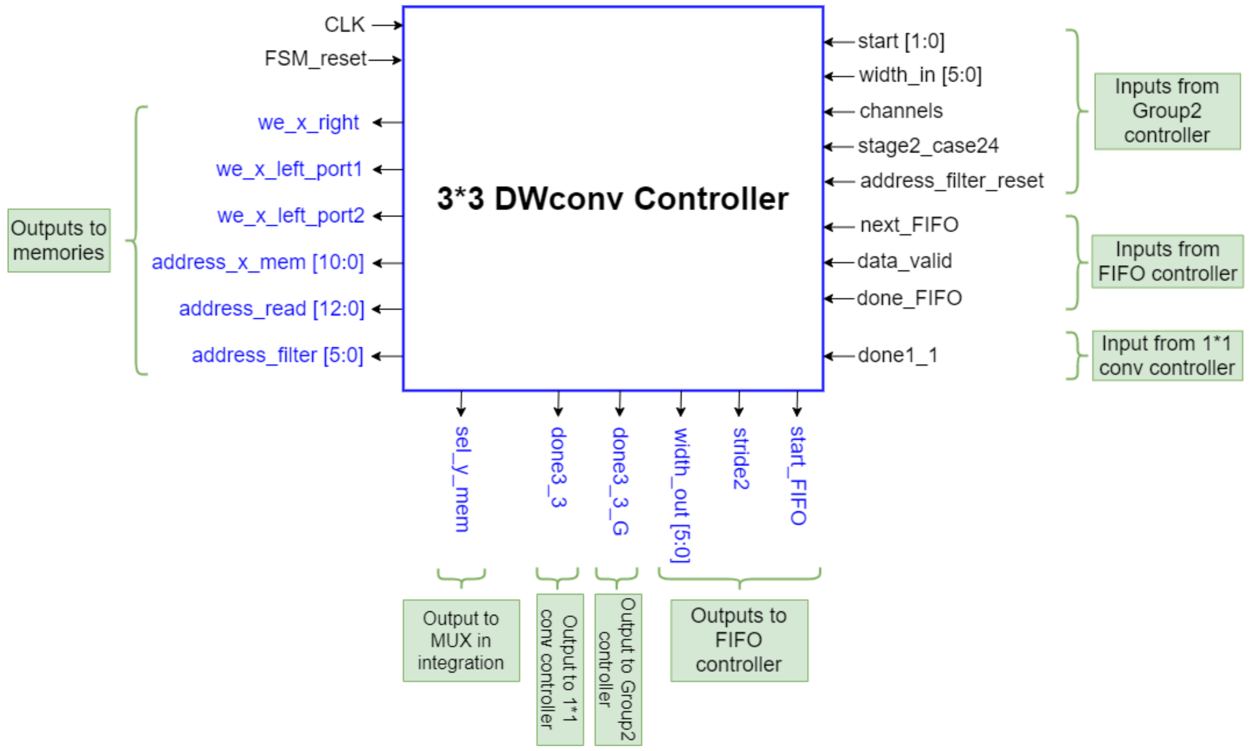


Figure 5.44 Block diagram of the 3by3 DWconv controller

Start_FIFO pulse generation logic

The Start signals are always in the form of a pulse. The Start FIFO signal is zero by default and the Start_FF signal is reset in the IDLE state, when the FIFO starts it sends a signal that will enable the flip flop and therefor the start signal will end. This simple circuit and the associated waveforms are shown in Figure 5.45.

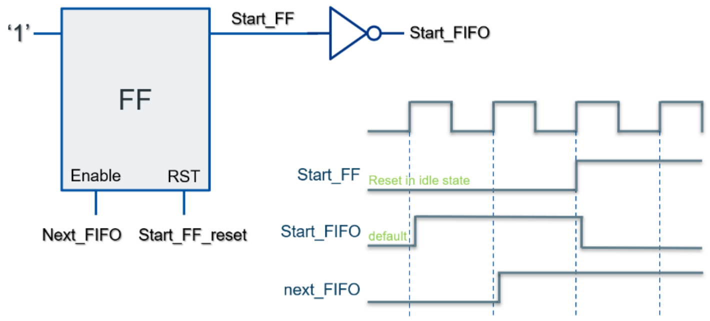


Figure 5.45 Start_FIFO pulse generation logic

FSM state diagram of the 3by3 DWconv controller

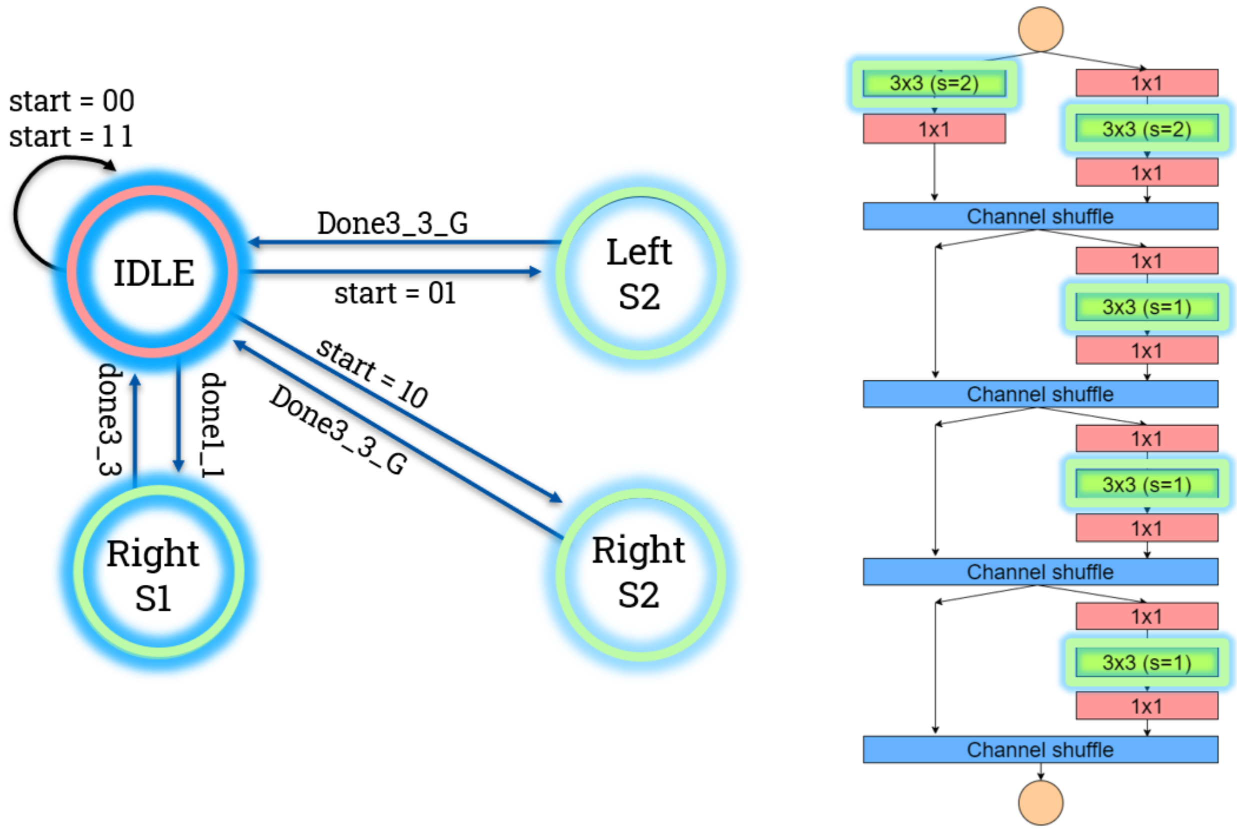


Figure 5.46 FSM state diagram of the 3by3 DWconv controller

The FSM has 4 states as shown in Figure 5.46, it's normally in the IDLE state till the start signal comes from the Group 2 controller to start the 3by3 DWconv operation. It starts by the first layer of 3by3 DWconv which is the Left_S2 then go back to IDLE state till the simultaneous layer of 1by1 Conv finishes, so the Group 2 controller sends a new start for the Right_S2 layer, when it finishes it goes back to IDLE state till the simultaneous layer of 1by1 Conv finishes and the 1by1 Conv starts its operation solely twice and send the done1_1 signal to start alternating between the 1by1 and 3by3 convolutions. One of them is working and the other in IDLE state till all the layers are done.

Counters used in the 3by3 DWconv controller

Figure 5.47 shows all the counters used in the 3by3 DWconv controller and shows that the width is just passed to the FIFO controller without any operations on it. It has four counters. The Channels counter counts the channels of pixel as multiples of 29 which is used in the initialization and jumping in other counters and to know when we start or finish a pixel. The Read Address counter is to generate the read address to read from the Maxpool memory or the Shuffle memory or the Y memory. The X_mem Address counter is used to generate the write address to write either in the X_Left or X_Right memories. Finally, the Filter address counter is used to generate the address of the filter and bias memory which is the same.

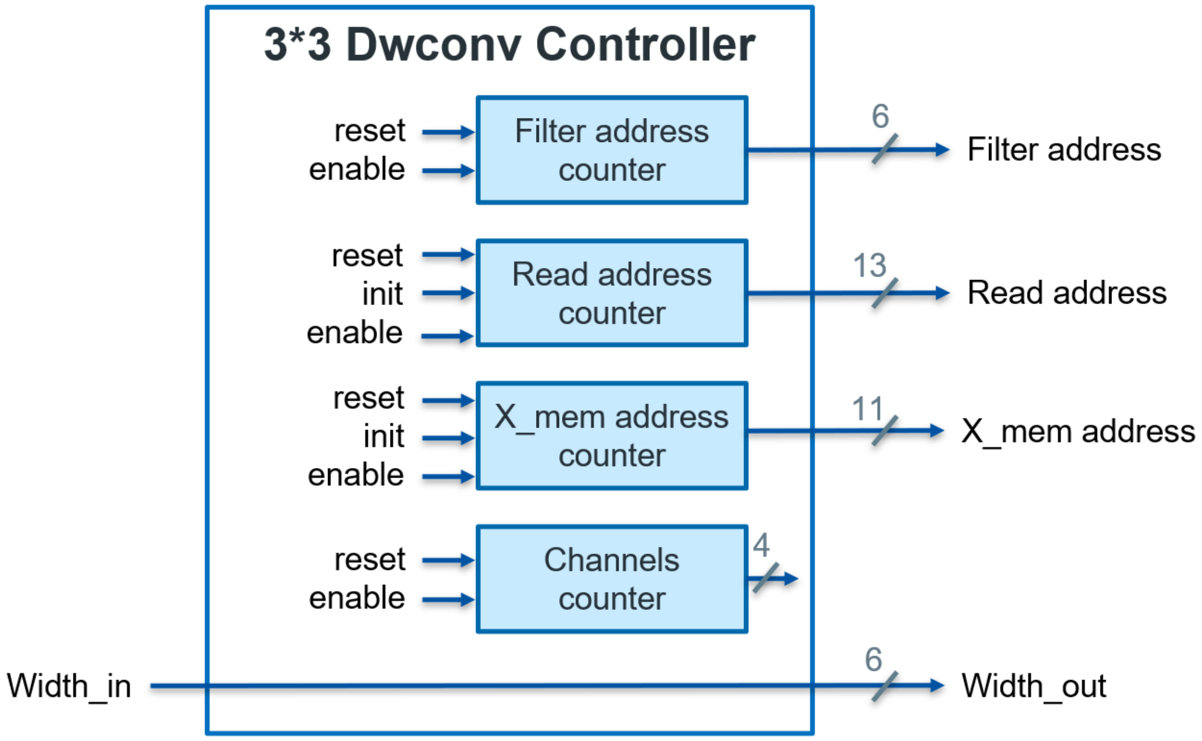


Figure 5.47 Counters used in the 3by3 DWconv controller

Explanation of each state in the 3by3 DWconv controller

(1) IDLE state

- As long as start signal is 00 or 11, it keeps in the in the IDLE state.
- Inside the state:

    i. All output signals are low by default.

    ii. All counters are reset, except filter address counter is reset when the Group 2 controller sends its reset signal.

- When start signal becomes 01, it goes to Left_S2 state, when it becomes 10, it goes to Right_S2 state and when don1_1 signal is high, it goes to Right_S1.

(2) Left_S2

- It's a stride 2 layer, so stride2 signal is high.
- Start_FIFO is sent as a pulse to the FIFO controller.
- Reading from Shuffle memory or Maxpool memory using the same read address which is incremented when next_FIFO is high.
- Writing in X_Left memory using write address which is incremented when data_valid is high. In case of reading from Maxpool memory, write enable is sent to port 1 only. Otherwise, write enable is sent to port 1 & port 2.

- When the FIFO is done, Channels counter and Filter address counter are incremented, the read and write counters are initialized to new channels, and the Start flip flop is reset to have a new start_FIFO pulse.
- When all channels are done, done3_3_G is sent to Group 2 controller, the read, write and channels counters are reset and it goes to IDLE state.

(3) Right_S2

- It’s a stride 2 layer, so stride2 signal is high.
- Start_FIFO is sent as a pulse to the FIFO controller.
- Reading from Y memory using the read address which is incremented when next_FIFO is high.
- Writing in X_Right memory using write address which is incremented when data_valid is high and write enable is sent.
- When the FIFO is done, Channels counter and Filter address counter are incremented, the read and write counters are initialized to new channels, and the Start flip flop is reset to have a new start_FIFO pulse.
- When all channels are done, done3_3_G is sent to Group 2 controller, the read, write and channels counters are reset and it goes to IDLE state.

(4) Right_S1

- It’s a stride 1 layer, so stride2 signal is low.
- Start_FIFO is sent as a pulse to the FIFO controller.
- Reading from Y memory using the read address which is incremented when next_FIFO is high.
- Writing in X_Right memory using write address which is incremented when data_valid is high and write enable is sent.
- When the FIFO is done, Channels counter and Filter address counter are incremented, the read and write counters are initialized to new channels, and the Start flip flop is reset to have a new start_FIFO pulse.
- When all channels are done, done3_3 is sent to the 1by1 Conv controller, the read, write and channels counters are reset and it goes to IDLE state.

5.4.7.3 1by1 Conv Controller

The 1by1 Conv controller controls the operation of the 1by1 Conv core and its connected read and write memories by determining the write enable signals and addresses of the memories at the correct clock cycle to write the processed data either in the Y memory or Shuffle memory and read from the Maxpool memory, the Shuffle memory, the X_Left or the X_Right memories by choosing between them using the MUX select signals. It also outputs the address of the filter and bias memory. It also manages writing in the Extra memory when group 2 finishes its operation. It generates the select signals for the MUXs in the Shuffling unit to pass the 29 input channels either from the first half or the second half of memory channels.

It also interfaces with the 3by3 DWconv controller to organize the stages when only one of the 1by1 Conv and 3by3 DWconv are working solely. This is demonstrated using the done1_1 and the done3_3 signals to let the other entity know it can start.

It also interfaces with the Group 2 controller to know which layer of 1by1 Conv is it and to know the width, number of channels and when to reset the address filter counter and whether this is the stage where the 1by1 Conv reads from the Maxpool memory using the signal stage2_case24, and to organize the stages when both the 1by1 Conv and 3by3 DWconv are working simultaneously in parallel. This is demonstrated using the done1_1_G signals to let the Group2 controller know when both cores are done to start the next layer.

Figure 5.48 shows the block diagram of the 1by1 Conv controller and its inputs and outputs. The controller is implemented using a finite state machine.

Block diagram of the 1by1 Conv controller

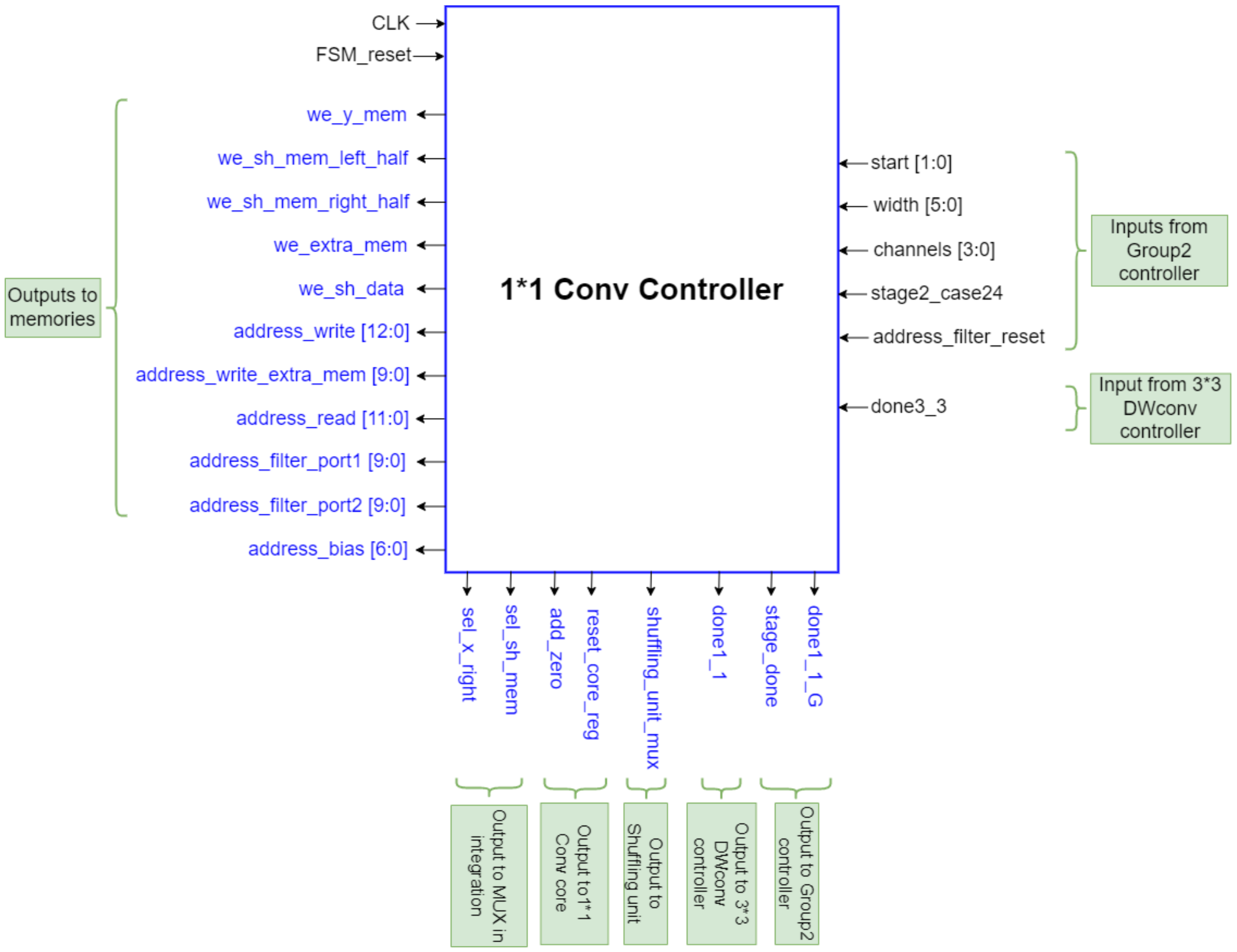


Figure 5.48 Block diagram of the 1by1 Conv controller

FSM state diagram of the 1by1 Conv controller

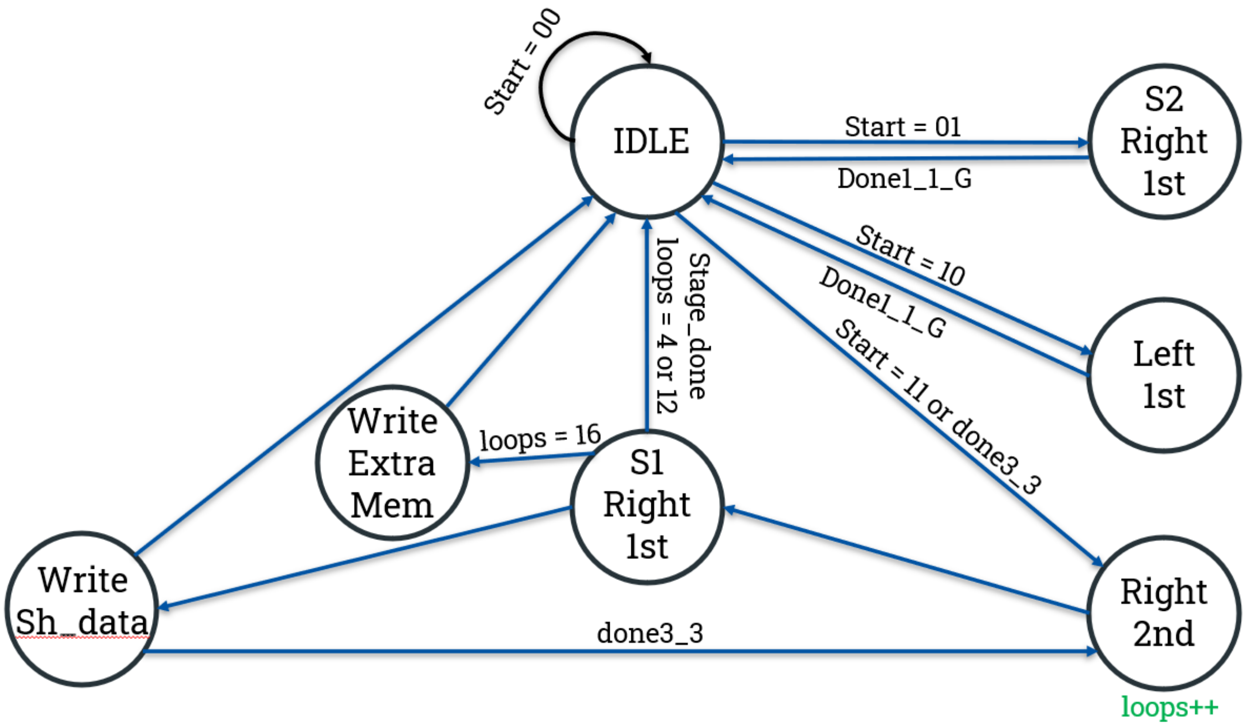


Figure 5.49 FSM state diagram of the 1by1 Conv controller

The FSM has 7 states as shown in Figure 5.49, it's normally in the IDLE state till the start signal comes from the Group 2 controller to start the 1by1 Conv operation. It starts by the first layer of 1by1 Conv which is the S2_Right_1st then goes back to IDLE state till the simultaneous layer of 3by3 DWconv finishes, so the Group 2 controller sends a new start for the Left_1st layer, when it finishes it goes back to IDLE state till the simultaneous layer of 3by3 DWconv finishes, so Group 2 controller sends a new start to the Right_2nd layer which when finished goes immediately to S1_Right_1st, then shuffled data is written in the Shuffle memory and when the 3by3 DWconv is done we go back to the Right_2nd layer. This sequence of S1_Right_1st, Write_Sh_data, Right_2nd is carried out till the stage is done at loops counter equals 4 or 12, so we go back to IDLE and start this whole stage sequence again. When all the 3 stages are done at loops counter = 16, this means all layers of Group 2 are done and the data will be written in Extra memory.

Counters used in the 1by1 Conv controller

Figure 5.50 shows all the counters used in the 1by1 Conv controller. It has 8 counters. The Channels counter counts the channels of pixel as multiples of 29 which is used in the initialization and jumping in other counters and to know when we start or finish a pixel. The Filter counter is used to count the filters of the 1by1 Conv layer as a multiple of 29 and it's incremented by 2 as we have 58 filters. The Read Address counter is to generate the read address to read from the Maxpool memory, the Shuffle memory, the X_Left or the X_Right memories.

The Write Address counter is used to generate the write address to write either in the Y memory or Shuffle memory. The Loops counter is used to count the stride 1 and stride 2 blocks, where stages are done at loops = 4, 12 and 16. The Bias address counter is used to generate the address of the bias memory. Finally, we have 2 ports for the Filter memory, the address of the second port is offset by channels compared to the first port.

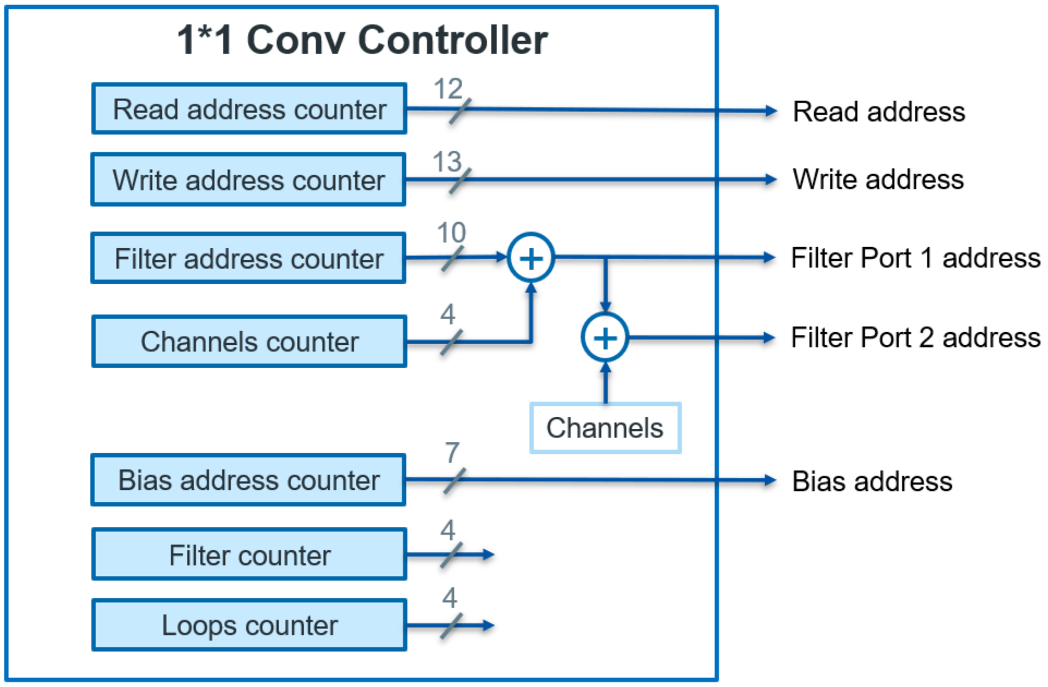


Figure 5.50 Counters used in the 1by1 Conv controller

<u>Explanation of each state in the 1by1 Conv controller</u>

(1) IDLE state

- As long as start signal is 00 or 11, it keeps in the in the IDLE state.
- Inside the state:

i. All output signals are low by default.

ii. All counters are reset, except filter address counter, bias address counter and loops counter that are reset when the Group 2 controller sends its reset signal.

- When start signal becomes 01, it goes to S2_Right_1st state, when it becomes 10, it goes to Left_1st state and when it becomes 11 or done3_3 signal is high, it goes to Right_2nd state.

(2) S2_Right_1st

- It's a stride 2 layer, so the shuffling unit MUX select signal will toggle every clock cycle except when reading from the Maxpool memory, the MUX selects the left half of shuffle memory.
- Reading from Shuffle memory except in the very first layer it reads from Maxpool memory using the same read address.
- Channels counter is incremented except if it's the very first S2_Right_1st layer.
- The add_zero signal going to the 1by1 Conv core is raised in the beginning of each pixel.
- Writing in the Y memory using write address.

- When all channels are done, done1_1_G is sent to Group 2 controller, the read, write and channels counters are reset and it goes to IDLE state.

(3) Left_1st

- Reading from X_Left memory using the read address.
- Channels counter is incremented every clock cycle.
- The add_zero signal going to the 1by1 Conv core is raised in the beginning of each pixel.
- Writing in the right half of Shuffle memory using write address.
- When all channels are done, done1_1_G is sent to Group 2 controller, the read, write and channels counters are reset and it goes to IDLE state.

(4) Right_2nd

- Reading from X_Right memory using the read address.
- Writing in X_Right memory using write address which is incremented when data_valid is high and write enable is sent.
- When all channels are done, write and channels counters are reset, read counter is initialized not reset as it always goes to S1_Right_1st state that skips the first half of channels.
- Loops counter is incremented as a stride block is done.

(5) S1_Right_1st

- It's a stride 1 layer, so the shuffling unit MUX select signal will toggle every clock cycle as we are reading from shuffle memory.
- Reading from Shuffle memory using the read address.
- Channels counter is incremented every clock cycle.
- The add_zero signal going to the 1by1 Conv core is raised in the beginning of each pixel.
- Writing in the Y memory using write address.
- When all channels are done, read and channels counters are reset and goes to the Write_Sh_data state.
- When a stage is done (at loops = 4 or 12), the next state will be IDLE and when all stages are done (at loops = 16), the next state will be Write_Extra_mem.

(6) Write_Sh_data

- Write enable for the shuffled data is raised and write enable for the left half of shuffle memory will toggle.
- When shuffling is done, if the simultaneous 3by3 DWconv Layer is also done, go to Right_2nd state. Otherwise, go to IDLE state and wait for the done3_3 signal to go to Right_2nd state.

(7) Write_Extra_mem

- The data is transferred from the Shuffle memory to the Extra memory through the Shuffling unit.
- The 1by1 Conv core is not used, so add_zero is high, core registers are reset
- When all data is transferred to the Extra memory, all counters are reset and it goes to IDLE state.

5.4.7.4 Group 2 Controller

The Group 2 controller controls the flow of group 2 layers between 1by1 Conv and 3by3 DWconv by sending them the Start signal, width, number of channels in the current layer and whether they are reading from the Maxpool memory or not. It also interfaces with the 1by1 Conv and 3by3 DWconv controllers to organize the stages when both the 1by1 Conv and 3by3 DWconv are working simultaneously in parallel. This is demonstrated using the done1_1_G and done3_3_G signals to let the Group2 controller know when both cores are done to start the next layer.

It also interfaces with the Accelerator controller to know when to start Group 2 operation after Group 1 finishes writing in the Maxpool memory and to let the Accelerator controller know when Group 2 is done writing in the Extra memory to start Group 3. A signal is sent to the Accelerator controller to let it know when reading from the Maxpool memory is done to allow Group 1 start again. Figure 5.51 shows the block diagram of the Group 2 controller and its inputs and outputs. The controller is implemented using a finite state machine.

Block diagram of the Group 2 controller

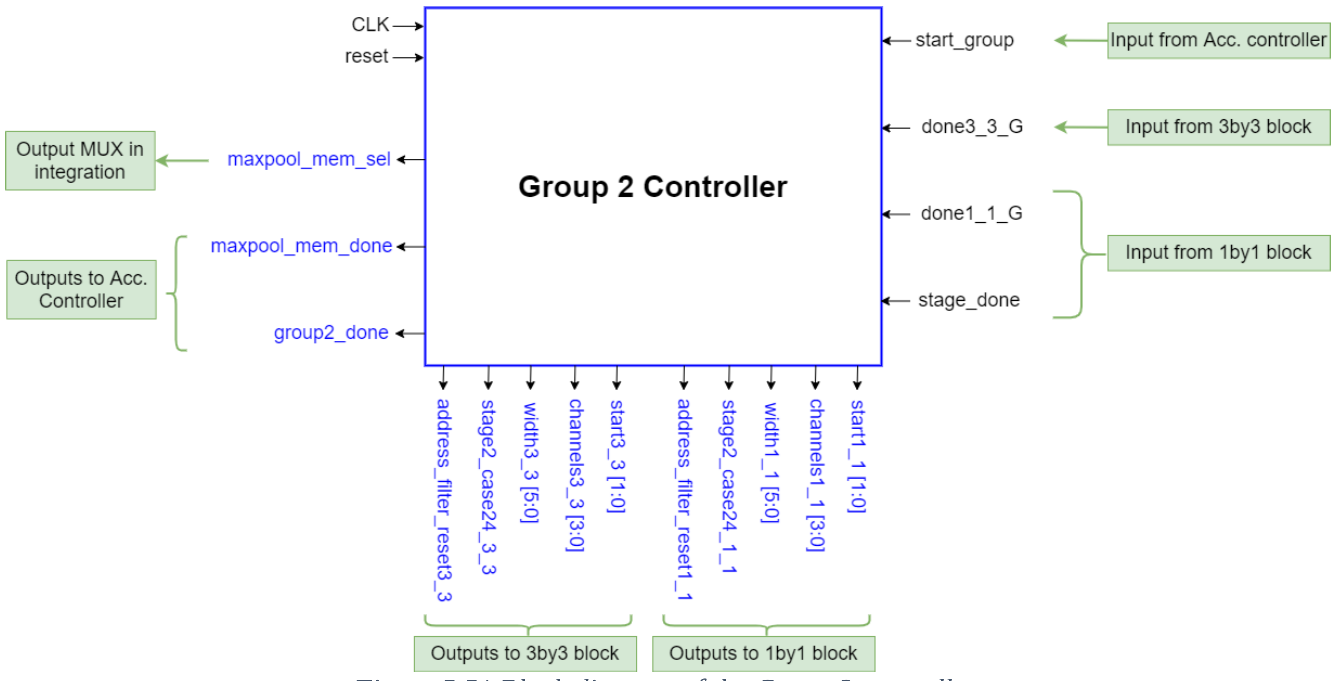


Figure 5.51 Block diagram of the Group 2 controller

FSM state diagram of the Group 2 controller

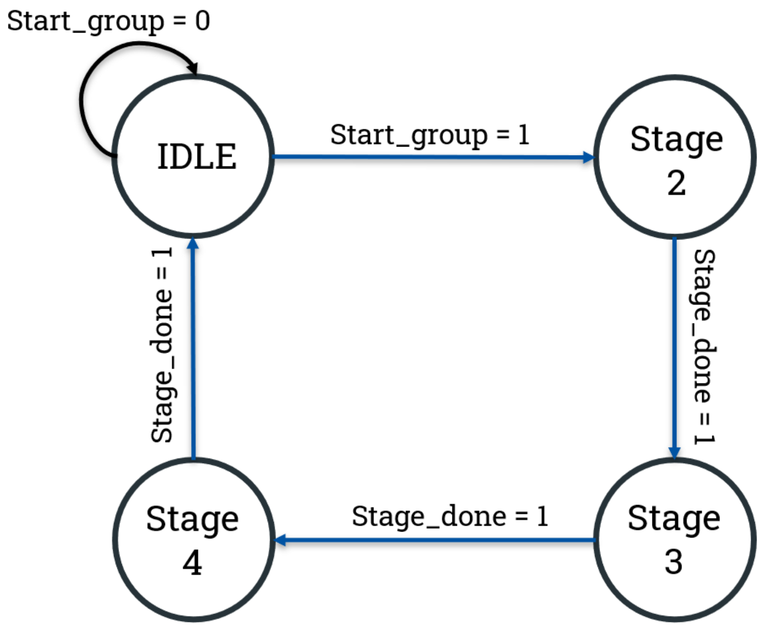


Figure 5.52 FSM state diagram of the Group 2 controller

The FSM has 4 states as shown in Figure 5.52, it's normally in the IDLE state till the start signal comes from the Accelerator controller to start Group 2 operation. It starts by Stage 2 which consists of a stride 2 block followed by three stride 1 blocks, then Stage 3 which consists of a stride 2 block followed by seven stride 1 blocks and finally Stage 4 which consists of a stride 2 block followed by three stride 1 blocks. Transition between stages happens when stage done signal becomes high, this signal is generated from the 1by1 Conv controller at the last layer of each stage.

Counters used in the Group 2 controller

Figure 5.53 shows all the only one counter used in the Group 2 controller which is the start counter that is incremented each time done3_3_G or done1_1_G comes from the lower-level controllers and used to send the start signal to both the 1by1 Conv controller and the 3by3 DWconv controller.

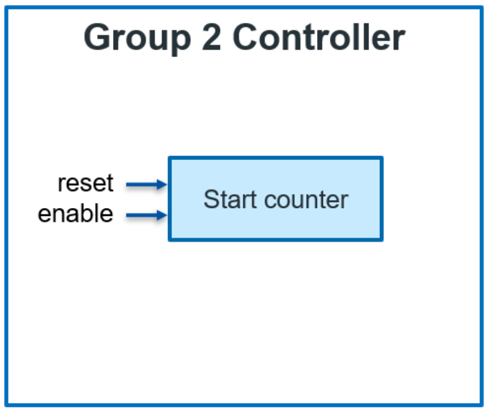


Figure 5.53 Counter used in the Group 2 controller

Explanation of each state in the Group 2 controller

(1) IDLE state

- As long as start signal is low, it keeps in the in the IDLE state.
- Inside the state:
  - i. Reset the address filter counter as all filters of all layers are in the same memory and the biases too.
  - ii. All counters are reset.
- When start signal becomes high, it goes to Stage 2 state.

(2) Any Stage states

All signals are initialized to the default values when the start_group signal comes, for example the widths are initialized to 56 and all other signals are initialized to zero. When the start signal become 01, both the 1by1 Conv and the 3by3 DWconv start together. One of them will finish first and send its done signal, let it be the 1by1 Conv. The Group 2 controller will wait till the other block finishes too and send its done signal, so the start signal will become 10 and the same operation happens again till the simultaneous blocks finish their operation. Finally, the 1by1 Conv will operate and when it finishes it sends the stage done signal indicating that the stride block is done. Figure 5.54 shows the waveforms associated with group 2 operation.

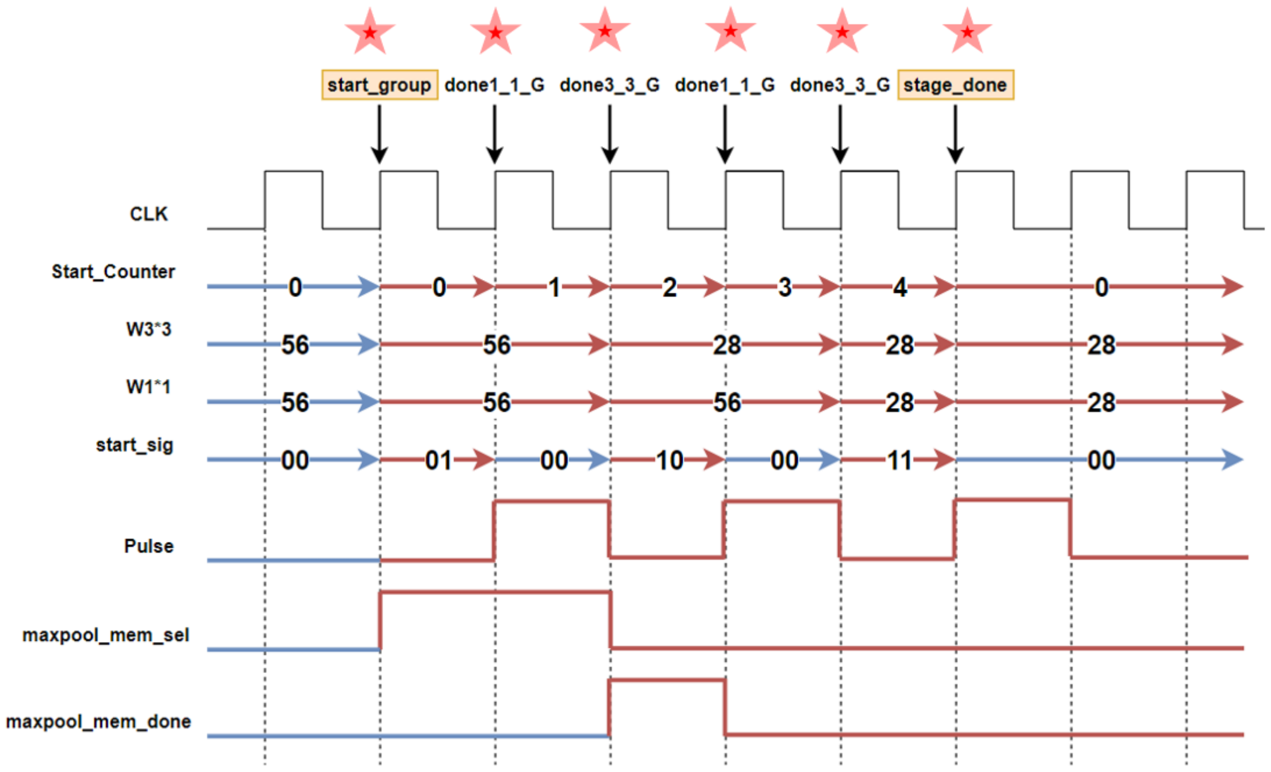


Figure 5.54 The waveforms associated with group 2 operation

5.5 Group 3 Design

In this section, we discuss the design of Group 3 in the architecture. Group 3 contains the last layers of ShuffleNet CNN; 1*1 convolution, average pool, and FC layers as well as a classification unit.

Group 3 takes the feature map (processed image by Group 2) stored in Extra memory as input and passes it through its units and finally outputs the estimated class of the image as shown in Figure 5.55.

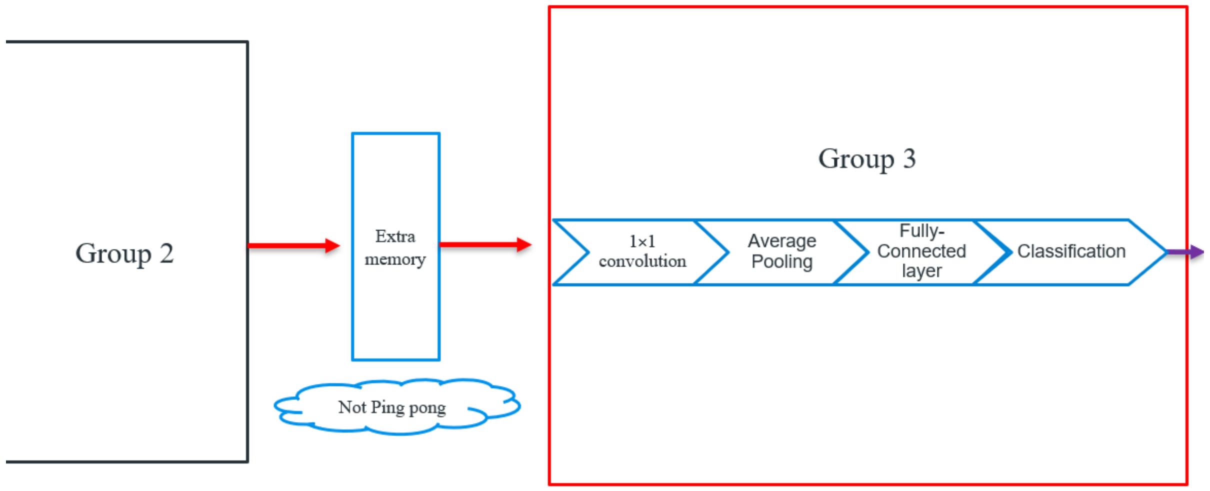


Figure 5.55 Layers in Group 3 and data transfers between Group 2 and Group 3 through Extra memory

The hierarchy of Group 3 consists of computational cores, feature map memories, filter and bias memories, and group controller. As shown in Figure 5.56, this hierarchical system can be structured like follows:

(1) 1by1 conv block
  a. 1by1 conv computational core
  b. 1by1 conv filter memories
  c. 1by1 conv bias memories
(2) Average-pool block
  a. Average-pool computational core
(3) Feature map storage (Register file)
(4) FC block
  a. FC computational core
  b. FC filter memories
  c. FC bias memories
(5) Classification unit.
(6) Group 3 controller

Throughout this section we will explain how these blocks are designed and assembled together.

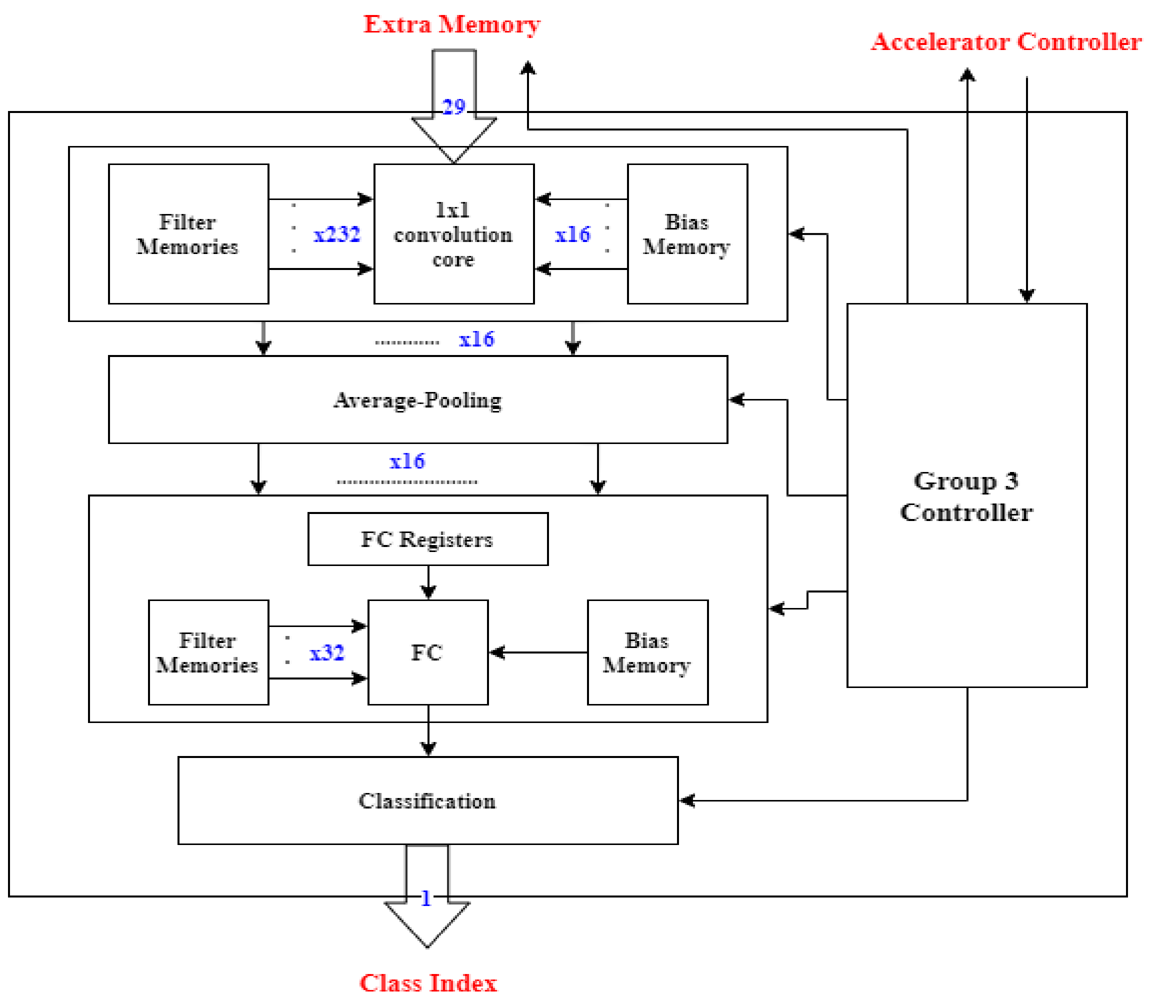


Figure 5.56 Block diagram of Group 3

5.5.1 Computational Cores

As we have mentioned in the design approach, we designed a computation core for each different type of layer. In Group 3, there is 3 different layers so we designed 3 computation cores, one for the 1*1 convolution layer, one for the average-pool layer, and one for the FC layer. In each of them we will discuss the parallelism used in filters, channels and window, The pipelining registers used to break the long timing path and how quantization is done to have a 15-bit fixed point output from the core without overflow.

5.5.1.1 1by1 Convolution

As mentioned in earlier chapters, 1*1 convolution convolves each pixel in a channel in the input feature map with its counterparts in other channels through multiplying each one of them by a weight from a certain filter, then adding the results to produce a pixel in the output feature map. In this layer we have a 7*7*464 input feature map and 1024 filters and it produces a 7*7*1024 output feature map. Based on our design approach, we use parallelism in computation cores to enhance the throughput. The parallelism used in this layer is as follows:

Parallelism

Table 5.9 Parallelism in 1by1 Convolution

Parallelism in Filters	Parallelism in Channels	Parallelism in window
16	29	NA

This means that 16 filters are executed in parallel for 29 channels of each filter then in the next cycle the next 29 channels are convolved and so on till all the channels are done. That's why an accumulator is placed to sum up all the products of all channels. Then it processes the next pixel with the same way and so on. So, we have 16 1by1 conv units in this layer, one for each filter.

Inputs

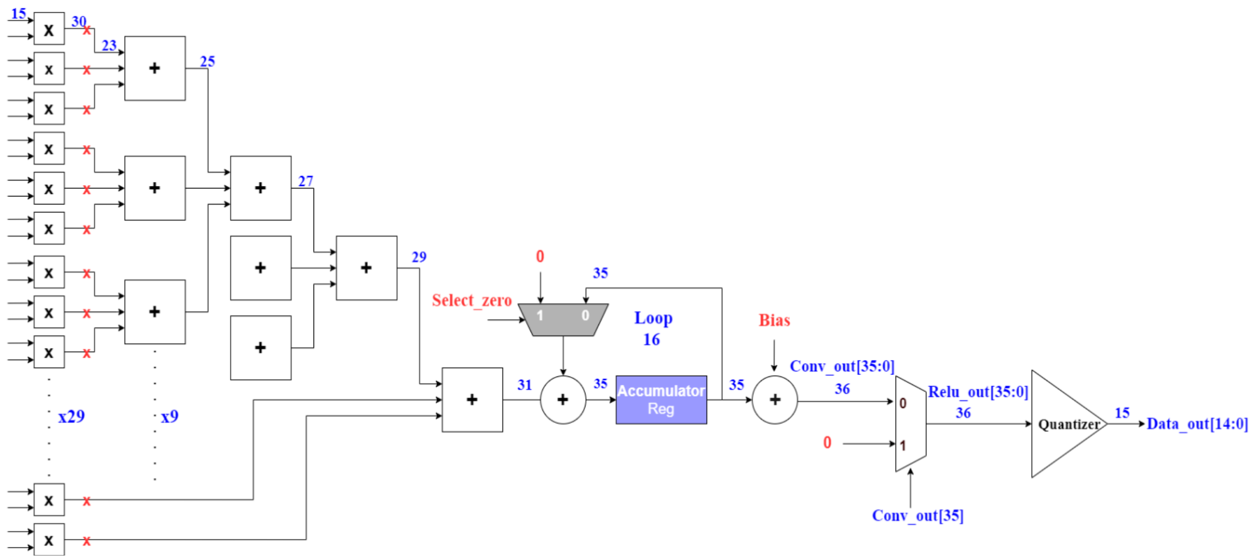


Figure 5.57 1by1 Conv Unit Block Diagram

The inputs to the 1*1 convolution core are as follows:

1. Data: 29 parallel inputs coming from a feature map memory (Etxtra_memory).
2. Weights: (16*) 29 parallel inputs from filter memory.
3. Bias: (16*) 1 input from bias memory.
4. Control signals like select_zero signal.

Output

The output of the 1*1 convolution core is (16*) 1 pixel and it goes direct to the average-pool layer without any intermediate storage between the two layers.

Methodology

The input 29 elements from the feature map memory (Extramemory) enters this layer and the operation is held by convolving it with 16 filters in parallel. The convolution operation is executed by multiplying each feature map pixel to the corresponding pixel in the filter by using DSPs, then adding the 29 products together using an adder tree, where the Adder3 block is used to add each 3 products. So, 27 products are added in a 3-level adder tree then the remaining two products are added to them using an extra 3-input adder level. Then accumulating the result to the result of the next 29 channels until all the channels are convolved. The accumulator loops 16 times, as feature maps and filters have 464 channels. In the feedback of the accumulator, we either add the accumulated value or add zero using a MUX. This is used to add zero instead of accumulation in the beginning of each pixel to save the clock cycle wasted to reset the accumulator register. After the accumulation of all channel convolution, the bias is added to the result.

A ReLU activation function follows the 1*1 convolution in this architecture. It rectifies the output to only positive values and force the negative values to be zero. As it is a simple operation, we designed it as a part of each 1by1 conv unit. The ReLU checks if the number is negative using the MSB of the convolution output instead of using comparators which lead to high power consumption and long logic delay. The pipelining registers are used after the multipliers to break the long timing path and to make sure the inputs to the adder tree are constant when not using them so as to save dynamic power.

Regarding the size of the internal signals through each 1by1 conv unit, multipliers double the number of bits. Thus, number of fraction bits increases by 8. No need for all these fraction bits so we drop 7 bits of them. It will not cause an error in the value; this is just a quantization noise. The 3-input adder increments the number of bits by 2 and the 2-input adder increments the number of bits by 1. Moreover, the accumulator increments the number of bits by $log_2(M)$, where M is the number of accumulations. Thus, the output of each unit is not 15-bits and overflow may occur. To make sure that the output is the correct 15-bits without overflow, a quantizer is placed at the end of each computation core to compare the output of convolution after the ReLU to the maximum possible value of the 15-bit fixed point output. The quantizer consists of a multiplexer and a simple logic for maximum checking. We check for the maximum value using a simple logic circuit as in the 1by1 conv in Group 2 instead of a comparator to save power and area. If the number is smaller than the maximum value, the number is passed as it is. Otherwise, the maximum value is passed (saturation case to avoid overflow). Checking for the minimum case is not required here as the ReLU output will never be a negative value.

5.5.1.2 Average Pooling

Average pooling layer takes the output feature map which is 7*7*1024 pixels from the 1*1 convolution layer and prepares it to go through the fully-connected layer which takes a vector of data so the average pooling layer flattens the input feature map to be a vector of data by getting the average of each input channel $\frac{(\sum_{j=1}^{49} P_{ij})}{49}$, where ‘i’ is the number of channels ranges from 1 to 1024; and ‘j’ is the number of pixels. We applied parallelism in this layer as follows:

Parallelism

Table 5.10 Parallelism in Average Pooling

Parallelism in Filters	Parallelism in Channels	Parallelism in window
NA	16	NA

There are no filters in the average pooling layer, and since the average pooling layer takes its inputs directly from the 1*1 convolution layer without an intermediate memory between the two layers and the 1*1 convolution layer outputs 16 pixels from different channels so we must have 16 average pooling units to process 16 channels in parallel. There is no parallelism in window which is 7*7, as each unit cannot take multiple pixels from a channel at the same time because each 1by1 conv unit outputs a pixel at a time. And we don't need to do that because it will make Group 3 finishes its process faster than other groups and since Group3 must wait other groups to finish, this will cause overhead and add extra resources in vain.

Inputs

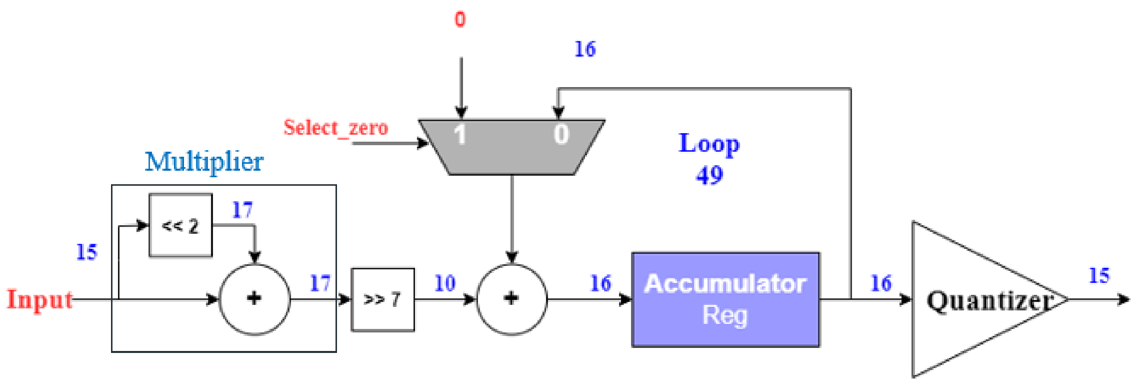


Figure 5.58 Average-pooling Block Diagram

The inputs to the average-pool core are as follows:

1. Data: 16 parallel inputs coming from the 1*1 convolution layer.
2. Control signals like accumulate_en and select_zero signals.

Output

The output of the average-pool core is (16*) 1 pixel (channel) and it goes to the register file between the average-pooling and FC layers.

Methodology

In average-pooling operation, there is a division by constant (49) operation, we can replace it by a multiplication operation (Sum*(1/49)).

We have two approaches to do the operation of the average-pooling. The first approach is accumulating all the 49 inputs (channel pixels) entering each unit one by one, then do the multiplication operation. The second approach is multiplying each input entering each unit by 1/49, then accumulating the result. The second approach is better than the first approach in terms of data widths as the output width of a multiplier is in order of 2*N, where N is the number of input bits, and the accumulator increments number of the bits by $log_2(M)$, where M is the number of accumulations. So, applying the multiplication operation first on the input data is better and we used it in our design.

As the multiplier is constant (1/49) which is 000000000000101 in a 15-bit fixed point representation with 8 bits fraction, no need for using a real multiplier, we can do this multiplication by a shift left and addition operations as shown in Figure 5.58. The reason for that can be shown in the example in Figure 5.59. Let’s take an example, 9-bit signed number, the fraction bits are 4 and the integer bits are 4.

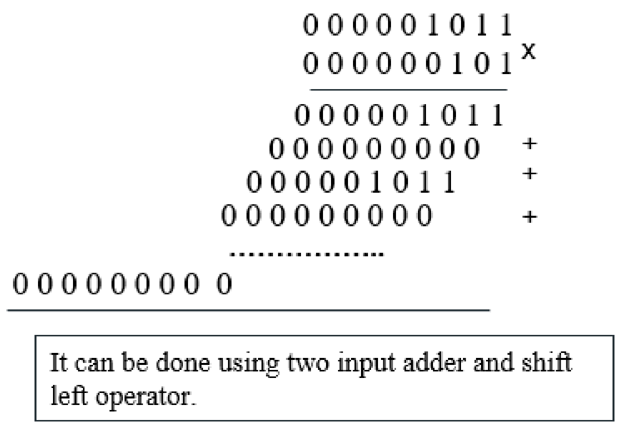


Figure 5.59 Multiplication by constant and how it is done using 1 shift left and a 2-input adder

Then accumulating the current result from the multiplier to the previous results of the input pixels from the same channel until all the pixels in a channel are accumulated. The accumulator loops 49 times to average all pixels in a channel. In the feedback of the accumulator, we either add the accumulated value or add zero using a MUX. This is used to add zero instead of accumulation in the beginning of each pixel to save the clock cycle wasted to reset the accumulator register. Since the average-pool layer takes its inputs directly from the 1*1 convolution layer and each 1by1 conv unit outputs a correct pixel each 16 clock cycles, the results of the average-pool accumulator must be sampled by the accumulator register each 16 clock cycles. This is done by activating the accumulate_en signal each 16 clock cycles as it is connected to the enable of the accumulator register.

Regarding the size of the internal signals through each average-pooling unit, the multiplier by the constant (1/49) increments the number of bits by 2. After the multiplication operation, number of fraction bits increases by 8. No need for all these fraction bits so we drop 7 bits of them. It will not cause an error in the value; this is just a quantization noise. The accumulator increments the number of bits by $log_2(M)$, where M is the number of accumulations. Thus, the output of each unit is not 15-bits and overflow may occur. To make sure the output is the correct 15-bits without overflow, a quantizer is placed at the end of each computation core to compare the output of convolution to the maximum and minimum possible values of the 15-bit fixed point output. The quantizer consists of two multiplexers, one for maximum checking and the other for minimum checking. Firstly, we check for the minimum value using a comparator and if the number is larger than the minimum value, the number is passed as it is. Otherwise, the minimum value is passed (saturation case to avoid overflow). Secondly, we check for the maximum value using a simple logic circuit instead of a comparator and if the number is smaller than the maximum value, the number is passed as it is. Otherwise, the maximum value is passed (saturation case to avoid overflow).

5.5.1.3 Fully-Connected

Fully-connected layer takes the flattened output of the average pooling layer (1024 pixels) stored in the register file between the two layers and passes them through a layer of 1000 neurons, each neuron takes the 1024 input pixels and calculates the probability of each class that it is the class of the input image. We can consider that the input data is a feature map has 1024 channels and each channel has one pixel. Also, the FC layer is like a 1*1 convolution has 1000 filters. The applied parallelism in this layer is as follows:

Parallelism

Table 5.11 Parallelism in Fully Connected

Parallelism in Filters	Parallelism in Channels	Parallelism in window
NA	32	NA

No parallelism in filters, we designed only one computation core for the FC layer, so it calculates the probability of classes one by one. In addition to no parallelism in window as each channel has one pixel. Therefore, the parallelism applied in the FC layer is in channels, it takes 32 channels in parallel from the 1024 channels in the register file. then in the next cycle the next 32 channels are processed and so on till all the channels are done. That's why an accumulator is placed as shown in Figure 5.60 to sum up all the products of all channels.

Inputs

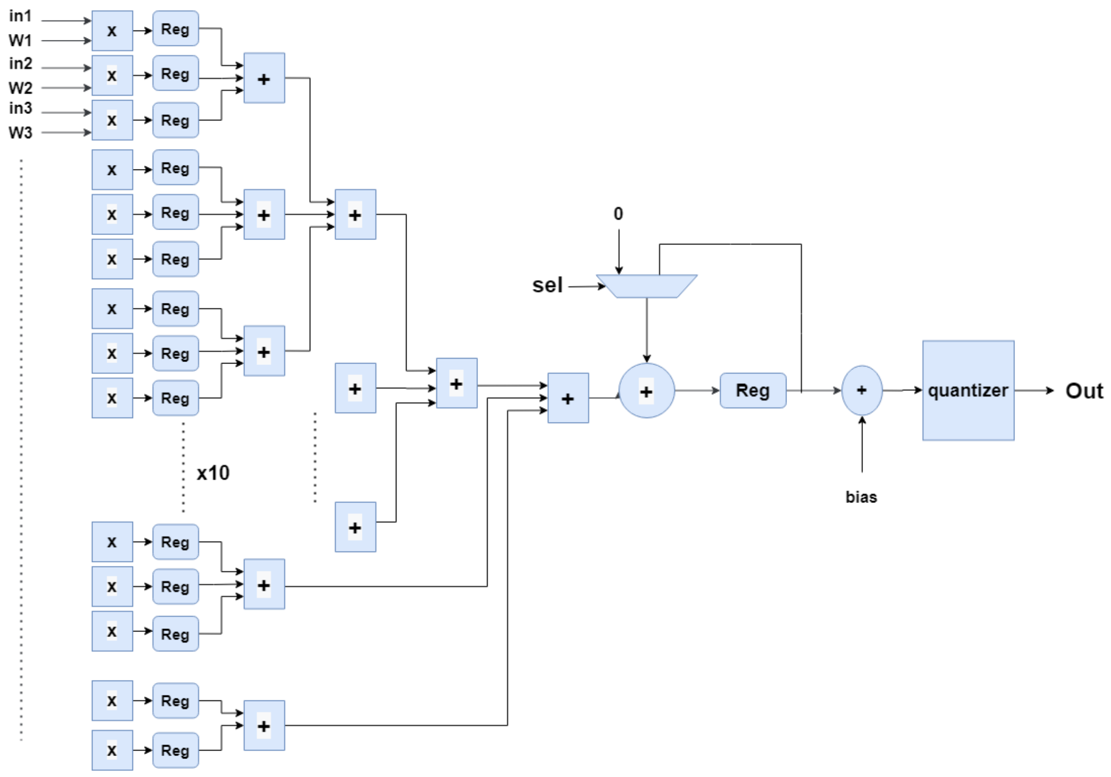


Figure 5.60 FC Block Diagram

The inputs to the FC core are as follows:

1. Data: 32 parallel inputs coming from a register file.
2. Weights: 32 parallel inputs from filter memory.
3. Bias: one input from bias memory.
4. Control signals like select_zero signal.

Output

The output of the FC core is one output represents the probability of each class and it goes directly to the classification unit.

Methodology

As shown in Figure 5.60, the 32 input pixels (channels) from the register file enter this computation core and the operation is held like a 1*1 convolution as it is executed by multiplying each pixel to the corresponding weight in the filter using DSPs, then adding the 32 products together using an adder tree, where the Adder3 block is used to add each 3 products. So, 30 products are added in a level of 3-Adder 3 blocks producing 10 results, 9 of them are added in a 2-level adder tree then the remaining two products and the last result of the first level of Adder 3 (the 10th result) are added together to them using an extra Adder 3 level. Then accumulating the current result to the previous results of the input pixels (channels) until all the results of the 1024 channels are accumulated. The accumulator loops 32 times. In the feedback of the accumulator, we either add the accumulated value or add zero using a MUX. This is used to add zero instead of accumulation in the beginning of each pixel to save the clock cycle wasted to reset the accumulator register. After the accumulation of all channel results, the bias is added to the stored value in the accumulator.

The pipelining registers are used after the multipliers to break the long timing path and to make sure the inputs to the adder tree are constant when not using them so as to save dynamic power.

Regarding the size of the internal signals through the FC core, multipliers double the number of bits. Thus, number of fraction bits increases by 8. No need for all these fraction bits so we drop 7 bits of them. It will not cause an error in the value; this is just a quantization noise. The 3-input adder increments the number of bits by 2 and the 2-input adder increments the number of bits by 1. Moreover, the accumulator increments the number of bits by $log_2(M)$, where M is the number of accumulations. Thus, the output of each unit is not 15-bits and overflow may occur. To make sure that the output is the correct 15-bits without overflow, a quantizer is placed at the end of the computation core to compare the output to the maximum possible value of the 15-bit fixed point output. The quantizer consists of two multiplexers, one for maximum checking and the other for minimum checking. Firstly, we check for the minimum value using a comparator and if the number is larger than the minimum value, the number is passed as it is. Otherwise, the minimum value is passed (saturation case to avoid overflow). Secondly, we check for the maximum value using a simple logic circuit instead of a comparator and if the number is smaller than the maximum value, the number is passed as it is. Otherwise, the maximum value is passed (saturation case to avoid overflow).

5.5.1.4 Classification Core

The last computation core in Group 3 and the whole accelerator. This computation core performs a different function from the function of convolution and pooling; it takes the outputs of the FC layer which are 1000 probabilities, one for each class, and compares them to get the top 1 probability and outputs the corresponding class which is the estimated class for the input image and the main output of the accelerator. There is no parallelism in this core, as the FC has one computation core and outputs one probability from the 1000 probabilities at a time.

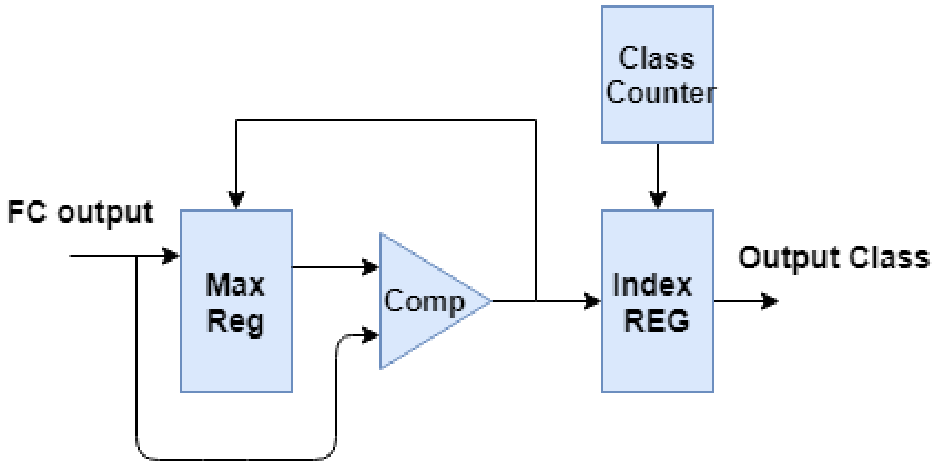


Figure 5.61 Classification Core Block Diagram

Inputs

The inputs to the FC core are as follows:

1. Data: one probability from the fully-connected layer at a time and the class counter value.
2. Control signals like max_reg_en signal.

Output

The output of the classification core is the estimated class for the input image and it is the final and main output of the whole accelerator.

Methodology

At the beginning of Group 3 operation, the Max Reg is set to the minimum value it can hold as well as the Index Reg is reset to 0. Then as shown in Figure 5.61, the output of the FC layer is compared with the stored value in the Max Reg using a comparator. The output of the comparator determines the maximum one of them and it is connected to the enable of the Index Reg. It is also ANDed with the max_reg_en, then the result is connected to the enable of the Max Reg. We defined the signal max_reg_en because the FC core outputs a correct class probability each 32 clock cycles so when the correct output from FC is ready (each 32 clock cycles), the signal max_reg_en is activated. If the input to this core is the maximum one, it will be stored in the Max Reg and the Index Reg will capture the value of the class counter. Otherwise, the Max Reg and the Index Reg keep their values. This operation is repeated till the probabilities of the 1000 classes are compared. Finally, the output class is the last stored value in the Index Reg.

5.5.2 Filter and Bias Memories

All constant parameters of the layers of Group 3 like filter weights and biases are stored in on-chip memories instead of off-chip memories to avoid their higher latency as our first priority is speed. Since, these parameters are constants, their memories will be ROMs (Read Only Memories). We used the BRAMs and LUTs of the FPGA to store the parameters. Weights and biases of Group 3 are in the 1*1 convolution layer and FC layer only. Average pooling layer doesn't have weights or biases.

In this section, the weight and bias memories of both the 1by1 convolution and FC will be explained including their organization, address size, number of ports, and number of memory instances.

*5.5.2.1 Convolution 1*1 Weight Memories*

In the 1*1 convolution layer, we have 1024 filters, each one has 464 channels, and each channel has a window of 1*1. So totally, we have 475,136 weights in this layer to be stored. The number of memory instances is indicated by the parallelism used in the 1by1 convolution core, which is 16 in filters and 29 in channels, so we need 29*16 weights simultaneously. We will use 29*8 = 232 memory instances of dual port BRAM memories to provide the 29*16 parallel weights. Each instance contains 2048 weights and each weight is represented in a 15-bit fixed point number. Therefore, each instance is stored in one BRAM (36kb), and the total BRAMs occupied for the 1*1 convolution weights is 232 BRAMs. The size of the address of the 1*1 filter memories is 11 bits ($log_2(2048)$).

<u>Filter memories organization</u>

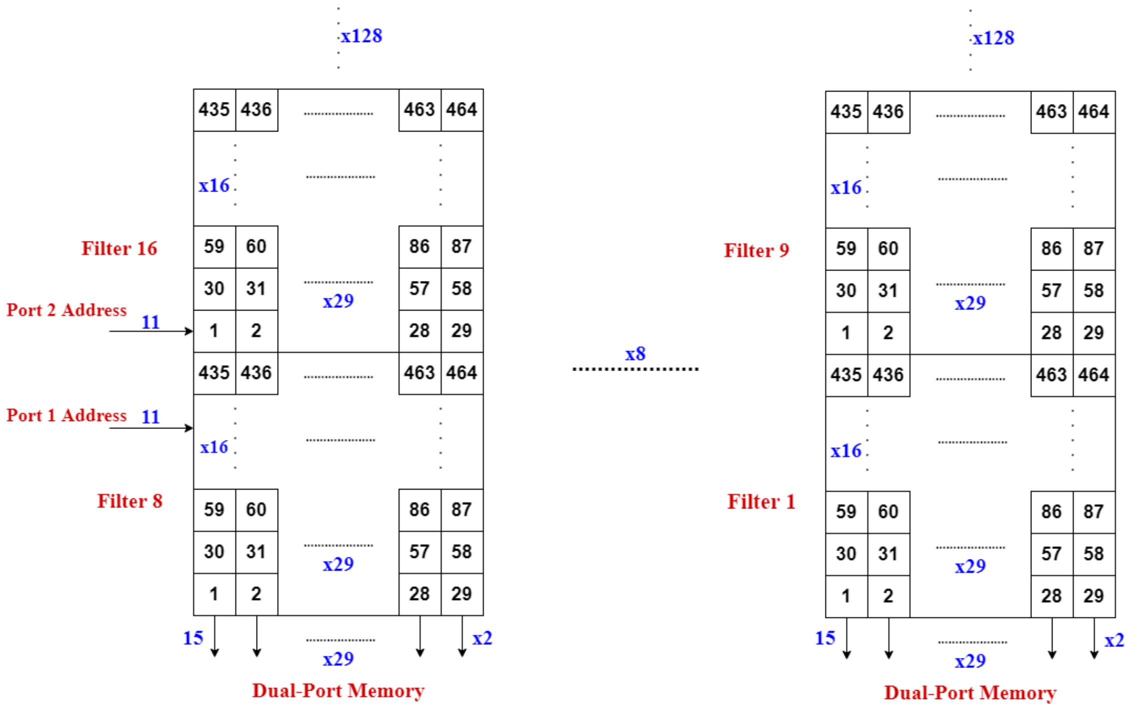


*Figure 5.62 1*1 Convolution Filter Memories Organization*

As shown in Figure 5.62, the weights of each filter (464 weights) are distributed on the 29 memory instances as the operation of 1*1 convolution needs 29 channels from the same filter in parallel. And it takes 16 rows (words) from each instance. The first eight filters (filters from 1 to 8) are stored in the first 16 rows of the memory instances (8*29), then the second eight filters (filters from 9 to 16) are stored in the next 16 rows and so on. Hence, after fetching each 29*16 weights from any 16 filters in parallel we just increment the filter memory address counter to get the next 29*16 weights from the same 16 filters till all channels of these filters are fetched, then start fetching the next 16 filters.

*5.5.2.2 Convolution 1*1 Bias Memories*

Each filter in a convolution has one bias, so totally the 1*1 convolution layer has 1024 biases. And since we have 16 parallel filters for the 1*1 Convolution core, we need the memory to provide 16 biases simultaneously. We will not just connect the biases to VDD and GND as we have a large number of biases, instead we will use 16 instances of distributed ROM memories to provide the 16 parallel biases. We will use LUTs instead of BRAMs as each instance is 64 biases only that will give a very poor utilization of the BRAMs. Thus, it's better to store them in LUTs. The size of the address of the 1*1 Bias memories is 6 bits ($log_2(64)$).

Bias memories organization

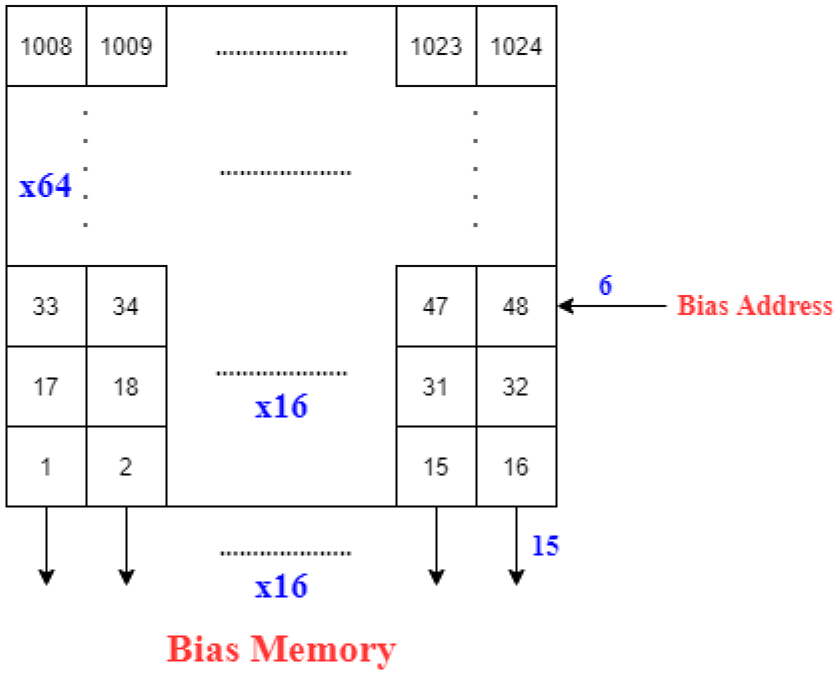


*Figure 5.63 1*1 Convolution Bias Memories Organization*

As shown in Figure 5.63, the biases of 1*1 convolution layer filters (1024 biases) are distributed on 16 memory instances as the operation of the 1*1 convolution need to take 16 biases in parallel for the 16 current applied filters in the computation units. Each instance has 64 rows (words). The first sixteen filter biases (filters from 1 to 16) are stored in the first row of the memory instances, then the second sixteen filter biases (filters from 17 to 32) are stored in the next row and so on. Hence, after finishing each 16 filters in parallel we just increment the bias memory address counter to get the next 16 biases.

5.5.2.3 Fully-Connected Weight Memories

In the fully-connected layer, we have 1000 filters, each one 1024 channels, and each channel has a window of 1*1, It's like a 1*1 convolution. So totally, we have 1,024,000 (~ 1 million) weights in this layer to be stored. It has almost half the parameters of the ShuffleNet CNN (~ 2.3 million parameters). The number of memory instances is indicated by the parallelism used in the FC core, which is 32 in channels, so we need 32 weights from the same filter simultaneously. Hence, 32 memory instances of single port BRAMs are used to provide the 32 parallel weights. Each instance contains 32,000 weights and each weight is represented in a 15-bit fixed point number. We used single port BRAMs instead of dual port BRAMs as number of BRAMs in both cases is the same. Moreover, we used the memory IP of Vivado design suit with its minimum area algorithm to generate the memory instances because it reduces number of BRAMs needed in case of the FC layer. Thus, each instance is stored in 14 BRAMs (36kb), and the total BRAMs occupied for the FC weights is 448 BRAMs. The size of the address of the FC filter memories is 15 bits ($ceil(log_2(32{,}000))$).

Filter memories organization

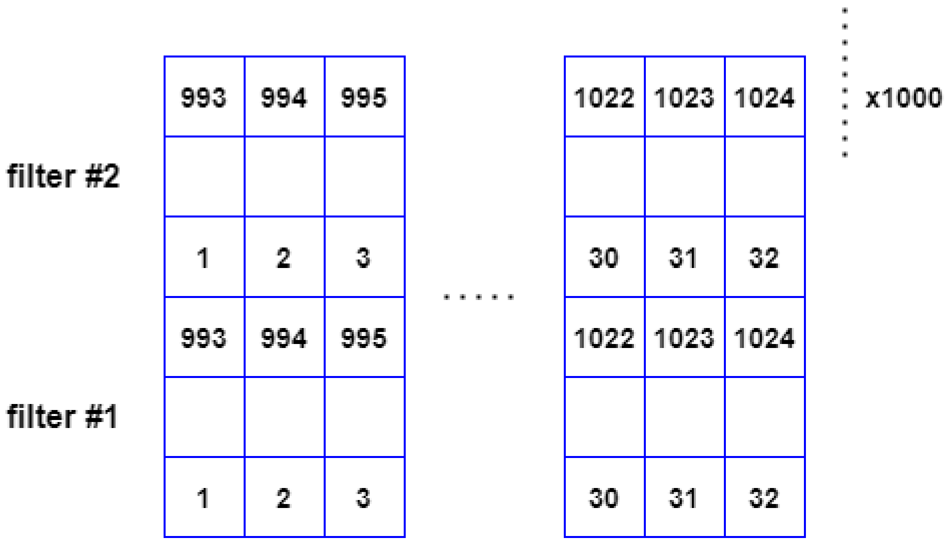


Figure 5.64 FC Filter Memories Organization

As shown in Figure 5.64, the weights of each filter (1024 weights) are distributed on the 32 memory instances as the operation of FC needs 32 channels from the same filter in parallel. And it takes 32 rows (words) from each instance. The first filter is stored in the first 32 rows of the memory instances, then the second filter is stored in the next 32 rows and so on. Hence, after fetching each 32 weights from a filter in parallel we just increment the filter memory address counter to get the next 32 weights from the same filter till all channels of the filter are fetched, then start fetching the next filter.

5.5.2.4 Fully-Connected Bias Memories

Each filter in the FC layer has one bias, so totally the FC layer has 1000 biases. And since the FC core operates on one filter at a time, we need the memory also to provide one bias at a time. Thus, one memory instance is used to store all these biases. Actually, no need for implementing the bias memory using LUTs or BRAMs as this layer has few biases and they are constants. Simply, these biases will be just connected to VDD and GND and bits of the memory address. The size of the address of the FC Bias memory is 10 bits ($ceil(log_2(1000))$).

Bias memory organization

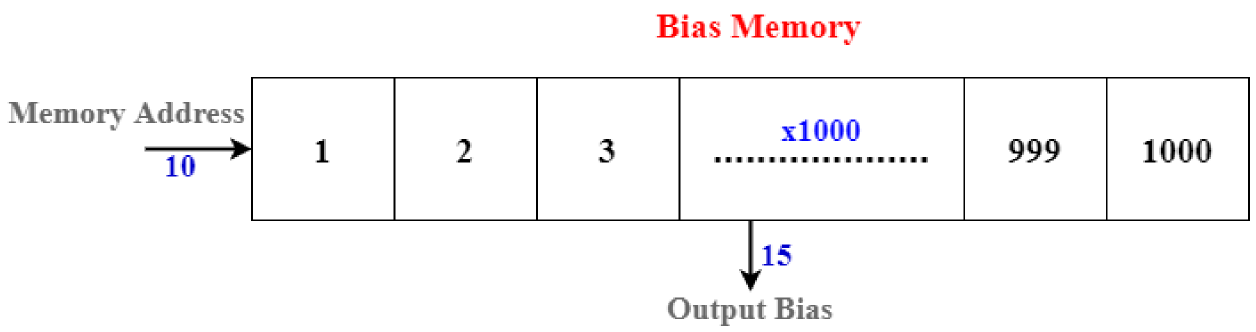


Figure 5.65 FC Bias Memory Organization

As shown in Figure 5.65, the biases of FC layer filters (1000 biases) are stored in one memory instance as the operation of the FC layer needs to take one bias at a time for the current applied filter in the computation unit. Thus, the memory instance has 1000 rows (words). After finishing each filter in the computation core, we just increment the bias memory address counter to get the next filter bias and so on.

5.5.3 Feature Map storage

In Group 3, no intermediate storage between the 1*1 convolution layer and average pooling layer while there is an intermediate storage between the average pooling layer and fully-connected layer to store all the 1*1*1024 feature map first before passing them to the FC layer. This intermediate storage is a register file has 1024 registers and each one is a 15-bit width register, we used register file instead of memories because the average pooling layer writes in this storage more than two words at a time and the FC layer reads more than two words at a time and BRAMs has only two ports at maximum. If we used multiple memory instances, it would exhibit very poor BRAM utilization. So, it is better to use registers. The register file is a special type of register files as it is designed to perform a specific writing and reading methodology based on the parallelism in average pooling and FC layers; it has inputs and outputs as follows:

Inputs

The inputs to the register file are as follows:

1. Data: 16 parallel pixels (words) coming from the average pooling layer.
2. Control signals like shift and sel_channels signals.

Output

The outputs of the register file are 32 parallel pixels (words) going to the fully-connected layer.

Writing Methodology

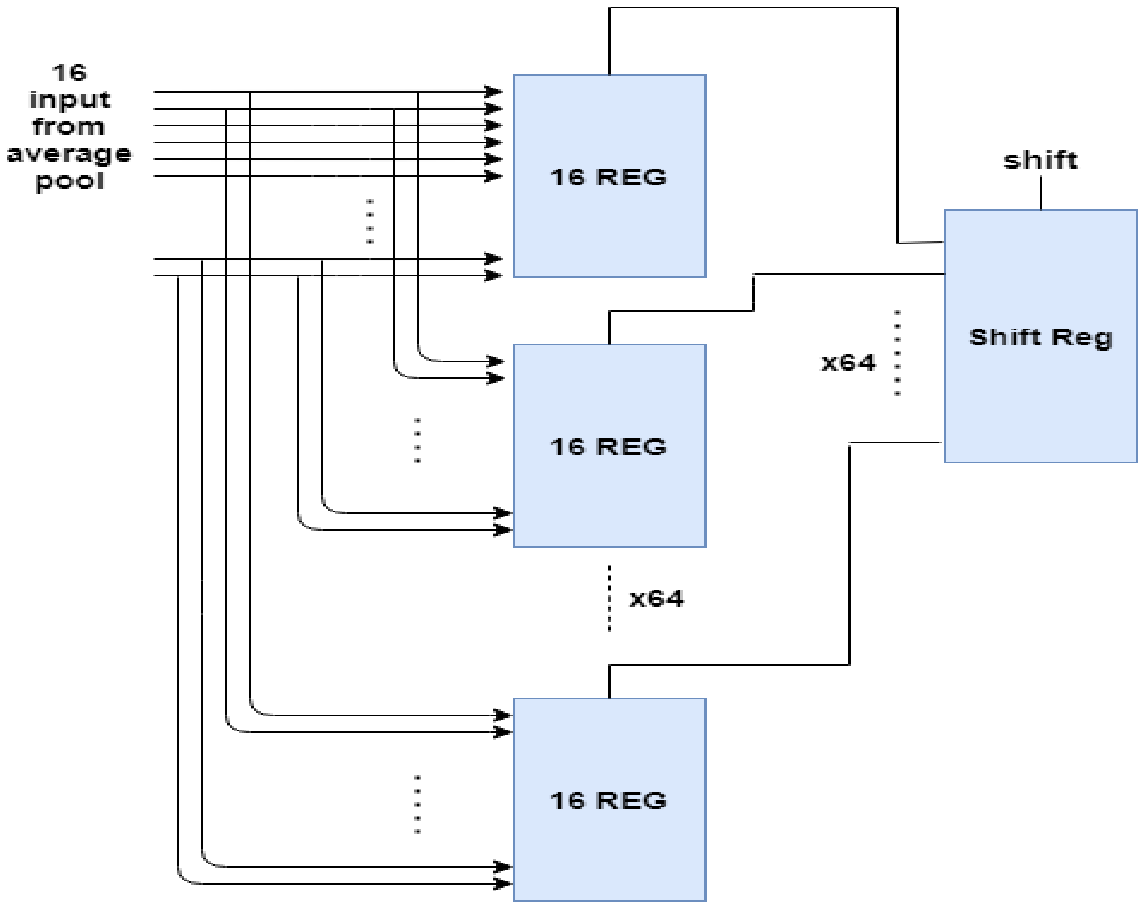


Figure 5.66 Register File Writing structure

Average pooling layer writes 16 parallel words at a time in the register file. So, in writing we consider the 1024 registers are divided into 64 register banks and each bank contains 16 registers. First sixteen words are stored in the first register bank and the second sixteen words are stored in the second register bank and so on. As shown in Figure 5.66, the sixteen parallel words are connected to inputs of all banks. The required register bank to store words are selected by activating the enable signals of its registers while other banks are deactivated. Enable signals are generated from a shift register of 64-bit width where enable signals of the first bank are connected to the first bit (LSB) in the shift register and those of the second bank are connected to the second bit and so on. This shift register has only one bit activated (high) at a time, and the other bits are low. After the correct words are written in the register file (as the average pool generates correct words each 49*16 = 784 clock cycles), Group 3 controller activates the shift signal to shift the active bit of the shift register to the next bit. Hence, the next register bank is ready to capture the next correct words from the average pooling layer.

Reading Methodology

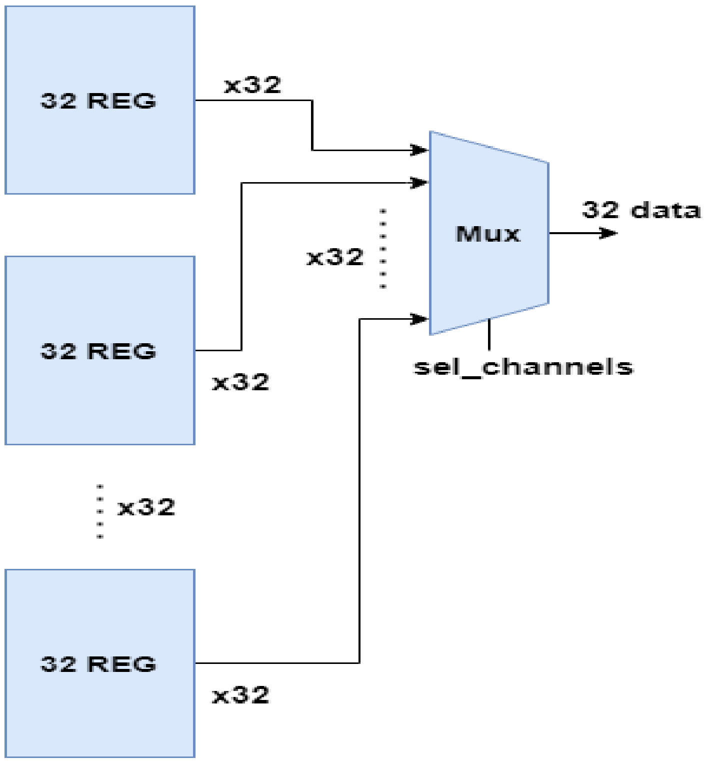


Figure 5.67 Register File Reading Structure

Fully-connected layer reads 32 parallel words at a time from the register file. So, in reading we consider the 1024 registers are divided into 32 register banks and each bank contains 32 registers. First thirty-two words are read from the first register bank and the second thirty-two words are read from the second register bank and so on. As shown in Figure 5.67, outputs of the register banks are passed through 32:1 MUXs to select the required bank to read from. After the FC reads each thirty-two words from the register file, Group 3 controller increments the sel_channels signal to select the words of the next register bank.

5.5.4 Group 3 Controller

We use the Controller–Datapath architecture in our design approach. In the previous subsections, we explained the Datapath of Group 3 and in this subsection, we will explain the control part of Group 3. As mentioned in the beginning of this chapter, we have distributed controllers so Group 3 has its own controller. It manages the sequence of operations, data flow in Group 3, however it works under the supervision of the accelerator controller. Group 3 controller manages 5 big operations; 1*1 convolution, average-pooling, register file, FC and classification unit. And it has inputs and outputs as shown in Figure 5.68:

Inputs

The inputs to Group 3 controller are as follows:

Control signals from accelerator controller: group3_start, reset.

Output

The outputs of the Group 3 controller are as follows:

1. **Control Signals for Extra memory:**
   - extramem_read_address
2. **Control Signals for 1*1 Convolution:**
   - conv1by1_filter_address1
   - conv1by1_filter_address2
   - conv1by1_bias_address
   - conv1by1_sel _zero
3. **Control Signals for Average Pooling:**
   - avgpool_acc_en
   - avgpool_sel _zero
4. **Control Signals for Register File:**
   - shift
   - sel_channels
5. **Control Signals for Fully-Connected:**
   - fc_filter_address
   - fc_bias_address
   - fc_sel _zero
6. **Control Signals for Classification Unit:**
   - reset
   - class_count
   - max_reg_enable
7. **Status signal for the Accelerator Controller:**
   - group_done

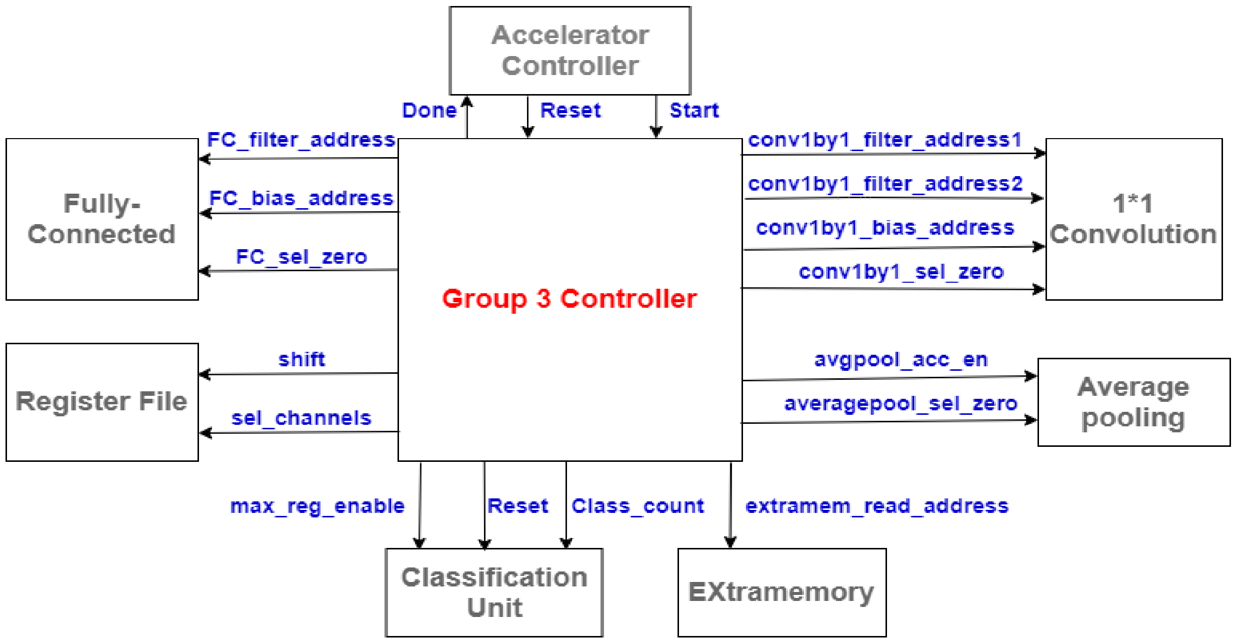


Figure 5.68 Shows the input and output signals of Group 3 and its interaction with other blocks

The function of each control signal of them are illustrated in the computation cores and memories subsections unless the control signals go outside Group 3, they will be illustrated here in the methodology.

Methodology

All the operations in Group 3 can be managed using counters and simple logic so no need for design FSM. This can be seen from the sequence of the operations in Group 3 that must be handled by the controller, it is as follows:

Group 3 waits a start signal from the accelerator controller to start processing the feature map stored in the Extra memory so it then sends a read address to the Extra memory to fetch words from it each clock cycle. These words are processed by the computation core of the 1*1 convolution layer as illustrated before. Weights from filter memories are fetched each clock cycle. Moreover, the conv1by1_sel_zero signal must be activated after a clock cycle from the first parallel words in any filter in the 1*1 convolution enter the computation core. This 1 clock cycle delay because of the pipeline in the computation core Each 784 (16*49) clock cycles, the conv1by1_bias_address is incremented to fetch new filter bias. Due to the pipeline in the core, bias must be delayed by 2 clock cycles from the data.

While the 1*1 convolution works, the computation core of the average pooling works on the output of the 1*1 convolution simultaneously. Thus, each 16 clock cycles, avgpool_acc_en signal is activated to allow the accumulator register in the average pooling core capture the correct data. Due to the pipeline in the 1*1 conv core as well as the outputs of 1*1 convolution go directly to the average pooling, this enable signal must be delayed by 2 clock cycles from the corresponding data enters the 1*1 convolution. Moreover, the avgpool_sel_zero signal is activated each 784 (16*49) clock cycles and it must be delayed 2 clock cycles from the corresponding data enters the computation core of the 1*1 convolution. Average pooling writes its outputs in the register file so each 784 (16*49) clock cycles, the shift signal is activated. For the same reason; the pipeline in 1*1 convolution and average pooling as shown in Figure 5.57 and Figure 5.58 respectively, it must be delayed 3 clock cycles from the corresponding data enters the computation core of the 1*1 convolution too.

After the 1*1 convolution and average pooling computation cores finish their work and the 1024 words (pixels) are stored in the register file, their operations are hold by stopping their control signals to save dynamic power and make sure that the data in the register file will not change. Also, fully-connected layer start working on the data in the register file, thus, the sel_channels signal is activated and increments each clock cycle to fetch new words from the register file. Moreover, the fc_filter_address increments each clock cycle to fetch new weights from the FC weights memories. The fc_bias_address is also activated and increments each 32 clock cycles to fetch new filter bias. Due to the pipeline in the FC computation core as shown in Figure 5.60, the fc_bias_address must be delayed by 2 clock cycles from the data enters the FC computation core. Like the previous two computation cores, there is a fc_sel_zero signal. It is activated first time after one clock cycles from the beginning of the FC operation and it is then activated each 32 clock cycles. There is a one clock cycle delay at the beginning due to the pipeline in the FC computation core.

While the FC core works, the classification unit works on the outputs of the FC layer simultaneously. Hence, the max_reg_enable is activated first time after 34 clock cycles from the beginning of the FC operation and it is then activated each 32 clock cycles. There is a two clock cycles delay at the beginning due to the pipeline in the FC computation core as well as data goes directly from the FC to the classification unit. Also, classification unit uses the class_count value to store the index of the class of maximum probability. After the value of the class_count reaches 1000 classes, and the FC and classification unit finish their work, the operations of the FC and the classification unit are hold by stopping their control signals to save dynamic power and to make sure that the class index will not change until the next operation. The group_done signal is activated as well to inform the accelerator controller that Group 3 finish its operation and it's free to start new operation.

Table 5.12 Signals of Group 2

Signals	Activation/Increment	Delay
extramem_read_address	Each clock cycle	NA
conv1by1_filter_address1	Each clock cycle	NA
conv1by1_filter_address2	Each clock cycle	NA
conv1by1_bias_address	Each 784 clock cycles	2 clock cycles
conv1by1_sel _zero	Each 16 clock cycles	1 clock cycles
avgpool_acc_en	Each 16 clock cycles	2 clock cycles
avgpool_sel _zero	Each 784 clock cycles	2 clock cycles
shift	Each 784 clock cycles	3 clock cycles
sel_channels	Each clock cycle	NA
fc_filter_address	Each clock cycle	NA
fc_bias_address	Each 32 clock cycles	2 clock cycles
fc_sel _zero	Each 32 clock cycles	1 clock cycles
class_count	Each clock cycle	2 clock cycles
max_reg_enable	Each 32 clock cycles	2 clock cycles
group_done	After class_count = 1000	1 clock cycles

In Group 3 controller, there is a start flag to capture the start signal and hold it high while the group works because the coming start signal from the accelerator controller is a pulse signal. Its architecture based on 10 counters used to manage the sequence of operations in Group 3 and to generate the above control signals. They are:

1) **extramem_read_count:** Counts number of words read from each instance in the Extra memory which means it represents the Extra memory read address. So, it is used to generate the extramem_read_address. Its initial value is 0 and it counts up to 783.
   - ❖ **extramem_read_address = extramem_read_count.**
2) **filter1_offset:** Its value represents the address of the beginning of the filter weights read from the first port of the 1*1 convolution weight memories. So, it is used to generate the conv1by1_filter_address1. Its initial value is 0 and it increments by 32. Also, It Increments each 16 clock cycles.

3) **filter2_offset:** Its value represents the address of the beginning of the filter weights read from the second port of the 1*1 convolution weight memories. So, it is used to generate the conv1by1_filter_address2. Its initial value is 16 and it increments by 32. Also, It Increments each 16 clock cycles.
4) **filter_elem_count:** Each filter in the 1*1 convolution weight memory takes 16 rows from the memory instances. The value of this counter represents the applied filter current row number to be read either from the first port or the second port of the 1*1 convolution weight memory. So, it is used to generate the conv1by1_filter_address1 and the conv1by1_filter_address2 as well as conv1by1_sel_zero. It can be considered as the number of accumulations occurs from the 16 accumulations required to produce a correct output from the 1*1 convolution computation core. Its initial value is 0 and it counts up to 15.
   - ❖ **conv1by1_sel_zero = (filter_elem_count == 15).**
   - ❖ **avgpool_acc_en = (filter_elem_count == 15).**

   From 3 & 4:
   - ❖ **conv1by1_filter_address1 = filter1_offset + filter_elem_count**
   - ❖ **conv1by1_filter_address2 = filter2_offset + filter_elem_count**
5) **window count:** It counts number of pixels processed from a window of the input feature map either by the 1*1convolution or the average pooling as they are working simultaneously and they have the same window size 7*7. So, it is used to generate the avgpool_sel _zero signal. Its initial value is 0 and it counts up to 48. Also, It Increments each 16 clock cycles.

   From 4 & 5:
   - ❖ **avgpool_sel _zero = (window count == 48) & (filter_elem_count == 15).**
6) **bias1by1_add_count:** Its counts number of biases read from each 1*1 convolution bias memory instance which means it represents the address of the bias memory. Hence, it is used to generate the conv1by1_bias_address signal. Its initial value is 0 and it counts up to 63. Also, It Increments each 16 clock cycles.
   - ❖ **conv1by1_bias_address = bias1by1_add_count**
7) **accumulate_count:** It counts number of accumulations occurs from the 32 accumulations required for each filter in the FC computation core. So, it is used to generate the fc_sel_zero signal and the max_reg_enable signal. Its initial value is 0 and it counts up to 31.
   - ❖ **fc_sel_zero = (accumulate_count ==31)**
   - ❖ **max_reg_enable = (accumulate_count ==31)**
8) **sel_channels_count:** Its value represents the required register bank to be read by the FC core from the register file which means the selection of the output mux in the register file. So, it is used to generate the sel_channels signal. Its initial value is 0 and it counts up to 31.
   - ❖ **sel_channels = sel_channels_count**
9) **fc_add_count:** Its value represents the number of weights read from each FC weight memory instance which means the address of the FC weight memories. It is used to generate the fc_filter_address. Its initial value is 0 and it counts up to 31,999.
10) **class_counter:** Its value represents the number of the current applied filter in the FC core which means the current class number. So, it is used to generate the class_count and the

fc_bias_address. Its initial value is 1 and it counts up to 1000. Also, It Increments each 32 clock cycles.

- ❖ **class_count = Class_counter**
- ❖ **fc_bias_address = Class_counter – 1**

From 7 & 10:

- ❖ **group_done = (Class_counter == 1000) & (accumulate_count == 31)**

When the start signal becomes high, all these counters are reset to their initial values. The enables and reset of these counters are controlled through a simple logic on their values. In addition to using delay units to achieve the required delay for the controller signals as in the above table.

5.6 Accelerator Controller Design

After we have illustrated the architecture of the three groups including their internal controllers, and the intermediate memories between them in the previous sections, it is time to illustrate the design of the top-level controller which is called the accelerator controller. It supervises the operation of the sub-controllers and it decides which groups will work and which will not, depending on the number of images in the pipeline and the existence of new incoming image. Also, it handles the communication with the outside world (source of images such as a processor). The accelerator needs to know that the source of images has written an image in the photo memory and it is ready to be processed. On the other hand, the source of images needs to know when it is allowed to write a new image in the photo memory as the photo memory can store only two images at most. Also, it needs to know when the image class is ready.

As our architecture has 3 pipeline stages; there are many scenarios that must be handled by the accelerator. They are as follows:

- **Scenario no.1:** new incoming image to the accelerator and the pipeline is free; no previous images. (Group 1 only must work)
- **Scenario no. 2:** no new incoming image to the accelerator after Group 1 has finished its work and other groups didn’t work last time. (Group 2 only must work)
- **Scenario no. 3:** new incoming image to the accelerator while Group 2 is working only. (Group 1 and Group 2 only must work)
- **Scenario no. 4:** no new incoming image to the accelerator after Group 2 has finished its work and other groups didn’t work last time (Group 3 only must work)
- **Scenario no. 5:** new incoming image to the accelerator after Group 2 has finished its work and other groups didn’t work last time (Group 1 and Group 3 only must work)
- **Scenario no. 6:** no new incoming image to the accelerator after Group 1 and Group 2 have finished their work. (Group 2 and Group 3 only must work)
- **Scenario no. 7:** new incoming image to the accelerator after Group 1 and Group 2 have finished their work. (Group 1 and Group 2 and Group 3 must work)
- **Scenario no. 8:** new incoming image to the accelerator while Group 2 and Group 3 are working only. (Group 1 and Group 2 and Group 3 must work)
- **Scenario no. 9:** no new incoming image to the accelerator after Group 2 and Group 3 have finished their work and Group 3 didn’t work last time. (Group 3 only must work)

- **Scenario no. 10:** new incoming image to the accelerator after Group 1 and Group 3 have finished their work and Group 2 didn't work last time. (Group 1 and Group 2 only must work)
- **Scenario no. 11:** no new incoming image to the accelerator after Group 1 and Group 3 have finished their work and Group 2 didn't work last time. (Group 2 only must work)
- **Scenario no. 12:** no new incoming image to the accelerator after Group 3 has finished its work and other groups didn't work last time. (All groups will not work until the next incoming image)

Now and before talking about the design of the accelerator controller and how it handles the above scenarios, we must define the inputs and outputs of the accelerator controller. Inputs and outputs of the accelerator controller as shown in Figure 5.69 are as follows.

Inputs

The inputs to the accelerator controller are as follows:

Control signals: reset (Global asynchronous reset)

Status signals: group1_done, group2_done, group3_done and maxpool_mem_done (from Group 2).

Handshaking signal: photo_ready (from the source of images).

Output

Control signals: group1_strart, group2_start, group3_start, and ping_pong_sel.

Handshaking signal: busy (to the source of images).

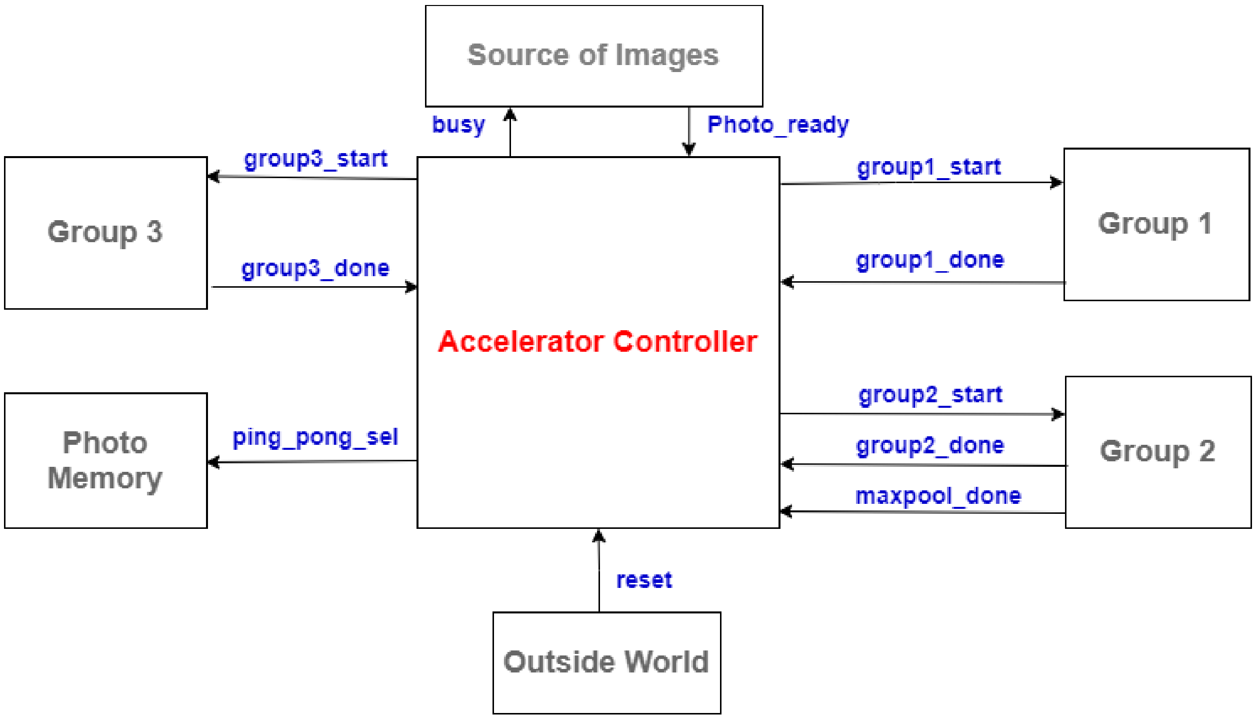


Figure 5.69 The input and output signals of the accelerator controller

The function of each signal of them will be illustrated below in the methodology.

Methodology

When the source of images finishes writing an image in the photo memory and the image is ready to be processed, it must send a pulse on the photo_ready signal. So, the accelerator controller knows that there is a new incoming image. It can write only one image between two successive photo_ready pulses.

As explained in the section of the photo memory, it is a ping pong memory and it has a signal called ping_pong_sel which is used to choose which memory instance will store the next image and the other one will be for reading the current image by default. Thus, when the source of images sends a pulse on the photo_ready signal, the ping_pong_sel signal must be toggle to use the other memory instance to write a new image and allow the accelerator to read the current image. To make the ping_pong_sel signal toggles each time the source of images sends a photo_ready signal, we designed a flag to generate and store the state of the ping_pong_sel signal.

The source of the image cannot send photo_ready signal while the busy signal is high. When it is low, it is allowed to send the photo_ready pulse. The busy signal is high while Group 1 is working or when Group 2 is working and the maxpool_mem_done signal is low because Group will not be able to process the new image in the photo memory 1 in the both cases.

From the scenarios explained above, it is obvious that the best approach to design the controller is the FSM approach. Hence, we designed a finite state machine with 5 states to cover all the scenarios. The state diagram of the FSM is as shown in Figure 5.70.

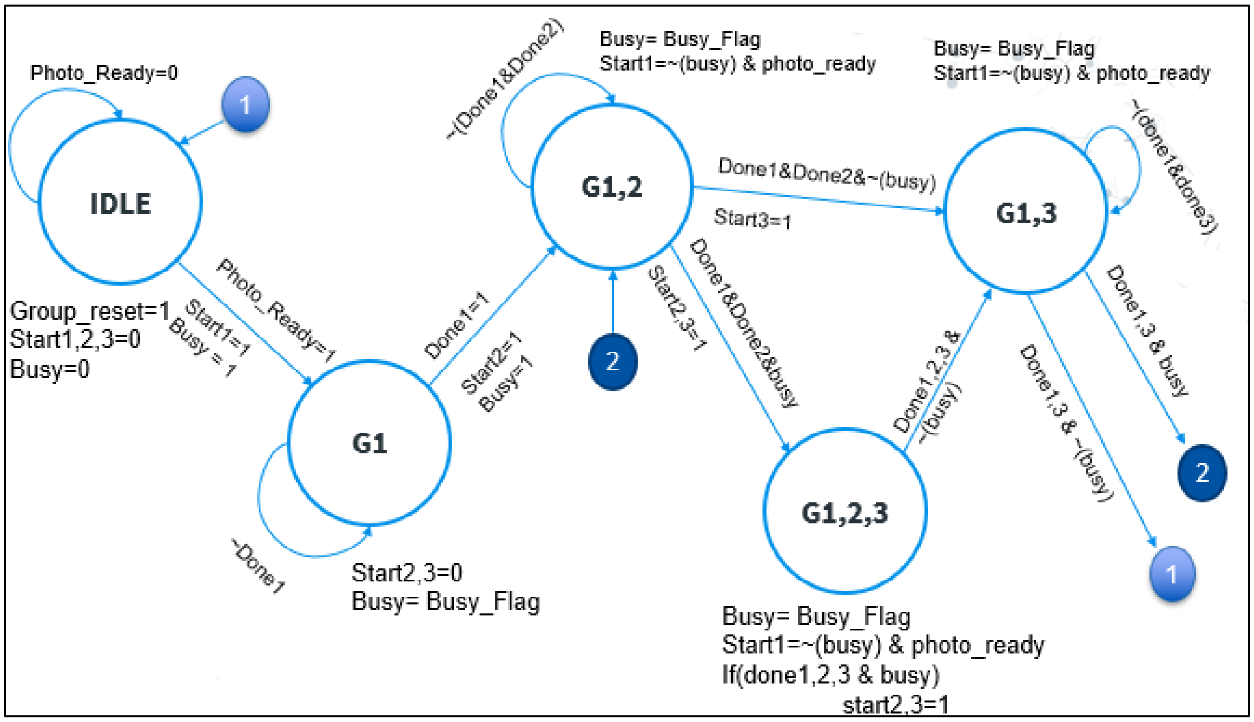


Figure 5.70 The Accelerator Controller State Diagram

The choice of the states is based on the all-possible combinations from the groups as in the scenarios so we have IDLE, G1, G1_2, G1_2_3, and G1_3 states. We can see that Group 1 are in the all states unless the IDLE state because Group 1 depends on the photo_ready signal. So, to avoid redundancy, when other Groups work, we will handle both cases; the case of a new incoming image (Group 1 works) and the case of no new incoming image (Group one doesn’t work). And we can use the state of the busy signal to know whether the Group 1 worked or not and hence choose the next state as shown in the state diagram in Figure 5.70.

group1_start = photo_ready

To transit from a state to another, all done signals of the current working groups must be asserted but as we designed them to be pulses and they are asserted at different times, the controller needs to keep the state of each one of them (when they become high) until the condition becomes valid. As well, the busy signal depends on the value of the maxpool_mem_done signal in the most of states because Group 2 works but the maxpool_mem_done signal is designed to be a pulse and we need when maxpool_mem_done signal becomes high, the busy signal becomes low and keep its state until Group 1 works again. To do that we designed a flag for each one of them to store their states as shown in figure 5.71.

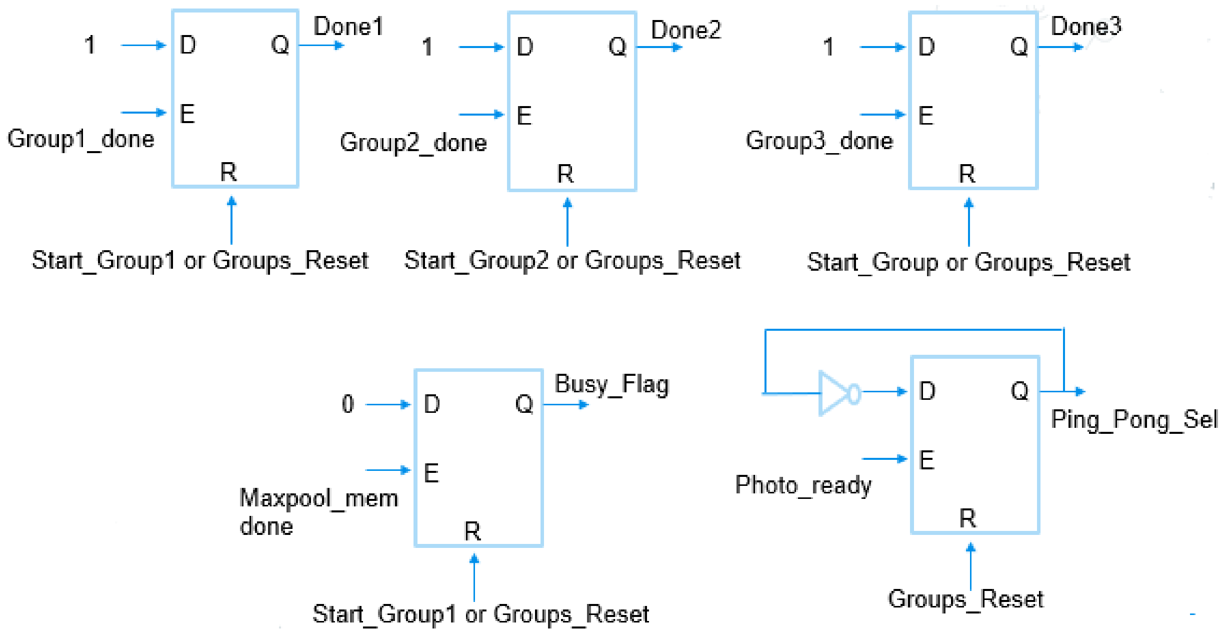


Figure 5.71 Flags of the Accelerator Controller

The start signals of the groups are designed to be pulses as well. For Group 1, it comes from the photo_ready signal which is also a pulse signal. Thus, no need to handle this case. But For Group 2 and Group 3, their start signals are activated depending on the state so we must handle this case here. Thus, we made the activation of these start signals occurs only on transitions to be performed only once and then return low again within any state.

5.7 Summary of RTL Design

This chapter provides a thorough discussion on the chosen design with the details of how the convolution layers of ShuffleNet are implemented, and introducing the different approaches of implementing filter memories and feature map intermediate memories, then the proper implementation of pooling layers, to end up with output classification implementation for getting the index of the maximum probability class. Figure 5.72 summarizes the hierarchy of all blocks used to build the ShuffleNet accelerator.

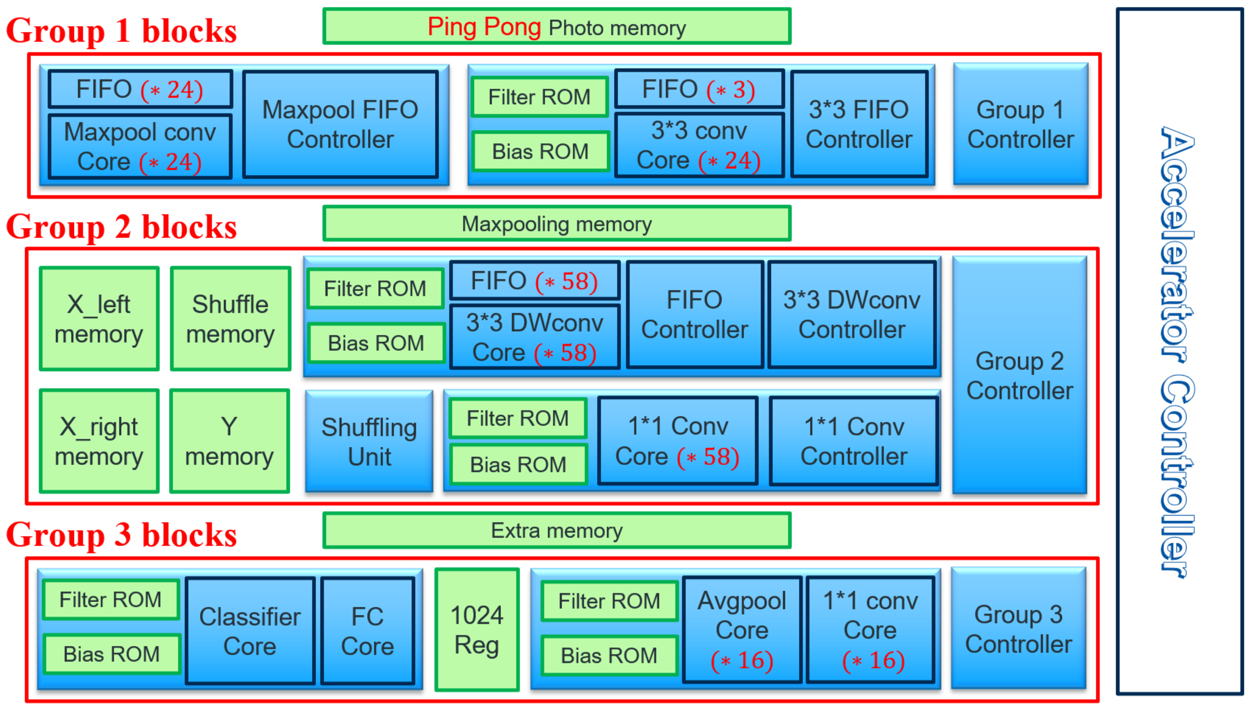


Figure 5.72 Hierarchy of all blocks used to build the ShuffleNet accelerator

Since we target the highest speed with affordable power, one of the main techniques used to improve the accelerator speed was the parallelism in computation cores. Table 5.13 summarizes all the used parallelism factors. We should mark out that the number of channels in all layers of the shuffle group is divisible by 29 which is a non-familiar prime number not divisible by any other factor. But we could exploit this demerit for our favor by making the parallelism in channels be of a factor 29 and parallelism in filters be of a factor 58, so any number of channels or filters will be divisible by the used parallelism. This leads to a complete fitting and better utilization of memories.

Table 5.13 Summary of all used parallelism factors in the ShuffleNet Accelerator

Block		Parallelism Used		
3*3 Conv		24 parallel in filters	3 parallel in channels	3 parallel in window
Maxpooling		24 parallel in channels		
Shuffle group	1*1 Conv	58 parallel in filters	29 parallel in channels	
	3*3 DWconv	58 parallel in filters	9 parallel in window	
1*1 Conv		16 parallel in filters	29 parallel in channels	
Avg pooling		16 parallel in channels		
FC layer		32 parallel in channels		

5.8 Verification of RTL Functionality

The validation step is one of the most important steps in the design to make sure that the design is behaving correctly comparing to the software model of the architecture. And also, the behavior is still correct after any optimization done to the model.

It also very important to verify the functionality not only to the hole system but also the sub-blocks before assembling the accelerator. This makes it easier to locate problems. To test and validate the design we need a testing strategy to automatically insert inputs and check the outputs.

5.8.1 Testing Strategy

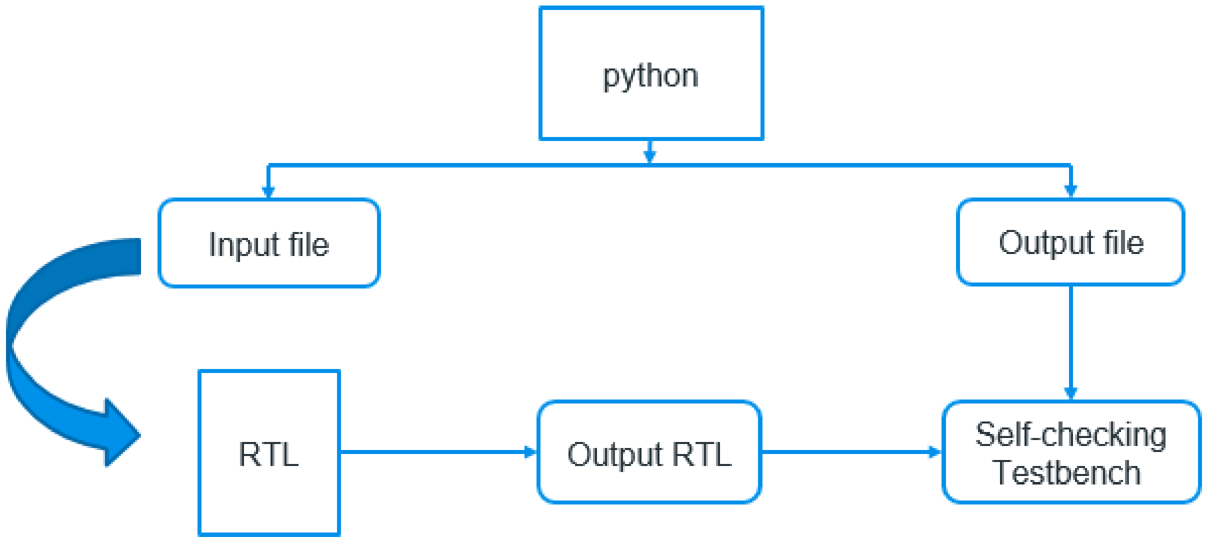


Figure 5.73 Testing strategy

We use the software implementation using python of the accelerator as the reference for the RTL design. Figure 5.73 illustrates the testing strategy we used. First, we use the python model to generate files containing the input of the model written in the correct format and quantization that the RTL model expect. And generation files containing the correct outputs to compare it with the outputs of the RTL model. Then we create a self-checking testbench to read the input files and pass them to the design then write the output of the RTL to files and compare them with the files generated from the python model.

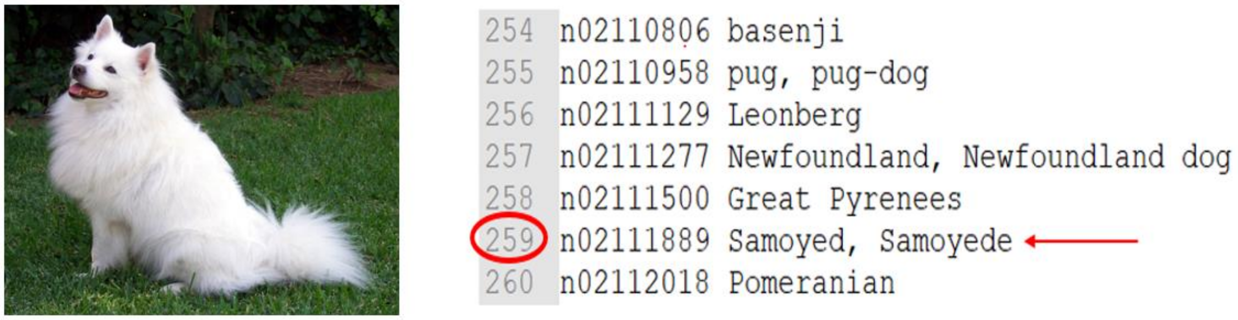
This procedure is performed for each group separately functional, post-synthesis and post-implementation. Then for the entire accelerator. If any miss-match between the two files appears, we debug the source of the problem to solve it.

5.8.2 Validation Results

5.8.2.1 Validation of the Design Functionality

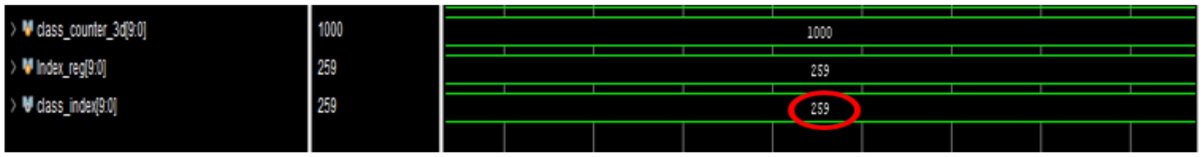
To validate the functionality of the model we used a photo from the validation set to use it as an input to the design and then check the output class generated from the RTL model. We used the photo and the correct class is 259 as shown in Figure 5.74. Which is equals to the class generated from the RTL model as shown in Figure 5.75.

This operation is not enough to validate the accuracy of the model.



```
254 n02110806 basenji
255 n02110958 pug, pug-dog
256 n02111129 Leonberg
257 n02111277 Newfoundland, Newfoundland dog
258 n02111500 Great Pyrenees
259 n02111889 Samoyed, Samoyede
260 n02112018 Pomeranian
```

Figure 5.74 ImageNet Sample and predicted class



```
class_counter_3d[9:0]   1000   1000
Index_reg[9:0]          259    259
class_index[9:0]        259    259
```

Figure 5.75 Simulation output class

5.8.2.2 Validation of the accuracy

To validate the accuracy, we and to use a large number of test photos to get a reasonable measure of the accuracy. The accuracy of this validation step is increasing by increasing the test photos used.

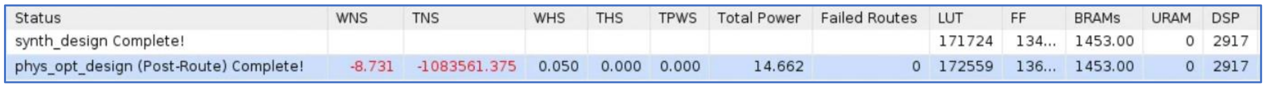
We used 100 photos in this step to get an acceptable measure of the accuracy with acceptable run time because the simulation of one photo requires large run time.

The accuracy of the RTL model for the 100 photo is 71% which is the same as the accuracy of the software model for the same 100 photos. This accuracy is predicted to be lower for the 50000 photos to be equals to the software model (68.14%).

6 Chapter 6: Optimizations

After the architecture blocks had been integrated into one top-level block and functionally verified, as explained in chapter 5, we proceeded with the rest of the FPGA flow (synthesis and implementation). Our target FPGA is the Virtex-7 FPGA. When we tried implementing, the process ended with a netlist that fits in the FPGA resources, but it has congestion of level 6. So, the tool couldn't complete routing and finish implementation. To solve this problem, we used the implementation directives of Vivado to force it to make much effort in the implementation steps. Also, we tried to reduce the fanout through the architecture. The implementation directives and handling the high fanout nets are used to solve the congestion are explained later in this chapter. When we implemented the design on 50 MHz, it was successful without any violations and worked on the FPGA with 608 frames per second, this result will be discussed later in chapter 9. When we tried implementing on 100 MHz, we ended with a netlist that is placed and routed on the FPGA successfully, but the critical path has a slack of -8.731 ns, as shown in Figure 6.1, because of routing. So, from this point, we started performing other optimizations on the architecture to help it meet timing, reduce area and reduce power.

Throughout the chapter, we will explain the software optimizations that will then lead to RTL level optimizations that we applied to the architecture to reduce its size and hence reduce routing delay. These optimizations enable us not just to meet timing but also help reduce the used resources and hence reduce power too.



Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
synth_design Complete!								171724	134...	1453.00	0	2917
phys_opt_design (Post-Route) Complete!	-8.731	-1083561.375	0.050	0.000	0.000	14.662	0	172559	136...	1453.00	0	2917

Figure 6.1 Timing before optimization

6.1 Software Modeling Optimizations

6.1.1 Verified Software model

After we finished our hardware model and we were sure that it was working correctly (output class is correct), we compared the output probabilities of the hardware model with the output probabilities of the software model and we found that there is about 10% difference between them which is an unexplained and unpredicted difference.

We found that we have done some edits on the hardware model to avoid divisions, so we returned back to the software model and did all the edits, that have been done on the hardware model, on software model to make it identical to the hardware model to predict the validation accuracy with zero error and zero difference between probabilities.

Edits are removing batch normalization layers from the software model, editing weights of the model to have the same effect of batch normalization, multiplying in avg pooling layer by 5/256, not 1/49 due to quantization, and separating the branches of the software model which are consists of three or two convolutions together without quantization between them, so we separate each convolution to quantize the feature map after each convolution like the hardware model.

After we made these edits, we retested the model on the whole 50000 photos of the ImageNet validation with different quantization to check the accuracy again and the results are shown in Table 6.1.

Table 6.1 Quantization results on 50000 photos

Floating Point 32-bit Accuracy			69.362%
Bit Width	**Integer length**	**Fractional Length**	**Accuracy**
17	6	10	69.244%
16	7	8	66.578%
16	6	9	68.704%
16	5	10	68.068%
15	6	8	66.576%
15	5	9	67.532%
15	4	10	38.744%
14	4	9	38.476%
14	5	8	65.454%
14	6	7	55.950%
13	4	8	38.246%
13	5	7	55.054%

From Table 6.1, we can see that the edits decrease the accuracy of any quantization as the model has a lower number of weights and quantization after every layer of the model, not branches as before.

Also, we can see that the accuracy of our hardware model is not as expected before, it's actually 66.57%, not 68.3%. Also, we can conclude that if we use 15-bit quantization it's better to make a 9-bit fraction than an 8-bit fraction.

From these results and observations, we see that the fraction affects the model greater than the integer. The minimum acceptable bits of the integer are 5 bits but the minimum acceptable bits of the fraction are 8 bits. We want to increase the accuracy versus the number of quantization bits, to have better quantization.

6.1.2 Increasing accuracy

We started by studying the effect of feature map quantization and weights quantization and quantizing them independently. We observed that quantization of weights has a greater effect on the accuracy than feature map quantization. This can lead us to have a 15-bit quantization of the feature map and have an accuracy of 68.5% which is very close to 16-bit quantization.

Then we make a weights file that can save an accuracy of 68.5%. We will use the file with any feature map quantization. We called these weights optimum weights to talk about them easily. Optimum weights are mentioned in detail as shown in Table 6.2.

Table 6.2 Optimum weights quantization

Weights	width (bits)	fraction (bits)
3by3 conv. weights	12	9
Shuffle group 1by1 conv. weights	11	9
Shuffle group 3by3 conv. weights	15	8
Shuffle group biases	13	9
1by1 conv. weights	9	8
fully connected weights	15	8
fully connected biases	11	8

6.1.3 Editing the original shuffle model

From optimum weights, we see that fully connected weights have a 6-bit integer due to its maximum weight value of 40.5. This quantization of fully connected make it consume a large number of BRAMs as it has 1 million parameters. If we decrease this width, we will save a lot of BRAMs.

We can remove dividing on 49 from avg pooling layer and divide the fully connected weights by 49 as shown in Figure 6.2, this will make the maximum weight in it equal to 0.82 which doesn't need integer bits, so we can quantize fully connected weights with 9-bit width 8-bit fraction this will reduces BRAMs utilization.

$$\text{FC output} = \sum \text{AVG pool output} \times W_{FC} + b_{FC}$$

$$\text{FC output} = \sum \frac{\sum \text{Conv1by1 out}}{49} \times W_{FC} + b_{FC}$$

$$\text{FC output} = \sum \sum \text{Conv1by1 out} \times \frac{W_{FC}}{49} + b_{FC}$$

Figure 6.2 Divide FC weights by 49 instead of avg pooling

Also, we will get rid of multiplying by 1/49 on the hardware model which is quantized to 5/256, causes decreasing in output probabilities and decreasing the accuracy. So now, avg pooling doesn't have division we can call it sum pooling and we update fully connected weights.

The breakpoint of accuracy is the 14-bit quantization of the feature map. Are all layers preventing us from decreasing quantization bits to 14 bits or individual layers? When we investigated all layers, we find that the fully connected layer is the single layer that prevents us from decreasing quantization bits to 14 bits as its output needs at least 5 bits for integer. but all layers of the model need only 3 bits for integer.

We can deal with this problem by quantizing fully connected with different quantization, but we have a better way to deal with this problem which is dividing the fully connected equation by a fixed number. This way is slightly strange, but we can explain it simply. We determine the top-1 class depending on the maximum fully connected output, so if we divide all fully connected outputs by a fixed number, this doesn't affect the maximum (the top-1 class).

So, we will divide the fully connected outputs by 4 as shown in Figure 6.3 to make them need 3 bits for integer like all layers, not 5 bits. Also, number 4 has a powerful advantage, as we don't need to divide, it simply can be a shift operation which is very easy on hardware design.

$$\text{FC output} = \sum \sum \text{Conv1by1 out} \times \frac{W_{FC}}{49} + b_{FC}$$

$$\frac{\text{FC output}}{4} = \sum \frac{\sum \text{Conv1by1 out}}{4} \times \frac{W_{FC}}{49} + \frac{b_{FC}}{4}$$

Figure 6.3 Divide FC equation by 4

There are several benefits of this edit. First, we can divide the output of average pooling by 4 as shown in Figure 6.3 which makes the output of average pooling decrease again, as we removed dividing by 49. Second, fully connected output need only 3 integer bits so we can quantize all model on 12-bit width and 8-bit. Third, fully connected bias can be quantized 9-bit width and 8-bit fraction as the maximum value of it was 3, so when we divide it by 4 it doesn't need any integer bits.

These edits have never happened before on the CNN architectures in general. After these edits on the original shuffle model, we can quantize the model on 12-bit width and 8-bit fraction with an accuracy of 68.38%.

6.1.4 Photo Quantization

We can quantize the photo with 11-bit width and 8-bit fraction, as the transformations performed on the photo before sending it to the model make the maximum integer of it 2 only which needs 2 bits integer only. We apply this quantization to the whole 50000 photos of the ImageNet validation and it doesn't affect the accuracy at all, as the integer bits which more than 2 bits don’t carry any information.

We can quantize the photo with 8-bit width and 5-bit fraction, this affects accuracy only by 0.12% which is a very small effect but saves a lot of BRAMs due to the size of the photo. We apply this quantization to the whole 50000 photos of the ImageNet validation and the accuracy before it is 68.38% and the accuracy after it is 68.26%.

6.1.5 Dynamic Quantization

To get a better result, we decided to do dynamic quantization. After several trials of quantization for each layer of the model, we reach this dynamic quantization described in detail in Table 6.3.

Table 6.3 Dynamic quantization of layers

Layer name	width (bits)	fraction (bits)
photo	8	5
3by3 conv. output	10	6
Max-pooling output	10	6
Shuffle group output	12	8
1by1 conv and avg pooling	9	5
fully connected	9	5

This dynamic quantization has an accuracy of 68.184% on the whole 50000 photos of the ImageNet validation which is decreased by about 1.17% with respect to floating point accuracy. If we decrease any bit of this dynamic quantization, accuracy will decrease at the minimum by 0.22% which is comparable to the total decrease.

6.2 RTL Optimizations

6.2.1 Powerful Implementation Directives

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

1) **Opt Design:** Optimizes the logical design to make it easier to fit onto the target FPGA.
2) **Power Opt Design (optional):** Optimizes design to reduce the power demands of the FPGA.
3) **Place Design:** Places the design onto the target FPGA.
4) **Post-Place Power Opt Design (optional):** Additional optimization to reduce power after placement.
5) **Post-Place Phys Opt Design (optional)**: Optimizes logic and placement using estimated timing based on placement.
6) **Route Design:** Routes the design onto the target FPGA.
7) **Post-Route Phys Opt Design (optional):** Optimizes logic, placement, and routing using actual routed delays.
8) **Write Bitstream:** Generates a bitstream for the FPGA configuration.

With each of these sub-processes we can choose powerful directives to reach a better implementation that's optimized for timing, power or something else. The directives that we used are listed below with explanation of each of them.

- *opt_design -ExploreWithRemap*: Runs multiple passes of optimization with remapping and combining multiple LUTs into a single LUT to reduce the depth of the logic.
- *place_design -ExtraNetDelay_high*: Increases estimated delay of high fanout and long-distance nets. This directive can improve timing of critical paths that meet timing after place_design but fail timing in route_design due to overly optimistic estimated delays. Two levels of pessimism are supported: high and low. ExtraNetDelay_high applies the highest level of pessimism.
- *post-place phys_opt_design -AggressiveExplore*: Runs different algorithms in multiple passes of optimizations with more aggressive goals.
- *route_design -NoTimingRelaxation*: Prevents the router from relaxing timing to complete routing. If the router has difficulty meeting timing, it runs longer to try to meet the original timing constraints.
- *post-route phys_opt_design -AggressiveExplore*: Directs the router to further expand its exploration of critical path routes while maintaining original timing budgets. The router runtime might be significantly higher compared to the Explore directive because the router uses more aggressive optimization thresholds to attempt to meet timing constraints.

6.2.2 Handling High Fanout signals

One of the causes of design congestion on FPGA is the presence of many high fanout signals. So, by reducing this high fanout or reducing it, the routing will be easier and any congestion will be resolved. Our Design had a congestion of level 6 when first implemented, so we reported all the high fanout nets using this command *report_high_fanout_nets* and found that the congestion originates from some signals related to Group 2 as listed below:

- Shift_load for FIFO
- Reset FIFO
- Reset Core Registers in 3by3
- Reset Core Registers in 1by1
- Address of Maxpool memory

First, we tried to reduce the fanout of these control signals using the control sets in Virtex-7, where all registers in a slice share common control signals (CLK, reset/set, enable), and thus, the control sets can be packed into the same slice. We Used opt_design -control_set_merge and opt_design merge_equivalent_drivers commands to merge equivalent control sets after synthesis. But this wasn't enough, so we used some implementation directives like *post-place phys_opt_design* -explore and *post-route phys_opt_design -explore* which solved the congestion on account of routing delay. To help reduce the congestion so as to reach a better timing, we eliminated the high fanout control nets reported above and reduced the fanout of the Maxpool memory address by implementing it using BRAMs instead of LUTs.

Removing shift_load signal

The shift_load signal of the FIFOs in Group2 is always high except the last clock cycle after last window process, where data_valid is high. We will eliminate this clock cycle to make the shift_load signal always high and raise data_valid in the beginning of the next state after process as shown in Figure 6.4.

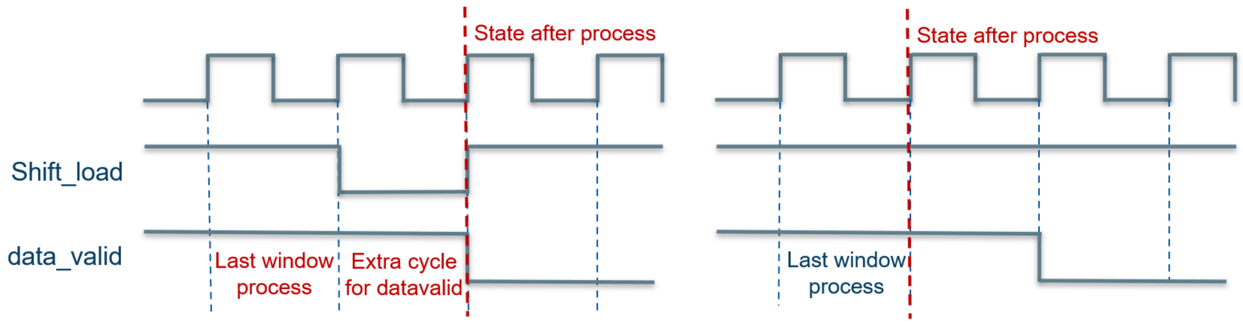


Figure 6.4 Removing shift_load signal

Removing all reset signal

We used to reset the FIFO in IDLE state to have the first zero padding row already when using the FIFO. To remove the FIFO reset, we added a state after IDLE to load the 1st row of padding in FIFO. In IDLE, we put padding_sel = 1, to fill the FIFO with zeros when not using it, this gives the same effect of reset.

The zeros in FIFO propagates to the Multiplier registers of 3by3 DWconv core after 1 clock cycle, so no need to reset the Multiplier registers in the 3by3 DWconv core. The Multiplier registers of 1by1 core doesn't require a reset too, as the 1by1 is operating most of the time.

Finally, after these optimizations, we were able to implement the design on 50 MHz without any violations and reached 608 frames per second. The upcoming optimizations aims at improving the timing, power and area and to increase the maximum operating frequency of the design.

6.2.3 Optimizing the Division in the Average Pooling Layer and the FC Weights

In the average pooling, the output is divided by 49, we designed it as a constant multiplier by 1/49 as shown in Figure 6.5, but if we put this number in a fixed-point representation, we will discover that it is only 101 at the least bits of the 15-bit representation and all the remaining bits are zeros. Actually, it is not a 1/49 exactly, it is 5/256 because of quantization noise. Hence, it will affect and reduce the accuracy. To overcome this issue, we can remove this multiplier from the average pooling and divide the FC weights by 49, as shown in Figure 6.6, because the outputs of the average pooling will be multiplied then by the FC weights so it will give the same effect. The FC weights will be divided by 49 and then quantized by software. Thus, we ensured that the accuracy is the same as the software model as well as the data width in the average pooling and the FC weights are minimized. Hence the required number of BRAMs and resources are reduced. And as a result, it will have a positive effect on timing.

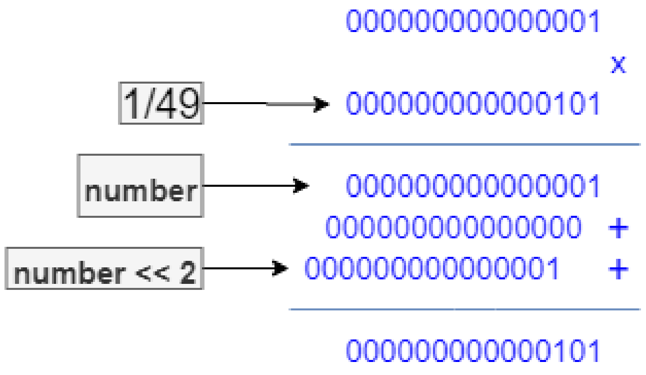


Figure 6.5 Average pool multiplier

6.2.4 Reduction for the Output Size of FC

As explained in chapter 5, the probabilities of the 1000 classes are passed from the FC layer to the classification unit, and it just compares them. So, we can divide them by four first and then compare them. This optimization will allow us to reduce the width of the signals and hence reduce the used resources and routing delays while it will not affect the accuracy as we are interested in the max probability, not the absolute value. Also, we ensured by the software model that there is no impact on the accuracy. We can apply the same effect of dividing the FC outputs by 4 through dividing the average pooling output and the bias of the FC by 4. It is a shift right by 2 in the case of the average pooling output and the bias parameters are updated by software. This optimization above as well as the optimization stated above for the division by 49 in the average pooling are, as shown in the FC output equation in Figure 6.6. So, we can gain the most benefits from this optimization as it results in reducing the size of the FC output data, average pool output, and the number of BRAMs used for weights and biases of the FC, also the worst negative slack is reduced.

$$\text{FC output} = \sum \text{AVG pool output} \times W_{FC} + b_{FC}$$

$$\text{FC output} = \sum \frac{\sum \text{Conv1by1 out}}{49} \times W_{FC} + b_{FC}$$

$$\text{FC output} = \sum \sum \text{Conv1by1 out} \times \frac{W_{FC}}{49} + b_{FC}$$

$$\frac{\text{FC output}}{4} = \sum \frac{\sum \text{Conv1by1 out}}{4} \times \frac{W_{FC}}{49} + \frac{b_{FC}}{4}$$

Figure 6.6 FC output size reduction

6.2.5 Applying Optimum Weights and Bias Sizes for All Layers

Applying optimum weights and bias width through reducing their widths in each layer while guaranteeing that the software model's accuracy is unaffected. So, the width of the weights and biases of 3x3 convolution will be 12 bits (9 bits for the fraction). And the biases of group 2 will be of size 13 bits (9 bits for the fraction) and the weights of 3x3 DW convolution will be 15 bits (8 fractions) and the weights for 1x1 convolution in group 2 will be 11 bits (9 fractions) and the weights and bias for group 3 will be 9 bits (8 fractions) as shown in Figure 6.9. As shown in Figure 6.8, this reduces the number of BRAMS consumed to 1155.5 BRAMs and lowers the WNS to -0.053ns.

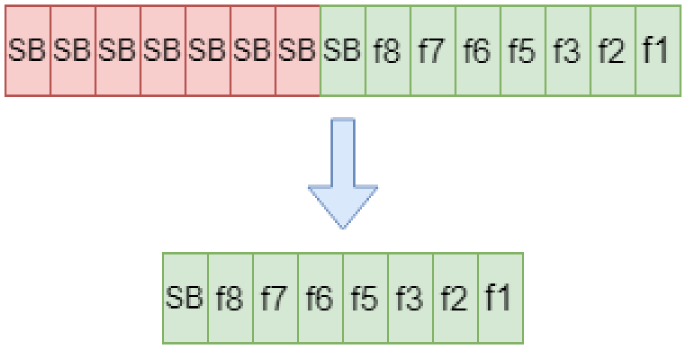
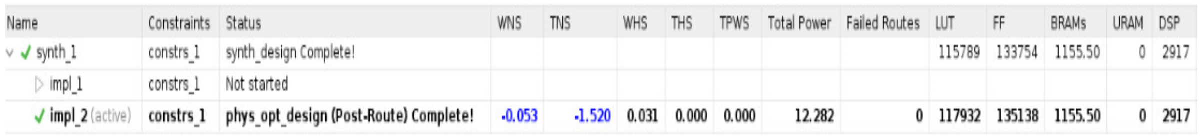


Figure 6.7 Reduction in the width of the Group 3 conv1by1 weights



Name	Constraints	Status	WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAMs	URAM	DSP
✓ synth_1	constrs_1	synth_design Complete!								115789	133754	1155.50	0	2917
impl_1	constrs_1	Not started												
✓ impl_2 (active)	constrs_1	phys_opt_design (Post-Route) Complete!	-0.053	-1.520	0.031	0.000	0.000	12.282	0	117932	135138	1155.50	0	2917

Figure 6.8 Timing after optimum weights and bias sizing

6.2.6 Reducing the width of the input Photo

Photos are formed from pixels, and each pixel value are represented in 8-bit integer number which means it can take values from 0 to 255 only. In the ShuffleNet CNN, input images must be preprocessed first by a certain transformation. This transformation changes the range of the pixel value and makes its value has integer and fraction parts. And as we use a 15-bit fixed point (8-bit fraction) representation in this architecture, the input photo pixels are represented in the same way too. But value of the photo pixel after the transformation is from -2 to 2 so no need for all the 15-bits, it only needs 2 bits for the integer part and 8 bits for fraction part. Hence, the total bits needed are 11 bits including the sign bit. This optimization will reduce the data width in the photo memory and in the computation cores in Group 1 and hence decrease number of BRAMs, LUTs and registers. It also will decrease routing wires used and help the design meet timing.

Actually, we reduced the width of the input photo to be 8-bits (5-bit fraction) because the accuracy will be a little bit affected (reduced by 0.01%) while it will reduce the number of BRAMs used. We validated the accuracy by the software model after applying this optimization to it.

6.2.7 Removing the quantizer after Group3 1*1 convolution

In the architecture, we put a quantizer at the end of each computation core. But we found out through the software model that when we put a qauntizer at the end of the computation core of the 1*1 convolution in Group 3, the accuracy degraded very much. So, we removed this quantizer from the RTL. It may increase the width of the data in the average pooling a little bit but it will save the accuracy. Also, average pooling is now just a summation unit and it has a quantizer at the end.

6.2.8 Applying Dynamic Quantization for All Model Layers

Using a dynamic quantization for all model layers based on the software model to determine the appropriate data width for each layer will not compromise accuracy. Beginning with the data from the photo memory, the data width will be 8 (5 bits for the fraction) due to some processing reducing the values of the pixels for the image, and moving to the output of group 1, the data width will be 10 bits (6 bits for the fraction), the output from group 2, the data width will be 12 bits (8 bits for the fraction), and the output of group 3 will have data width of 9 bits (5 bits for the fraction) as shown in Figure 6.9. This results in meeting timing with a positive slack of 0.014 ns and a positive hold slack of 0.046 ns targeting 100MHz clock frequency as shown in figure 6.10.

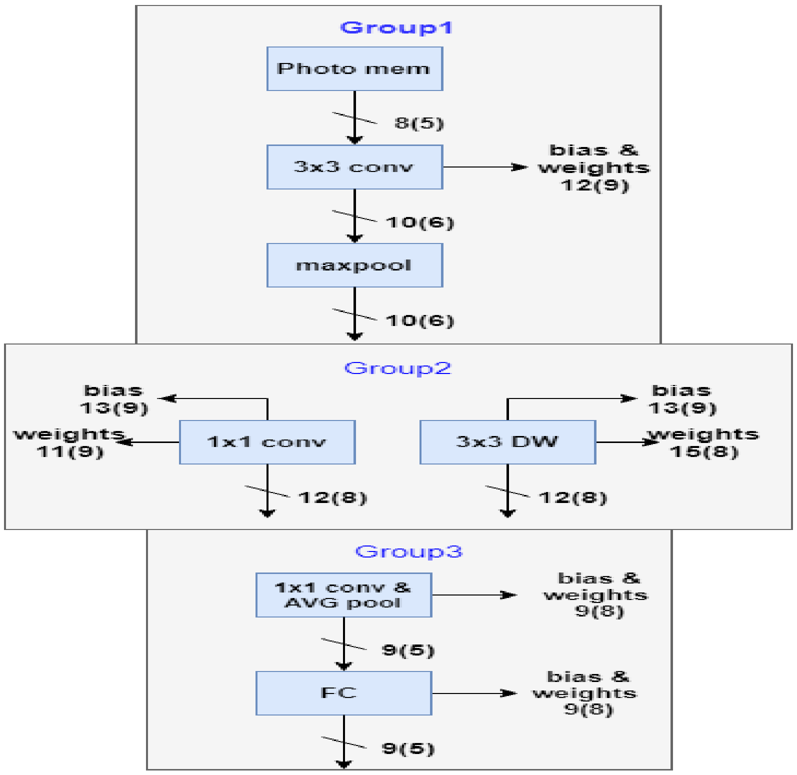
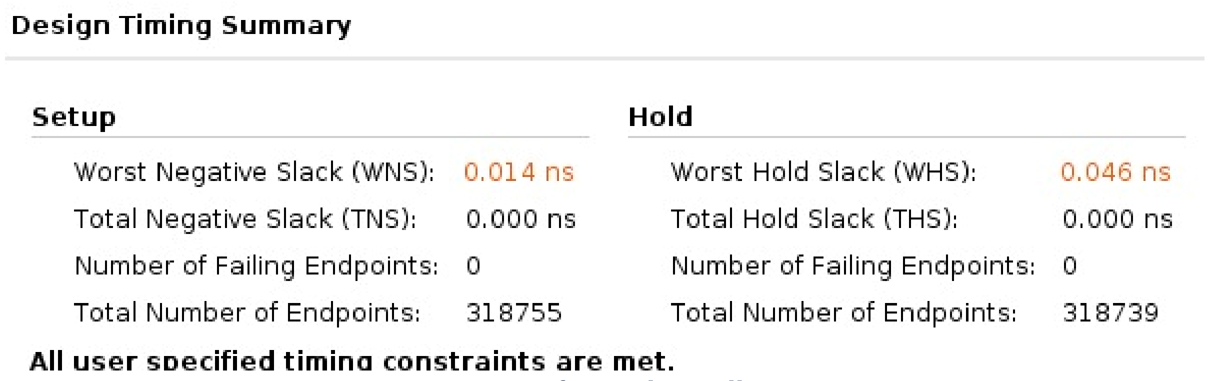


Figure 6.9 Optimum weights, bias size, and dynamic quantization



Design Timing Summary

Setup

Worst Negative Slack (WNS):	0.014 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	318755

Hold

Worst Hold Slack (WHS):	0.046 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	318739

All user specified timing constraints are met.

Figure 6.10 Timing after applying all optimizations

7 Chapter 7: High Level Synthesis (HLS)

7.1 Introduction

7.1.1 HLS Design Flow

High Level Synthesis (HLS) is a tool which takes a software model written in a high-level language like C, C++ or SystemC as input and generates RTL files as output which describes the model in terms of hardware components which is done in process. RTL files are written in Verilog or VHDL and are synthesizable so that we can use it with a synthesizer to implement the RTL files into real hardware whether the target hardware is ASIC or FPGA. In our case, we target FPGA so we will use **Vivado HLS** tool. We chose to use C++ as high-level language in software modelling.

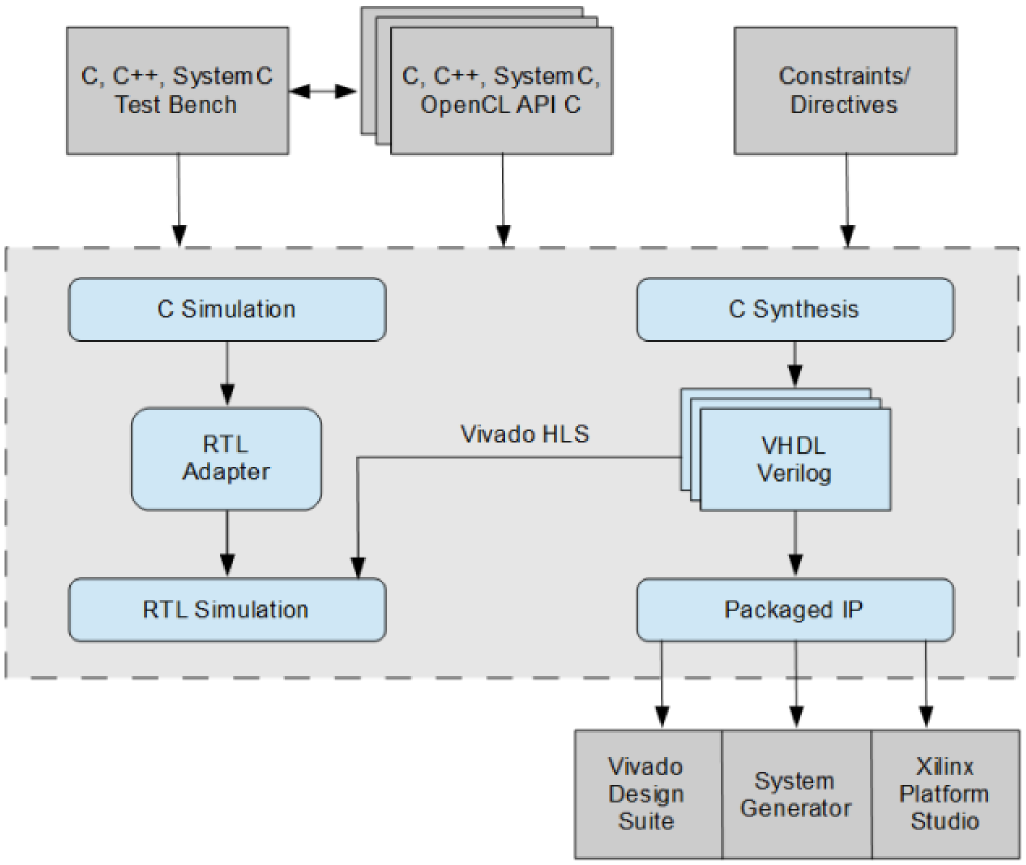


Figure 7.1 HLS design flow

As shown in Figure 7.1 [22], HLS tool give us the power of verification the behavior of the generated RTL files (RTL simulation and is referred to by Vivado HLS as C/RTL co-simulation) by writing simply high level testbench with high level language also which make it faster to develop then the tool will generate the RTL testbench also in Verilog or VHDL. Also, we can run the software model on the HLS tool to verify its functionality before making high level synthesis (C simulation) using the same high level testbench.

HLS tool also can take some Constraints and directives which help the designer to specify how the generated hardware will look like to make parallelism, pipelining and resource sharing to meet the specification of the hardware system and desired performance in terms of throughput, latency, speed, area and power. There are also some constraints which can guide the tool to do more effort to generate the desired hardware. HLS tool gives some reports about the estimated latency, maximum clock and the resources of each hardware block generated. In the Vivado HLS GUI, the Analysis Perspective allows you to interactively analyze the results in detail.

After generating the desired hardware and verify its functional behavior using RTL simulation, tool can export the RTL files as an Intellectual Property (IP). We can use the IP inside any other design, or make it the top design if we want, to implement the design into real hardware on FPGA and this can be done by using the IP inside Vivado design suite.

We can summarize the HLS design steps as following:

1) Develop algorithm at the high level (like C++).
2) Verify the algorithm at the high level. (C simulation)
3) Synthesize the C algorithm into an RTL (Optimize using directives and constraints).
4) Generate reports and analyze the design (Analysis Perspective).
5) Verify the generated RTL (RTL simulation).
6) Export the design as an IP.

7.1.2 HLS Benefits

High Level Synthesis bridges hardware and software domains aiming to [23]:

- Improved productivity for hardware designers: Hardware designers can work at a higher level of abstraction while creating high-performance hardware.
- Improved system performance for software designers: Software developers can accelerate the computationally intensive parts of their algorithms on a new compilation target, the FPGA.
- Develop algorithms at the C-level: Work at a level that is abstract from the implementation details, which consume development time.
- Verify at the C-level: Validate the functional correctness of the design more quickly than with traditional hardware description languages.
- Control the C synthesis process through optimization directives: Create specific high-performance hardware implementations.
- Create multiple implementations from the C source code using optimization directives: Explore the design space, which increases the likelihood of finding an optimal implementation.
- Create readable and portable C source code: Retarget the C source into different devices as well as incorporate the C source into new projects.

7.1.3 HLS Phases

High-level synthesis includes the following phases:

- **Scheduling**
  Determines which operations occur during each clock cycle based on:
    - Length of the clock cycle or clock frequency.
    - Time it takes for the operation to complete, as defined by the target device.
    - User-specified optimization directives.
- **Binding**
  Determines which hardware resource implements each scheduled operation. To implement the optimal solution, high-level synthesis uses information about the target device.
- **Control logic extraction**
  Extracts the control logic to create a finite state machine (FSM) that sequences the operations in the RTL design.

7.1.4 HLS Code Structure

High-level synthesis synthesizes the C code as follows:

- Top-level function arguments synthesize into RTL I/O ports.
- C functions synthesize into blocks in the RTL hierarchy: If the C code includes a hierarchy of sub-functions, the final RTL design includes a hierarchy of modules or entities that have a one-to-one correspondence with the original C function hierarchy. All instances of a function use the same RTL implementation or block.
- Loops in the C functions are kept rolled by default: When loops are rolled, synthesis creates the logic for one iteration of the loop, and the RTL design executes this logic for each iteration of the loop in sequence. Using optimization directives, you can unroll loops, which allows all iterations to occur in parallel. Loops can also be pipelined, either with a finite-state machine fine-grain implementation (loop pipelining) or with a more coarse-grain handshake-based implementation (dataflow).
- Arrays in the C code synthesize into block RAM or UltraRAM in the final FPGA design: If the array is on the top-level function interface, high-level synthesis implements the array as ports to access a block RAM outside the design.

7.1.5 Important Directives

1) **ALLOCATION**
   Specify a limit for the number of operations, cores or functions used. This can force the sharing or hardware resources and may increase latency.
2) **ARRAY_PARTITION**
   Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks. Vivado HLS provides three types of arrays partitioning as shown in Figure 7.2:

1. Block: The original array is split into equally sized blocks of consecutive elements of the original array.
2. Cyclic: The original array is split into equally sized blocks interleaving the elements of the original array.
3. Complete: The default operation is to split the array into its individual elements. This corresponds to resolving a memory into registers.

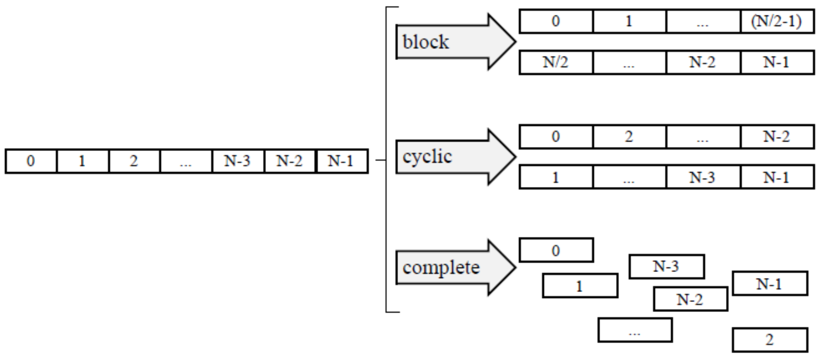


Figure 7.2 Array Partition Styles

3) DATAFLOW

Enables task level pipelining, allowing functions and loops to execute concurrently (Parallelism). Used to optimize throughout and/or latency as shown in Figure 7.3.

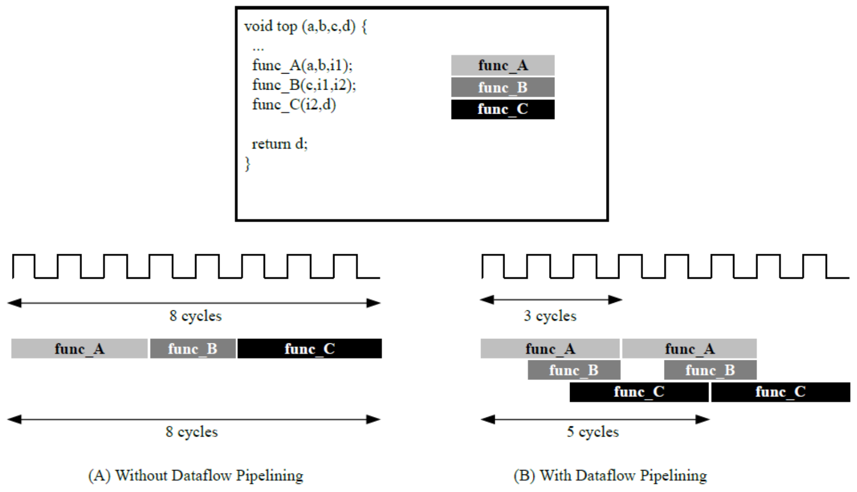


Figure 7.3 Dataflow Optimization

4) DEPENDENCE

Used to provide additional information that can overcome loop-carried dependencies and allow loops to be pipelined (or pipelined with lower intervals).

5) INLINE

Inline a function, removing function hierarchy at this level. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead.

6) **RESOURCE**

Specify that a specific library resource (core) is used to implement a variable (array, arithmetic operation or function argument) in the RTL.

7) **LOOP_FLATTEN**

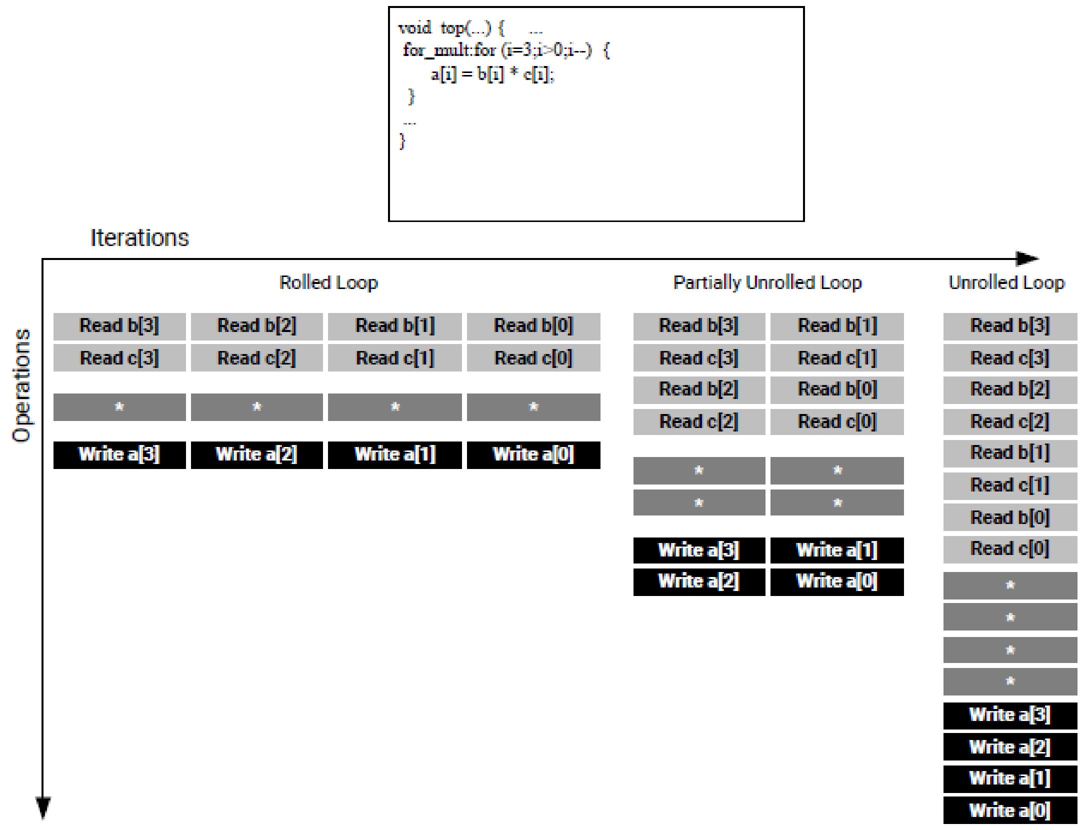
Allows nested loops to be collapsed into a single loop with improved latency.

8) **LOOP_TRIPCOUNT**

Used for loops which have variables bounds. Provides an estimate for the loop iteration count. This has no impact on synthesis, only on reporting.

9) **UNROLL**

Unroll for-loops to create multiple instances of the loop body and its instructions that can then be scheduled independently as shown in Figure 7.4.



```
void  top(...) {     ...
 for_mult:for (i=3;i>0;i--)  {
      a[i] = b[i] * c[i];
  }
 ...
}
```

Figure 7.4 Loop Unrolling Details

10) **PIPELINE**

Reduces the initiation interval by allowing the overlapped execution of operations within a loop or function. Pipelining allows operations to happen concurrently, each execution step does not have to complete all operations before it begins the next operation. Pipelining is applied to functions and loops. The throughput improvements in function and loop pipelining are shown in Figure 7.5 and 7.6 respectively.

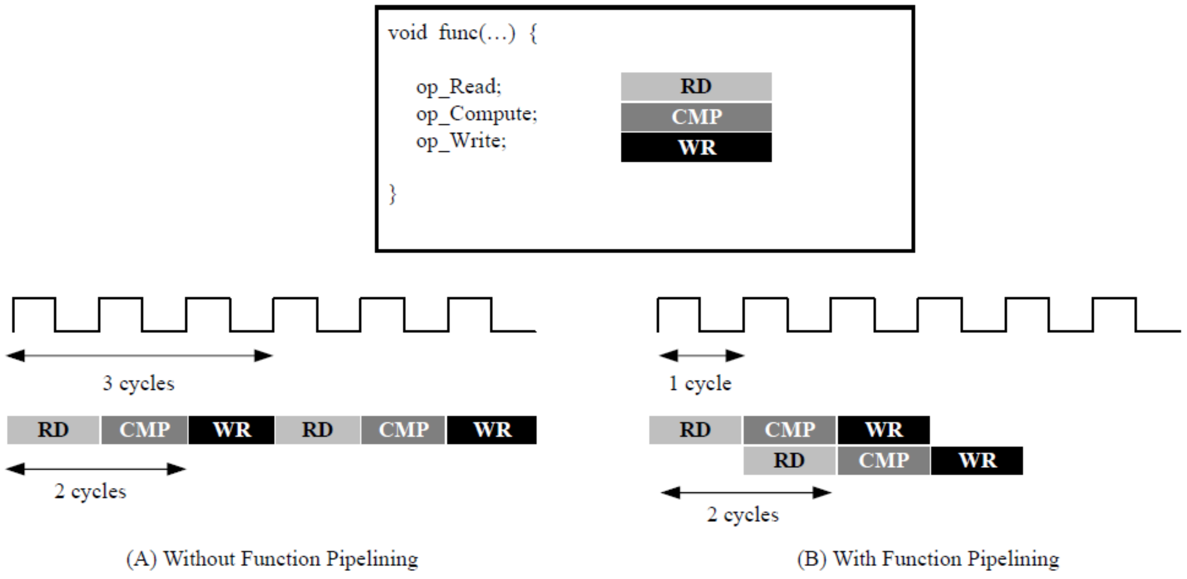


Figure 7.5 Function Pipelining

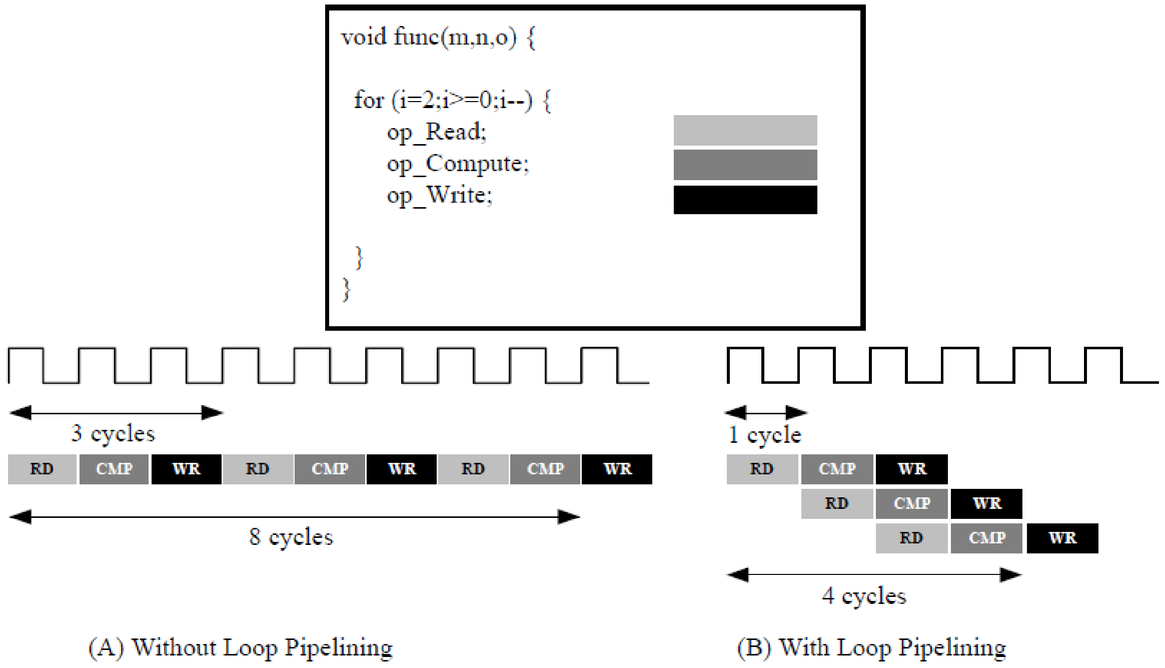


Figure 7.6 Loop Pipelining

7.2 Software Model Architecture

7.2.1 First Model

Firstly, we think in a simple way to build our high-level software model by making each convolution or building block of our CNN architecture (ShuffleNet) as a function which implements the computations needed. Each function can take its input data (input feature map) inside a C++ array and provide the output data inside another C++ array and so on as shown in Figure 7.7.

The advantage of this software architecture is easy to code but it has a disadvantage which make it impossible to implement. The disadvantage is that each array written in C++ will correspond to a memory in the generated RTL files or hardware so that we will have many memory instances each has big size of data which will not fit inside the FPGA as the BRAMs needed are larger than provided inside the FPGA so we have to find another software architecture.

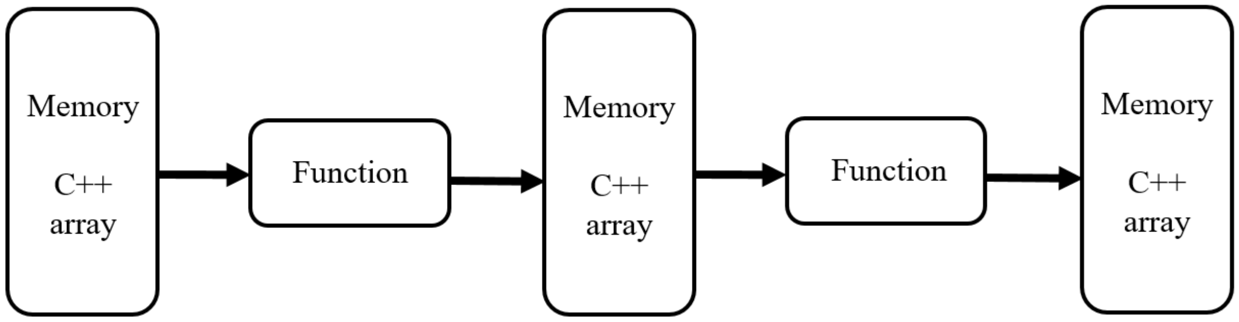


Figure 7.7 First Model

7.2.2 Second Model

The second and final model is based on shared memories (shared C++ arrays) which can be accessed by all functions in the design. These memories are implemented in a way to use suitable number of BRAMs making the generated hardware able to be fit and implemented on the FPGA. The class of the input photo which is the output of our CNN model will be provided by the FC layer (function) which is the last layer by comparing the 1000 output probabilities and selecting the index of the maximum probability as the output class.

As shown in Figure 7.8, we have 4 shared memories (C++ arrays) and each convolution, pooling layer and shuffling unit is implemented as a function. 1By1Conv and 3By3DWConv are grouped with shuffling unit to build a function called “Shuffle Group”. Last 1By1Conv and AVG Pooling are grouped together into one function also. All of the functions and memories (C++ arrays) are grouped into a top-level function called “Shuffle Model”. This top-level function is given to the Vivado high level synthesis tool synthesizer to generate the RTL files for the generated hardware after verify the software model functionality using C simulation.

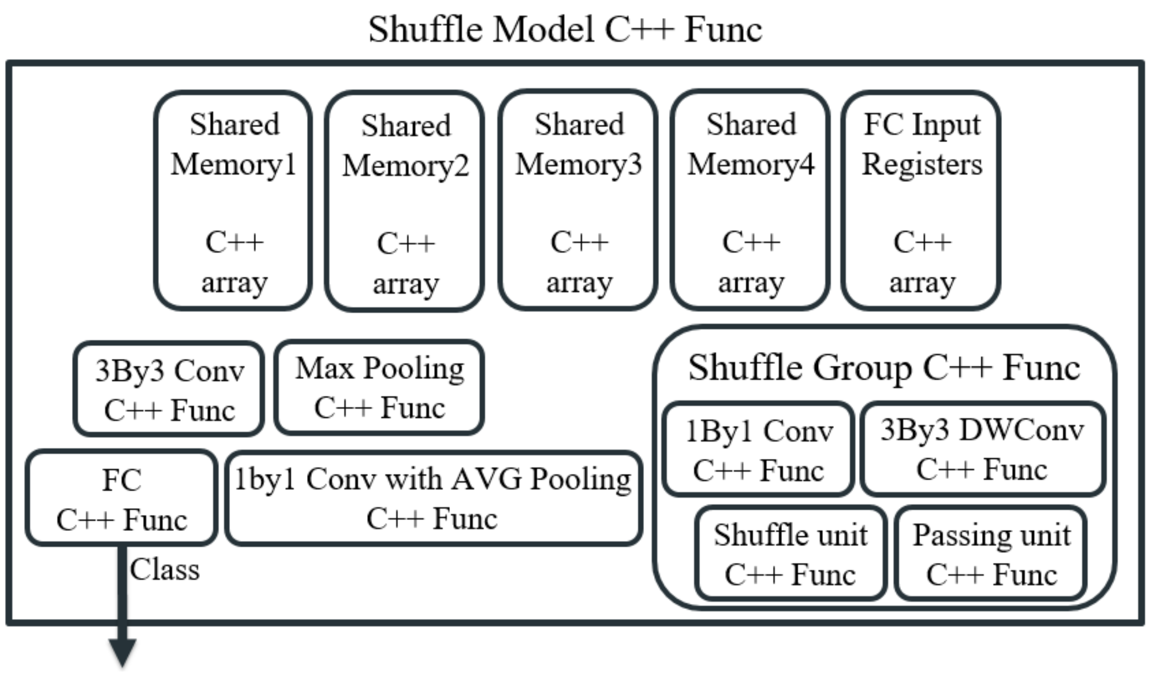
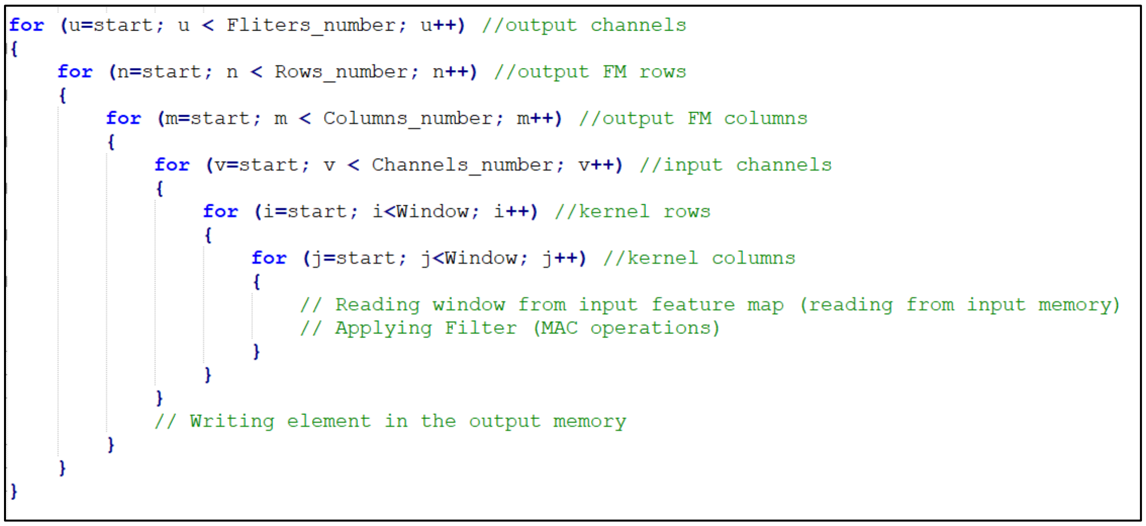


Figure 7.8 Final Model

7.2.3 Basic Elements Coding Style

7.2.3.1 Convolution and Pooling Layers

Convolution is coded using nested for loops. The outer nested loop is looping on the filters. The outer for loop has inside it 2 for loop to mainly fetch the window from the input feature map, one loop fetches horizontally and another fetch vertically. Inside these for loop, there is number of for loop equal to the dimensions of the convolution filter each loop is looping on one dimension. For example, if we have a 3By3 convolution with 3D filters we will have 3 for loops as shown in Figure 7.9, if it is depth wise convolution (2D) we will have 2 for loop and if it is 1by1 convolution we will have one for loop. Pooling layers has the same structure with different operation.



```
for (u=start; u < Fliters_number; u++) //output channels
{
    for (n=start; n < Rows_number; n++) //output FM rows
    {
        for (m=start; m < Columns_number; m++) //output FM columns
        {
            for (v=start; v < Channels_number; v++) //input channels
            {
                for (i=start; i<Window; i++) //kernel rows
                {
                    for (j=start; j<Window; j++) //kernel columns
                    {
                        // Reading window from input feature map (reading from input memory)
                        // Applying Filter (MAC operations)
                    }
                }
            }
            // Writing element in the output memory
        }
    }
}
```

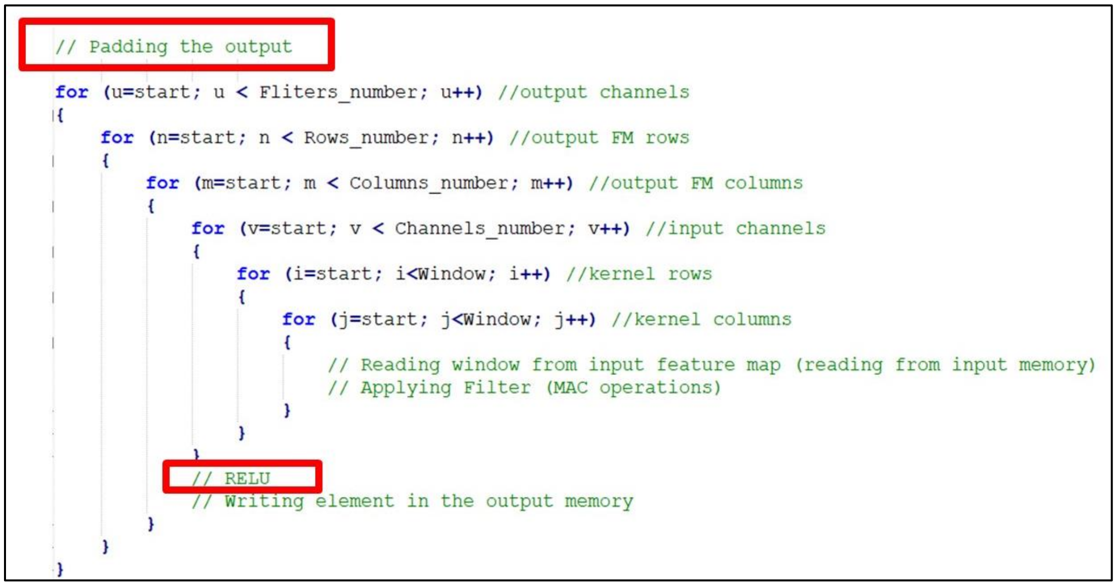
Figure 7.9 3By3 Convolution Code Structure

7.2.3.2 Padding and RELU

Padding is needed only before some convolutions only. But for the sake of simplicity, we will make the padding all the time meaning that input feature map for any convolution will be padded and the convolution will use the padding if needed. Padding is coded to be done on the output of the function so that when it became an input for the next function, the padding is ready. This code is written before the nested loops of the convolution or pooling computation.

RELU will not be a standalone function. It will be merged with any convolution have RELU after it. It will be only an if condition checks if the output element which has been calculated is zero or a positive number and will store zero or the output based on this condition. This will save the time of storing the output in a memory then reading all the data again to check if it greater than zero or not then store it again in memory as this operation will take a big time as reading from memory will be a bottleneck resulting in increasing latency.

In 3by3 convolution, we will make padding and RELU so the code structure shown in Figure 7.9 will be updated as shown in Figure 7.10.



```
// Padding the output
for (u=start; u < Fliters_number; u++) //output channels
{
    for (n=start; n < Rows_number; n++) //output FM rows
    {
        for (m=start; m < Columns_number; m++) //output FM columns
        {
            for (v=start; v < Channels_number; v++) //input channels
            {
                for (i=start; i<Window; i++) //kernel rows
                {
                    for (j=start; j<Window; j++) //kernel columns
                    {
                        // Reading window from input feature map (reading from input memory)
                        // Applying Filter (MAC operations)
                    }
                }
            }
            // RELU
            // Writing element in the output memory
        }
    }
}
```

Figure 7.10 3By3 Convolution Structure with Padding and RELU

7.2.3.3 Shuffling Unit

It is coded using nested for loops which send data from memory (C++ array) with old index (location) to another memory (C++ array) with new index (location) so that we can shuffle data as explained before.

7.2.3.4 Passing Unit

It passes data from memory to another in the same locations. The advantage of this function is to simplify the model by fixing the input memory for the shuffle block with stride 2.

7.3 Data Representation

As stated before, we will use fixed point representation in our hardware design and this is the case also in HLS as Vivado HLS tool give us the freedom to do any arbitrary precision data type so that we can specify any number of bits we needed in the design and not to be restricted with C++ data types. The smaller bit-widths result in hardware operators which are in turn smaller and faster. This is in turn allows more logic to be place in the FPGA and for the logic to execute at higher clock frequencies.

But we have a big problem. The problem is that ROM synthesis can be slow when using fixed point data type. In our case, we have large arrays which contain the filter weights of our CNN which will be synthesized as ROMs using BRAMs and as stated before we have around 2.3 million parameter and this turns into very long runtime of high-level synthesis.

As stated in Vivado HLS user guide, if we use integer data type instead of fixed point it will be much faster in synthesis. So, we will convert our code to use integer data type so that we can synthesis the model in a shorter time. It turns in giving us the freedom to make more trials with different directives and optimization techniques to get an optimized hardware which is like to be impossible with fixed point data type.

After reaching the desired results in terms of speed and area on FPGA, we will convert our model back to fixed point representation and do high level synthesis. In this case it will not be a problem to take large time as we do it once.

7.3.1 Integer Representation

In order to use integer data type, we extract all our CNN parameters and the input photo again in integer format. Then we use “short int” data type because it is 16-bit so that the resources and latency estimated for our design are very close to the results when using fixed point data type.

Unfortunately, integer data type cannot replace fixed point because of the difference in overflow handling after multiplication as fraction size doubles so that if we need to use smaller number of bits after multiplier, we will take bits on the middle but in integer it will take the least significant bits. As a result of that integer cannot replace fixed point, we will record the output class resulting from the model using integer data type as a reference. Then after any edit in the code we will compare the output class with this reference to be sure that the software models still behave functionally correct (C simulation).

After that we will synthesis the model and do RTL simulation (C/RTL co-simulation) and compare the results again with the integer output class reference to ensure that tool has done the synthesis correctly. If RTL simulation passed in integer model so it has a very big chance to pass in fixed point model but not vice versa. So, we need to convert back the model to fixed point datatype, synthesis it and do RTL simulation to ensure that the synthesis step performed correctly.

7.3.2 Fixed Point Representation

Vivado HLS tool has a header file called “ap_fixed.h” which we can use do define variables with a fixed-point format as following:

1) Include “ap_fixed.h” header file.
2) Declare a new variable which has a type of fixed point.

To declare a variable we should use the keyword **“ap_[u]fixed<W,I,Q,O,N>”** and the options inside the greater than and less than are used as following:

- **W & I**
  “W” is the total number of bits used in fixed point representation. “I” is the number of bits which is on the left side of the binary point which is the number of integer bits plus one bit sign as shown in Figure 7.11.

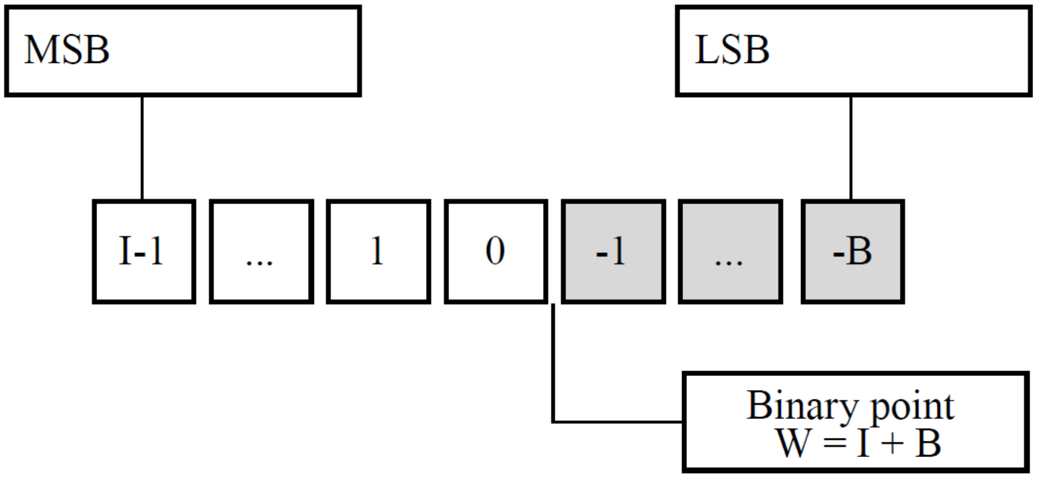


Figure 7.11 Fixed Point "W" and "I" options

- **Q**
  This dictates the behavior when greater precision is generated than can be defined by smallest fractional bit in the variable used to store the result. The default of this option is truncation and this is suitable in our case.

- **O**
  This dictates the behavior when the result of an operation exceeds the maximum (or minimum in the case of negative numbers) possible value that can be stored in the variable used to store the result. The default of this option is wrapping. In our case we will use the wrapping as default as it gives the same accuracy as saturation and this is tested by the software model on python on the 50000 photos of the validation set of ImageNet.

7.3.2.1 Data Format

To synthesis ROMs to hold our CNN parameters we define C++ arrays contain this data. We found that if we write this data in hexadecimal of binary format it will synthesis faster although we use fixed point data type (still slower than integer). But when we use these formats, the data are read wrong. Data are read as they are the maximum positive number can be hold by the fixed-point data type however the real data are much smaller and can be just a fraction making the output is not correct.

So, the only format to write data inside the arrays with is the decimal format like any fraction number to get the correct output result.

7.3.2.2 Multiplication and Addition Output Size

We must take care about the size of the variables when doing multiplication and addition operation as the output of theses operation is larger than the input to get the correct result from the software model.

In addition, we accumulate number of times equals to the size of filter of the convolution so we can define a new fixed-point variable which have a size larger than the input by “log2 M” where “M” is the number of additions. The new variable will put normally on the left-hand side of the addition statement and the tool will manage the difference in the sizes. Also, we can add 2 fixed point numbers with different sizes and the tool will manage the addition.

In Multiplication, we can multiply 2 numbers with different sizes also and the tool will manage the multiplication. But we must take care of that if we multiply 2 numbers one of Qn1.m1 and another Qn2.m2 the result must be stored in Qn1+n2.m1+m1.

7.4 Optimizations

After we finished writing the software model without any directives and simulate it using C simulation and validate the output. We synthesis the model with fixed point data type with 15-bit width and 8-bit fraction to analyze the generated reports from Vivado HLS tool and determine the speed and resources of the generated hardware so do optimizations if needed.

This model which we called **“Original Model”** after synthesis gets the following (detailed results mentioned in “Results” section):

- Latency: 690,776,599 clk cycles.
- Resources: 99% of BRAMs, 1% of DSPs, 2% of LUTs and, 0.005% of FFs on Virtex-7 VC709 Evaluation Platform.

It is very clear that we need to do strong optimizations to get acceptable speed in terms of frames/second.

7.4.1 Simplifying Expressions and Pipeline Optimization (Pipeline Model)

First optimization is done through 2 steps as following:

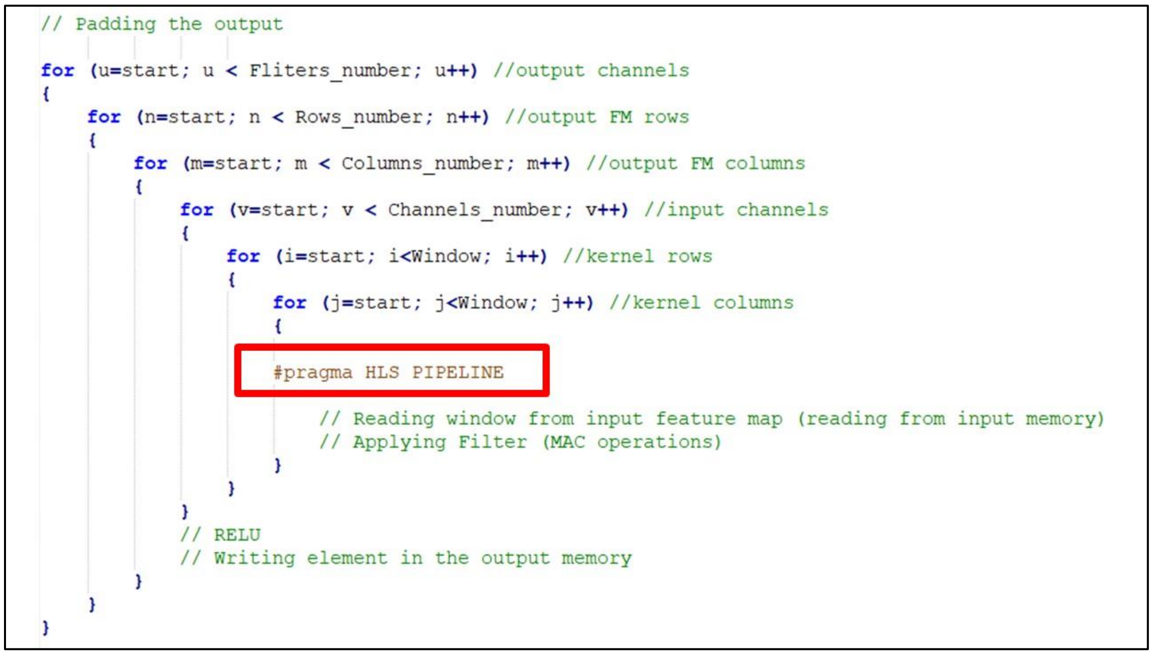
1) **Simplifying Expressions**
   While writing the code we coded some expressions which can ease the access of the arrays to fetch the window from input feature map, get the filter parameters, write output in the memory and shuffle the data. These expressions include multiplications and addition will be performed as hardware to generate address of the memory inside the hardware so if we simplify this expression latency will decrease and resources also can be decreased.

2) **Pipeline**
   We will use the "PIPELINE" directive inside the inner loop of each convolution, pooling, shuffling and passing unit so that the operations which take more 1 clk cycle can be pipelined and finished in clk cycles. For example, if we added this directive for 3by3 convolution function the code in Figure 7.10 will be updated as shown in Figure 7.12.

These 2 optimizations result in the following (detailed results mentioned in "Results" section):

- Latency: 237,717,509 clk cycles.
- Resources: 99% of BRAMs, 0.008% of DSPs, 2% of LUTs and, 0.006% of FFs on Virtex-7 VC709 Evaluation Platform.

These optimizations speed up the hardware by a factor of **2.9x**. But we still need to do more optimizations.


```
// Padding the output

for (u=start; u < Fliters_number; u++) //output channels
{
    for (n=start; n < Rows_number; n++) //output FM rows
    {
        for (m=start; m < Columns_number; m++) //output FM columns
        {
            for (v=start; v < Channels_number; v++) //input channels
            {
                for (i=start; i<Window; i++) //kernel rows
                {
                    for (j=start; j<Window; j++) //kernel columns
                    {

                    #pragma HLS PIPELINE

                        // Reading window from input feature map (reading from input memory)
                        // Applying Filter (MAC operations)
                    }
                }
            }
            // RELU
            // Writing element in the output memory
        }
    }
}
```
Figure 7.12 "PIPELINE" directive inside inner loop of 3by3 Convolution

7.4.2 Parallelism Based Optimization (Parallelism Model)

Parallelism is a common way used in hardware design to speed up the design and increase its performance. Unfortunately, the first technique doesn't get synthesized correctly as RTL simulation doesn't match the C simulation so we went with another technique. These 2 techniques are explained below in details.

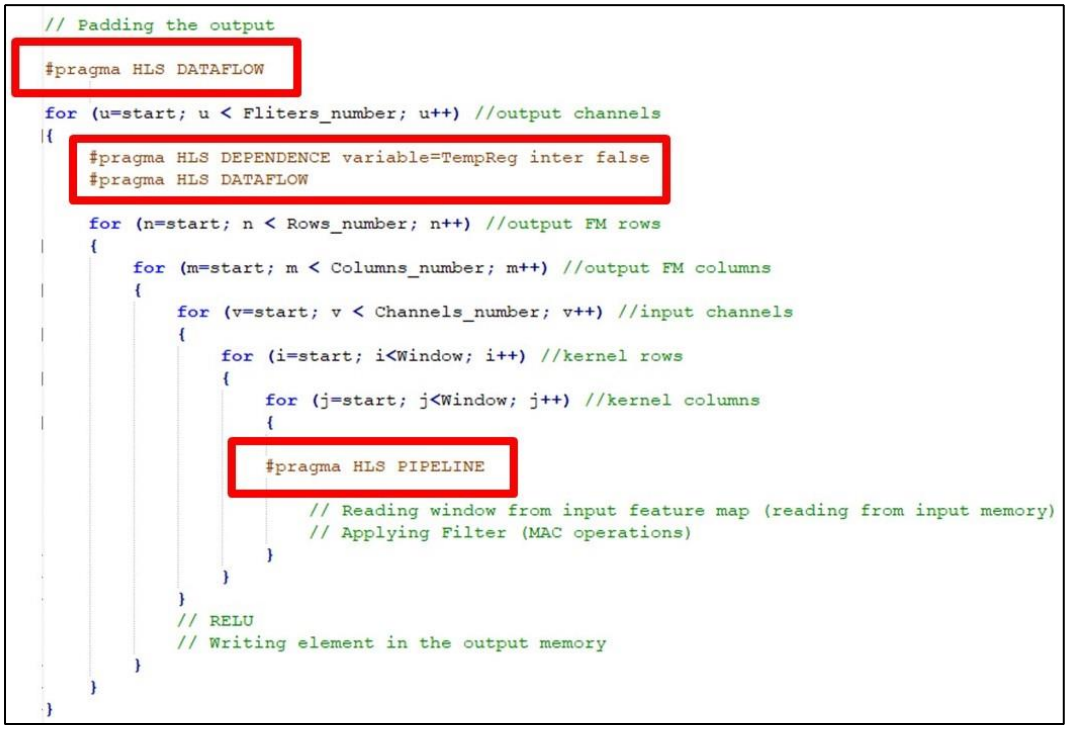
7.4.2.1 Parallelism on Filters

This is the first technique which is based on compute more than one filter of convolution in parallel or compute more than one window in pooling layer. As we take the same window from input feature map and compute the output of filters in parallel at the same time.

To implement this optimization, we will use some directives of Vivado HLS tool which will guide the tool to synthesis a parallel hardware without doing changes in the software model. The directives used are as following:

- **PIPELINE**: inside the inner loop to do all operation in 1 clk cycles.
- **DATAFLOW**: in order to make hardware in parallel so that we can compute the output of different filters simultaneously.
- **DEPENDENCE**: to remove the dependency on a specific variable which we accumulate the filter multiplications inside it so that all filters can be processed in parallel as we want to infer different hardware for every filter and parallel them using "DATAFLOW".

For example, if we have a 3by3 convolution so the code in Figure 7.12 will be updated as shown in Figure 7.13.



```
// Padding the output
#pragma HLS DATAFLOW
for (u=start; u < Fliters_number; u++) //output channels
{
    #pragma HLS DEPENDENCE variable=TempReg inter false
    #pragma HLS DATAFLOW
    for (n=start; n < Rows_number; n++) //output FM rows
    {
        for (m=start; m < Columns_number; m++) //output FM columns
        {
            for (v=start; v < Channels_number; v++) //input channels
            {
                for (i=start; i<Window; i++) //kernel rows
                {
                    for (j=start; j<Window; j++) //kernel columns
                    {
                    #pragma HLS PIPELINE
                        // Reading window from input feature map (reading from input memory)
                        // Applying Filter (MAC operations)
                    }
                }
            }
            // RELU
            // Writing element in the output memory
        }
    }
}
```

Figure 7.13 Parallel filter Code Structure.

The advantages of this optimization technique as following:

- We don't need to partition the input memory as the same input will be read by each filter. It is very suitable for us because the input memory is shared for all design so each block will need different partition.
- Filter ROM is easy to be partitioned if needed except for 1by1 Convolution and 3by3 DW Convolution as they hold weights for different stages with different sizes.

The disadvantages of this optimization technique as following:

- Output memory need to be partitioned to write all filters results at the same time.

There is some notes and modifications for this optimization technique as following

- We try to use UNROLL directive to unroll the for loop which loops on the filters the use DATAFLOW to make then in parallel but it doesn't work.
- When using these directives **without partition the output memory**, it works and latency **divided by 24 (number of filters as desired)** in 3by3 Convolution.
- We put the computation core (nested loops which perform the convolution) inside another function called core function.
- 1by1 Convolution and 3by3 DW Convolution nested for loops on max size (as they are repeated with different sizes as our CNN architecture need) to be suitable for directives.
- After integrating the whole model, directives didn't work inside all functions except 3by3 conv. So, we put internal memory inside every function takes the input first from the shared memory then do the rest of the function and it fix this problem without increasing memories. We think that tool optimized the internal memories as they don't add new logic for the code.
- Filter memories with these directives is doubled in size automatically by tool so that they can fetch the bias and weights in same cycle as we stored them in same memory (C++ array) so we split the them by storing weights in memory and bias in another memory and this solved the problem.

The most important step after any optimization is to do C/RTL co-simulation (RTL simulation) to ensure that high level synthesis step is done correctly as directives only affect the synthesis not the software model behavior.

Unfortunately, RTL simulation on 3by3 Convolution only and on the whole model after synthesis mismatches the C simulation resulting in false synthesis for this optimization. This can happen for many reasons, one of them is the misuse of the directives. So, we traced the model and found some things which may cause synthesis fails and tried to fix it as following:

- Removing internal memories as it may not correctly synthesized.
- Removing core function so that parallelism can be made without internal memories.
- Convert the accumulator variable into an array with number of filters so that every filter has its variable to store the result so after removing dependence it will synthesize correctly.
- Partitioning the output memory so that there will be no conflict while writing the filter outputs.

Unfortunately, after all these tries RTL simulation still gives wrong results so we need another technique with using directives to make synthesis done correctly.

7.4.2.2 Parallelism Inside Filters

The idea in this technique is to compute al the filter multiplication in the same cycle and then adding them together. If we **pipeline the multiplication and addition**, this will result in almost **compute all the filter in on cycle** but with some overhead.

One concept which make it suitable is that in convolution we take same window from input feature map and apply it on a filter then take another window and apply it on the same filter as so on. After we read all the input feature map and apply the first filter, we again read a window from input feature and apply another filter until we read all the input feature the read the input feature to the next filter and so on for each filter. So, we can read the input feature map window once and apply all filters sequentially as each filter after parallelism take only cycle.

In this way, we only **read the window once instead of reading same data multiple time** and that can give us the chance to read the window from input feature may **simultaneously** with applying the filter on the window read last time. In other words, we can **parallel reading window from input feature map with applying the filter on another window**.

The advantages of this optimization technique as following:

- We can reduce the latency by order of magnitude of the number of weights in the filter so as filter sized increase, speed will also increase.
- This is the common way to making parallelism in HLS as **we read some papers which used this technique in their design. So, RTL have a big chance to be synthesized correctly (and that what will end up with).**

The disadvantages of this optimization technique as following:

- As filter size increase, number of DSPs needed to parallel the multiplications increase so area and power increase but we will take care of them to be suitable.
- The reduction in latency will be small if filters size is small, which is the case in 3by3 DW Convolution and Max pooling layers.

To implement this optimization, we will use some directives of Vivado HLS tool which will guide the tool to synthesis a pipelined and parallel hardware. The directives used are as following:

- **UNROLL**: to unroll the inner nested loops which apply the filter on the window taken from the input feature map to be able to pipeline these operations.
- **PIPELINE**: to pipeline the operation unrolled which is multiplication and addition.
- **DATAFLOW**: to parallel between fetching a window from input feature map and applying the filter on another window.

Now, we will show the code structure after some modifications and the effect of optimization on every function as there is some differences and to know the amount of speed we get as following:

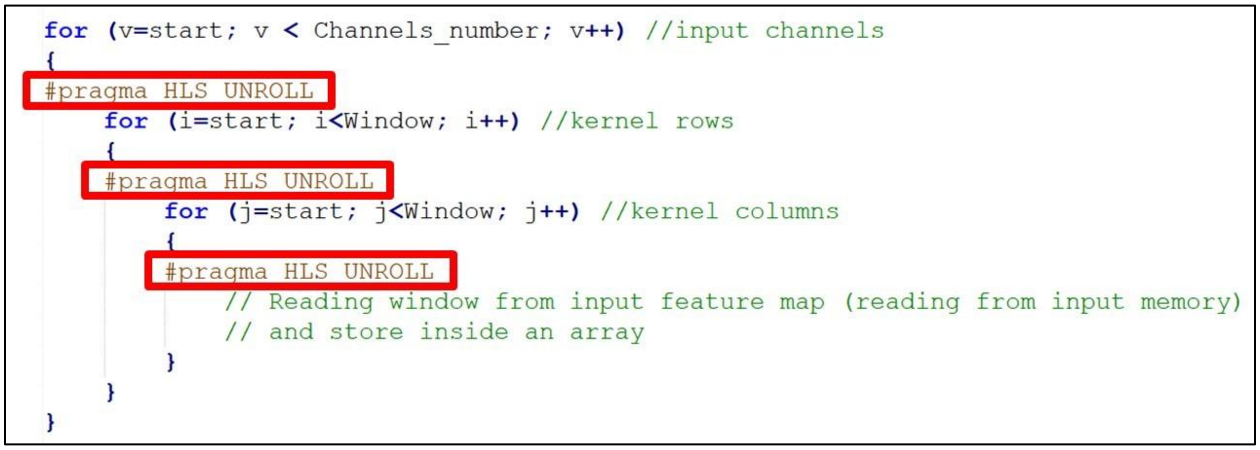
- **3By3 Convolution**

This function has been divided into 3 functions:

1) **Read Input Function**: to read an input window from the input feature using 3 nested for loop as filter is 3D dimensions. Then these for loop are **unrolled** so that we can parallel the reading operation by using 2 ports of the BRAM inside the FPGA so we can read 2 inputs simultaneously. The window is then stored in an array which is passed to the “Core Function”. Code structure for this function is shown in Figure 7.14.
2) **Core Function**: to apply filters on the window passed inside an array for “Read Input Function”. We need an outer loop to loop on number of filters inside it we have nested for loops to apply the 3D filter three for multiplication and one for accumulation and adding bias which are **unrolled** to be pipelined by the directive **“PIPELINE”** inside the outer loop as shown in Figure 7.15.
3) **3By3Conv Function**: to do padding then call the last 2 function inside 2 nested for loop which is looping on the input feature map to fetch all data and apply the parallelism on the last 2 functions using **“DATAFLOW”** as shown in Figure 7.16.

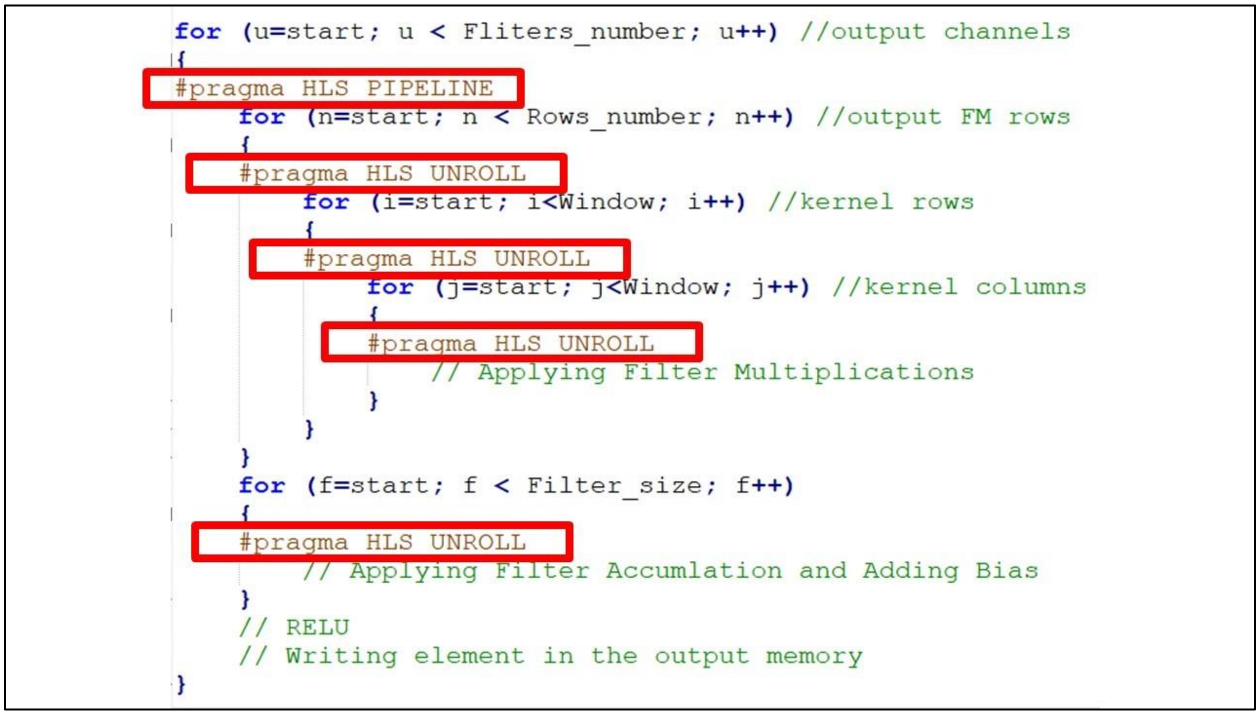
Filter and bias memories are split and forced to be implemented as LUTs as they are small.

As a result, **3by3 Convolution is speed up by a factor of 26x** as the filter size is 27 and we have some overheads resulting from “PIPELINE”, for loops and “DATAFLOW” directives.



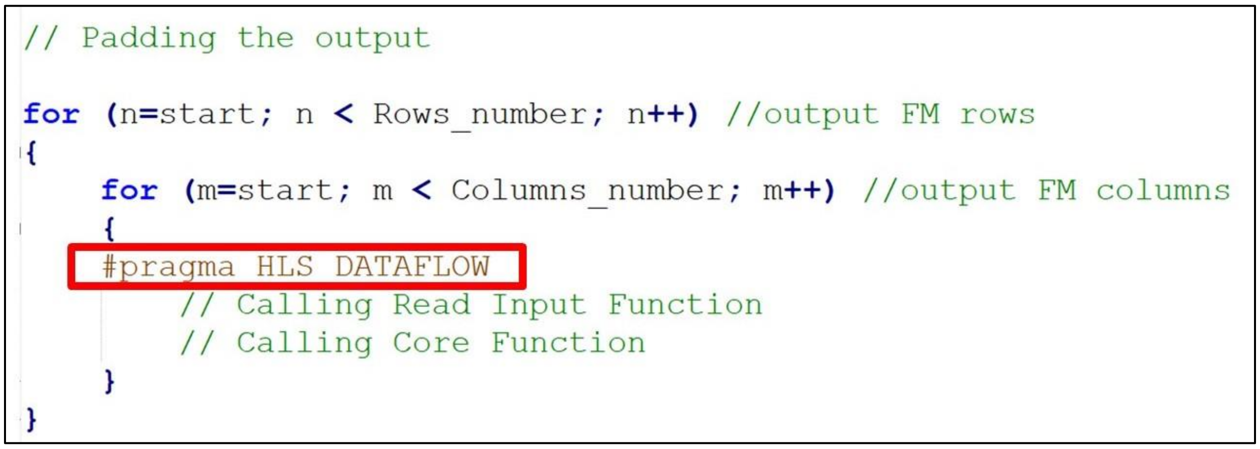
```
for (v=start; v < Channels_number; v++) //input channels
{
#pragma HLS UNROLL
     for (i=start; i<Window; i++) //kernel rows
     {
     #pragma HLS UNROLL
          for (j=start; j<Window; j++) //kernel columns
          {
          #pragma HLS UNROLL
               // Reading window from input feature map (reading from input memory)
               // and store inside an array
          }
     }
}
```

Figure 7.14 3By3 Convolution Read Input Function



```
for (u=start; u < Fliters_number; u++) //output channels
{
#pragma HLS PIPELINE
    for (n=start; n < Rows_number; n++) //output FM rows
    {
    #pragma HLS UNROLL
        for (i=start; i<Window; i++) //kernel rows
        {
        #pragma HLS UNROLL
            for (j=start; j<Window; j++) //kernel columns
            {
            #pragma HLS UNROLL
                // Applying Filter Multiplications
            }
        }
    }
    for (f=start; f < Filter_size; f++)
    {
    #pragma HLS UNROLL
        // Applying Filter Accumlation and Adding Bias
    }
    // RELU
    // Writing element in the output memory
}
```

Figure 7.15 3By3 Convolution Core Function



```
// Padding the output

for (n=start; n < Rows_number; n++) //output FM rows
{
    for (m=start; m < Columns_number; m++) //output FM columns
    {
    #pragma HLS DATAFLOW
        // Calling Read Input Function
        // Calling Core Function
    }
}
```

Figure 7.16 3By3 Convolution Function

- **Max Pooling & 3By3 DW Convolution**
  These two functions are 2D so they will have same structure but with different operation. The structure of these functions will be similar to 3By3 convolution except we will add a directive **“INLINE”** at the start of the **“Read Input Function”** and **“Core Function”** and remove “DATAFLOW” directive as it gives better results in terms of speed. Also, the “Core Function” will not include the loop which loops on the filter as the window will only be applied on one filter as filter works on one channel so it will be the 2 loops which loop on the window to fetch it and these two functions will be called inside them.

  3By3 DW convolution is repeated many times with different sizes so the header of the for loops will have variable upper bound depends on the calling which determine the size inside the “Shuffle Group” function as long as variable bounds don’t affect directives badly some directives restricted the for loop must have fixed upper bound. Also, because the window is always 9 elements so we don’t need different functions as the case in 1by1 convolution inside the shuffle units.

3By3 DW Convolution filter weights and biases are split in different memories. Filter memories also are partitioned using **“ARRAY_PARTITION, cyclic” with factor 9** to be able to fetch all weights in same clock cycle. Filter and bias memories are forced to be implemented as BRAMs using **“RESOURCE”** directive.

As a result, **Max Pooling is speed up by a factor of 2.3x and 3By3 DW Convolution is speed up by a factor of 1.32x**. This factor is small as the filter size is small only 9 elements and this is the drawback of this optimization technique.

- **1By1 Convolution (inside the shuffle units)**
  This function is repeated with different number of filter size so that we will split it into 3 functions depending on the number of filters. So, **we will get 3 functions operate on 58, 116 and 232 number of filters**. As the input feature map size varies, we don’t need to split functions on it as we can make the for loop bounds variable like 3by3 DW convolution. Splitting on the number of filters is mandatory to avoid large overhead and large area as we will design for the biggest size of for loops and largest number of DSPs needed.

  Every function will be similar to 3By3 convolution except that we can’t use “DATAFLOW” directive as we read from all the shared memories and that will violate the principal of one producer one consumer of the “DATAFLOW” directive. As a result of this we don’t need to make “Read Input Function” and “Core Function” and then call them inside a “1By1Conv Function” instead we will take their content and write the directly inside the “1By1Conv Function” in the same place of calling these functions.

  1By1 Convolution filter weights and biases are split in different memories. Also filter weights and biases are split for each number filters so **we have 6 memories, 2 memories per function** size to make it possible to fetch weights in parallel as partitioning will vary.
  In first stage of 1by1 convolution we have input with 24 channels but 58 filter so the filter has a depth of 24. The rest of filters whose number are 58 have a depth of 58. So, we **extend the filter with depth 24 with zeros** to have a depth of 58 and then control the number of elements which applied for a filter by if condition to ensure the functionality. This extension is mandatory for array partitioning as we need to partition all filters in the same memory with same factor and all 58-channel filter will be in the same memory to achieve best resources on FPGA.

  So, filter weights memories will be portioned using **“ARRAY_PARTITION, cyclic” with factor 58 for the first function size, 116 for the second and 232 for the third.** All filter and bias memories are forced to be implemented as BRAMs using **“RESOURCE”** directive.

As a result, **1By1 convolution is speed up by a factor of 27.7x** on average as we take the average latency of the 3 new functions and compared it with the old function.

- **Last 1By1 Convolution (after the shuffle units) and AVG Pooling**
  As stated, before these functions are merged as AVG Pooling can take the output of the 1by1 convolution after if condition which is doing the effect of RELU and multiply it by 1/49 as a hard coded value to avoid division in hardware then accumulate inside the memory with the new output.

  So, the 1by1 convolution will be exactly the same structure as 3by3 convolution except here the filter is 1D so we need only one loop for multiplication and here we can use the same for loop to accumulate and adding the bias as the problem which make us divide them into 2 for loops are that there is pipeline dependency but here in this 1by1 convolution only there is no pipeline dependency as the case in 3by3 DW Convolution there is no pipeline dependency.

  Filter weights and biases are split in different memories. Filter memories also are partitioned using **“ARRAY_PARTITION, cyclic” with factor 464** to be able to fetch all weights in same clock cycle. Filter and bias memories are forced to be implemented as BRAMs using **“RESOURCE”** directive.

As a result, **Last 1By1 convolution and AVG Pooling function is speed up by a factor of 365x** as the filter size is 464 and we have some overheads resulting from “PIPELINE”, for loops and “DATAFLOW” directives.

- **Fully Connected**
  It is a little bit different as it is only 3 for loops one for neurons (outer loop) and one for weights multiplication and one loop for accumulation (2 inner loops at the same level). The inner loop which is for multiplication of weights is **unrolled with factor of 256**. And the accumulation loop is **fully unrolled**. Then the outer loop is pipelined with **“PIPELINE”** directive.

  Multiplication for loop is unrolled with only 256 because we don’t need it fully as it will take 1024 DSPs so we use 256 with slightly increase in latency which can be ignored nut we **decrease the number of DSPs significantly**. Accumulation for loop is unrolled fully to decrease the overhead on parallelism and to do all addition in almost one cycle after filling the pipeline.

  There as a 1024-word array to store the input of fully connected from 1by1 convolution and average pooling. This array is partitioned completely with **“ARRAY_PARTITION”** directive to be suitable for pipelining.

  Neurons weights and biases are split in different memories. Neurons memories also are partitioned using **“ARRAY_PARTITION, cyclic” with factor 256** to be able to fetch all needed weights in the same clock cycle. Neurons weights and bias memories are forced to be implemented as BRAMs using **“RESOURCE”** directive.

As a result, Fully Connected **is speed up by a factor of 114.2x** as we unrolled by a factor of 256 and we have some overheads resulting from “PIPELINE” and for loops.

Parallelism based optimizations results in the following (detailed results mentioned in “Results” section):

- **Latency: 21,126,394 clk cycles in estimation and 9M in real RTL simulation**.
- Resources: 91% of BRAMs, 44% of DSPs, 61% of LUTs and, 21% of FFs on Virtex-7 VC709 Evaluation Platform.

It is clear that parallelism is very efficient optimization as it speeds up the hardware by a **speed up factor 26x**. But we still can get better results.

7.4.3 Speed Optimization

This optimization consists of two main parts, the first is removing any logic which another technique can replace, and the second is solving the drawbacks of parallelism on filter technique.

7.4.3.1 remove time-consuming logic

The passing unit is added to simplify the model by fixing the input memory, but now we need to increase the model speed, so we can remove the passing unit but increase the complexity of the model a little bit by adding extra modes depending on the input memory. This will save a lot of time as we call the passing unit 12 times.

The shuffling unit shuffles channels of the feature map. As all channels are padded, so we don’t need to shuffle channels with padding, we can only shuffle channels without padding. this will save some time. For example, size of a padded channel is 16*16=256, and size of the unpadded channel is 14*14=196, we have 232 channels of this size and we shuffle them 8 times across stage 3 of the model. which will save 8*232*(256-196) = 111360 operations in this stage only.

The padding is added to the output of the current function, so if the next function doesn’t need padding, we will not add this extra padding. This will save some time.

We add padding with respect to function output, but we are using shared memories so we can put padding in memories only 1 time until the size changes. this will save the time of padding the same-size channels many times.

These 4 optimizations result in the following:

- Latency: 17,930,226 clock cycles in estimation and 7.5M in real RTL simulation.
- Resources: 91% of BRAMs, 44% of DSPs, 65% of LUTs, and 21% of FFs on Virtex-7 VC709 Evaluation Platform.

7.4.3.2 speed up the Max pooling and 3by3 DW convolution

As we clarify before that the speed-up factor of the max pooling and 3by3 DW convolution is small due to filter size, so we need to speed up them with other techniques in addition to parallelism on filter technique.

First, we apply the pipeline directive with the inline directive on these two functions which make pipelining between the input reading, processing, and output writing. This technique speeds up the max pooling and 3by3 DW convolution by a speed-up factor of 1.4x. This technique has a huge effect on the model as we call the 3by3 DW convolution function 19 times. we also apply this technique to 3by3 convolution to make it faster.

This technique results in the following (detailed results mentioned in “Results” section):

- Latency: 12,982,523 clock cycles in estimation and 5.6M in real RTL simulation.
- Resources: 91% of BRAMs, 61% of DSPs, 71% of LUTs, and 28% of FFs on Virtex-7 VC709 Evaluation Platform.

This technique has upset limitations, it increases the DSPs of 3by3 convolution from 30 DSPs to 650 DSPs for unknown reasons as it’s applied to 3by3 DW convolution and doesn’t increase the number of DSPs anymore. The reason may be related to the depth of the channels of 3by3 convolution which is not in 3by3 DW convolution. As a result of this limitation, we remove pipeline and inline directives from the 3by3 convolution function and return back to the dataflow directive. Also, this increase of DSPs happens when we applied this technique to 1by1 convolution which strengthens the idea of relation to the depth of the channels.

Another limitation is the interval of this pipelining technique, we read 9 elements of input in 5 clock cycles with dual ports, then process these inputs parallelly in 1 clock cycle, then write the output in 1 clock cycle. Now it’s clear that the interval is 5 clock cycles which are for reading input and can’t reduce due to limitations of BRAM ports. Despite we can reduce them by partitioning the memories, this is very difficult as memories are shared memories.

As a solution for interval limitation, we can take two windows in the same iteration. the two windows are shared six elements of inputs (if stride = 1) and 3 elements of inputs (if stride = 2). Let’s consider the case of stride = 1 as shown in Figure 7.17, we need to read 12 elements of input in **6 clock cycles** with dual ports, then process these inputs parallelly in 1 clock cycle, then write the output in 2 clock cycles. this solution will make the interval 6 clock cycles for 2 outputs instead of 5 clock cycles for each 1 output. This solution speed up the max pooling and 3by3 DW convolution by an extra speed-up factor of 1.66x. This solution also has a huge effect on the model as we call the 3by3 DW convolution function 19 times. we call this solution the dual window technique.

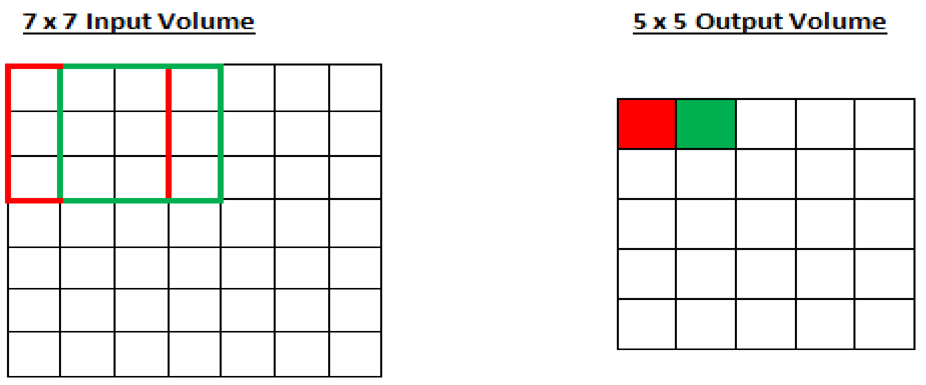


Figure 7.17 Convolution with stride=1

We can apply the dual window technique to any function to reduce the latency of any function by calculating 2 outputs by 2 outputs especially the functions that have shared elements of the input. We applied the dual window technique to 3by3 convolution, max pooling, 3by3 DW convolution, and 1by1 convolutions of shuffle group.

We can also apply the multiple windows technique (take more than dual windows) in the max pooling and 3by3 DW convolution to speed up them more, as pipelining is applied to them which makes the interval of iteration is the interval of reading input only, and each window consumes low DSPs (9 DSPs for 3by3 DW convolution and no DSPs for max pooling). So, we took four windows in each iteration of the max pooling and 3by3 DW convolution.

These techniques result in the following:

- Latency: 10,814,741 clock cycles in estimation and 3.7M in real RTL simulation.
- Resources: 88% of BRAMs, 49% of DSPs, 70% of LUTs, and 21% of FFs on Virtex-7 VC709 Evaluation Platform.

To reach higher frames per second, we took four*two windows in each iteration of the max pooling and 3by3 DW convolution. four windows in the horizontal direction and two windows in the vertical direction.

The order of the loops affects the latency of the model a little bit. It’s better to have the outer loop with the minimum iterations and the inner loop with the maximum iterations if the order of the loops doesn’t affect the functionality of this code. this effect happens due to overheads of transitions between loops.

For example, if we have 2 loops, first with 5 iterations and second with 10 iterations, assume the overhead of transition is 2 clock cycles and the core takes 7 clock cycles. if the outer loop with 10 iterations, then total clocks = [(7+2) *5 + 2] * 10 = 470 clock cycles. But if the outer loop with 5 iterations, then total clocks = [(7+2) *10 + 2] * 5 = 460 clock cycles. this is a small difference, but with bigger iterations and calling the function many times, this makes a noticed difference.

We lower the fully connected parallelism factor from 256 to 32 to save DSPs and only increase latency from 11K to 39K clock cycles which is a small increase.

These edits result in the following:

- Latency: 10,902,810 clock cycles in estimation and 3.2M in real RTL simulation.
- Resources: 88% of BRAMs, 42% of DSPs, 72% of LUTs, and 21% of FFs on Virtex-7 VC709 Evaluation Platform.

7.4.4 Resources and Area Optimization

After we have finished software modeling optimization (Dynamic Quantization for each layer), we applied its results in the HLS model. We multiply avg pooling by 1/4 instead of 1/49, We divide fully connected weights on 49 and fully connected bias on 4. Table 7.1 below concludes the quantization of each layer:

Table 7.1 Dynamic quantization of layers

Layer name	width (bits)	fraction (bits)
photo	8	5
3by3 conv. output	10	6
Max-pooling output	10	6
Shuffle group output	12	8
1by1 conv and avg pooling	9	5
fully connected	9	5

Also, we quantized weights of the HLS model dynamically as a result of software modeling optimization. Table 7.2 below concludes the quantization of each layer's weights:

Table 7.2 Dynamic quantization of weights

Weights	width (bits)	fraction (bits)
3by3 conv. weights	12	9
Shuffle group 1by1 conv. weights	11	9
Shuffle group 3by3 conv. weights	15	8
Shuffle group biases	13	9
1by1 conv. weights	9	8
fully connected weights	9	8

This dynamic quantization led to reducing the number of used BRAMs, LUTs, and FF as detailed in “Results” section as named “Final Model”.

7.5 Verification

Our verification process has 3 steps.

7.5.1 HLS C Simulation

This is a pre-synthesis simulation which targets verifying the functionality of the software model itself before any consideration of the hardware which will be generated. It like running any other C program on an IDE. It is done by writing a testbench in high level C++ language where we can call the top function of the model and give inputs and take outputs and check if the outputs are correct or not and print messages which show the status of the testbench pass or fail.

In our case we validate that the software model gives the same class output for the input photo as our pretrained python model. This was done on more than one photo and the resulting accuracy is almost the same as python model. So, our software model function validation step passes.

Also, we use this simulation when we edit the model for any optimization as we edit some functions in the code so that we need to ensure that the functionality of the software model still correct.

We did this step firstly on the model written with floating point data type to be easy to debug and compare results with python then we can change the data type to fixed point or integer. This is simply done using “typedef” keyword in C++ high level language.

In integer model, as stated before, it can’t replace the fixed point so after checked the functionality of the model using floating point datatype, we change it to integer and rerun C simulation to get an output which will be our golden reference after that in integer to ensure after any edit the software functionality is still correct.

In order to run the C simulation inside Vivado HLS, you will only press on the icon shown in the Figure 7.18. Also, we can use the debugging option to analyze the flow of the data through whole the software model to get the point where it fails by choosing the debugging option in the window opened after pressing the icon as shown Figure 7.19 which will take you to the debugging perspective.

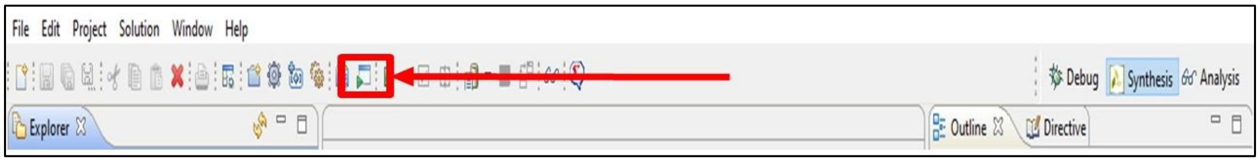
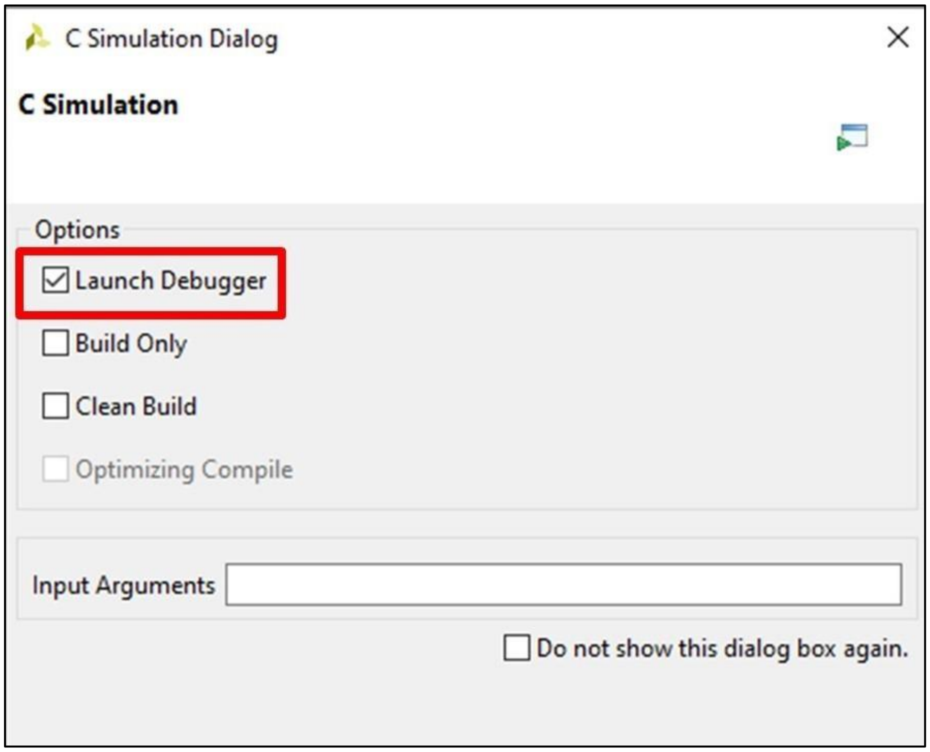


Figure 7.18 HLS C simulation Icon



- [✓] Launch Debugger
- [ ] Build Only
- [ ] Clean Build
- [ ] Optimizing Compile

Input Arguments

- [ ] Do not show this dialog box again.

Figure 7.19 Debugging Option in C simulation

7.5.2 HLS C/RTL co-simulation

This is a **post-synthesis simulation** which targets verifying the functionality of the synthesized RTL with the original C-based testbench.

We use this kind of simulation after any edit in software model or directives so that we ensure that the tool still can synthesis correctly the model after editing. It is very important step to **validate the generated RTL functionality**.

We use it mainly with integer version of the model as we make many edits and trials with integer model, as it is fast in synthesis step, and to ensure the directives works as desired and check the **real latency of the generated RTL** as the latency in HLS reports is just an estimation as we use **"LOOP_TRIPCOUNT"** directive to specify average number of iterations for the for loops with variable upper bound and we overestimate this average to make a stronger optimization to get better results.

In order to run the C/RTL co-simulation inside Vivado HLS, you will only press on the icon shown in the Figure 7.20. We can choose which simulator to go with also if we need to save a waveform or show it while simulation running from options in the window opened after pressing on the icon as showing in Figure 7.21.

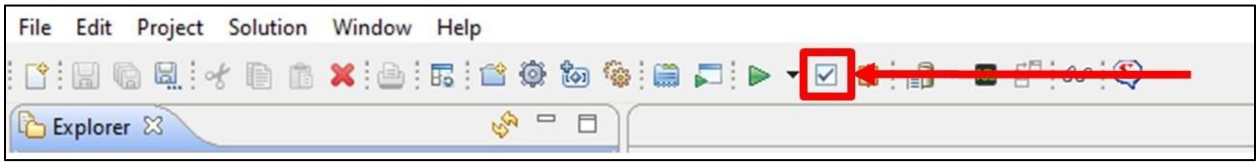
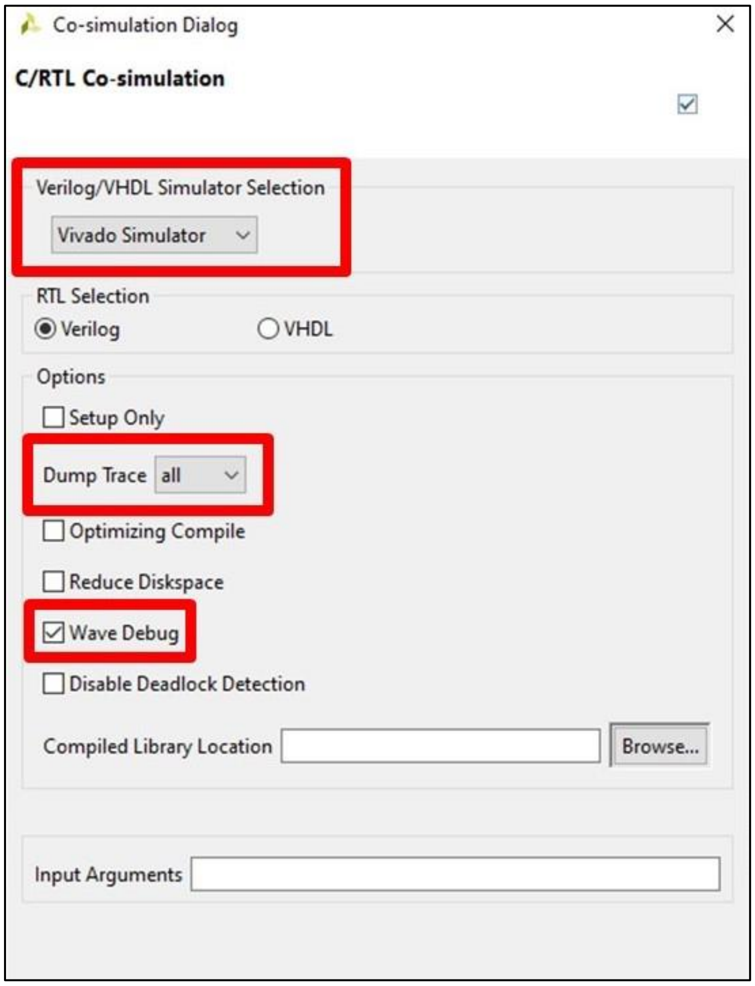


Figure 7.20 C/RTL co-simulation Icon



Co-simulation Dialog

C/RTL Co-simulation

Verilog/VHDL Simulator Selection: Vivado Simulator

RTL Selection: [✓ Verilog] VHDL

Options

- ☐ Setup Only
- Dump Trace: all
- ☐ Optimizing Compile
- ☐ Reduce Diskspace
- ☑ Wave Debug
- ☐ Disable Deadlock Detection

Compiled Library Location: Browse...

Input Arguments

Figure 7.21 C/RTL co-simulation simulator and waveform options

There is a problem with this type of simulation which is when we run C/RTL co-simulation, tool starts C simulation automatically before C/RTL co-simulation and if print an output in a text file C simulation print it firstly then when running C/RTL co-simulation it can't overwrite this text file so we could think that the output is identical to C simulation but it doesn't. We can solve it by printing data in console window. But this solution is limited as the output of some layers is very large to print in console. So, we will synthesis the whole model and print the output class in the console and check if it matches the C simulation or not.

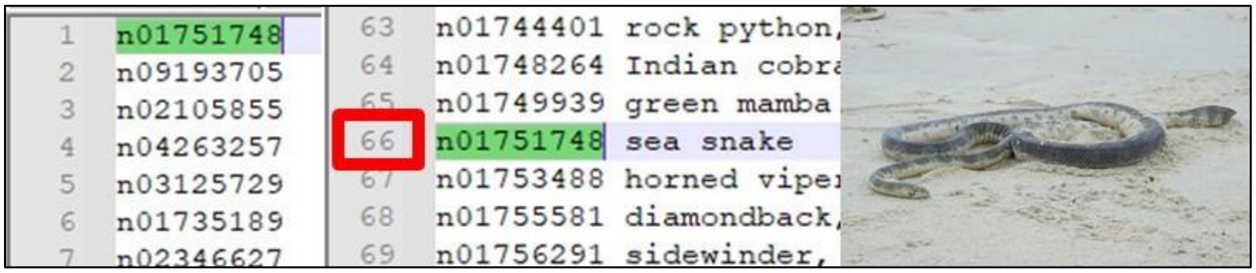
7.5.3 Vivado Behavioral Simulation

In C/RTL co-simulation, fixed point data type gives an error so we solve this by exporting our generated hardware RTL files (exporting will be explained in synthesis and implementation section) then go to Vivado design suite and perform functional simulation using a Verilog testbench to validate the functionality of the generated hardware and determine the real latency of the design.

Also, we use this simulation to make sure the generated RTL file which will be synthesized by Vivado synthesizer behaves as intended before synthesis.

In testbench we instantiate the design under test with the IP exported from Vivado HLS then we generate a clock and reset the design then we start it by making the start signal high and wait for the output class to be printed in the waveform.

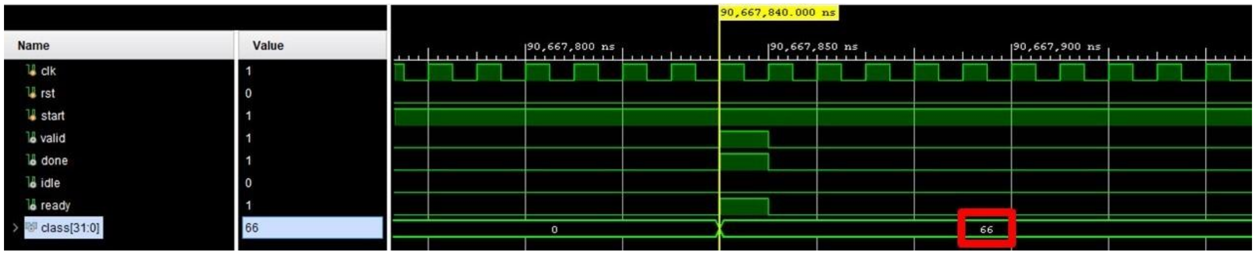
We put a real photo inside our design of the software model and synthesis with it so that the hardware can work on this photo and output its class which is 66 (sea snake) as this is the first photo inside the ImageNet validation set. The class of the first photo is shown in Figure 7.22. Behavioral simulation result is shown in the Figure 7.23.



```
1  n01751748
2  n09193705
3  n02105855
4  n04263257
5  n03125729
6  n01735189
7  n02346627
```

```
63  n01744401 rock python,
64  n01748264 Indian cobr
65  n01749939 green mamba
66  n01751748 sea snake
67  n01753488 horned vipe
68  n01755581 diamondback,
69  n01756291 sidewinder,
```

Figure 7.22 Testing Photo with its ID and Class Number



Name	Value
clk	1
rst	0
start	1
valid	1
done	1
idle	0
ready	1
class[31:0]	66

Figure 7.23 Functional Simulation Result

7.6 Synthesis and Implementation

Before we can start synthesis and implementation. We need to export our design from Vivado HLS. We will export the design as an IP as following:

1) Pressing on “Export RTL” icon as shown in Figure 7.24.
2) Choosing IP Catalog with RTL written in Verilog as shown in Figure 7.25.

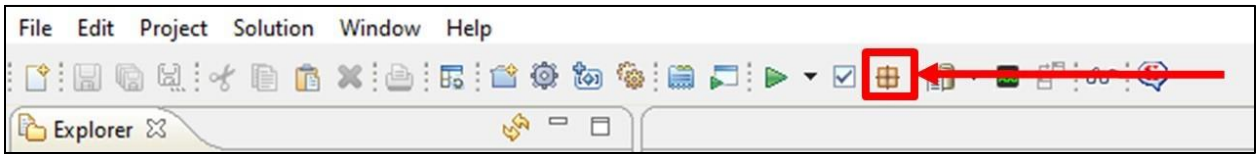
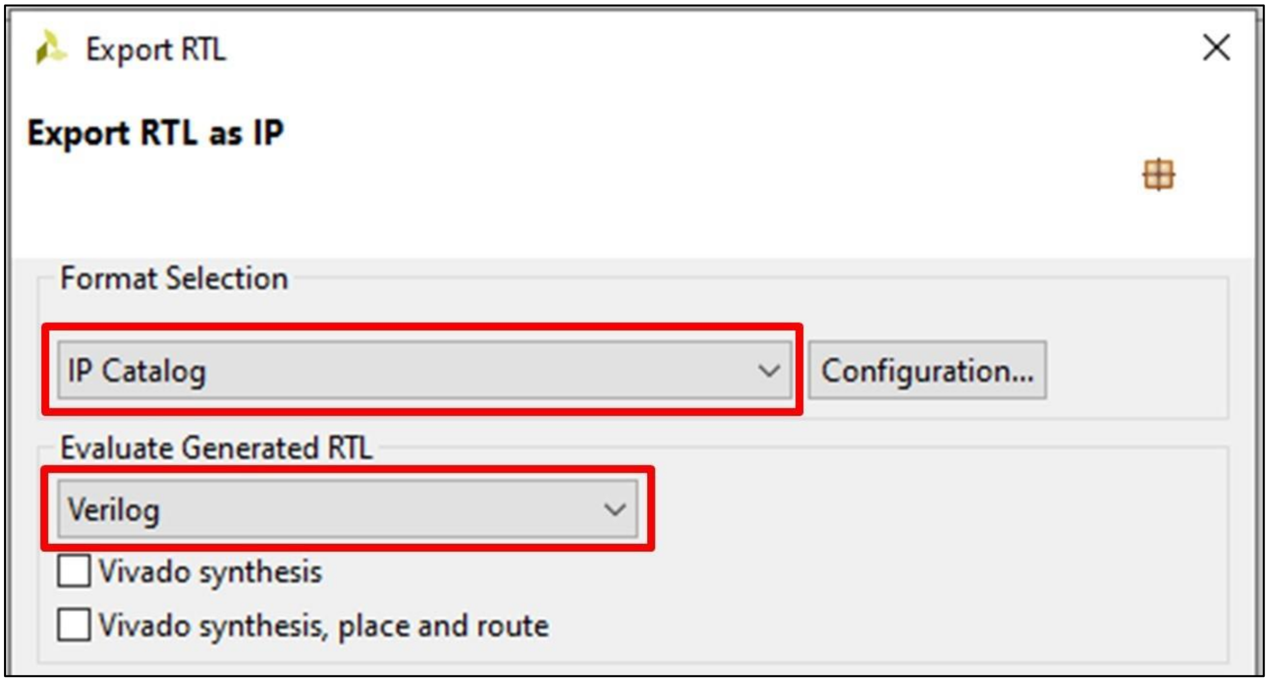


Figure 7.24 Export RTL icon


Export RTL

Export RTL as IP

Format Selection

IP Catalog

Evaluate Generated RTL

Verilog

☐ Vivado synthesis

☐ Vivado synthesis, place and route

Figure 7.25 Export RTL Options

Once the IP has been exported, we need to add it inside Vivado design suite project to work with it so we will add the IP as following:

1) Open project settings.
2) Choose repository from IP.
3) Add the exported IP from the solution inside the HLS project.

After the steps above, we can use the IP as normal as any other IP exists in IP catalog. We will click on the IP and generate one and we will choose **"Global"** option so that we can synthesis the IP by ourselves and determine our constraints like the operating clock.

One important step in Vivado HLS is choosing the clock and constraints which will has a big effect in the combinational delay of the generated hardware. As we target a high clock frequency as the design can meet so we can put a period of the clock inside Vivado HLS with 10ns. But Vivado HLS in most of cases couldn't meet the desired clock so we tight it and make it 7ns. We choose this value because the latency increased as we decreased the clock period as the loop take more clock cycles to perform same operations. At 7ns the estimated clock will be around 7.5ns which is smaller than 10ns at the same time the increasing in latency will be small and acceptable. Clock uncertainty is kept as default 12.5%.

7.6.1 Synthesis

In this step, the exported IP which is a generated RTL files will be synthesized into gate level netlist. In our case we will use the default settings of Vivado design suite to synthesis our design.

We will just create a constraints file which has constraints on the clock which we target and the jitter to make synthesis more robust and efficient.

7.6.2 Implementation

In this step, the gate level synthesis which is resulted from the synthesis step will be implemented using the cells inside the FPGA then the tool will place the design and route between them. This is vital step as it has large effect for the design to meet timing constraints and operate on the desired clock frequency without and setup or hold timing violations.

So, we develop our strategy in Vivado project implementation settings to make implementation more aggressive and to do more effort to meet timing. The directives with options used are as following:

- *opt_design -directive ExploreWithRemap*
- *place_design -directive ExtraNetDelay_High*
- *phys_opt_design -directive Aggressive Explore* (post-place)
- *route_design -directive NoTimingRelaxation*
- *phys_opt_design -directive AggressiveExplore* (post-route)

7.7 Results

7.7.1 HLS Results

In this section, we will discuss the results of HLS reports which is only an estimation for the final results after synthesis and implementing the design on Vivado design suite.

7.7.1.1 Latency

We will compare between "Original Model" and all Optimizations (as their names as in optimization section) to know how the optimizations affected the design and how much the design speed up. Latency is measured in terms of **"clk cycle"**.

Function\Design	**Original Model**	**Pipeline Model**	**Parallelism Model**	**Final Model**	**Total Speed Up Factor**
3By3 Convolution	32,223,939	9,639,291	408,363	204,033	**157.9x**
Max Pooling	2,037,674	1,056,506	532,394	217,758	**9.36x**
1By1 Convolution	16,231,186	5,490,514	207,603	141,818	**114.45x**
3By3 DW Convolution	1,370,846	293,729	191,400	133,710	**10.25x**
Shuffling unit	640,785	146,625	146,626	29,776	**21.52x**

Passing unit	215,993	154,049	154,050	-----------	-----------
1by1 Convolution with AVG Pooling	46,682,113	23,436,289	62,440	62,392	**748.2x**
FC	2,051,001	1,028,001	20,001	47,001	**43.64x**
Total Estimated Latency	690,776,599	237,717,509	21,126,394	10,862,591	**63.6x**
RTL Accurate Latency	------------	156,160,401	9,066,784	3,244,576	----------

Note: We will calculate the average latency for functions which are divided in any optimizations to be able to compare and determine the speed up factor.

7.7.1.2 Resources

All resources here with respect to **XC7VX690T-2FFG1761C** FPGA chip.

Resource	Function\Design	Original Model	Pipeline Model	Parallelism Model	Final Model
BRAM 18K	**3By3 Convolution**	1	1	0	0
	Max Pooling	0	0	0	0
	1By1 Convolution	960	960	646	755
	3By3 DW Convolution	30	30	39	40
	Shuffling unit	0	0	0	0
	Passing unit	0	0	0	----------
	1by1 Convolution with AVG Pooling	480	480	465	465
	FC	960	960	1025	577
DSP	**3By3 Convolution**	1	1	29	56

	Max Pooling	1	0	0	0
	1By1 Convolution	5	4	822	994
	3By3 DW Convolution	21	20	37	214
	Shuffling unit	4	1	1	0
	Passing unit	1	1	1	----------
	1by1 Convolution with AVG Pooling	2	2	464	464
	FC	1	1	256	32

Resource	Function\Design	Original Model	Pipeline Model	Parallelism Model	Final Model
FF	**3By3 Convolution**	43	521	3815	5196
	Max Pooling	286	322	340	708
	1By1 Convolution	587	843	52383	57681
	3By3 DW Convolution	1965	1949	2723	6619
	Shuffling unit	261	303	307	710
	Passing unit	115	176	192	----------
	1by1 Convolution with AVG Pooling	209	193	32941	36978
	FC	107	109	51745	25010
LUTs	**3By3 Convolution**	1589	1916	4748	20855

	Max Pooling	1026	1275	1546	8290
	1By1 Convolution	1255	1659	75011	87313
	3By3 DW Convolution	3408	3615	3961	8826
	Shuffling unit	515	565	635	2076
	Passing unit	214	286	341	----------
	1by1 Convolution with AVG Pooling	506	552	56261	50269
	FC	194	249	102922	81985

<u>Note</u>: We will calculate the sum of any resource for functions which are divided in any optimizations to be able to compare.

The **total utilization** can be summarized as following:

Resource\Design	**Original Model**	**Pipeline Model**	**Parallelism Model**	**Final Model**
BRAM 18K	2912 (99%)	2912 (99%)	2695 (91%)	2305 (78%)
DSP	36 (1%)	30 (0.008%)	1610 (44%)	1760 (48%)
FF	4706 (0.005%)	5240 (0.006%)	184969 (21%)	164011 (18%)
LUTs	11279 (2%)	12657 (2%)	267272 (61%)	264394 (61%)

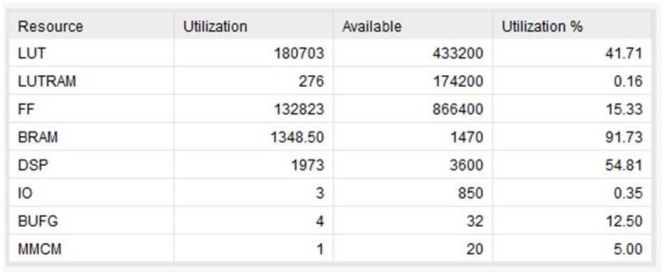
7.7.2 Vivado Implementation Results

7.7.2.1 Implementation of Parallelism Model on 100 MHz

Firstly, we implement the parallelism model on 100 MHz which will result in a speed of 11 frame/sec.

Utilization

The resources of our design on **Xilinx Virtex-7 FPGA VC709 Connectivity Kit** with **XC7VX690T-2FFG1761C** chip after implementation is shown in figure 26.



Resource	Utilization	Available	Utilization %
LUT	180703	433200	41.71
LUTRAM	276	174200	0.16
FF	132823	866400	15.33
BRAM	1348.50	1470	91.73
DSP	1973	3600	54.81
IO	3	850	0.35
BUFG	4	32	12.50
MMCM	1	20	5.00

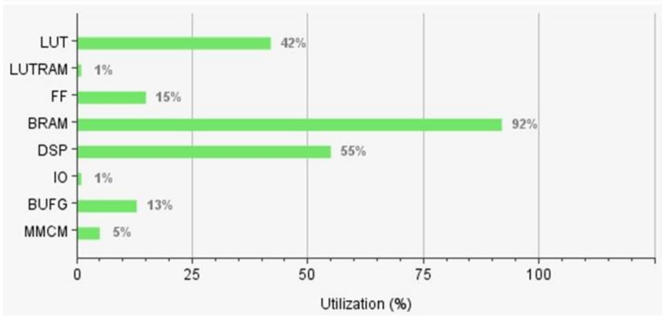
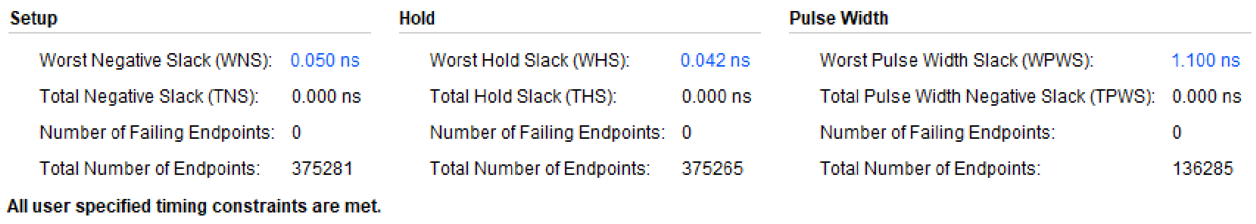


Figure 7.26 Utilization after Implementation on 100 MHz

Timing Analysis

As shown in figure 27 and 28, our design meets the timing constraints on 80 MHz without any negative slack in setup or hold time.



Setup

Worst Negative Slack (WNS):	0.050 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	375281

Hold

Worst Hold Slack (WHS):	0.042 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	375265

Pulse Width

Worst Pulse Width Slack (WPWS):	1.100 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	136285

All user specified timing constraints are met.

Figure 7.27 Timing Summary on 100 MHz

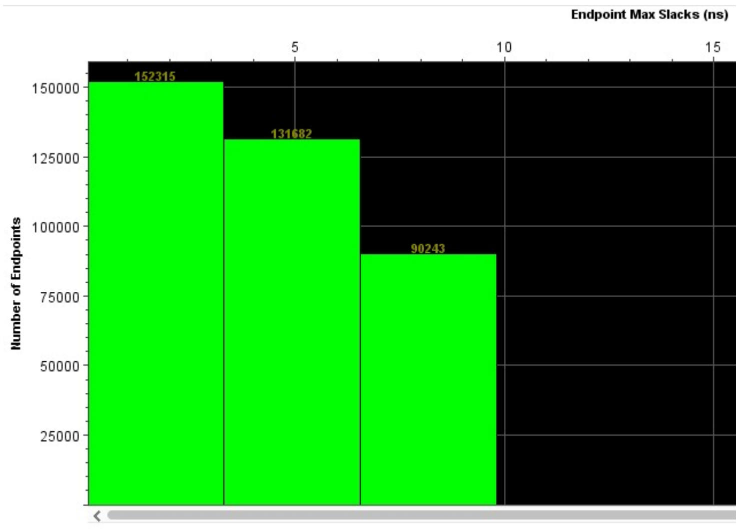
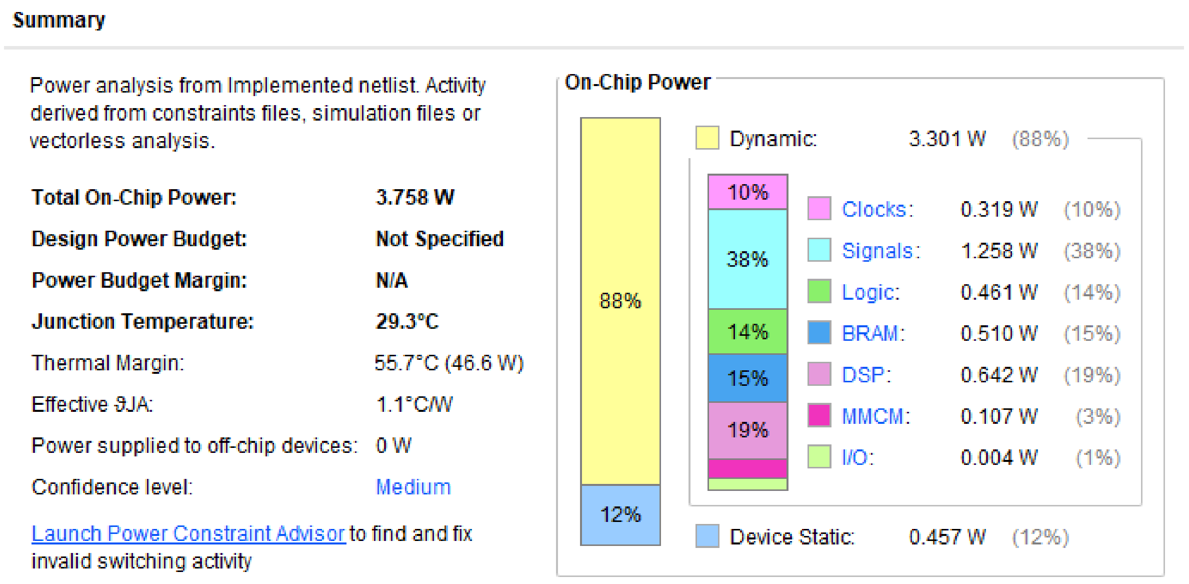


Figure 7.28 Timing Histogram on 100 MHz

Power Analysis

As shown in figure 29, the power consumption is suitable for our design speed as the energy per frame equals to **0.342 J/frame**.



Summary

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

Total On-Chip Power:	**3.758 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**29.3°C**
Thermal Margin:	55.7°C (46.6 W)
Effective ϑJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:		3.301 W	(88%)
	Clocks:	0.319 W	(10%)
	Signals:	1.258 W	(38%)
	Logic:	0.461 W	(14%)
	BRAM:	0.510 W	(15%)
	DSP:	0.642 W	(19%)
	MMCM:	0.107 W	(3%)
	I/O:	0.004 W	(1%)
Device Static:		0.457 W	(12%)

Figure 7.29 Power Consumption on 100 MHz

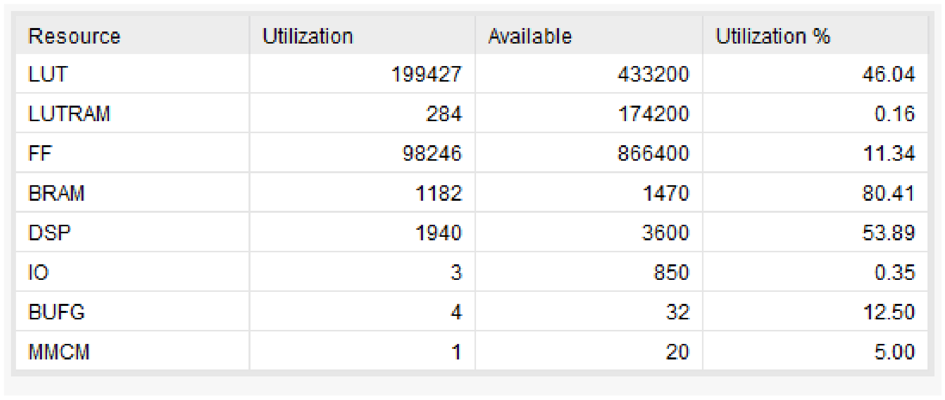
7.7.2.2 Implementation of Final Model on 80 MHz

After the first implementation we made many optimizations in speed as stated before. But, when we implemented the model, it didn't meet the timing neither on 100MHz or 80MHz. Thanks to dynamic quantization we would be able to implement our design on 80 MHz and it will give more frame rates the first implementation. This implementation would get 31 frame/sec if implemented on 100 MHz so if we implement on 80 MHz it will give us 24.8 frame/sec which is higher than 11 frame/sec and also will give better energy per frame. So, we will implement the

"Final Model" on Vivado to burn it on the FPGA at the end as it is the best in speed which is our main target to achieve real time operation.

Utilization

The resources of our design on **Xilinx Virtex-7 FPGA VC709 Connectivity Kit** with **XC7VX690T-2FFG1761C** chip after implementation is shown in Figure 7.26.



Resource	Utilization	Available	Utilization %
LUT	199427	433200	46.04
LUTRAM	284	174200	0.16
FF	98246	866400	11.34
BRAM	1182	1470	80.41
DSP	1940	3600	53.89
IO	3	850	0.35
BUFG	4	32	12.50
MMCM	1	20	5.00

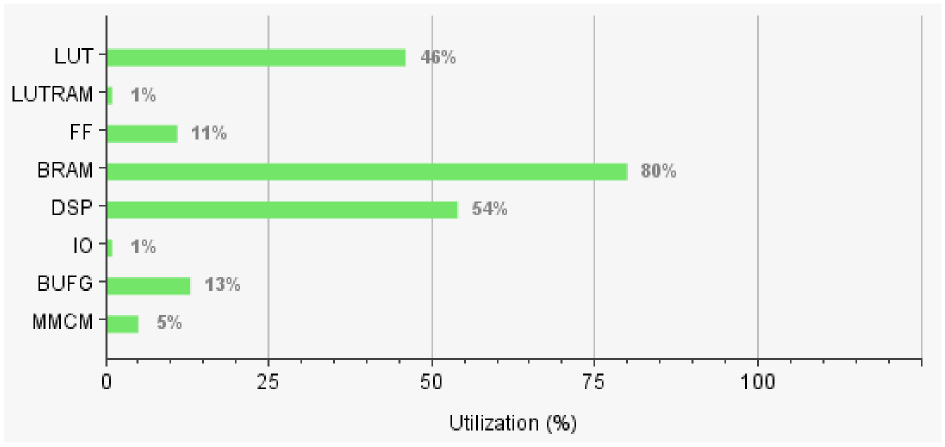
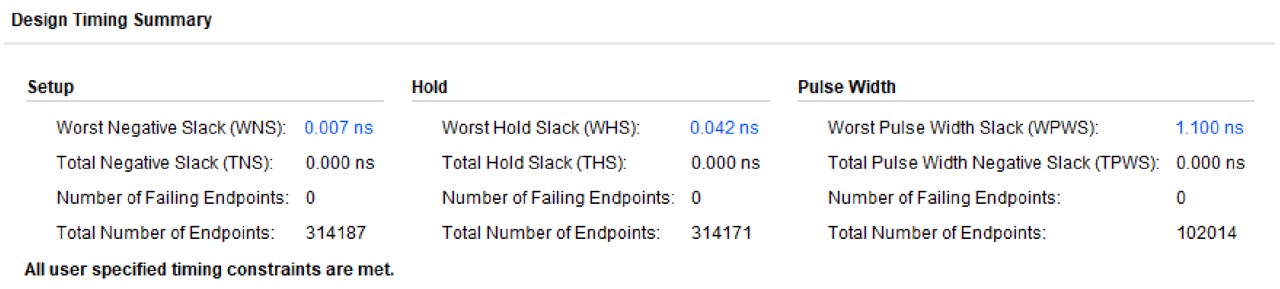


Figure 7.30 Post-implementation Utilization on 80 MHz

Timing Analysis

As shown in Figure 7.27 and 7.28, our design meets the timing constraints on 80 MHz without any negative slack in setup or hold time.



Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.007 ns	Worst Hold Slack (WHS):	0.042 ns	Worst Pulse Width Slack (WPWS):	1.100 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	314187	Total Number of Endpoints:	314171	Total Number of Endpoints:	102014

All user specified timing constraints are met.

Figure 7.31 Timing Summary on 80 MHz

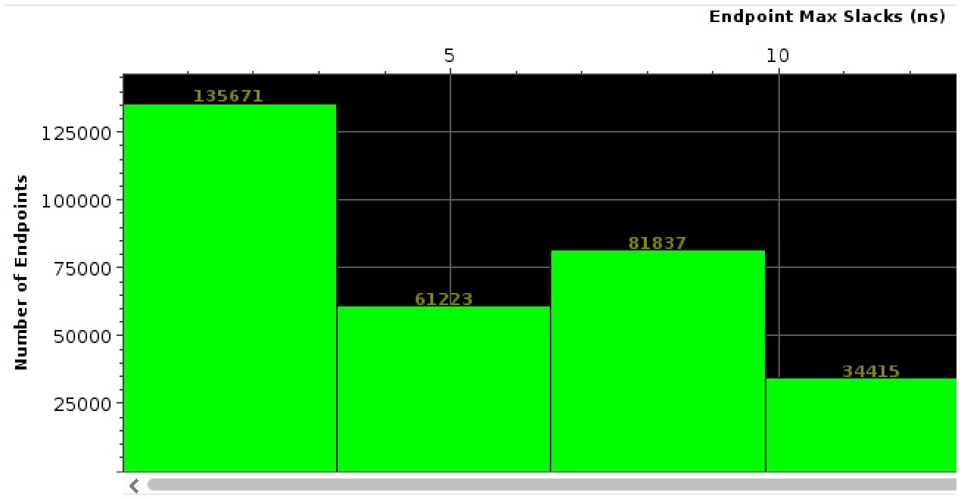
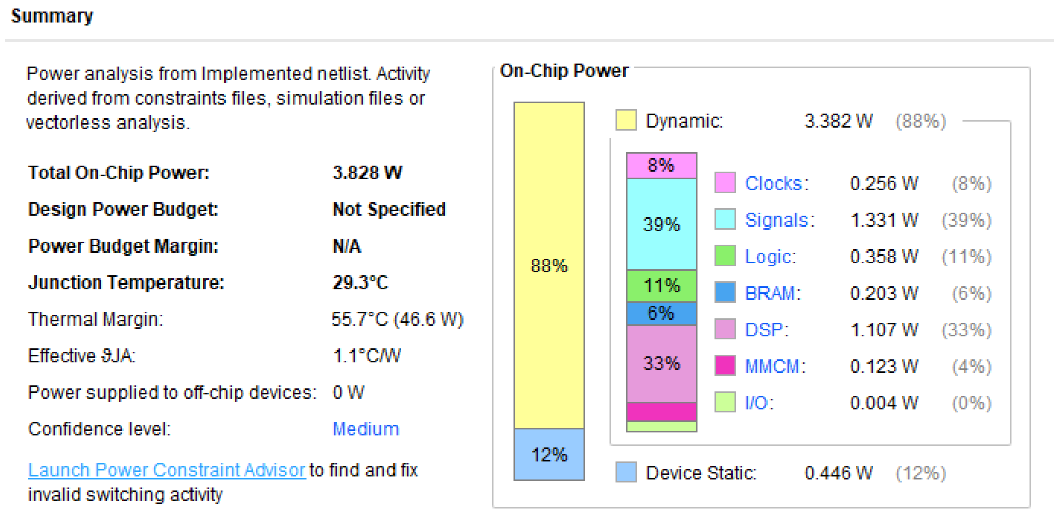


Figure 7.32 Timing Histogram on 80 MHz

Power Analysis

As shown in Figure 7.29, the power consumption is suitable for our design speed as the energy per frame equals to **0.154 J/frame**.


Summary

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

Total On-Chip Power:	**3.828 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**29.3°C**
Thermal Margin:	55.7°C (46.6 W)
Effective ϑJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:	3.382 W	(88%)	
	Clocks:	0.256 W	(8%)
	Signals:	1.331 W	(39%)
	Logic:	0.358 W	(11%)
	BRAM:	0.203 W	(6%)
	DSP:	1.107 W	(33%)
	MMCM:	0.123 W	(4%)
	I/O:	0.004 W	(0%)
Device Static:	0.446 W	(12%)	

Figure 7.33. Power Consumption on 80 MHz

7.7.3 Benchmark

To be able to judge on our design, we should compare it with the literature. So, we searched about published papers using HLS in their designs and the results are shown in Table 7.3.

Table 7.3 Benchmark

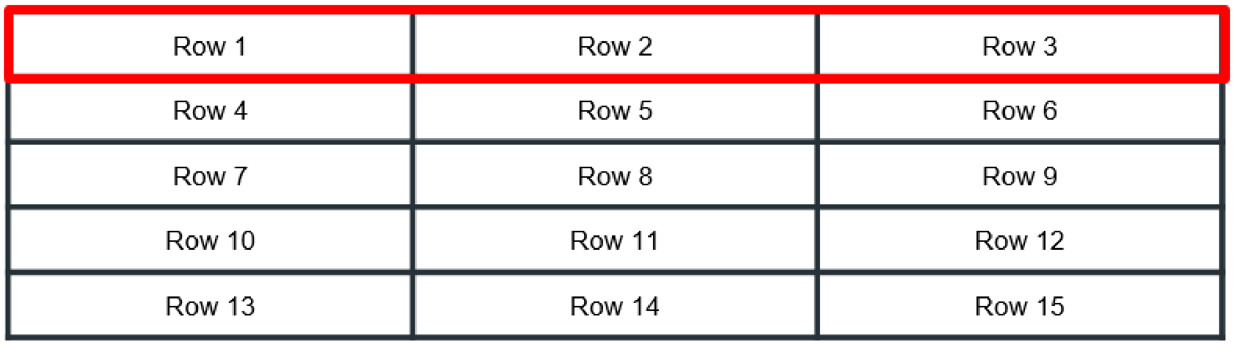
	Our Work	ZynqNet [24]	ZynqNet [25]	ResNet 50 (Software + Hardware) [26]
Target Board	Virtex-7 VC709	Zynq XC-72045	Kintex XC-ku060	Zynq UltraScale+ MPSoC ZCU104
No. Classes	1000	1000	32	1000
No. Parameters	2.3 M	2.5 M	1.8 M	25.5 M
LUT	199,427	154,000	113,000	125,926
FF	98,246	137,000	35,000	136,586
DSP	1940	739	700	1591
Off-Chip Memory	No	Yes	No	Yes
BRAM (18k)	2364	996	1492	376 + 78 URAM
Frequency	80 MHz	100 MHz	100 MHz	150 MHz
Inference time (ms)	40.25	1955	222	239.8
Frame rate	24.84	0.5	45	4.1
Power (W)	3.828	7.8	-------	8
Energy/Frame (J)	0.154	15.6	-------	0.5

7.8 Conclusion

7.8.1 HLS Limitations

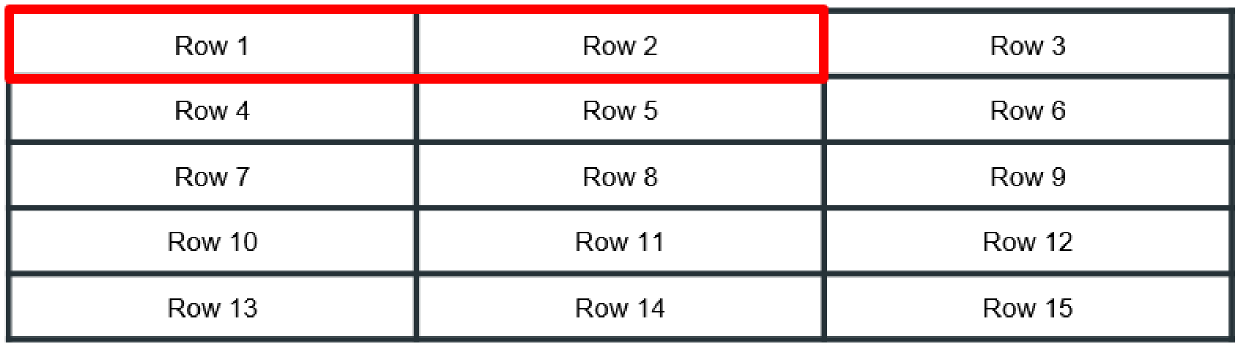
High Level Synthesis (HLS) is a powerful tool but the road has still many obstacles and you shouldn't think it will be smooth and comfortable especially with larger designs. HLS still has some limitation which need to be developed and some of these are listing below:

- **Array partitioning** have some limitations in indexing the array which is partitioned to be able to fetch data in parallel without overhead in latency or hardware.
  In cyclic portioning, if we have a photo inside a memory where pixels are stored sequential by row. If we partition with factor of "3*number of data in a row" so we can think about it as we have "3*number of data in a row" instances of memory which hold data.
  We should read data in integer multiple or factor of the number of instances so that tool can parallel the reading of data from all instances at the same time as shown in Figure 7.30, if not as shown in Figure 7.31, as next time you will read row3 and row4, the tool will infer a large mux and read data from more than one instance then choose one of the data.
  Same limitation exists also with block partition.



Row 1	Row 2	Row 3
Row 4	Row 5	Row 6
Row 7	Row 8	Row 9
Row 10	Row 11	Row 12
Row 13	Row 14	Row 15

Figure 7.34 Right Technique in Reading Data



Row 1	Row 2	Row 3
Row 4	Row 5	Row 6
Row 7	Row 8	Row 9
Row 10	Row 11	Row 12
Row 13	Row 14	Row 15

Figure 7.35 Wrong Technique in Reading Data

- **Pattern Detection Limitations**
  In the previous example, if we read "row1" and "row2" we will read "row3" and "row4" then "row 5" and "row6" and so on. HLS tool, as stated before, will infer mux and read from all instances. So, HLS tool can't detect this pattern and make a controller unit for it however in RTL we can make controller, it will not be easy but we can do it.

- **Loops with variable bounds**
  Some directives can't work if this the case of this type of for loop like "PIPELINE" and "UNROLL". In our CNN have the function with many sized so variable bounds is very suitable for us.
- **C simulation for large arrays**
  In this case simulation can run out of memory as in our case in fixed point model. The solution is to use the dynamic allocation as stated in HLS user guide but it is not synthesizable.

7.8.2 Pros and Cons

Form our experience we can summarize the pros of HLS as following:

- Ease of use.
- Ease of code editing.
- Reusability.
- Fast verification testbench development.
- Very good to implement arithmetic hardware.
- Small time to market with respect to RTL.
- Small number of project files.

We can summarize the cons as following:

- Hard to debug the hardware.
- Hard to read the output generated RTL like signals names.
- Testbench is sequential as it is high level, we need it to be some sort of parallelism.
- Flow control synthesized hardware is not optimum.
- Less controllability.
- Has many limitations as stated before.
- Very long run time to compile the whole model every time we edit.
- Need time to be familiar with HLS flow and directives.
- Results can be sometimes unexpected.

8 Chapter 8: Hardware Testing on FPGA

After exporting the hardware design in HLS and Writing the RTL code, verifying their functionality, synthesis and implementation both, this is the time to burn the design on the FPGA and test it on a real hardware as it is one of our targets which is to develop an accelerator which can be used in real life not just on software simulation tools. This is done by generating a bitstream file which will be uploaded on the FPAGA and will configure it. We will use **Xilinx Virtex-7 FPGA VC709 Connectivity Kit** with **XC7VX690T-2FFG1761C chip**.

In order to test on FPGA, we should think about how the clock will be generated and connected to our design and how we can drive the inputs and monitor the output of the hardware.

8.1 Clock Generation

The source of our clock will be the FPGA board System Clock which is an **LVDS 200 MHz oscillator** [27]. It will provide us with 200MHz differential clock which will be fed into a buffer **IBUFGDS** which will convert the clock to single ended with same 200MHz frequency.

The 200MHz single ended clock will be fed into an IP called **Mixed-Mode Clock Manager (MMCM)** as input. MMCM can generate output clock as desired up to 7 outputs clock. So, we will generate one output clock with the desired frequency to be the operating frequency for our hardware and it will be fed to our "Shuffle Model" IP which is exported from Vivado HLS or to the RTL design we developed.

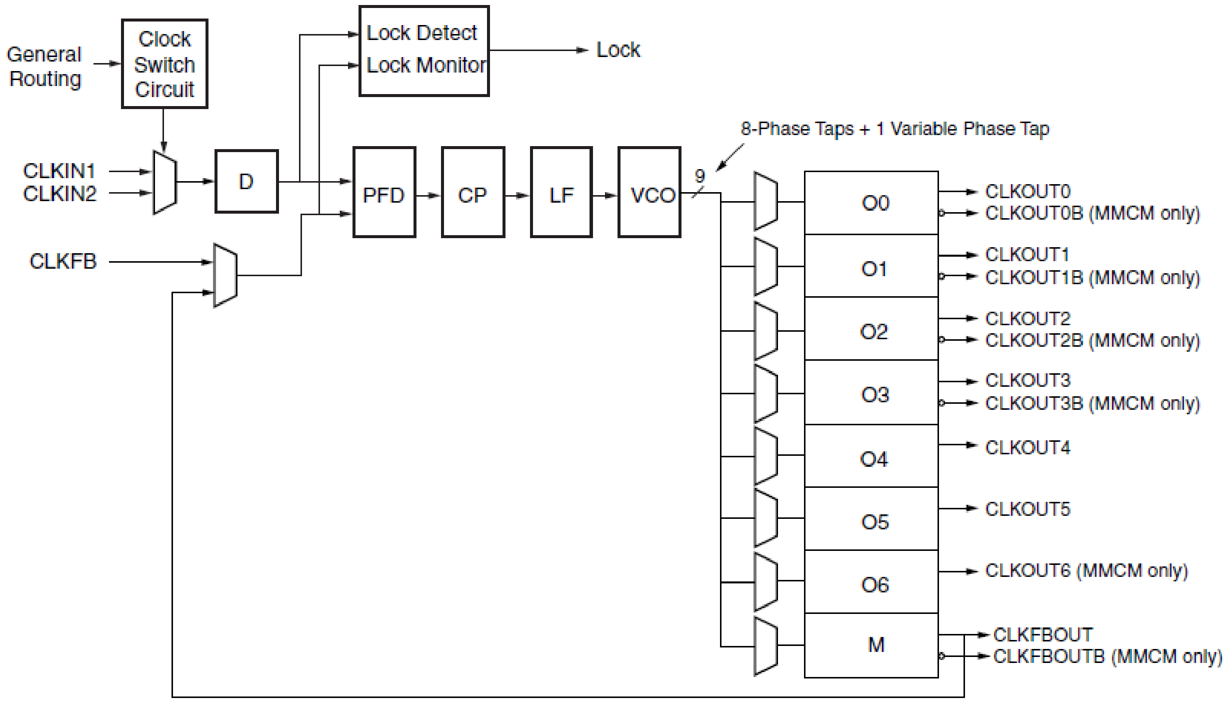


Figure 8.1 MMCM Block Diagram

8.2 Driving Input and Monitoring Output

We will drive the inputs and monitor outputs using Xilinx hardware IPs which can interact with Vivado design suite software in real time while our hardware operating on FPGA. Also we will drive the LEDs on FPGA with signal which is indicating that the output is ready and available at the output port so when the led flashes once in the RTL the output value on the software should be the desired output. In HLS, we drive the led with signal which tell us that the design can accept a new photo to process so if it OFF the output value should be the desired output like it is an active low signal.

We use 2 IPs to do this and the usage is explained as following:

- **Virtual Input/Output (VIO)**

The IP is a customizable core that can both monitor and drive internal FPGA signals in real time. The number and width of the input and output ports are customizable in size to interface with the FPGA design. This is generated and configured with IP catalog inside Vivado Project.

We connected it with same clock as our design. The inputs of our hardware are outputs for VIO to be able to drive them and the outputs of our hardware design are the inputs for VIO to be able to observe them.

- **Integrated Logic Analyzer (ILA)**

It is a customizable core which is a logic analyzer that can be used to monitor the internal signals of a design. The number of probes monitored and their widths are customizable in size to interface with the FPGA design. The number of samples taken by the IP can be chosen from some values provided with the IP. This is generated and configured with IP catalog inside Vivado Project.

We connected it with same clock as our design. We only monitor the output class and a signal which is become high when the output is available so that we can trigger the ILA to capture samples data at the rising edge of this signal to capture the real output value.

By monitoring the output by two IPs not only one we can be sure that the results are correct and the hardware testing step on FPGA succeed.

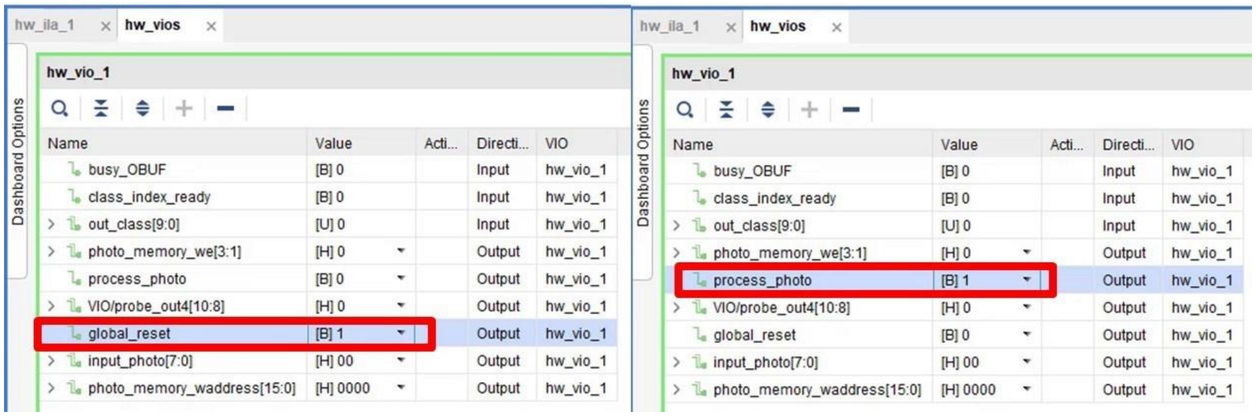
8.3 Testing Procedure and Results

We open the hardware manger and connect to the FPGA then then burn the bitstream file. After that the window of VIO and ILA are opened so that we can interact with FPGA in real time. We use the photo with class 66 (sea snake) which is the 1st photo in the ImageNet validation set.

8.3.1 RTL

We use VIO to reset the hardware then we make “process_photo” signal “HIGH” so that the hardware will start the operation and the output value will be observed as shown in Figure 8.2.

We use ILA to capture real output data on the rising edge of the "class_index_ready" signal as shown in Figure 8.3. From ILA we can verify that the real time hardware output is correct.



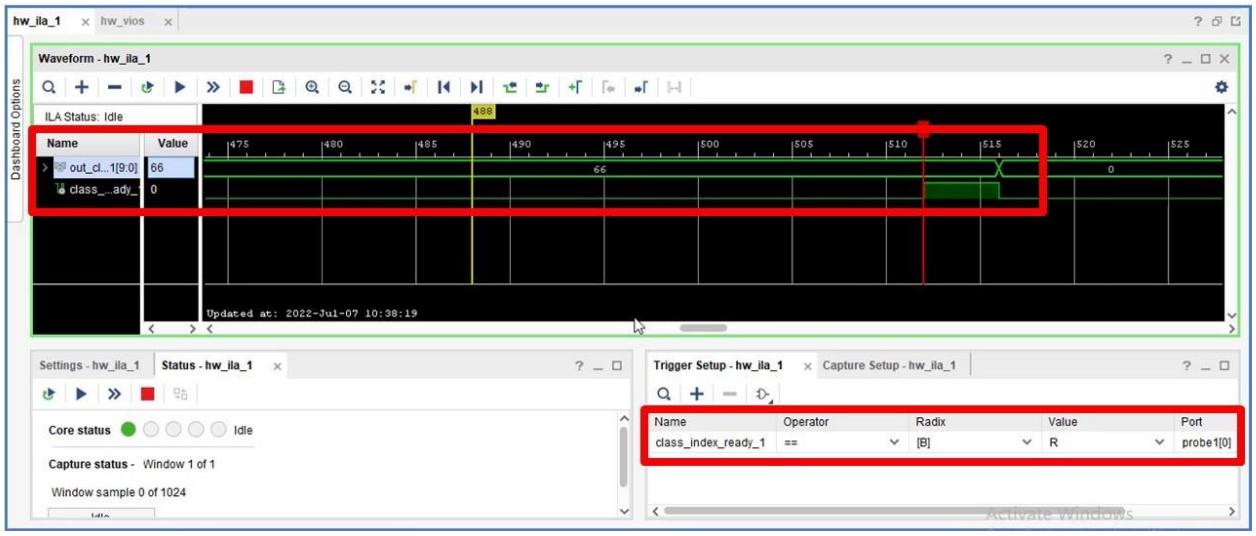
hw_vio_1

Name	Value	Acti...	Directi...	VIO
busy_OBUF	[B] 0		Input	hw_vio_1
class_index_ready	[B] 0		Input	hw_vio_1
out_class[9:0]	[U] 0		Input	hw_vio_1
photo_memory_we[3:1]	[H] 0		Output	hw_vio_1
process_photo	[B] 0		Output	hw_vio_1
VIO/probe_out4[10:8]	[H] 0		Output	hw_vio_1
global_reset	[B] 1		Output	hw_vio_1
input_photo[7:0]	[H] 00		Output	hw_vio_1
photo_memory_waddress[15:0]	[H] 0000		Output	hw_vio_1

hw_vio_1

Name	Value	Acti...	Directi...	VIO
busy_OBUF	[B] 0		Input	hw_vio_1
class_index_ready	[B] 0		Input	hw_vio_1
out_class[9:0]	[U] 0		Input	hw_vio_1
photo_memory_we[3:1]	[H] 0		Output	hw_vio_1
process_photo	[B] 1		Output	hw_vio_1
VIO/probe_out4[10:8]	[H] 0		Output	hw_vio_1
global_reset	[B] 0		Output	hw_vio_1
input_photo[7:0]	[H] 00		Output	hw_vio_1
photo_memory_waddress[15:0]	[H] 0000		Output	hw_vio_1

Figure 8.2 RTL VIO Interface



Waveform - hw_ila_1

ILA Status: Idle

Name	Value
out_cl...1[9:0]	66
class_...ady_	0

Updated at: 2022-Jul-07 10:38:19

Core status Idle

Capture status - Window 1 of 1

Window sample 0 of 1024

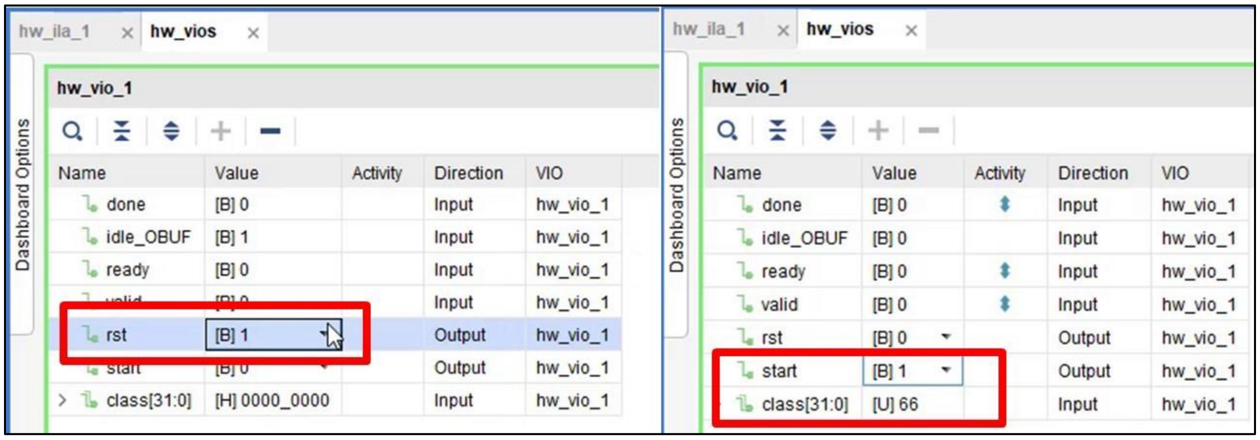
Trigger Setup - hw_ila_1

Name	Operator	Radix	Value	Port
class_index_ready_1	==	[B]	R	probe1[0]

Figure 8.3 RTL ILA Interface

8.3.2 HLS

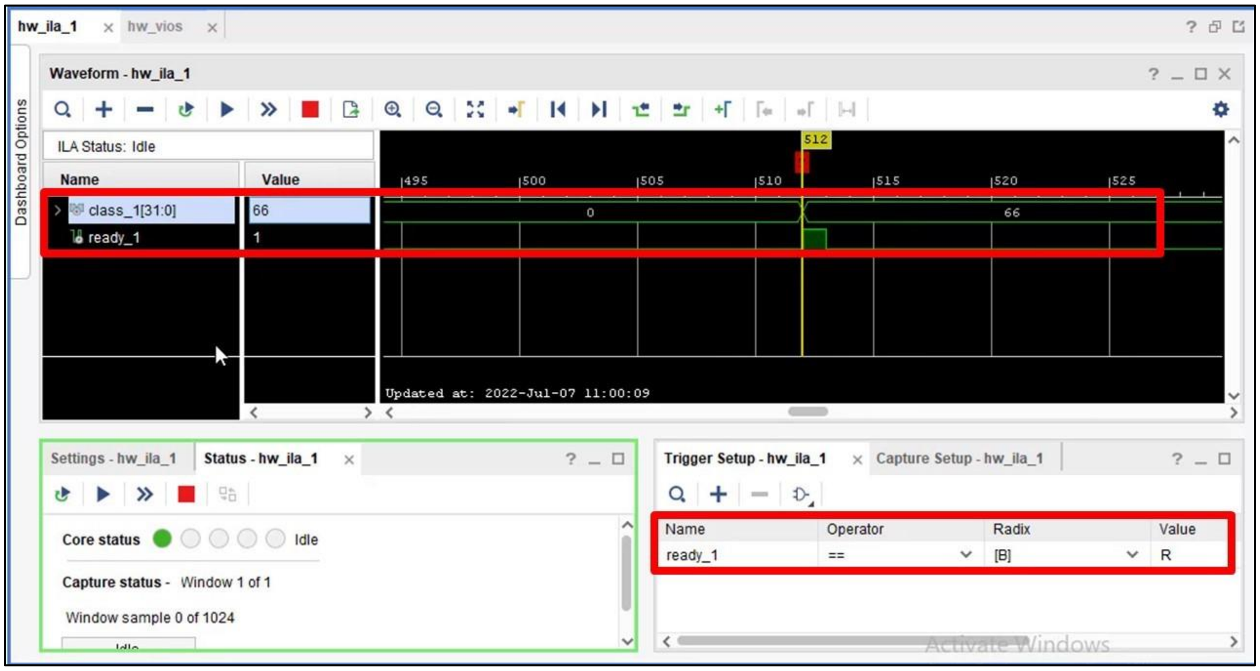
We use VIO to reset the hardware then we make "start" signal "HIGH" so that the hardware will start the operation and the output value will be observed as shown in Figure 8.4. We use ILA to capture real output data on the rising edge of the "ready" signal to as shown in Figure 8.5. From VIO and ILA we can verify that the real time hardware output is correct.



Name	Value	Activity	Direction	VIO
done	[B] 0		Input	hw_vio_1
idle_OBUF	[B] 1		Input	hw_vio_1
ready	[B] 0		Input	hw_vio_1
valid	[B] 0		Input	hw_vio_1
rst	[B] 1		Output	hw_vio_1
start	[B] 0		Output	hw_vio_1
class[31:0]	[H] 0000_0000		Input	hw_vio_1

Name	Value	Activity	Direction	VIO
done	[B] 0	⬍	Input	hw_vio_1
idle_OBUF	[B] 0		Input	hw_vio_1
ready	[B] 0	⬍	Input	hw_vio_1
valid	[B] 0	⬍	Input	hw_vio_1
rst	[B] 0		Output	hw_vio_1
start	[B] 1		Output	hw_vio_1
class[31:0]	[U] 66		Input	hw_vio_1

Figure 8.4 HLS VIO Interface



ILA Status: Idle

Name	Value
class_1[31:0]	66
ready_1	1

```
Updated at: 2022-Jul-07 11:00:09
```

Core status Idle

Capture status - Window 1 of 1

Window sample 0 of 1024

Name	Operator	Radix	Value
ready_1	==	[B]	R

Figure 8.5 HLS ILA Interface

9 Chapter 9: Results

In this chapter we will mark out our effort throughout the project and introduce the results obtained from the RTL design and the HLS design after implementation on the FPGA. The results include utilization, power, and timing for both implementations before and after optimizations. Also, we will introduce a benchmark for our RTL design with two other different designs implementing a machine learning algorithm on FPGA. Finally, we will end the chapter with a comparison between the RTL design flow and the HLS design flow.

9.1 RTL Design Results

9.1.1 Results Before Optimizations

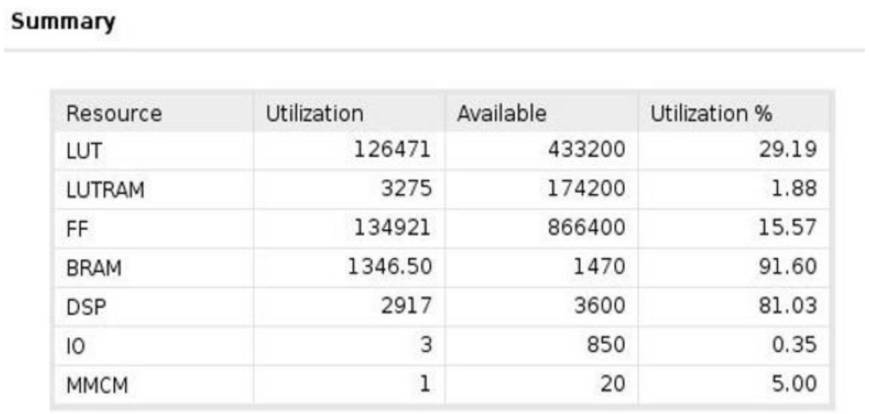
9.1.1.1 Utilization

Before reporting the RTL design utilization on the FPGA, we must first define the target FPGA used to implemented the design on. Our target FPGA is Xilinx Virtex-7 FPGA VC709 Kit and Table 9.1 below describes its features. One important note from the table below is that it has 1470 BRAM of 36Kbit and double this number of 18Kbit. Hence, it’s the best choice for us as our design requires large number of BRAMs to store the large number of parameters in the ShuffleNet CNN.

Table 9.1 Virtex-7 FPGA kit

DSPs	3600
Slice LUTs	433,200
Slice Regs	866,400
BRAMs	1470 (36Kbit each, can be used as 2 18Kbit BRAMs)

Figure 9.1 displays the detailed utilization for the design, and it is clear that while the utilization of BRAMs and DSPs is nearly high, that of LUTs and FFs is very low. Most utilization comes from group2 as it has all shuffle units in our design. The utilization we obtained by the design is good regarding large number of layers and parameters in the ShuffleNet CNN.



Summary

Resource	Utilization	Available	Utilization %
LUT	126471	433200	29.19
LUTRAM	3275	174200	1.88
FF	134921	866400	15.57
BRAM	1346.50	1470	91.60
DSP	2917	3600	81.03
IO	3	850	0.35
MMCM	1	20	5.00

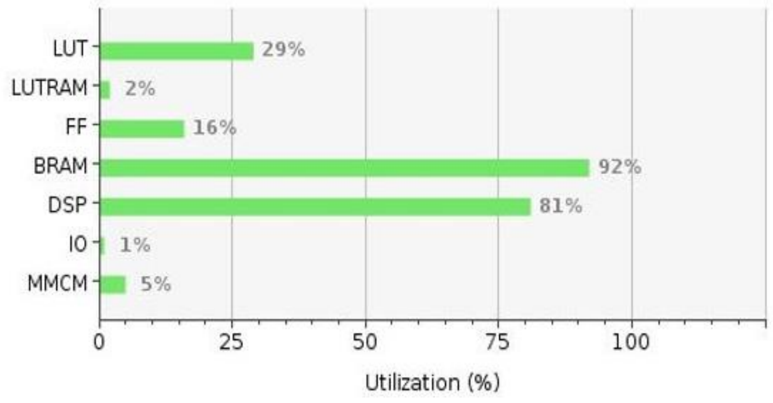
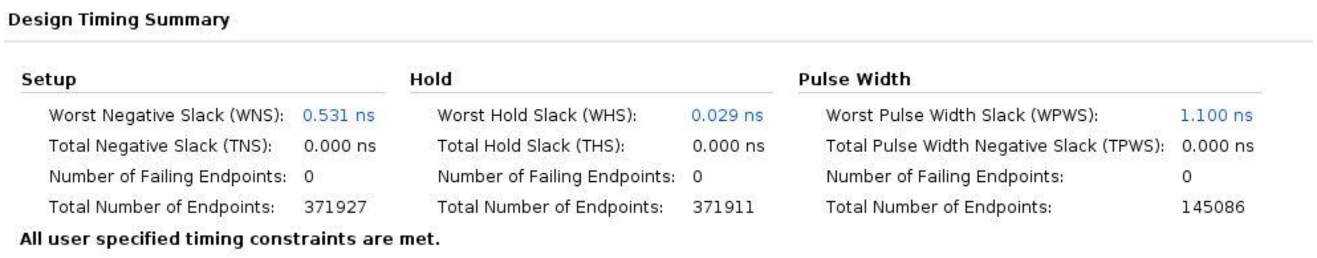


Figure 9.1 Design Utilization before optimizations

9.1.1.2 Timing

We implement the RTL design on the FPGA operation on 50MHz. Figure 9.2 illustrates how the design meets timing with a positive slack of 0.531ns and a positive hold slack of 0.029ns after design implementation.



Design Timing Summary

```
Setup                                     Hold                                      Pulse Width
  Worst Negative Slack (WNS):    0.531 ns   Worst Hold Slack (WHS):         0.029 ns   Worst Pulse Width Slack (WPWS):             1.100 ns
  Total Negative Slack (TNS):    0.000 ns   Total Hold Slack (THS):         0.000 ns   Total Pulse Width Negative Slack (TPWS):    0.000 ns
  Number of Failing Endpoints:   0          Number of Failing Endpoints:    0          Number of Failing Endpoints:                0
  Total Number of Endpoints:     371927     Total Number of Endpoints:      371911     Total Number of Endpoints:                  145086
All user specified timing constraints are met.
```

Figure 9.2 Timing constraints before optimizations

Table 9.2 shows the max time to get the output from each group (Group latency) so we find out that the max group delay equals 1644 us in Group3 and as the operating clock frequency equals 50Mhz, our RTL design can achieve up to 608 frames/sec.

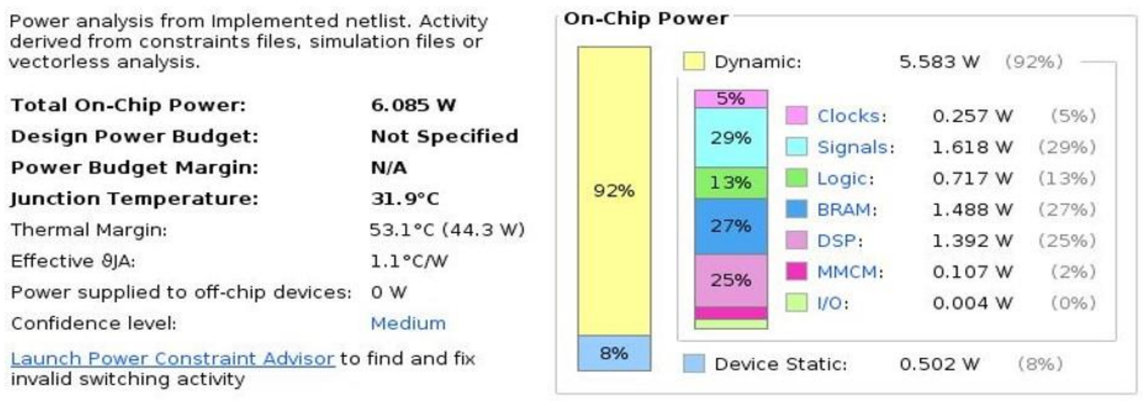
Table 9.2 Groups latency before optimizations

	Group1	Group2	Group3
Time	1266 us	1596 us	1644 us

9.1.1.3 Power

Power is an important factor in designing accelerators and choosing a platform to implement it on, as we can implement the accelerator on a GPU and achieve very high speed in short time but the cost is in power. It consumes very high power. FPGA is the best choice for achieving high speed as seen in the last section with reasonable power consumption as we will see in this section.

We first performed power analysis based on the default power estimation of Vivado. According to Figure 9.3, the total on-chip power equals 6.085 watt, consisting of 5.583 watt for dynamic power, which accounts for 92% of the total power, and 0.502 watt for static power, which accounts for 8% of the total power. From this power results and the latency of the design, we can calculate the power delay product which is equal to 0.01 joule/frame.



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

Total On-Chip Power:	**6.085 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**31.9°C**
Thermal Margin:	53.1°C (44.3 W)
Effective ϑJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

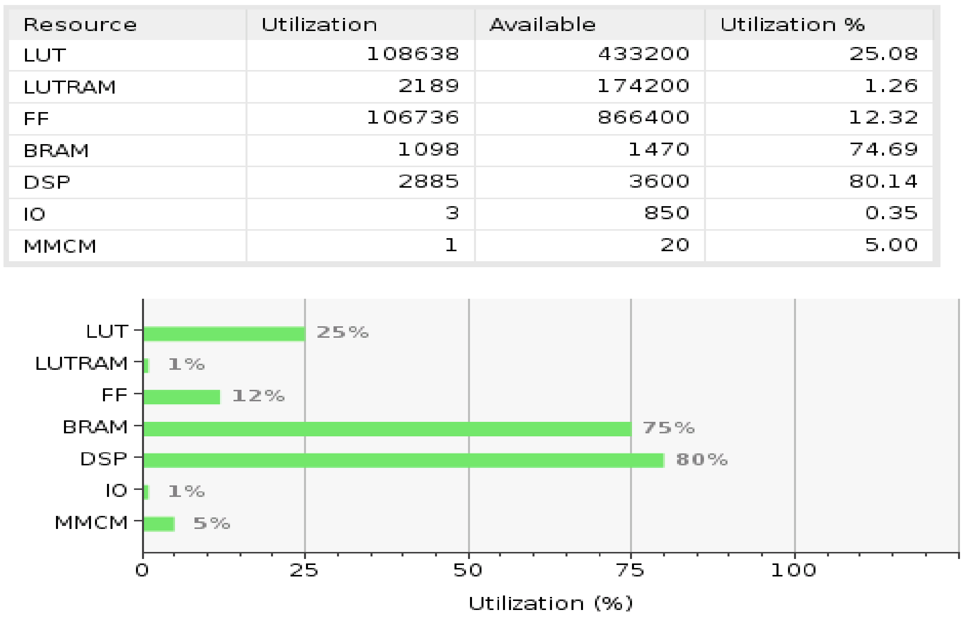
Dynamic:		5.583 W	(92%)
	Clocks:	0.257 W	(5%)
	Signals:	1.618 W	(29%)
	Logic:	0.717 W	(13%)
	BRAM:	1.488 W	(27%)
	DSP:	1.392 W	(25%)
	MMCM:	0.107 W	(2%)
	I/O:	0.004 W	(0%)
Device Static:		0.502 W	(8%)

Figure 9.3 Power consumption based on default settings before optimizations

9.1.2 Results After Optimizations

9.1.2.1 Utilization

Figure 9.4 displays the detailed utilization for the design, and it is clear that while the utilization of BRAMs and DSPs is nearly high but less than before optimizations, and that of LUTs and FFs is very low and also less than before optimizations. The utilization we obtained by the design is much good regarding large number of layers and parameters in the ShuffleNet CNN. This good results in the utilization of BRAMs occurs after applying the optimizations explained in chapter 6. Utilization of BRAMs is reduced significantly from 97% to 74.64%.

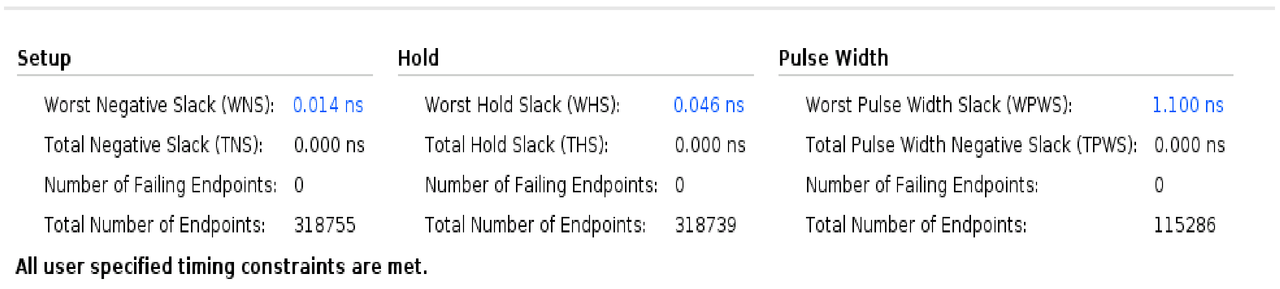


Resource	Utilization	Available	Utilization %
LUT	108638	433200	25.08
LUTRAM	2189	174200	1.26
FF	106736	866400	12.32
BRAM	1098	1470	74.69
DSP	2885	3600	80.14
IO	3	850	0.35
MMCM	1	20	5.00

Figure 9.4 Design Utilization after optimizations

9.1.2.2 Timing

After Applying all the optimizations in chapter 6, we managed to implement the RTL design on the FPGA operation on 100MHz. Figure 9.5 illustrates how the design meets timing with a positive slack of 0.014ns and a positive hold slack of 0.046ns after design implementation.



Setup

Worst Negative Slack (WNS):	0.014 ns
Total Negative Slack (TNS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	318755

Hold

Worst Hold Slack (WHS):	0.046 ns
Total Hold Slack (THS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	318739

Pulse Width

Worst Pulse Width Slack (WPWS):	1.100 ns
Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0
Total Number of Endpoints:	115286

All user specified timing constraints are met.

Figure 9.5 Timing constraints after optimizations

From Table 9.3, the max time to get the output from each group (Group latency) so we find out that the max group delay equals 822 us in Group3 and as the operating clock frequency equals 100Mhz, our RTL design can achieve up to 1216 frames/sec. Figure 9.6 is a timing histogram at a frequency of 100MHz shows the utilization of the clock frequency used, indicating how good the design is.

Table 9.3 Groups latency after optimizations

	Group1	Group2	Group3
Time	633 us	798 us	822 us

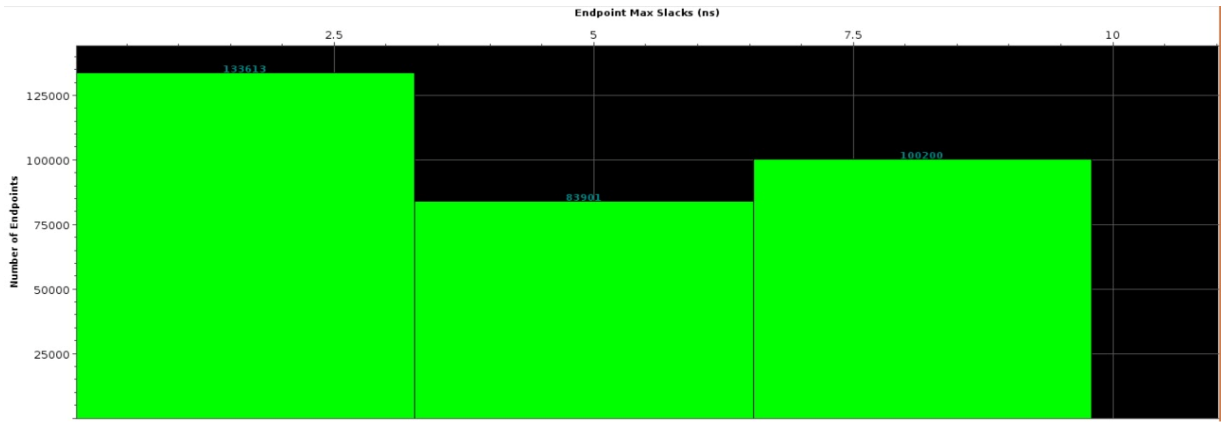
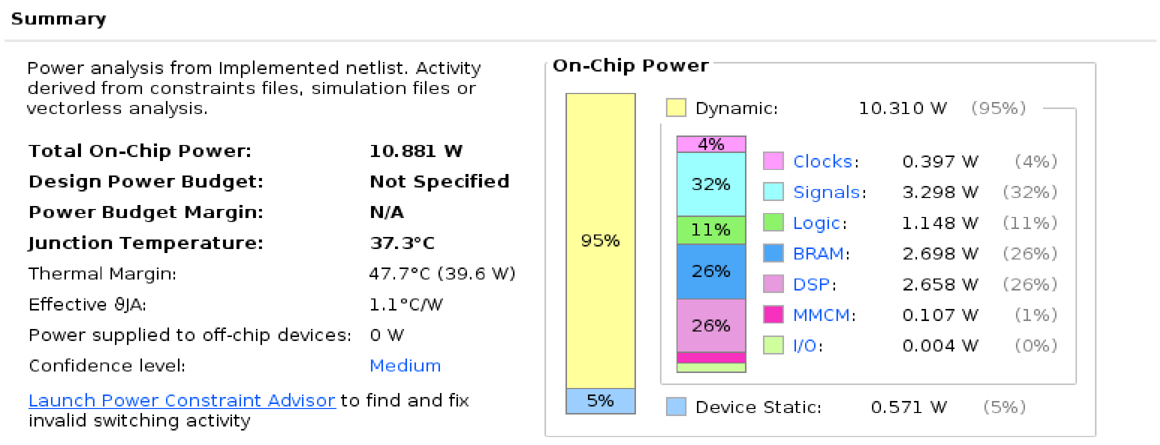


Figure 9.6 Timing histogram after optimizations

9.1.2.3 Power

We first performed power analysis based on the default power estimation of Vivado. According to Figure 9.7, the total on-chip power equals 10.881 w, consisting of 10.310 w for dynamic power, which accounts for 95% of the total power, and 0.571 w for static power, which accounts for 5% of the total power.



Summary

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

Total On-Chip Power:	**10.881 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**37.3°C**
Thermal Margin:	47.7°C (39.6 W)
Effective ϑJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

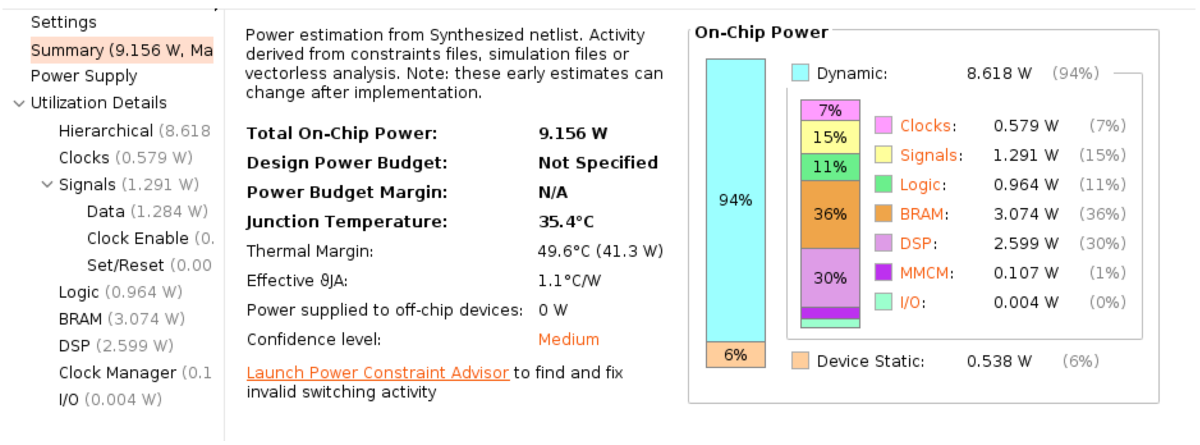
Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

	Dynamic:	10.310 W	(95%)
	Clocks:	0.397 W	(4%)
	Signals:	3.298 W	(32%)
	Logic:	1.148 W	(11%)
	BRAM:	2.698 W	(26%)
	DSP:	2.658 W	(26%)
	MMCM:	0.107 W	(1%)
	I/O:	0.004 W	(0%)
	Device Static:	0.571 W	(5%)

Figure 9.7 Power consumption based on default settings after optimizations

Then, we used the saif file which is a file contains the switching activity of the design signals during simulation. It was extracted from the post-implementation simulation. Applying power analysis using saif file is more accurate because it uses the actual switching rate of the signals. As shown in Figure 9.8, the total on-chip power is reduced to 9.156 w, consisting of 8.618 w for dynamic power, which accounts for 94% of the total power, and 0.583 w for static power, which accounts for 6% of the total power. Also, from this power results and the latency of the design, we can calculate the power delay product which is equal to 0.007526 joule/frame.



Settings
Summary (9.156 W, Ma
Power Supply
Utilization Details
Hierarchical (8.618
Clocks (0.579 W)
Signals (1.291 W)
Data (1.284 W)
Clock Enable (0.
Set/Reset (0.00
Logic (0.964 W)
BRAM (3.074 W)
DSP (2.599 W)
Clock Manager (0.1
I/O (0.004 W)

Power estimation from Synthesized netlist. Activity derived from constraints files, simulation files or vectorless analysis. Note: these early estimates can change after implementation.

Total On-Chip Power:	**9.156 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**35.4°C**
Thermal Margin:	49.6°C (41.3 W)
Effective ϑJA:	1.1°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:		8.618 W	(94%)
	Clocks:	0.579 W	(7%)
	Signals:	1.291 W	(15%)
	Logic:	0.964 W	(11%)
	BRAM:	3.074 W	(36%)
	DSP:	2.599 W	(30%)
	MMCM:	0.107 W	(1%)
	I/O:	0.004 W	(0%)
Device Static:		0.538 W	(6%)

Figure 9.8 Power consumption using Saif file after optimizations

9.1.3 Benchmark

In this section, we compare our design with two other implementations of the image classification machine learning algorithm on FPGA but with different CNN architectures as benchmarks to evaluate how good our design and results. From Table 9.4, we can see that our design has a large number of weights (2.3M parameter), a large number of layers (50 layers), and working on ImageNet validation set so it has 1000 class but our design has fewer resources in the number of FFs used and manage to achieve the highest frame rate (equals 1216 fps) which, when compared to the two other designs, is almost extremely high and as the power is roughly the same for all three designs hence it achieves the smallest energy per image which equals 0.007526 joule.

Table 9.4 Benchmark

Parameter\Design	This Work	Reference [28]	Reference [29]
Target Board	Virtex 7-VC-709	Virtex 7-VC-709	Virtex 7-VC-709
CNN	Shuffle Net V2	ZynqNet	Squeeze-Net
Accuracy (%)	69.4%	57.5%	69.6%
No of weights	2.3M	2.5M	1.2M
No of layers	50	10	18

No of classes	1000	1000	10
Frequency (MHz)	100	100	100
LUTs	109K	339K	84K
FFs	107K	184K	133K
DSPs	2885	3552	2656
BRAMs	1098	1413	908
Frame rate (fps)	1216	12.5	139
Power(watt)	9.156	10.97	8.9
Energy/image(j)	0.007526	0.88	0.0357

9.2 Comparison between RTL and HLS Results

From our experience throughout the project, we can compare between the RTL and HLS flows from different aspects as the following:

- It is clear that HLS is simpler to use, takes less time to market, and makes verification development easier, whereas RTL is more controllable, simpler to debug, and offers better performance than HLS.
- Also, utilization in RTL is less than HLS as RTL uses 25.08% LUTs and 74.69% BRAMs whereas HLS uses 46.04% LUTs and 80.41% BRAMs but HLS uses fewer DSPs and approximately number of FFs is the same in RTL and HLS.
- The cost of using HLS is more in terms of run time as there is a need to recompile codes in case of editing or modifying any of them.
- One more point is that number of files or sources needed in RTL is more than in HLS as there are more files for weights and biases in RTL whereas HLS has a CPP file and a number of header files and will generate weights and biases automatically.
- In HLS, RTL Testbench will be generated automatically from a C++ testbench which is easier to write, but writing and developing test benches in RTL will require more effort.
- One important point to note is that the result guarantee after each modification is more reliable from RTL than HLS.
- Finally, RTL code generated from HLS is difficult to read and understand.

This comparison also can be summarized as in Table 9.5.

Table 9.5 RTL and HLS comparison

	RTL	HLS
Utilization	Less	Larger
Easy to use and edit	Harder	Easier
Controllability	Better	Harder
Time to market	Larger	Less
Verification Development	Harder	Easier
Debugging	Easier	Harder
Testbench effort	More	Less
Sources or files used	More	Less
Result guarantee	More reliable	Less reliable
Run time	Less	Larger
Performance	Better	Worse

10 Chapter 10: Conclusion and Future Work

10.1 Conclusion

We have illustrated the importance of the CNN networks in computer vision applications and how they require high-speed computation resources and consume so much power. And to make the most benefit from them, they must run on a hardware accelerator. Thus, our project was to implement and optimize a hardware accelerator for a CNN on FPGA. Throughout the thesis, we discussed and illustrated in detail all the steps we followed in the project, from choosing the CNN network to hardware implementation on the FPGA. Problems and results also are included.

The CNN network we have chosen is the ShuffleNet CNN, and to our knowledge, there is no previous hardware implementation for it in the literature, so the project is the first hardware implementation for it. The golden reference for the network was a software model written in PyTorch and is pretrained on the ImageNet data set. We designed the ShuffleNet accelerator through two design flows; they are the RTL design flow and the HLS design flow, and our target in both flows was high throughput (real-time) with affordable power consumption. Fixed-point was used as data representation instead of float. Moreover, we applied many optimizations like dynamic quantization. The accelerator works on 224*224*3 RGB images and classifies them among 1000 classes. It can also process three images simultaneously. The tools that we used in both design approaches are the Vivado design suite and Vivado HLS from Xilinx.

The best figure of merit to evaluate a hardware design and compare different design approaches is the power delay product (the energy per frame in our case) to compromise between throughput and power. Hence, we used the energy per frame as well as the accuracy to evaluate our design. The accuracy of the hardware generated from both design flows, is 68.184%, and it matches that of the golden reference after quantization while the accuracy of the golden reference before quantization is 69.362%, so the error is small and acceptable. The RTL design has a throughput of 608 frame/second with a power of 6.085 watts (0.01 joule/frame) on 50 MHz and 1216 frame/second with a power of 9.156 watts (0.007529 joule/frame) on 100 MHz, while the HLS design has a throughput of 24.84 frame/second with a power of 3.828 watts (0.154 joule/frame) on 80 MHz.

From the above summary of results, it is clear that the hardware generated either from the RTL design approach or the HLS design approach satisfies the required specs and proves that using FPGA instead of GPU guarantees higher speed and lower power, however, it needs much more effort and design time.

Finally, we compared the two design approaches, and we found out that the RTL design flow is better than the HLS design flow in many aspects and the above results show that. However, the HLS design approach is better than the RTL design approach in some aspects like time to market and verification development. Hence, the RTL approach is the most efficient design approach for digital electronics at that time, but HLS is a good design approach for small designs. In the future, The HLS can be an efficient design approach by paying more effort to the HLS tool development and optimization, in addition to allowing more controllability and observability for designers.

10.2 Future Work of RTL

This chapter summarizes points to be done as a future work to increase the performance of our design. This work will be summarized concisely in bullets.

- using double data rate memory to increase the speed of loading the photo memory with an image to make it faster by 2 and we can do this by three different ways.
    - We can interface with the DDR using 16-bit width data bus, so we can fetch 2 pixels by 2 pixels from the DDR memory to the Photo memory as the photo pixel width is 8-bits.
    - Using the rising and falling edge of the clock in DDR to capture different data so that to get a data word on each edge.
    - Use dual-port photo memory, each port has a separate clock write in the photo memory from the DDR using 200MHz and read from the photo memory to the model using 100MHz.
- make training for a certain application which reduces the number of classes needed and increases the accuracy of the model.
- ASIC implementation for our model to increase performance as there will be more control on our design.

10.3 Future Work of HLS

Future work are more techniques to speed up the hardware and to interface it with real designs to work in the industry as following:

- We can add more memories and **break the whole model into groups** so that the whole latency can be divided on the same number of groups.
- We can change the interface if the design so that we can interface it with **DDR interface** which can feed photos to the design in real time operation.

References

[1] Asifullah Khan1, Anabia Sohail1, Umme Zahoora1, and Aqsa Saeed Qureshi1, “A Survey of the Recent Architectures of Deep Convolutional Neural Networks”.

[2] Shiv Ram Dubey, Satish Kumar Singh, Bidyut Baran Chaudhuri, “Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark”, 2021.

[3] Forrest N. Iandola, “SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size”, 2016.

[4] Amir Gholami, Kiseok Kwon, Bichen Wu, Zizheng Tai, Xiangyu Yue, Peter Jin, Sicheng Zhao, Kurt Keutzer, “SqueezeNext: Hardware-Aware Neural Network Design”, 2018.

[5] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, Liang-Chieh Chen, “MobileNetV2: Inverted Residuals and Linear Bottlenecks”, 2019.

[6] Nasri Sulaiman, Zeyad Assi Obaid, M. H. Marhaban and M. N. Hamidon, “Design and Implementation of FPGA-Based Systems - A Review”, 2009.

[7] Zhang, Xiangyu, et al. "Shufflenet: An extremely efficient convolutional neural network for mobile devices." Proceedings of the IEEE conference on computer vision and pattern recognition, 2018.

[8] Ma, Ningning, et al. "Shufflenet v2: Practical guidelines for efficient cnn architecture design." Proceedings of the European conference on computer vision (ECCV), 2018.

[9] Kurama, V. (2020, March 19). A Review of Popular Deep Learning Architectures: DenseNet, ResNeXt, MnasNet, and ShuffleNet v2. Retrieved from PaperspaceBlog: https://blog.paperspace.com/popular-deep-learning-architectures-densenet-mnasnet-shufflenet/ [Accessed: 27 Jan, 2022].

[10] The ShuffleNet Series. (n.d.). Retrieved from https://iq.opengenus.org/shufflenet/ [Accessed: 27 Jan, 2022].

[11] Hollemans, M. (2020, April 8). New mobile neural network architectures. Retrieved from https://machinethink.net/blog/mobile-architectures/ [Accessed: 27 Jan, 2022].

[12] Brownlee, J. (2019, April 7). How Do Convolutional Layers Work in Deep Learning Neural Networks? Retrieved from Machine Learning Mastery : https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/ [Accessed: 27 Jan, 2022].

[13] “ShuffleNet V2”, PyTorch Hub for Researchers. [Online]. Available at: https://pytorch.org/hub/pytorch_vision_shufflenet_v2/ [Accessed: 5 March, 2022].

[14] Ningning Ma, Xiangyu Zhang, Hai-Tao Zheng, Jian Sun: ShuffleNet V2: Practical Guidelines for Efficient CNN Architecture Design. arXiv preprint arXiv:1807.11164, 2018.

[15] Steve Arar, “Fixed-Point Representation: The Q Format and Addition Examples”, All About Circuits, 2017, [Online]. Available: https://www.allaboutcircuits.com/technical-articles/fixed-point-representation-the-q-format-and-addition-examples/ [Accessed: 5 March, 2022].

[16] Philipp Gysel, “Ristretto: Hardware-Oriented Approximation of Convolutional Neural Networks”, 2016.

[17] Steve Arar. “Multiplication Examples Using the Fixed-Point Representation”, All About Circuits, 2017, [Online]. Available: https://www.allaboutcircuits.com/technical-articles/multiplication-examples-using-the-fixed-point-representation/ [Accessed: 9 March, 2022].

[18] Alcaraz, F., 2020. Fxpmath. Available at: https://github.com/francof2a/fxpmath [Accessed: 9 March, 2022].

[19] Harris, C.R., Millman, K.J., van der Walt, S.J. et al. Array programming with NumPy. Nature 585, 357–362 (2020). DOI: 10.1038/s41586-020-2649-2.

[20] “TORCH.TENSOR”, PYTORCH DOCUMENTATION. Available at: https://pytorch.org/docs/stable/tensors.html [Accessed: 9 March, 2022].

[21] Xilinx, 7 Series FPGAs Memory Resources, UG473, v1.14, 3-July-2019.

[22] “Basic HLS Tutorial”, so-logic, 25-January-2017.

[23] Xilinx, Vivado Design Suite User Guide, High-Level Synthesis, UG902, v2020.1, May 4, 2021.

[24] D. Gschwend, Master Thesis Report “ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network”, 2016.

[25] Ricardo Núñez Prieto, “Implementation of an 8-bit Dynamic Fixed-Point Convolutional Neural Network for Human Sign Language Recognition on a Xilinx FPGA Board”, 2019.

[26] Muhammad Sarg, Ahmed H. Khalil, Hassan Mostafa, “Efficient HLS Implementation for Convolutional Neural Networks Accelerator on an SoC”, International Conference on Microelectronics (ICM), 2021.

[27] Xilinx, VC709 Evaluation Board for the Virtex-7 FPGA, UG887, v1.6, March 11, 2019.

[28] Ahmed J. Abd El-Maksoud, Amr Gamal, Aya Hesham, Gamal Saied, Mennat-Allah Ayman, Omnia Essam, Sara M. Mohamed, Eman El Mandouh, Ziad Ibrahim, Sara Mohamed, Hassan Mostafa, “Hardware-Accelerated ZYNQ-NET Convolutional Neural Networks on Virtex-7 FPGA”, 2021.

[29] Ahmed J. Abd El-Maksoud, Abdallah Mohamed, Ahmed Tarek, Amr Adel, Amr Eid, Farida Khaled, Fatma Khaled, Ziad Ibrahim, Eman El Mandouh, and Hassan Mostafa, “FPGA Design of High-Speed Convolutional Neural Network Hardware Accelerator”, 2021.