An Automated Flow for Configuration and Generation of CNN based AI accelerators for HW Emulation & FPGA Prototyping

By

Ahmed Emad Ibrahim
Ahmed Hossam Mohammed Younes
Abd-Elrahman Khaled Abd-Elbadee
Abd-Elrahman Tarek Mohammed
Basem Essam Abd-Elsadeq
Mohammed Ahmed Mohammed

Under supervision of

Dr. Karim Abbas Ossama

A Graduation Project Report

Submitted to

The Faculty of Engineering at Cairo University
In Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science
in

Electronics and Electrical Communications Engineering

Faculty of Engineering, Cairo University

July 2022

Table of Contents

Table of Figures

Table of Tables

List of Abbreviations

ASIC	Application-Specific Integrated Circuit
CNN	Convolutional Neural Network
CPU	Central processing unit
CSV	comma-separated values
Conv	Convolution
FC	Fully Connected
FPGA	Field Programmable Gate Array
GPU	Graphics processing unit
HDL	Hardware description language
IC	Integrated Circuit
IFM	Input Feature Map
LUT	Lookup table
ML	Machine Learning
NRE	Non-recurring engineering
OFM	Output Feature Map
RGB	Red-Green-Blue
RTL	register-transfer level
ReLU	Rectified Linear Unit
WM	Weight Memory
e.g.	For example
i.e.	That is

Acknowledgments

We would like to take this opportunity to show our gratitude and to thank the people that made this project possible and gave us this great adventure and opportunity and without them we wouldn't have reached towards where we are now and the fruitful results that occurred at the end of the project.

We would like to thank specially Dr. Karim Ossama Abbas for being our mentor and supervising our project and teaching us about Digital Electronics all the past years and making us fall in love with it.

We would also like to thank Dr. Mohamed AbdelSalam from Mentor, A Siemens Business for sponsoring our project, for always following up with us and providing us with the needed tools and knowledge throughout the project, and a special thanks for Eng. Mahmoud Ali and Eng. Ahmed Elgendy for always being there for us and helping us if we had any technical difficulties.

Last but definitely not least we would like to thank our families for being patient and supportive with us through this year, giving us all the encouragement and always believing in us.

Abstract

Machine learning (ML) algorithms is some algorithms used to help AI learn without being explicitly programmed to perform the desired action. By learning a pattern from sample inputs, the machine learning algorithm predicts and performs tasks solely based on the learned pattern and not a predefined program instruction. Also, ML algorithms have proven to be a concrete component in various fields that aim to be fully automated. Therefore, many researchers have shed the light towards the modifications of ML algorithms to be fully automated for more complicated tasks. Convolutional Neural Networks (CNNs) is a class of deep learning neural networks that has major breakthroughs in the field of image recognition.

However, the acceleration of such algorithms is extremely hard due to high computations and memory required. Therefore, in our project we implemented an automated flow using Perl scripts to generate AI accelerators for various image classification based CNNs like Drone Agent, Lenet-5 and VGG16.

Our target is a high throughput, configurable and scalable RTL design that is generated by the Perl scripts. Our flow is designing and verifying using Veloce emulator due to its high capacity and fast simulation speed.

Chapter 1: Introduction

1.1 Motivation

Hardware acceleration is the approach of designing set of chips to be more convenient to run ML algorithms than general purpose computing solutions [1]. This kind of acceleration was introduced for the first time nearly 20 years ago. It rose with 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 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.

To achieve both training and inference processing constraints, 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 schedulers and multi-core processors. There are many products from different companies are introduced to fulfill the tasks of hardware acceleration [2]. These products can be categorized into four different computing devices: Graphical Processing Units (GPUs), Field Programmable Gate Array (FPGA), Application Specific Integrated Circuit (ASIC) and Central Processing Units (CPUs). GPUs (Graphical Processing Units) were deployed for only graphical processing at first. However, it has been found that they can be adopted for AI and achieved great results. FPGA is a class of computing elements which can be re-programed through its Look-Up Tables (LUTs). ASIC can be described as a family of processors to carry out certain specific task. For instance, in neural networks, the tasks are mainly multiplying and accumulating operations in addition to convolution.

Finally, CPUs is the electronic circuitry that executes instructions comprising a computer program. The CPU performs basic arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program. Among different ML algorithms, we have chosen CNNs to be accelerated for their huge number of applications including classification, image recognition and face detection. These primary applications are used in different numerous fields including autonomous driving, security systems and many other systems.

1.2 Problem Definition

Autonomous cars rely on CNNs that needs heavy computational power, so GPUs and CPUs are used but their disadvantage is that they have a lot of power consumption and are relatively slow, so another approach is to use a dedicated AI accelerator as it has low power consumption and better performance, but the down side is that it's designed to a specific CNN, So it has high nonrecurring engineering (NRE) costs to migrate to another CNN, it also lacks the ability to be retrained after deployment, So a generalized and configurable accelerators are needed.

1.3 Aim of the Project

The end of mind goal of the project is having a working configurable and scalable AI accelerator generated from automated flow using Perl scripts for different convolutional neural network models such as LeNet-5, AlexNet, VGG16, etc. Our target is a high throughput, configurable and scalable RTL design that is generated by Perl scripts. Our flow is designing and verifying using FPGA and Veloce emulator.

1.4 Solution Approach

We will divide our solution into two parts

- Designing the configurable AI accelerator
- Automated Flow for Configuration and Generation of RTL

1.4.1 Designing the configurable accelerator

CNN consists of many layers (convolution layer, pooling layer, fully connected layer... etc.), we will design each layer in RTL and test/verify them and for the optimization we have to consider the computational resources and memory bandwidth, then generalizing each layer this will give us the ability to write a script that will have the number of layers as well as the size of each layer as an input (config file) then it will generate an RTL file (top module) that has the fully connected CNN.

1.4.2 Automated Flow for Configuration and Generation of RTL

To solve the problem of the high NRE costs and the time consumed to migrate to another CNN an automated flow for the configuration and generation of the AI accelerator using perl scripts and a config file as in input which has the necessary parameters needed by the script to generate the required accelerator.

1.5 Organization

A brief introduction of the contents in each chapter is explained in the following:

Chapter 2 provides a background and an overview on convolutional neural networks and explains the different layers that it consists of in detail.

Chapter 3 gives examples on different CNN architectures and demonstrates their building block and a brief history on them and various information about the architecture.

Chapter 4 provides details about our automated flow to generate and configure the AI accelerator and explains how the perl scripts works to generate the AI accelerator's RTL in Verilog language using a config file as an input to the perl script to provide the needed information and parameters by the scripts.

Chapter 5 provides an overview on the emulator and FPGA flows and explains their steps and the pros and cons of using an emulator for the design.

Chapter 6 provides details about our hardware architecture and the different optimizations that we came up to increase the throughput through pipelining our design and explains in details, and how the memory is managed in the design.

Chapter 7 provides details about the software models and their training and way of extracting the weights in text format files in specific order suitable for the memory in ConvA and ConvB data flows.

Chapter 8 provides the design simulations results that we got using different data representations whether it was fixed-point representation or floating-point representation, and the simulation was done by Questa simulation tool and Veloce emulator tool.

Chapter 9 provides the results that we got from the different data representations and the throughput parameters and the design utilization data also it provides the info on power estimations and time analysis.

Chapter 10 concludes our project and work and introduces future works that can be done.

Chapter 2: Background and Related work

2.1 CNN Overview

Convolutional Neural Networks (CNNs) are a special type of Neural Networks which are commonly used with visual data, which have shown state-ofthe-art performance on various competitive benchmarks. The powerful learning ability of deep CNN is largely due to the use of multiple feature extraction stages (hidden layers) that can automatically learn representations from the data. The topology of CNN is divided into multiple learning stages composed of a combination of the convolutional layer, non-linear processing (ReLU) units, and subsampling (Pooling) layers. Each layer performs multiple transformations using a bank of convolutional kernels (filters). Convolution operation extracts locally correlated features by dividing the image into small slices, making it capable of learning suitable features. Output of the convolutional kernels is assigned to nonlinear processing (ReLU) units, which not only helps in learning abstraction but also embeds non-linearity in the feature space. This non-linearity generates different patterns of activations for different responses and thus facilitates in learning of semantic differences in images. Output of the non-linear function (ReLU) is usually followed by subsampling (Pooling), which helps in summarizing the results, and also makes the input invariant to geometrical distortions [3].

2.2 CNN Layers

Standard Convolution Layer

Convolution is the first layer to extract features from an input image. A convolution layer receives n input feature maps (IFM) and generates m output feature maps (OFM). Each input feature map is convolved by a shifting window with
K x K kernel (filter) to generate one element in one output feature map Eqn. (1). The stride of the shifting window is S, which is normally smaller than K. A bias value is added to each pixel in OFM as shown in Eqn. (1) [4].

$$Y_i = f\left(bias + \sum_{j=1}^{n} X \oplus k_{ij}\right) \quad i = (1,2,\ldots..,m) \qquad (1)$$

The main computations in convolutional layers are multiplication and accumulation (MACs) and the number of MACs is K x K.

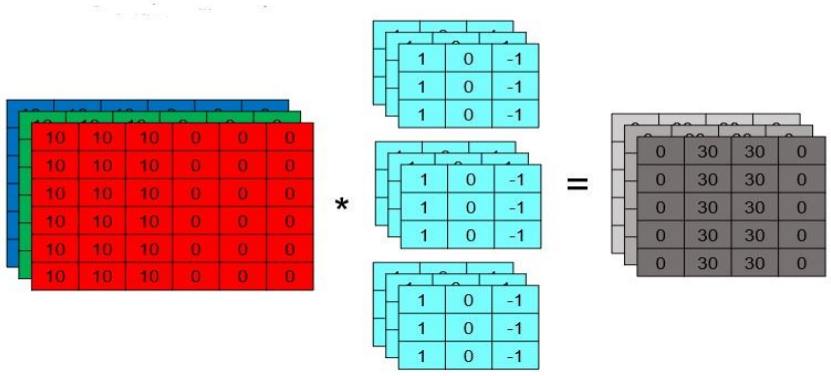


Figure 1: Convolution Layer Operation

Winograd Convolution Algorithm

Winograd proposed the minimum filtering algorithm of finite impulse response (FIR) filtering in 1980. The minimum filtering algorithm gives m output generated by the FIR filter of r taps, namely F(m, r), the minimum number of multiplications

needed μ(F(m, , r)) is m + r − 1. Taking F(2,3) as an example, the input d = [d0, d1, d2, d3] is a vector of size 4, filter g = [g0, g1, g2]^T , then:

$$F(2,3) = \begin{pmatrix} d_0 & d_1 & d_2 \\ d_1 & d_2 & d_3 \end{pmatrix} \begin{pmatrix} g_0 \\ g_1 \\ g_2 \end{pmatrix} = \begin{pmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 - m_4 \end{pmatrix}$$

where

$$m_1 = (d_0 - d_2)g_0 , \quad m_2 = (d_1 + d_2)\frac{g_0 + g_1 + g_2}{2}$$

$$m_4 = (d_1 - d_3)g_2 , \quad m_3 = (d_2 - d_1)\frac{g_0 - g_1 + g_2}{2}$$

When calculating [m_1, m_2, m_3, m_4], the number of multiplications involved in the algorithm is μ(F(2,3)) = 2 + 3 − 1 = 4 multiplications, the number of addition operations that need to be performed on d is 4, and the number of addition operations that need to be performed on g is 3 (the value of $g_0 + g_2$ can be calculated only once); using[m_1, m_2, m_3, m_4] to get the result of F(2,3) requires 4 additions. The number of multiplications required by the algorithm has been reduced from 6 to 4.

Winograd minimum filtering algorithm can be expressed in the form of a matrix:

$$y = A^T[(Gg) \odot (B^T d)]$$

where$g$ is the filter vector, d is the input data vector, Y is the output data vector, G is the filter transformation matrix, B^T is the data transformation matrix, ⊙ is the corresponding bit multiplication of the matrix (Hadamard product), A^T represents the output transformation matrix.

For (2,3), the matrices are:

$$B = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}, \qquad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix},$$

$$A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}, g = [g_0 \quad g_1 \quad g_2]^T, d = [d_0 \quad d_1 \quad d_2 \quad d_3]^T$$

So, in general A^T, G, B^T are fixed matrices consisting of zeros, ones and 1/2 which maps to addition, subtraction and shift right operations.

By nesting the one-dimensional minimum filtering algorithm (m, r) , we can get the two dimensional minimum filtering algorithm $F(m \times m, r \times r)$:

$$y = A^T[(GgG^T)_o(B^T dB)]$$

Now the size of the filter g is $r \times r$, the size of output Y is $m \times m$, and the size of input d is $(m + r - 1) \times (m + r - 1)$. The number of multiplications required by the two-dimensional minimum filtering algorithm is $(m + r - 1)^2$, while the number of multiplications required by the original convolution algorithm is $m \times m \times r \times r$.

For $(2 \times 2,3 \times 3)$, the number of multiplications is reduced from 36 to 16, which is a reduction of 2.25 times. Even if the additional addition operations are included, there are great benefits.

Depth-Wise Convolution

Depth-wise Separable Convolution becomes a sophisticated evolution in the CNN modern architectures like MobileNet and Xception architectures. The main advantage of the DSC is concluded on the number of multiplication operations per

layer in DSC is less than in the standard convolution. It replaces internal multiplication operations with additions through separate the standard convolution layer into two layers; Depth-wise and Point-wise convolution layers.

We can say that the Depth-wise Separable Convolution has two advantages due to this reduction in multiplication as following:

- They have smaller number of parameters to adjust as compared to the standard CNN's, which **reduces over-fitting** during architecture training.
- They are **computationally cheaper** because of fewer computations which make them suitable for mobile vision applications.

Differences Standard CNN and DWS Convolution

1. Standard CNN

Standard/usual convolution performs the channel-wise and spatial-wise computation in one step. As we have mentioned before that the standard CNN is to do filter to image part multiplication and add partial sums in three dimensions to get one output per operation.

2. DWS Convolution

It contains two layers of convolution. The first one is Depth-wise convolution it's look like the usual convolution except it doesn't sum up the partial sum across the channels, where the output channel of input with the corresponding channel in filter have the responsibility of the corresponding output channel. The second layer is Point-wise convolution; it's a usual convolution with kernel 1x1, its important to combine the features of the input with others to satisfy the training and feature dependency.

3. Comparison according to multiplication cost

3.1 Standard CNN

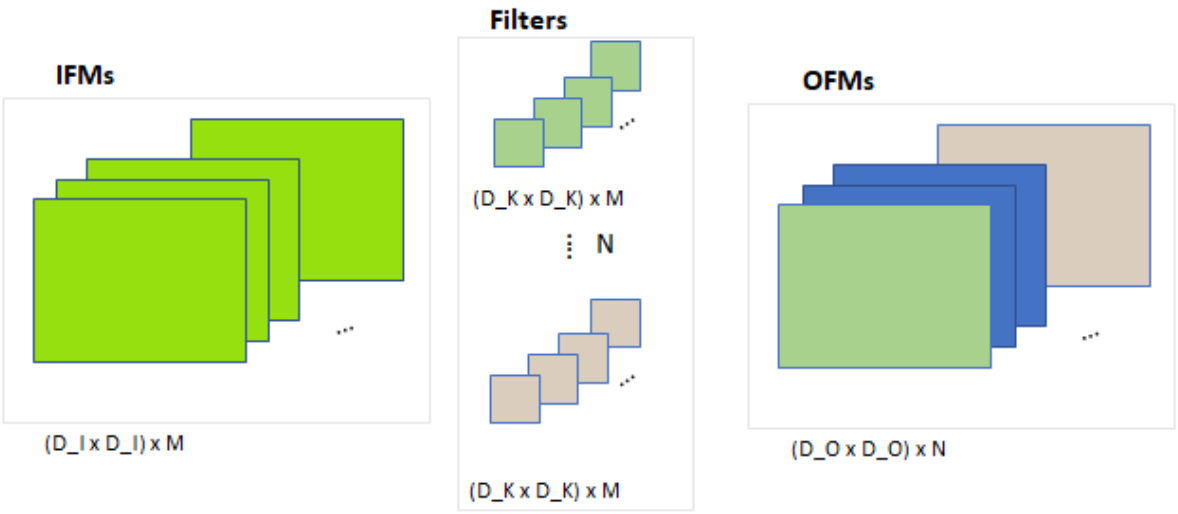


Figure 2: Standard convolution describes multiplication cost

According to Figure 2, we can conclude the multiplication cost of Standard CNN as following:

- # Multiplication for one pixel = size of one filter = (D_K x D_K) x M = $D\_K^2 * M$
- # Multiplication for one OFM = pixel cost * size of OFM = $D\_K^2 * M * D\_O^2$
- # Multiplication for all OFM = one OFM cost * # Filters = $D\_K^2 * M * D\_O^2 * N$

3.2 DWS Convolution

Figure 3: Depth-wise convolution shows the Multiplication cost per layer

From Figure 3, we can conclude the multiplication cost of Depth wise layer as following:

- Each input channel generates one output channel without merging any channels
- Multiplication cost per channel = size of one filter * size of output channel

$$= (D\_K \times D\_K) * (D\_O \times D\_O) = D\_K^2 * D\_O^2$$

- DW: Multiplication cost = Channel cost * M = $D\_K^2 * D\_O^2 * M$

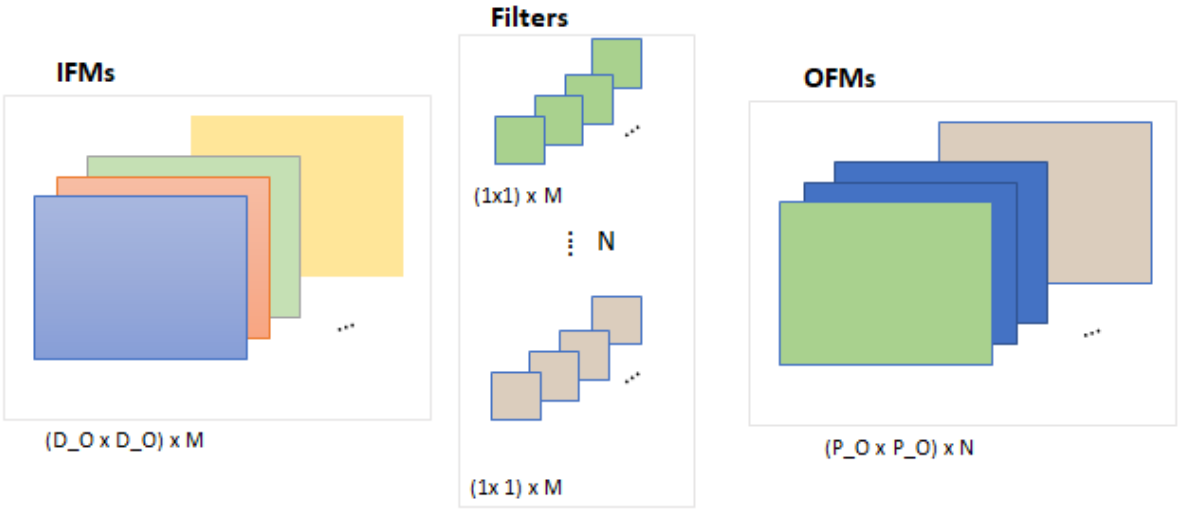


Figure 4: Point-wise convolution shows the multiplication cost per one layer

According to Figure 4, we can conclude the multiplication cost of Point-wise layer as following:

- Point wise convolution is a standard convolution with filter size (1 * 1 * M).
- P_O = D_O
- PW : Multiplication cost = $1^2 * D\_O^2 * M * N = D\_O^2 * M * N$

The total multiplication cost of DWS convolution layer is DW cost and PW cost.

- Total cost = PW cost + DW cost = $D\_O^2 * M * (D\_K^2 + N)$
- Ratio between standard CNN and DWS Convolution = $\frac{Depthwise\ cost}{Standard\ Conv\ Cost}$

$$= \frac{1}{N} + \frac{1}{D\_K^2}$$

So we get a reduction with ($\frac{1}{N} + \frac{1}{D\_K^2}$) in multiplication cost, that is what makes it suitable for mobile neural network applications.

Padding layer

Padding is usually used to preserve the information that exists near the edge of the image. In addition, it's used to preserve the original input size. No padding is called Valid Padding, and using padding to preserve the original input size is called Same Padding as shown in Figure 5.

0	0	0	0	0
0	25	12	10	0
0	11	5	22	0
0	9	23	7	0
0	24	32	10	0
0	0	0	0	0

Figure 5: Padding Layer

Rectified linear unit (ReLU)

The ReLU layer is a piecewise-linear operation that performed after every Convolution layer [5]. The output is given by max (0, input), meaning that if the input is less than zero or equal to zero the output is zero, otherwise the output is the same as the input. Figure 6 shows the functionality of ReLU activation function.

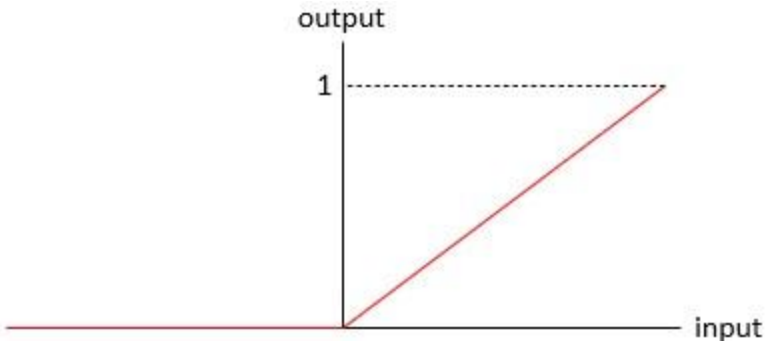


Figure 6: ReLU operation

Pooling Layer (Pool)

Pooling layer (also called sub-sampling) reduce the dimensionality of each feature map of its input feature maps, but retain 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. Pooling can be of different types: Average pooling and Max pooling. As shown in Figure 7 and 8 respectively.

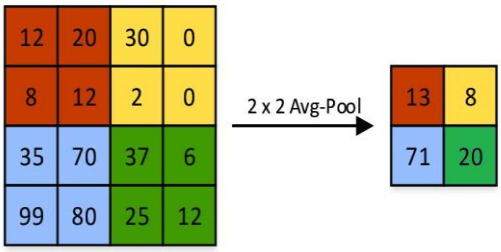


Figure 7: Average Pooling

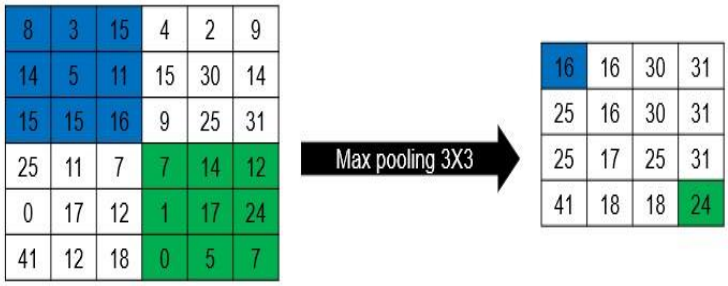


Figure 8: Max Pooling

Fully Connected Layer (FC)

The way this fully connected neural network layer (FC) works is that it looks at the output of the previous layer (which represent 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. Also, this layer is followed by soft max to show the most correlated class to the input as shown in Figure 9 which is an example for image classification [6] .

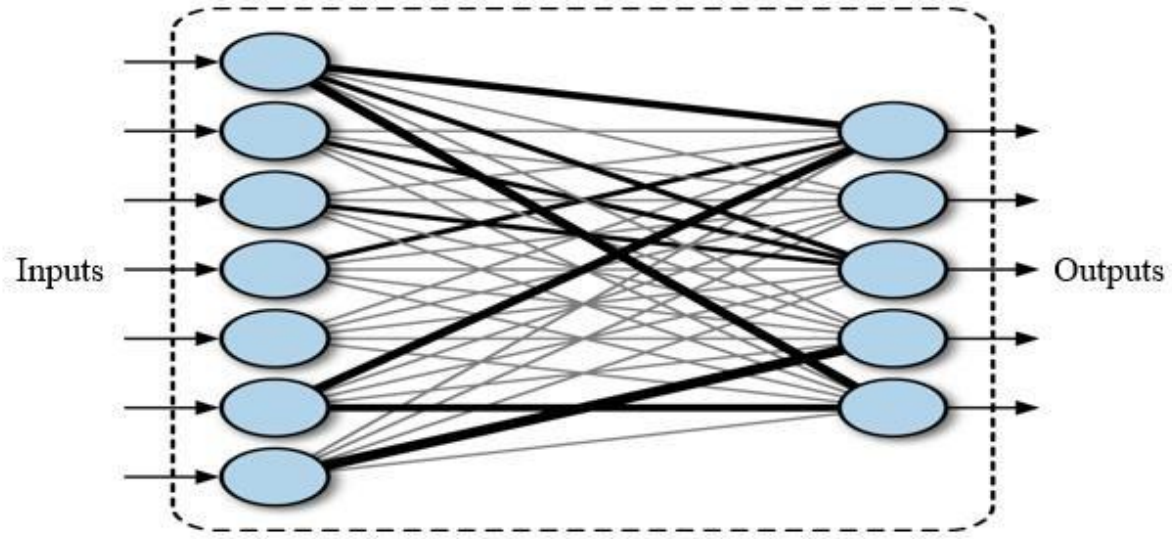


Figure 9: Fully Connected

Stride

Stride is the step in which the filter moves with across the image until it reaches the upper right-hand corner. Stride is used mainly to decrease the output size so processing will be easier.

Flatten

Flattening is always the last layer before the fully connected layer and it's used to convert the data into 1-dimensional array for inputting it to the next layer which is usually the fully-connected layer, we flatten the output of the convolutional layers (convolution, pooling, …etc.) to create a single long feature vector, and it's connected to the final classification model, which is called a fully-connected layer, in other words, we put all the pixel data in one line and make connections with the final layer.

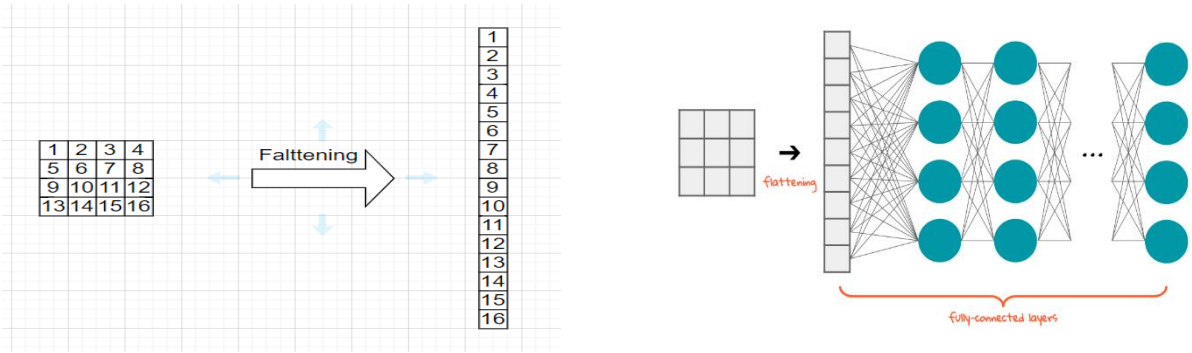


Figure 10: Flatten operation

That was the concept but from the hardware aspect how to implement this layer. It depends on the previous layer, if the previous layer was slow and gives only one output every cycle or more than cycle it will be easy to implement the flatten layer in the hardware as the output will pass directly to the FC-layer with a control signal, but if the previous layer was faster and give more than one output every cycle, we must store this output in a two port memory and read it using address from the FC-layer side to be able to read the whole output from the previous layer and in order also.

Soft Max

Soft Max is used to generate the probabilities of the outputs. Its main advantage is that it gives accurate output results, so it is used during the training of the model and tuning the parameters.

Its main disadvantage is the complexity in design as it has exponential and division operations which takes much area and consumes high power.

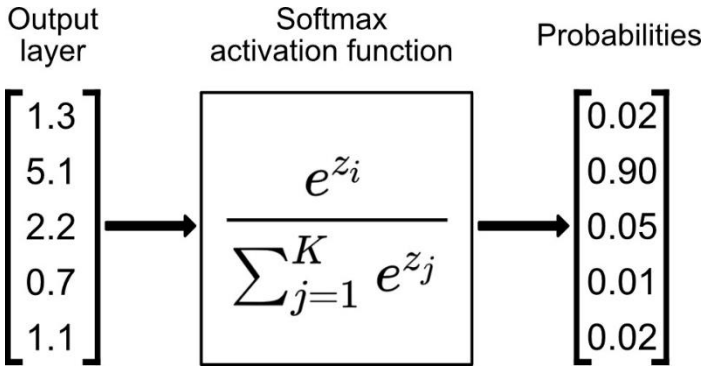


Figure 11: Soft Max operation

Hard Max

Hard Max is the last layer in the design instead of the SoftMax because it is simple in design and has less area than the SoftMax.

It is simply a comparator that compares the output of the input numbers together then set the largest number equal to one and the rest is equal to zero, so it is mainly used during testing.

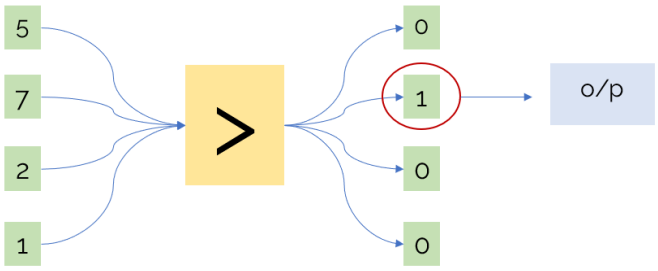


Figure 12: Hard Max operation

Chapter 3: CNN Architectures

Our goal is to have a configurable and generic architecture for image classification, and we used a custom model (AI drone Agent) as a reference in our work.

3.1 AI drone Agent

This model was graduation project sponsored by siemens for drone Autonomous Navigation it used digital twin simulator to extract data and train the model

The model achieved 96% accuracy

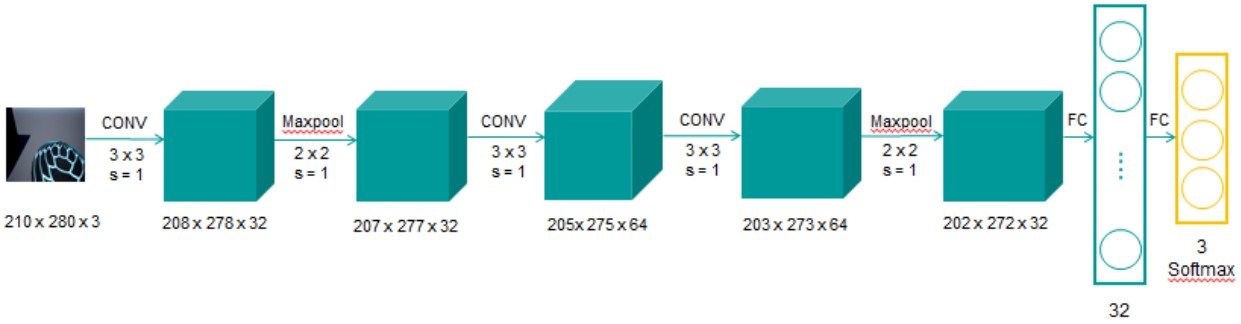


Table 1: AI drone Agent Architechure Dimensions

Layer	Size			
	Input	Filter	Bias	Output
C1	280x210x3	3x3x3x32	32	278x208x32
P1	278x208x32	2x2	N/A	277x207x32
C2	277x207x32	3x3x3x64	64	275x205x64
C3	275x205x64	3x3x3x32	32	273x203
P4	273x203x32	2x2	N/A	272x207x32
FC1	1x1x32	272x202x32x32	32	1x1x32
FC2	1x1x3	1x1x32x3	3	1x1x3

3.2 MobileNet

MobileNet is a CNN architecture that was developed by researchers at Google in 2017 that is used to incorporate Computer Vision efficiently into small, portable devices like mobile phones and robots without significantly reducing accuracy. The total number of parameters in a standard MobileNet is 4.2 million, which is significantly lesser than some of the other CNN architectures.MobileNets also give model developers the flexibility to control the size of their model(at the cost of accuracy) depending on their requirements by introducing two new global hyperparameters that can be tuned according to the requirements of the model developer[7].

Table 2: MobileNet Architecture Dimensions

Layer	Size					
	Input	Filter	Padding	Stride	Bias	Output
C1	224x224x3	3x3x3x32	Valid	2	32	112x112x32
DW1	112x112x32	3x3x32 DW 1x1x32x64 PW	Same	1	64	112x112x64
DW2	112x112x64	3x3x64 DW 1x1x64x128 PW	Same	2	128	56x56x128
DW3	56x56x128	3x3x128 DW 1x1x128x128 PW	Same	1	128	56x56x128
DW4	56x56x128	3x3x128 DW 1x1x128x256 PW	Same	2	256	28x28x256
DW5	28x28x256	3x3x256 DW 1x1x256x256 PW	Same	1	256	28x28x256
DW6	28x28x256	3x3x256 DW 1x1x256x512 PW	Same	2	512	14x14x512

DW7 : DW11	**14x14x512**	**3x3x512 DW 1x1x512x512 PW**	**Same**	**1**	**5x512**	**14x14x512**
DW12	**56x56x128**	**3x3x512 DW 1x1x512x1024 PW**	**Same**	**2**	**1024**	**7x7x1024**
DW13	**7x7x1024**	**3x3x1024 DW 1x1x1024x1024 PW**	**Same**	**2**	**1024**	**7x7x1024**
P14	**7x7x1024**	**7x7 AgvPool**	**Valid**	**1**	**N/A**	**1x1x1024**
FC1	**1x1x1024**	**1024x1000**	**Valid**	**1**	**1000**	**1x1x1000**

3.3 LeNet-5 Architecture

LeNet-5 is used for number detection using MNIST data set for training, and it consists of 7 layers, three convolution layers, two pooling layers and two fully connected layers [6].

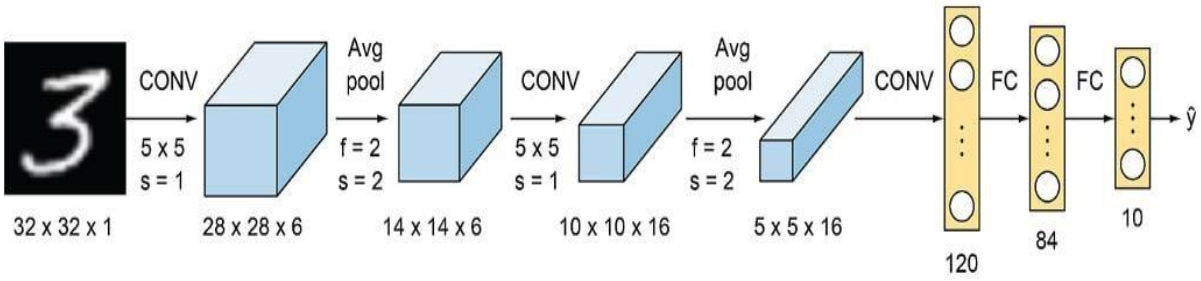


Figure 13: LeNet5 Architecture

In our implementation the input is 32x32x1, and the following table illustrates the dimension of each layer and their input and output.

Table 3: LeNet-5 Architecture Dimensions

Layer	Size			
	Input	Filter	Bias	Output
C1	32x32x1	5x5x1x6	6	28x28
P1	28x28x6	2x2	N/A	14x14x6
C2	14x14x6	5x5x6x16	16	10x10x16
P2	10x10x16	2x2	N/A	5x5x16
C3	5x5x16	5x5x16x120	120	1x1x120
FC1	1x1x120	1x1x120x84	84	1x1x84
FC2	1x1x84	1x1x84x10	10	1x1x10

3.4 VGG-16 Architecture

VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper “Very Deep Convolutional Networks for Large-Scale Image Recognition”. The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. Figure 9 shows architecture of VGG16.

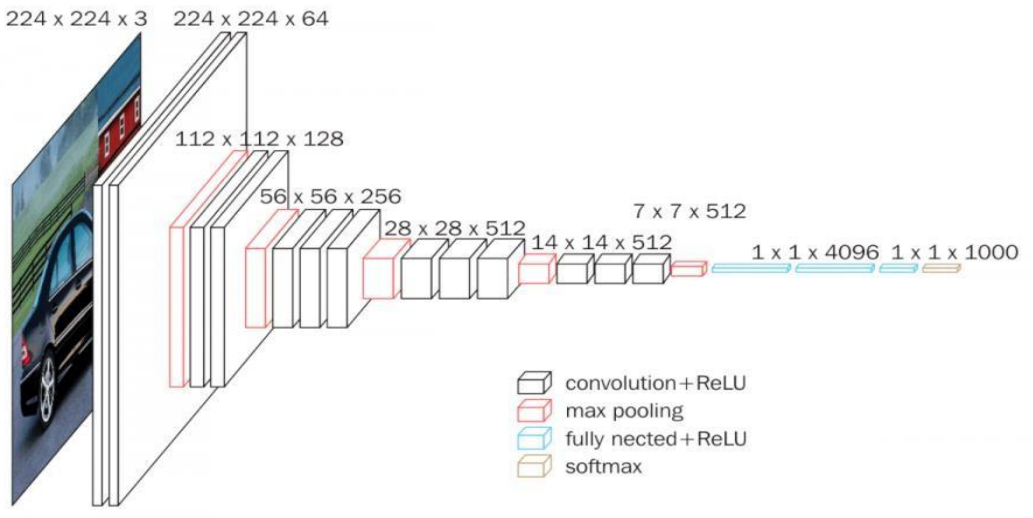


Figure 14: VGG16 Architecture

Chapter 4: RTL Automation

4.1 Overview

Our goal is to have a generic and configurable AI accelerator peripheral, to achieve this goal many Perl scripts are used to generate the synthesizable RTL for the different architectures.

Each layer consists of many Perl scripts and each script generates a component in the layer depending on the needs of the layer then another script for connecting the top module together and the final synthesizable RTL is generated where the whole flow is automated.

The choice in the scripting language and choosing specifically the Perl language is made due to the many capabilities of Perl, so we need to shed the light on this

essential and durable language, the next section will discuss the history of the Perl Language.

4.2 Perl Language History

Perl was developed by Larry Wall in 1987 as a multi-purpose scripting language for UNIX to make the processing of report easier. [1] Since then, Perl language has undergone many adjustments and versions [8].

The fact that Larry Wall graduated with bachelor's degree in Natural and Artificial Languages made the Perl language better for text manipulation from the other scripting languages out there which for our goal of generating a synthesizable RTL written in Verilog Language is a main advantage that tipped the scale in favor for Perl for writing our scripts.

The Perl language borrow features from other programming languages including C, shell script (sh), AWK, and sed. The Perl language provide text processing capabilities without the limitation of data lengths, it is called and nicknamed “the Swiss Army chainsaw of scripting languages” due to its flexibility and power and the versatility of applications Perl can be used in like system administration, network programming, finance, bioinformatics, and other applications like GUIs.

4.3 Advantages of Having a Generic and Configurable Architecture

There is many advantages in having an automated flow for CNN generation and having a generic architecture design that can be molded into different architectures of image classification CNNs like LeNet-5, VGG16, and any custom model that could be constructed by layers supported in the tool, as a start it would save for companies and different projects a lot of time when migrating from an architecture to another and also a big saving in money by reducing the NRE costs (Non-recurring

engineering costs) for writing and modifying the RTL of the accelerator, so we don't have to rebuild the wheel from scratch every time we need to change the architecture also the luxury of testing different architectures in the same time and measuring the difference in accuracy and performance between them is now a very simple task, where you only need to change the parameter of the current architecture to any other architecture needed.

The automation and the ability to easily change the used architecture also would help academic researchers in comparing between different architectures and calculate the different performance parameters like throughput and inference time.

4.4 The Configurability

The layers are writing in a scalable matter as the script can duplicate the core hardware for intensive operations (i.e., Multipliers and Adders) to increase the parallelism with the idea of mind that our design should be easily customized to be able to obtain the needed throughput at the cost of additional hardware.

The configurability of the design is achieved by sending a CSV (comma separated values) formatted file which acts as a .config file to the Perl script where the necessary information is written in it which is required by the Perl script to run and generate the synthesizable RTL of the architecture.

The following figure illustrates the process where the parameters are given as an input to the script and different modules are generated and then connected together forming the final AI accelerator.

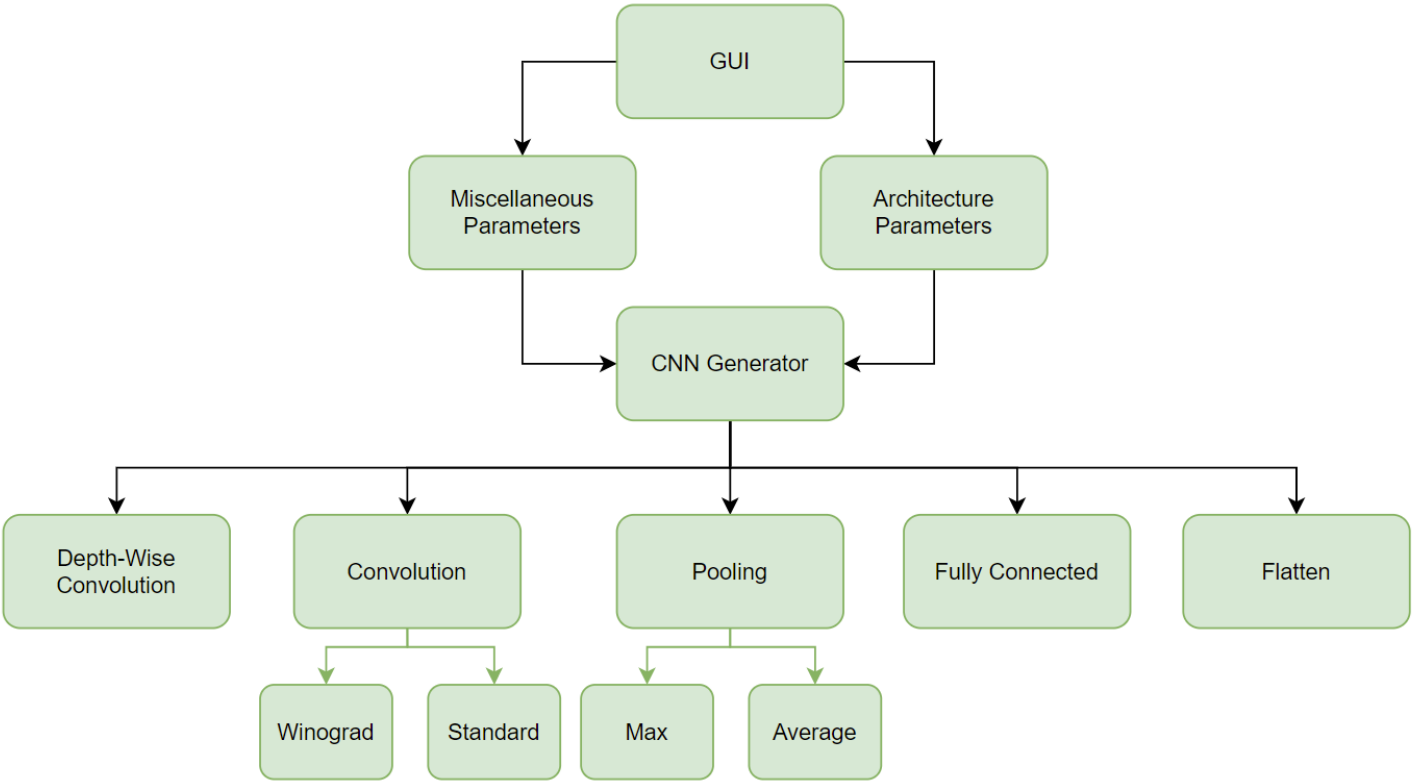


Figure 15: Script Parameters Diagram

The information required by the Perl is the main information that characterizes the given architecture which are as follow

- The data width
- Address bits
- Operation Type (Float or Fixed-Point mathematical operations)
- Input image is RGB or Grayscale
- Convolution type (Standard or WinoGrad Convolution).

The previous parameters are general parameters of the whole architecture, but the user must also provide the parameters of each layer like:

- Layer type and position in the architecture
- The kernel size
- Stride
- The Input feature map size and channels
- Output Channels (Number of filters)
- Number of units of resources needed for the layer
- Padding state (In convolution layers)

Depending on these inputs the Perl can generate the required architecture and layers that the user required.

The layers RTL is manipulated by Perl depending on certain conditions in model for example the layers sequence, the layers parameters, the miscellaneous parameters, and the number of units in each layer will all determine the final shape of each layer's control unit and data path.

4.5 RTL Automation

The core of the Perl scripts is the generic RTL that has been wrote with the idea of mind that our design should be easily customizable so the ability of generating different architectures from it become achievable.

Each Layer has its own Perl script where the layer RTL is manipulated by Perl, to gain the property of customizability and generating the needed control signals from the control unit of the layer.

Depending on the parameters given to Perl script from the config file that we discussed earlier the number of resources, memories, data bus and control signals are

generated and integrated together from each layer to have a fully functioning CNN architecture written in Verilog Language where all the logic of generating the architecture is hidden from the user and the only job for the user is to input the parameters needed.

The parameters that are given to the Perl script must be written correctly or else the generated RTL will not work as expected, having a great and intelligent tool where the RTL generation is automated is useless if the one using it, doesn't understand what the tool needs to work that's why with each Perl script a clear and concise documentation of the arguments given to the script as input are stated in the beginning of each file and also a guide that is formatted in an excel sheet is made to make the process of giving the inputs to the script more easy.

Chapter 5: Emulator and FPGA Flows

5.1 Introduction of Emulator

Emulator is designed to simulate the behavior of one or more hardware pieces with another hardware platform. Hardware emulation is generally used to debug and verify a system under design. Emulator is an array of FPGAs connected together and used to emulate RTL on real gates. It has a design flow to implement the design in the gates. As with everything it has some advantages and disadvantages which we are going to list.

Advantages of the Emulator

1. It is more helpful in debugging and increase the visuality of the design.
2. User can access any node in design.
3. It has a capacity of gates up to 50 billion gates.
4. It has different configurations depending on the capacity.

Disadvantages of the Emulator

FPGA is faster than emulator, as the FPGA can speed up to Giga Hz and emulator's speed up to Mega Hz.

5.2 Emulator Modes

Hardware acceleration mode

Write synthesizable RTL and test bench and burn them on emulator.

In circuit emulation mode

There is a data cable connected between the emulator and box and it burn part of RTL on emulator and the rest of the RTL is burned on FPGA inside the box. This method saves the emulator's resources. Sometimes this box is used for speed adaptation as the emulator is slow and the USB is fast to help us in verification of the input.

Co-modeling

Like in circuit emulation mode part of the RTL is burned on emulator and the rest is burned in test bench environment that simulated in block connected by emulator.

Figure 16 shows the three modes of the emulator.

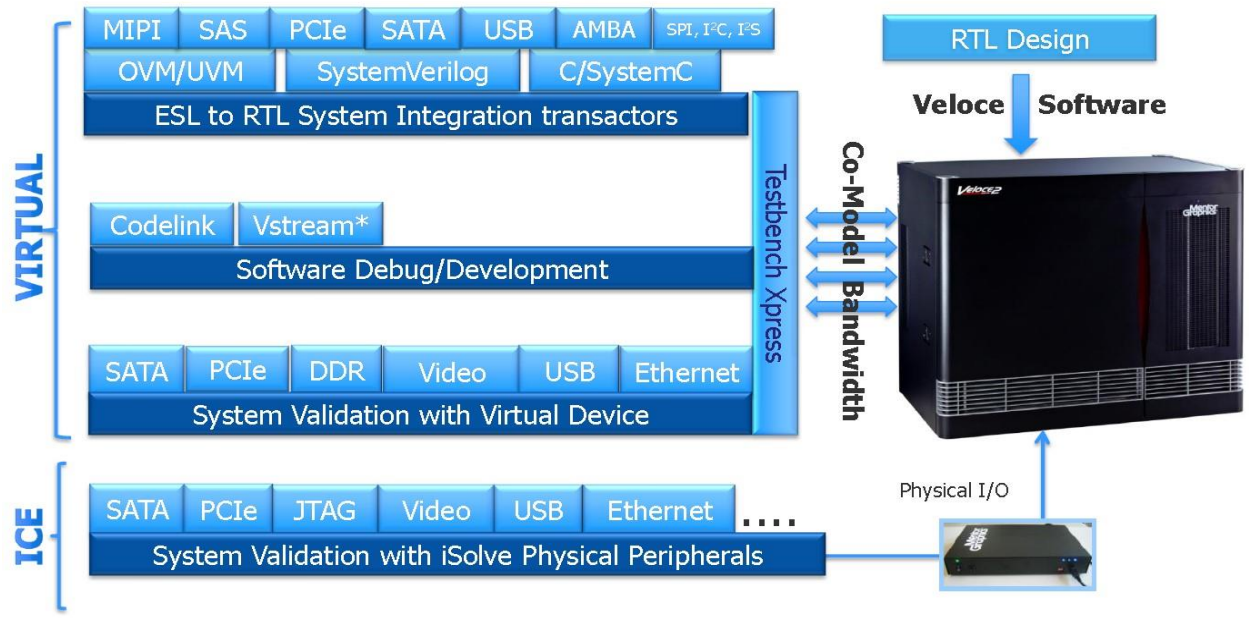


Figure 16: Emulator Modes

5.3 Emulator Flow

Emulation has two steps compile flow and runtime flow.

5.3.1 Compile Flow

1- RTL Compilation: it is divided into two steps analyze and RTLC (RTL Compiler). The user inputs RTL in VHDL/Verilog, System Verilog, Verilog gates, Translation libraries memories Synthesizable / parameterizable, patterns.

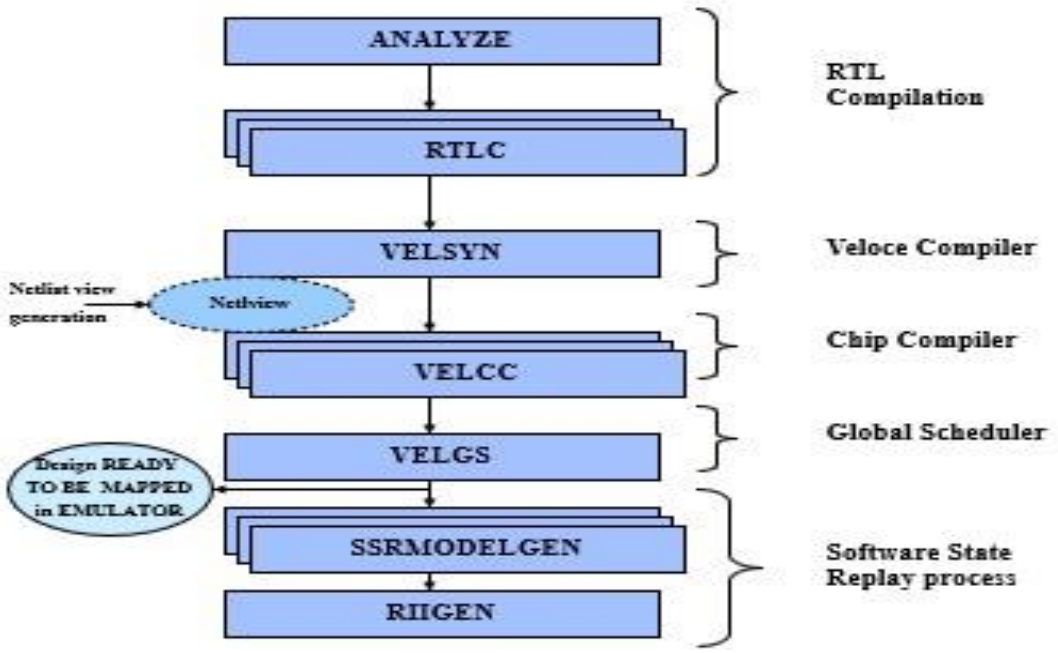
- Analyze: Used to check the syntax.
- RTL Compiler: used to generate structural Verilog netlist (Generic), debug database and do parallel task.

2- Synthesizer: User inputs Platform, Timing Specifications, IO mapping, bonded out cores and Design annotations.

- Veloce Synthesizer: Re-synthesis, performs partitioning & placement System-level routing (picking which signals go where), ASIC netlists for each crystal chip, Timing analysis, SSR model extraction (create chip level design database for SW State Replay).

3- Backend: No user input and it is divided into four steps.

- Veloce Chip Compiler: used to Place and Route the Crystal chip.
- Veloce Global Scheduler: Final Timing analysis, Generates timing information for emulator events and resources access.
- Software State Replay Model Generation: Generates models to replay states for each chip, Parallel tasks (one per chip), Necessary only for visibility (waveform analysis).
- Recon Input Index Generation: Creates databases visibility system



- Figure 13 shows the emulator compile flow

Figure 17: Emulator Compile Flow

5.3.2 Runtime Flow

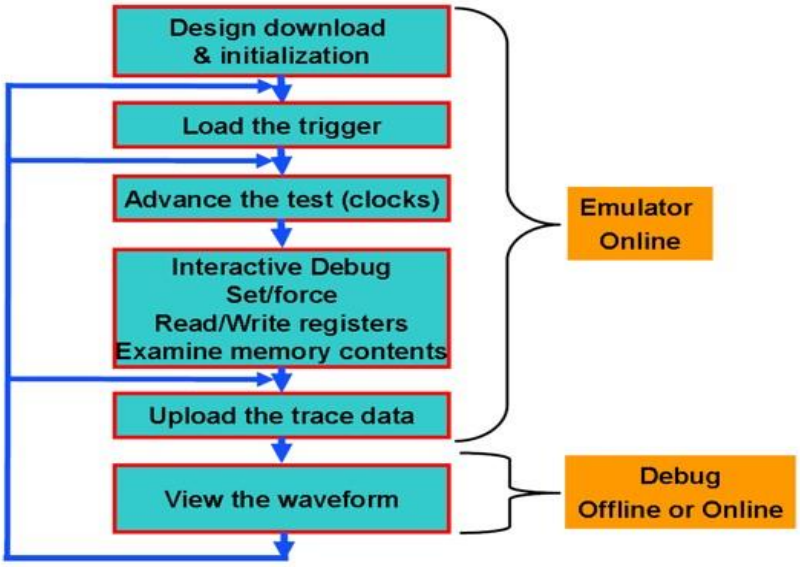


Figure 18: Emulator Runtime Flow

5.4 FPGA Flow

An FPGA is an IC consisting of programmable logic gates and interconnections that can be programmed using an HDL (hardware descriptive language) to do a specific function. It is an integrated circuit designed to be configured by a customer or a designer after manufacturing – hence "field-programmable". We will discuss the steps in FPGA flow.

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

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

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

delay of the application and calculate the maximum clock speed achieved for the application.

4. Place & Route: The logic blocks and cells in the FPGA are connected together, and Static Timing Analysis is done again to calculate the exact delay model of the application.
5. Download & verify in circuit: The HDL code is burned on the FPGA.

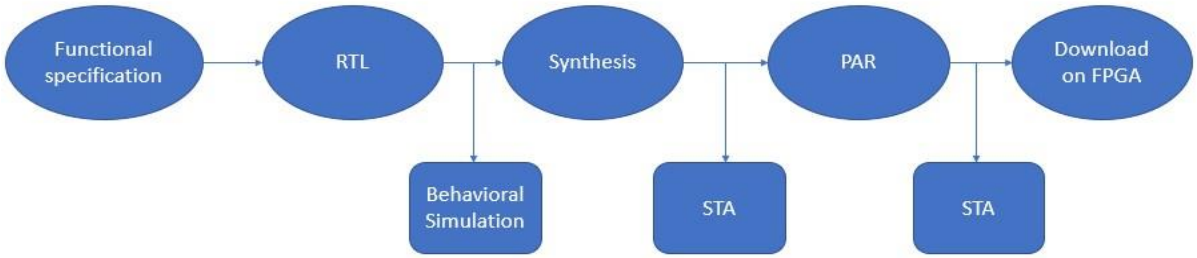


Figure 19: FPGA Flow

Chapter 6: Hardware Architecture

Our target is that we want to design the layers to be configurable and parameterized as the Perl scripts can generate each layer as we need in each model depending on the config input file that specify the parameters for each layer such as (filter size, stride, number of channel). Next, we will discuss our design in each layer and integrate our layers to build models. The most important layer in any model is the Convolution layer because it demands high resources, and it is the bottleneck in our design.

6.1 Winograd Convolution

We can split Winograd convolution into four separate stages as shown in the next

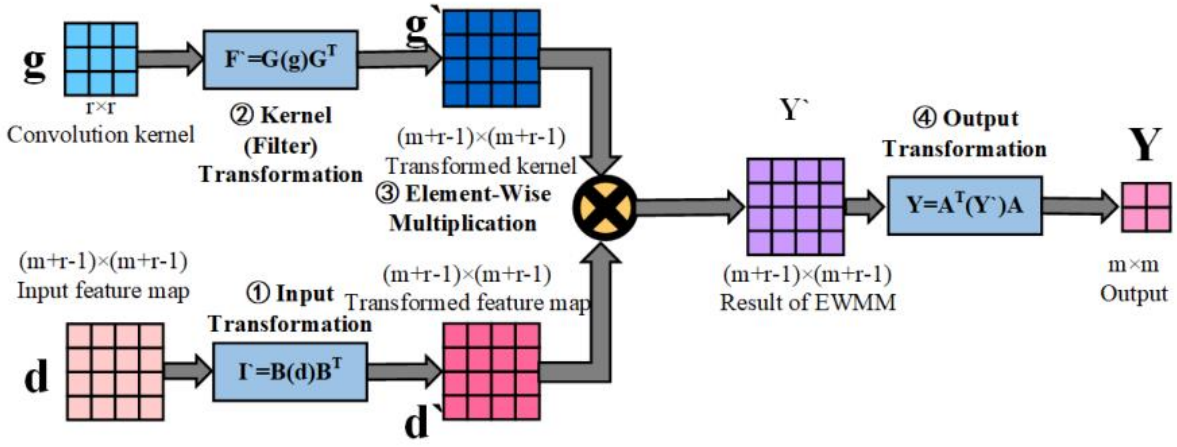


Figure:

Figure 20: Winograd Architecture

1- Input Transformation (ITrans): $d' = B^T dB$, transform the input tensor to Winograd domain, the size of d' is $(m + r - 1) \times (m + r - 1)$.
2- Kernel Transformation (KTrans): $g' = Gg$, transform the convolution kernel to Winograd domain, the size of g' is $(m + r - 1) \times (m + r - 1)$.
3- Element-Wise Matrix Multiplication (EWMM): $Y' = g' \odot d'$, which is the calculation stage of Winograd convolution, the size of Y' is $(m + r - 1) \times (m + r - 1)$.
4- Output Transformation (OTrans): $Y = A^T Y'$, inversely transform the result of EWMM from Winograd domain to feature map tensor domain, the size of Y is $m \times m$.

Normal Convolution operation

First: Point wise Multiplication

9 Multiplications

10	10	10
10	10	10
10	10	10

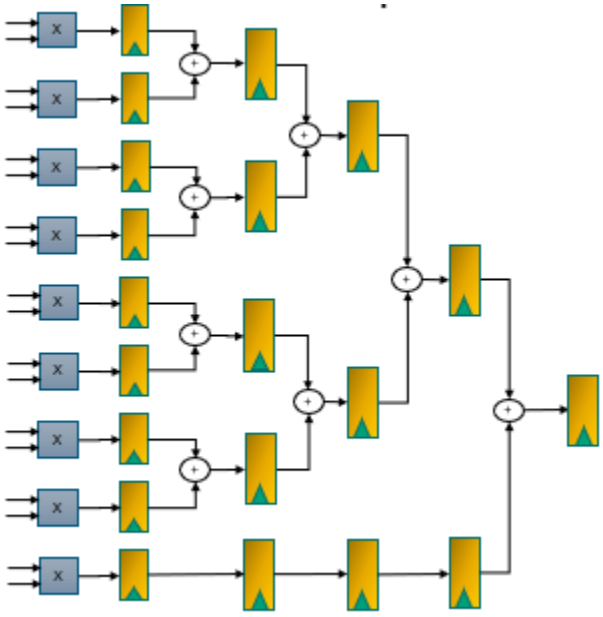
*

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

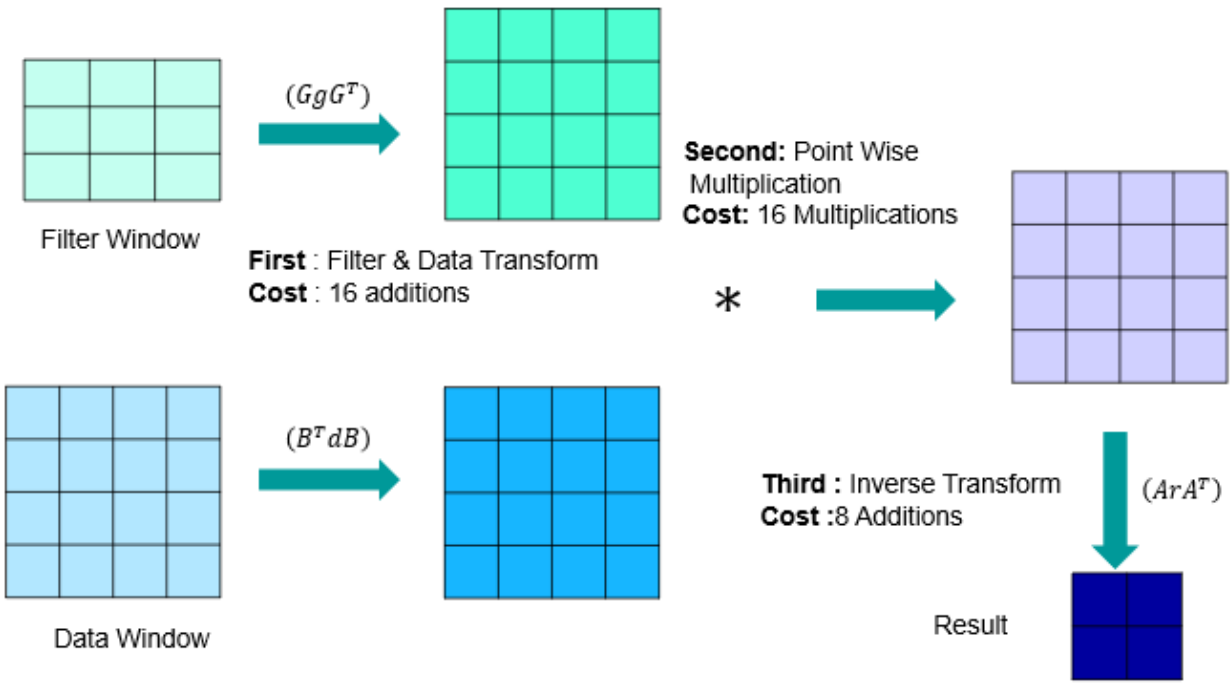
= 0

Second: Addition

8 Additions



Winograd Operation

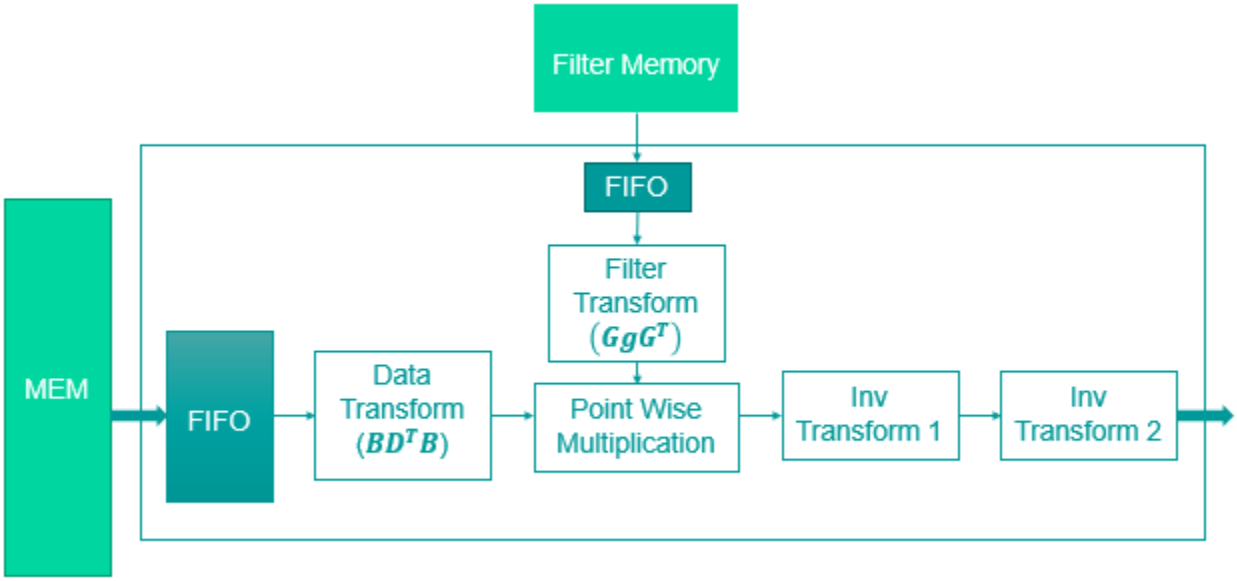


Winograd operation Cost	Normal Operation Cost
16 Multiplications + 24 Additions	36 Multiplications + 32 Additions

Limitations:

- It only efficient with 3*3 Convolution, once the filter size is bigger the transformation matrix values will have numbers and it will need Multipliers
- Also, it is only valid with stride 1 Convolution because It depends on the overlap of the data

Data Path Unit Flow

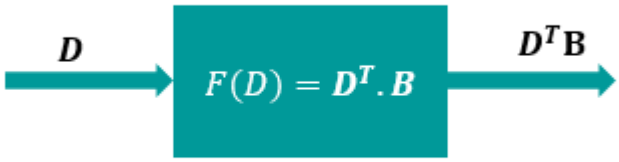


The data and filter will be loaded from the memory the transformation occurs the filter transform will only happen once with each channel, then pointwise multiplication finally inverse transform to get the final data

Matrix Transform operation

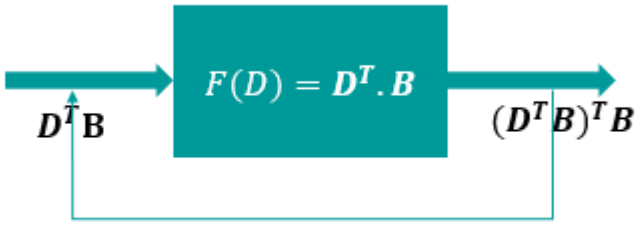
Matrix Transform Operation is nesting 2 transform matrices and we have stall Cycle so we may utilize same hardware by rewriting the equation as follows:

First Iteration:



First Iteration

Second Iteration



Second Iteration

6.2 Depth-wise Convolution

Our block diagram in depth-wise part is not different from the standard convolution as it consists of a data memory, weight memory, bias memory, data FIFO, weight FIFO and the convolution operation.

In the point-wise part, our block diagram consists from a bias FIFO, N Accumulators, N multipliers, N multiplexers and two memory arrays one for the weights of the filters & another one for the output data in which each output memory contains a channel.

Starting the operation from the beginning of the depth-wise, we have M input channels & M Filters in which each filter has only one channel, so each filter is multiplied by a corresponding channel from the input to produce an output channel.

In the point-wise we have N filters each with M channels, so the pointwise starts its operation by multiplying its input first channel with all N filters' first channel and its output is the depth of the OFM, then the second channel of the input is multiplied by the second channel of N filters to accumulate this output on the previous output and so on till the input channels are finished to reach our full sum.

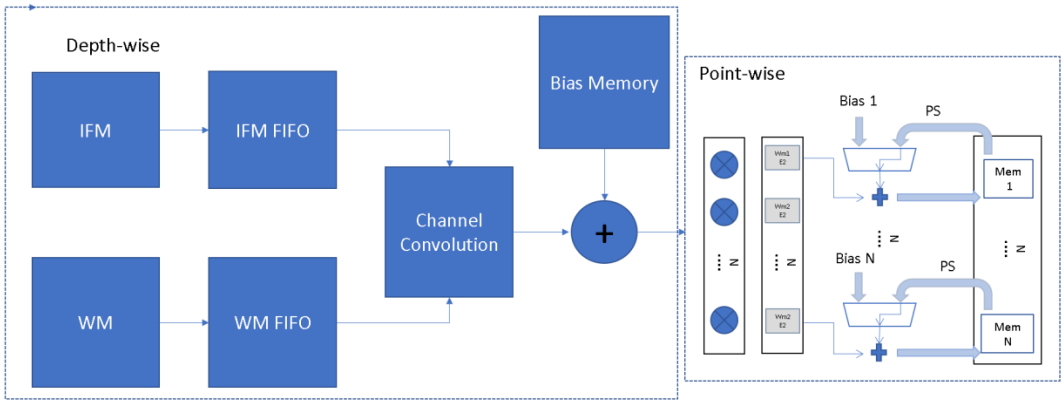


Figure 21: Depth-wise Architecture

6.3 Max Pooling

The design of the Max Pooling Layer is similar to the Average Pooling Layer design from the last year but here we don't have dividers. In here we have a 2x2 window and stride 2 so in the window we take 2 elements and choose the maximum between them, and we do the same for the other 2 elements and now we have the maximus 2 of the 4 elements in the window and we'll do the same operation for the maximum 2 to get the maximum element in the entire window.

To determine the maximum between 2 elements we have 2 cases (floating point representation – fixed point representation) and in both cases there were no comparators used as they require a lot of power consumption and instead, we used a subtraction algorithm, as shown in figure below.

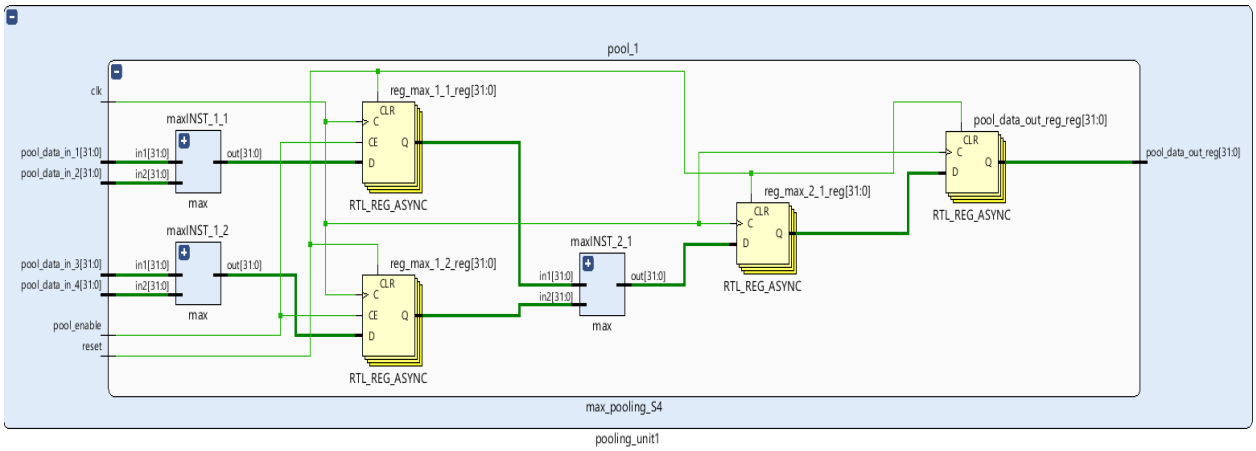


Figure 22: Max Pool Comparators

For the fixed point it was easier as subtracting the 2 numbers yields either a positive number, negative number or zero. The most significant bit tells us that the result is positive if it's equal to 0 or negative if it's equal to 1. A positive number indicates that the first number is greater so it's the maximum between them and the negative number indicates that the maximum is the second number and if they're equal we choose any of them.

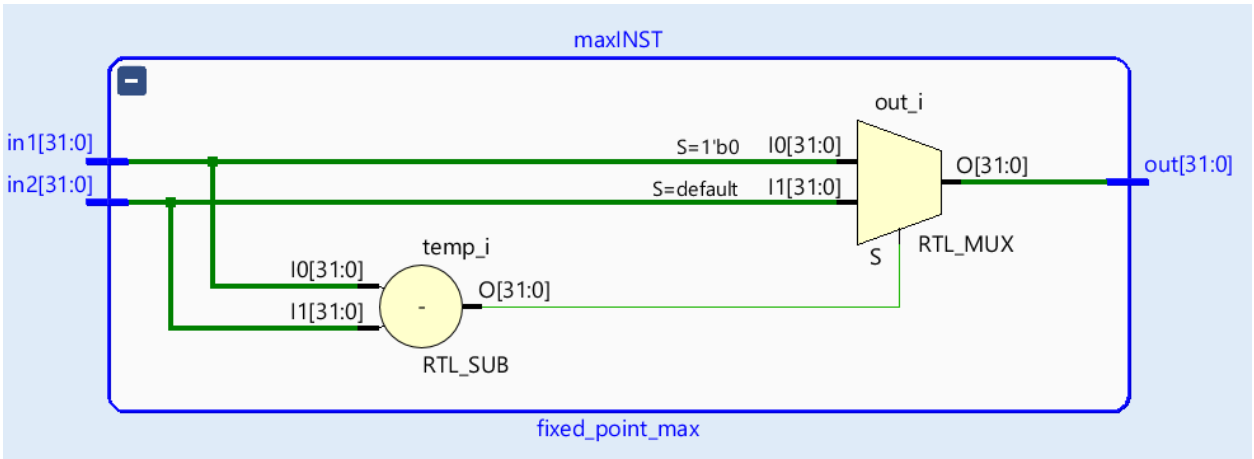


Figure 23: Fixed-Point Comparators

For the floating-point single precision, it was tricky:

- We first check the most significant bit for both numbers to check the signs if they are not equal then the positive is for sure the maximum between the two and if they are equal, we proceed to the next step.
- In here we compare the exponents by subtraction as well if the exponent in first number is greater than the other exponent than it's surely the maximum number of the two, otherwise if they are equal proceed to the next step.
- In here we compare the mantissas by subtraction as well if the mantissa of the first number is greater than the other mantissa than it's for sure the maximum number of the two, otherwise if they are equal then the maximum is any of them.

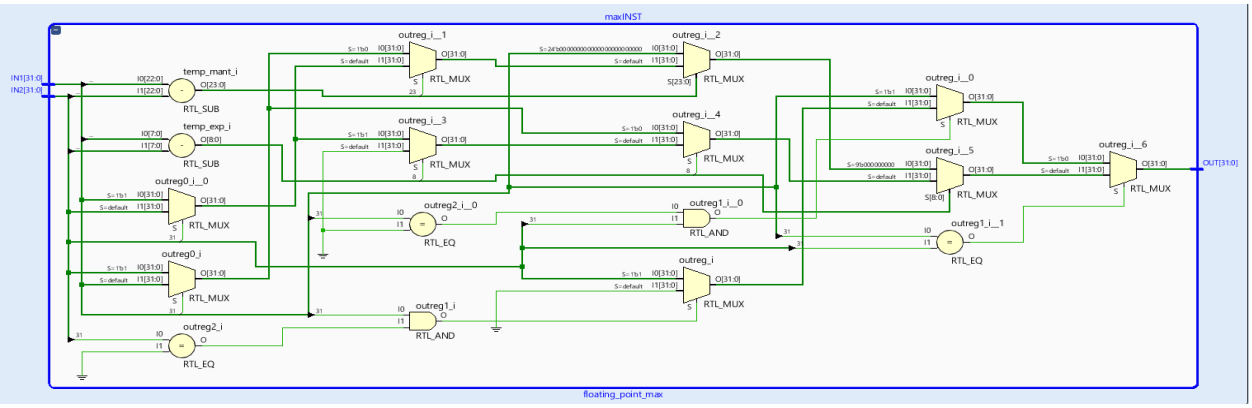


Figure 24: Floating-Point Comparators

6.4 Padding

The padded zeros are added on the IFM (image) before executing convolution operation. So, the design is applied on the IFM FIFO. The design algorism is simple, when reading the input, if the data that should be shifted in FIFO is actual data from IFM, the IFM output port is routed to FIFO, if the data is a pad zero, a constant zero is routed to FIFO. Obviously, the routing is done though a MUX, and the selection line of MUX is derived from FSM that determine the right time to route both actual and zero data, as shown in figure below.

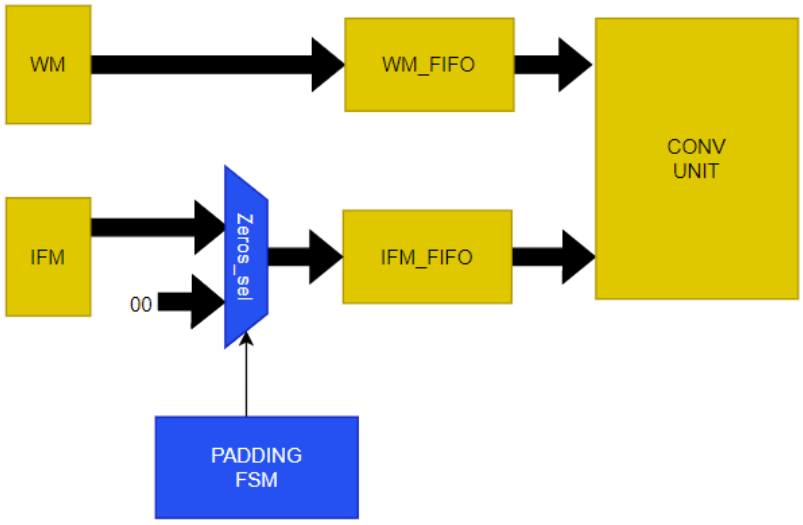


Figure 25: Padding block design

6.5.1 Types of padding

- Valid padding
- Same padding

Valid Padding

This type of padding can be considered as no padding. Why is there no padding we will understand after the example of a model? Let's just look at the figure below:

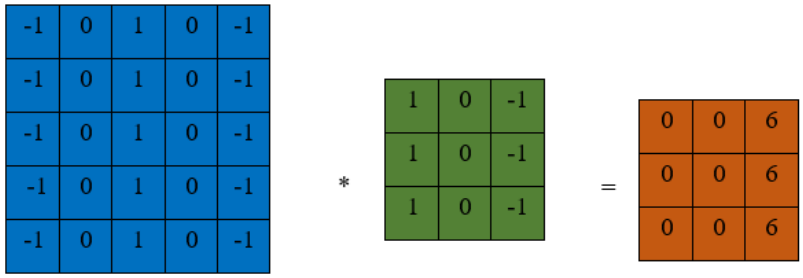


Figure 26: Valid Padding

So, in this, we really don't apply padding, but we assume that every pixel of the image is valid so that the input can get fully covered by the filter wherein a simple model assumes data in corners are less important. And do not consider them in the coverage area.

Same Padding

In this type of padding, the padding layers append zero values in the outer frame of the images or data so the filter we are using can cover the edge of the matrix and make the inference with them too. In case of stride 1 is used, the added pixels are used to maintain the OFMsize equals to IFMsize. If stride 2 is used, most of models use padding to convert even sized IFM to odd sized IFM, as the even sized IFM leads to loss in data without using padding, OFMsize in this case is exactly half of IFMsize. The number of added pixels needed "P" in both cases is the same, and it could be calculated through this equation:

$$\boldsymbol{OFM_{size} = \frac{IFM_{size} - Kernal_{size} + P}{Stride}, Where\ P\ is\ the\ depth\ of\ zeros\ around\ IFM}$$

$$\boldsymbol{P = Kernal_{size} - 1\ , in\ case\ stride\ 1\ is\ used}$$

$$\boldsymbol{P = Kernal_{size} - 2\ , in\ case\ stride\ 2\ is\ used}$$

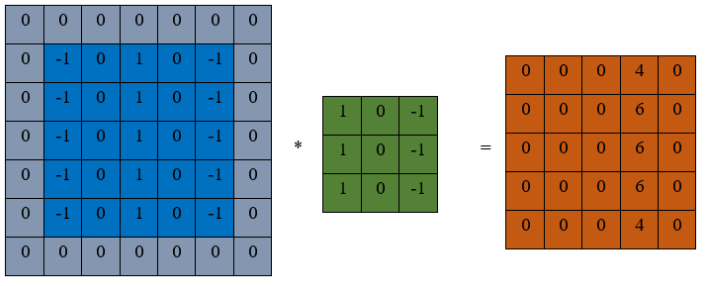


Figure 27: Same Padding with stride 1

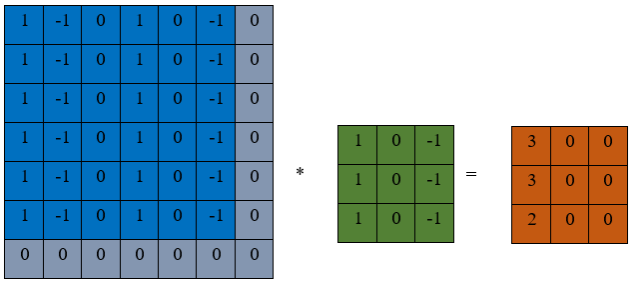


Figure 28: Same Padding with stride 2

6.4 Stride

Stride is the step in which the filter moves with across the image until it reaches the upper right-hand corner. Stride is used mainly to decrease the output size so processing will be easier, two types of strides are supported:

- Stride 1
- Stride 2

6.4.1 Stride 1

In general, the aim is to get the right number from the input image and filter to form the filter window to enter them in convolution block. This block is basically a FIFO (shift register) arranged to get the exact frame from the image and the filter. The following figure shows how the block operates and how it gives us the exact frame. The first pixel enters to the registers each cycle. Each register shifts its data to next register till the shift register is full after that we read the data from specific registers (the shaded registers in the shift register block figure) to get the right pixels.

Length of FIFO needed is calculated through this equation:

$$\boldsymbol{FIFO\ Length = (Kernal_{size} - 1) * IFM_{size} + Kernal_{size}}$$

So, in this example the FIFO length is = 2W+K (W image size, K filter size) but there is a wasted cycles = K-1 when shifting the data in the registers each time we get to the edge of the kernel.

The following figure shows the shift register block:

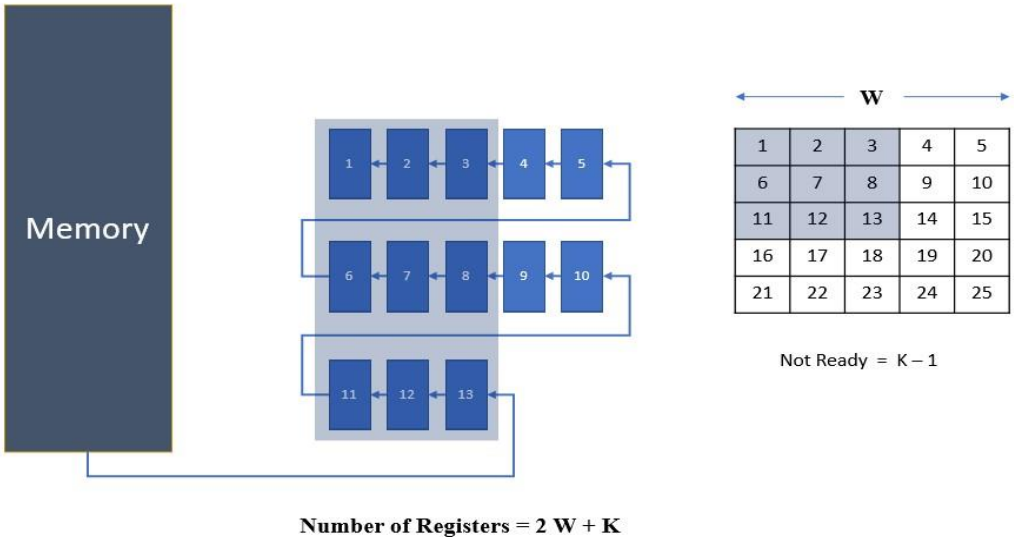


Figure 29: Input FIFO Flow in Stride 1

6.4.2 Stride 2

The goal is same as stride 1, is to get the right frame from input memory. But here the window is needed to be shifted by 2 cells in the FIFO to implement stride 2 operation. So, two port memory is used to get 2 data at the same time. The FIFO is implemented such that odd-indexed cells are shifted together, and even-indexed cells are shifted together also. In this case, the FIFO size is increased by 1 cell to convert it to even-sized number, as the input is fetched from IFM memory in pairs

$$\boldsymbol{FIFO\ Length} = (\boldsymbol{Kernal_{size}} - \mathbf{1}) * \boldsymbol{IFM_{size}} + \boldsymbol{Kernal_{size}} + \mathbf{1}$$

So, in this example the FIFO length is = 2W+K+1 (W image size, K filter size) but there is a wasted cycles = W/Stride +K-1 when shifting the data in the registers each time we get to the edge of the kernel.

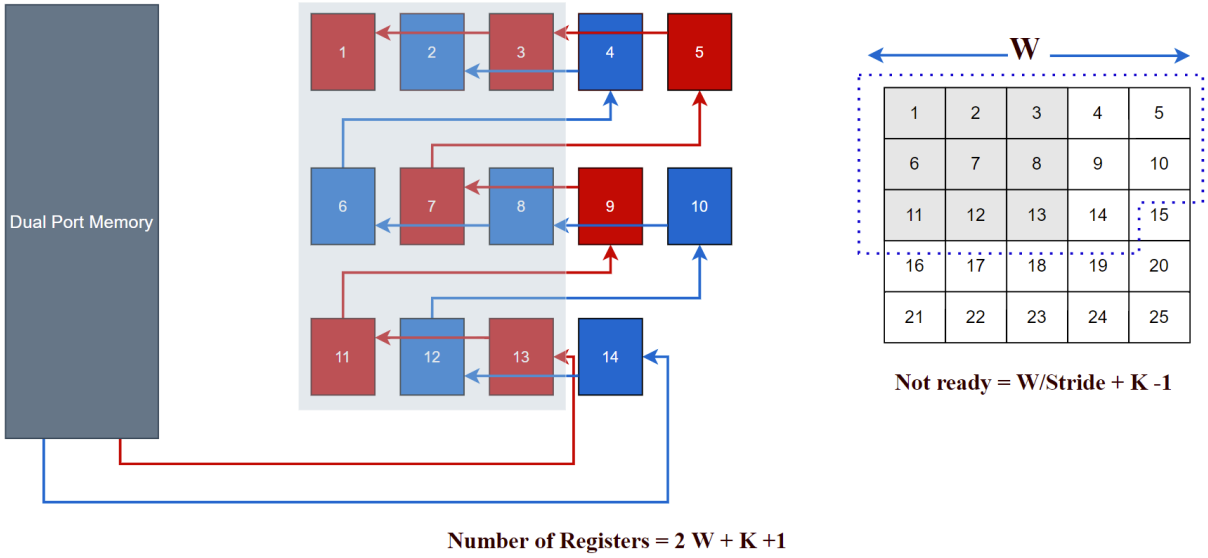


Figure 30: Input FIFO Flow in Stride 2

6.5 Flatten

There are two cases considered:

1. If the previous layer to FC is a type A layer (ConvA, PoolA and DwA), the output data comes out in order, and it is already flatted and serialized. So, there is no need to additional logic, only control manipulation between the FC and the previous layer.
2. If the previous layer to FC is a type B layer (ConvB, PoolB and DwB), the output data comes out from different filters, so memory is needed to hold the data, and a serializer is used.

6.6 Hard max

It is simple multiple levels of comparators that compares the output of the input numbers together then set the largest number equal to one and the rest is equal to zero, so it is mainly used during testing.

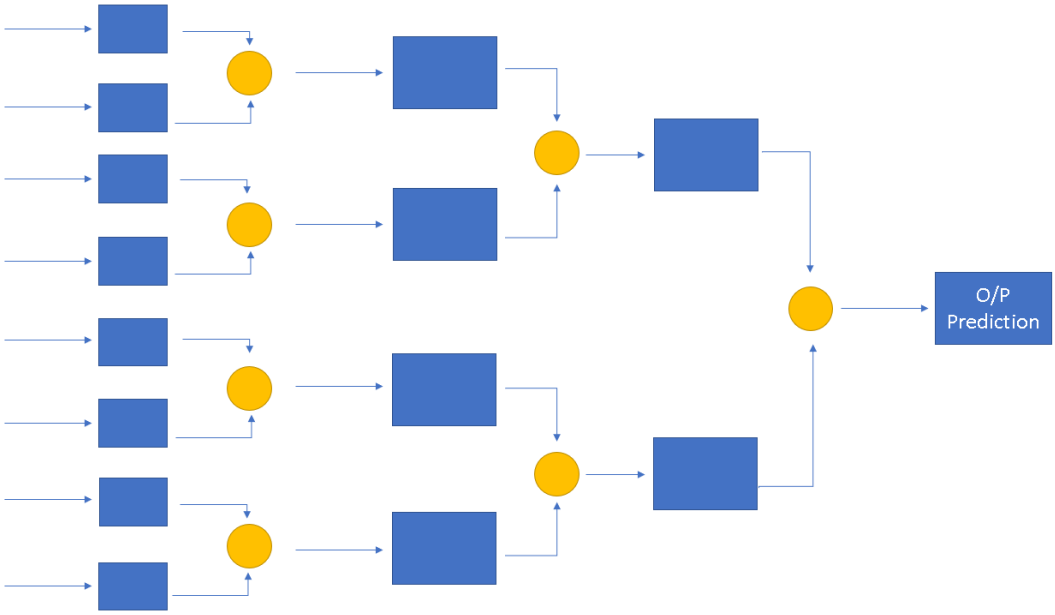


Figure 31: Hard Max implementation

Chapter 7: Software and Verification

CNN Training and Validation:

CNN was trained and validated on a set of labeled data from a drone. The labeled database contains images from drone that contains the direction of the drone. The images are represented in a 210 x 280-pixel RGB scale image. The labeled database has a training set of 8000images and a validation set of 2000 images. API Keras with TensorFlow backend was used to train and validate the CNN in python. Python was set up to use a floating-point representation to achieve high precision.

After training our model of Drone-Agent, we wrote a Python code to extract weights from the model in text format files and they are used in our HW design to initialize the memories in HW.

The trained model is stored in h5 format which can be converted to np (numpy) arrays, the array that has the weights for the conv layer is 2-dimensional tuple, the first dimension is 4-dimensional array for the weights, and the other is a one dimension for the biases, the dimensions are kernel height, kernel width, number of channels and number of filters respectively.

Our target was to reorder the array to suit the weight memory for each layer for example in ConvA each memory has different filters and the same channel is loaded at the same time so the reorder is number of filters, number of channels, kernel height and kernel width and if we want to add the units to the equation, we will add zeros to some memories based on the number channels divided by the number of units

$$ExtendedChannels = ceil(channels/units) * units$$

While in ConvB each memory will have the same filter and different channels loaded at the same time so the reorder is number of channels, number of filters, kernel height and kernel width and the zero extension will be in the number of filters

$$ExtendedFilters = ceil(filters/units) * units$$

The bias memory was straight forward in ConvA just load it in text format file while in ConvB it had to be reordered and split in different files, files' number equals to number of units and bias 1 is in file 1 and bias 2 is in file 2 and so on

The fully connected layers were straight forward, and the weights were loaded in the file directly.

Chapter 8: Results

Fashion model

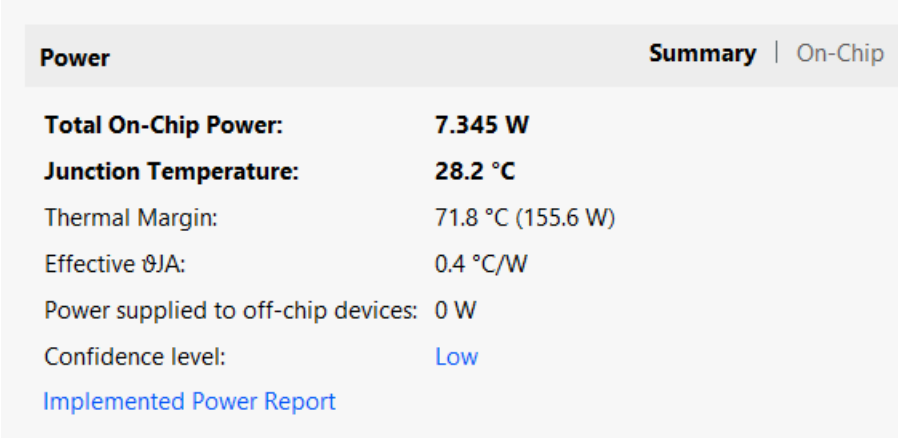


Figure 32: Power report of fashion model

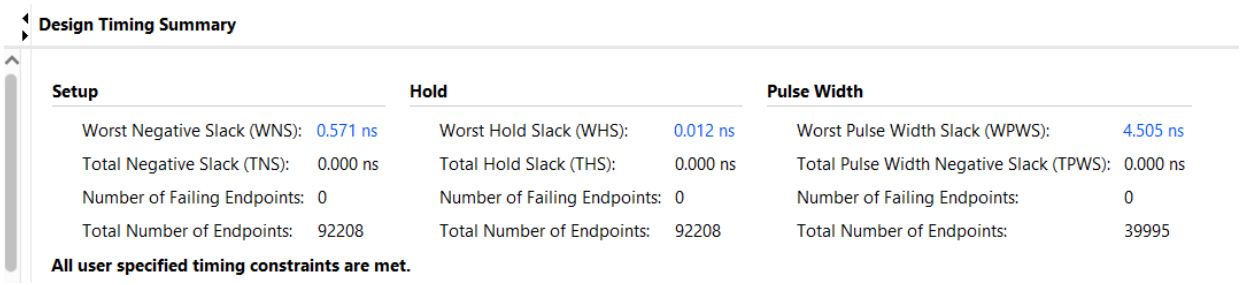


Figure 33: Timing report of fashion model

By using Virtex UltraScale 440 FPGA, we get these results:

Table 4: Fashion Model Results

Maximum Frequency	106 Mhz
Total Power	7.345 W
Functionality Accuracy	98.4 %
Image/Sec (HW Accelerator)	740
Image/Sec (CPU)	3

AI-Agent (Drone)

When we run the drone architecture on the emulator, we get the output pipelining as shown in the figure (), where we can insert input image and get output every 57 million Cycles. The emulation maximum frequency is 1562.5 Khz.

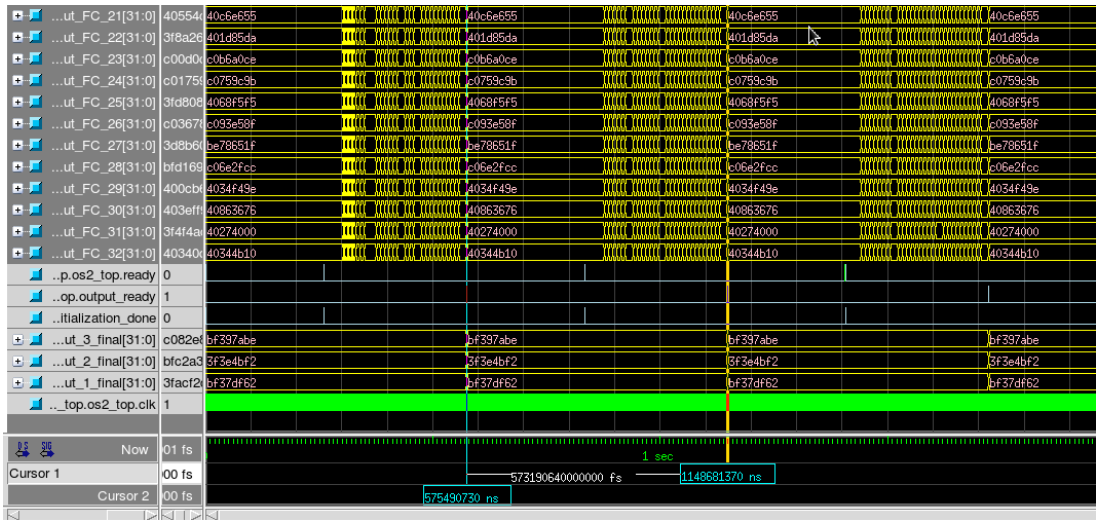


Figure 34: Pipelined output from Veloce waveforms

```
LUT AND FLOP COUNTS
Before Dead Logic Elimination
  Number of LUTs                  487017
  Number of flip-flops            248874
  Combinatorial synthesis LUTs    2417
  Synthesis flip-flops            16
  Memory Bytes                    288889728
After Dead Logic Elimination
  Number of LUTs                  436234
  Number of flip-flops            247316
  Memory Interface LUTs           0
  Combinatorial synthesis LUTs    1805
  Synthesis flip-flops            12
  Memory Bytes                    288889728
```

Figure 35: Architecture area

When we try to run the model in Xilinx Vivado and synthesis the design, we find that the architecture is very large and the number of input wires exceeds the maximum allowed input to a point in Vivado. So we try to decrease the architecture points through scaling down the input images. We have scaled down the input by 6, then we have ensured the functionality and then we get the following reports:

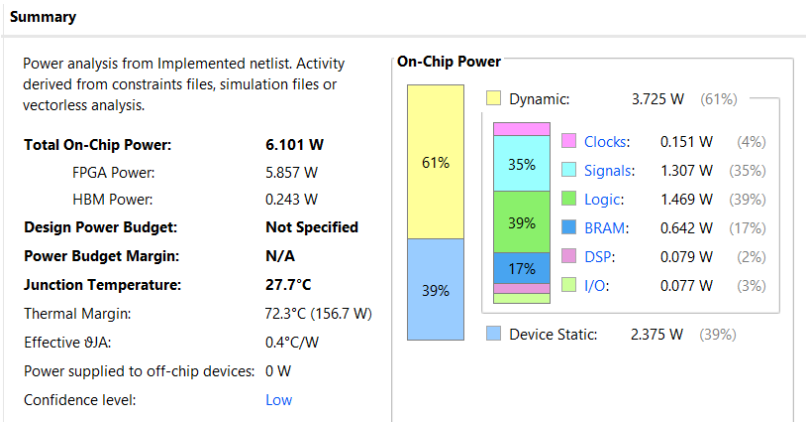


Figure 36: Power report of AI-Agent model

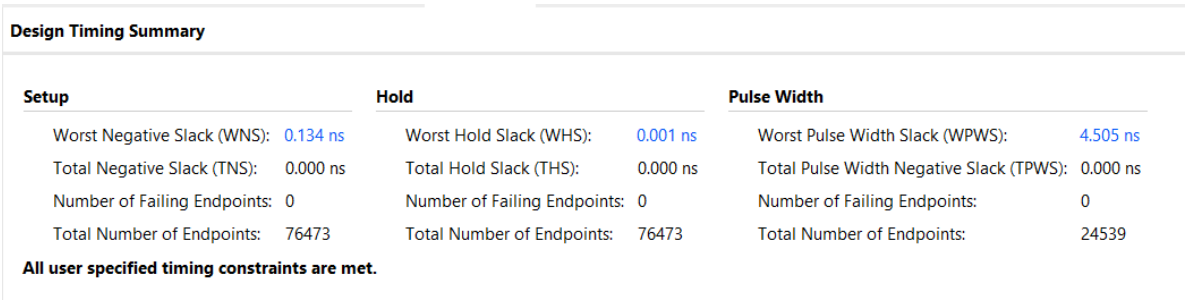


Figure 37: Timing report of AI-Agent model

By using the same FPGA (Virtex UltraScale 440), we get these results:

Table 5: Drone Model Results

Maximum Frequency	101 Mhz
Total Power	6.101 W
Functionality Accuracy	98 %
Image/Sec (HW Accelerator)	76
Image/Sec (CPU)	0.5

Chapter 9: Conclusion and Future work

Conclusion

In this thesis, we designed an automated flow using perl scripts for the generation and configuration of an AI accelerator for image classification CNNs and generated multiple accelerators for various CNNs and demonstrated their performance. We also introduced various optimizations in the design of the accelerator to make it scalable and configurable and to increase the throughput using pipelining and have a better memory utilization.

Verification and simulation of the design is carried out using Veloce Emulator and Vivado simulator.

Future Work

Adding more layers and features

Due the versatility of CNN designs, there is different HW architectures that has various features such as inception module and skip connection feature.

Our core RTL code that the Perl scripts are based on, as a future work should implement these layers so that we could generate more complex CNNs accelerators.

Integrate HW accelerator on Pulpino SoC

We modeled and designed our AI accelerator as a peripheral on a RISC-V based SoC like PULPino that's why the input and the weights are inputs to our accelerator and we use a memory controller to handle the address calculations

and a wrapper code is available to interface our accelerator with the PULPino's AXI bus, which in fact is going to make our design compatible with any AXI bus, so as a future work is the integration of our AI accelerator with the PULPino's AXI or APB bus.

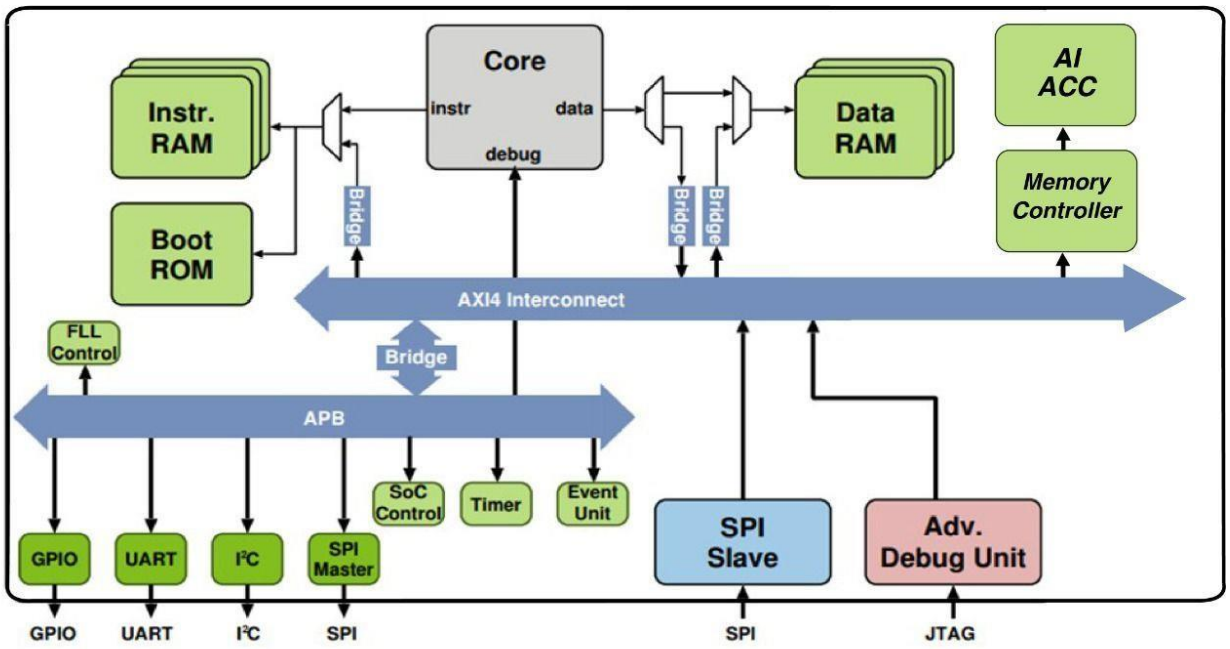


Figure 38: Integration with PULPino SoC

Using dynamic quantization

The dynamic quantization will need deeper study in the software model to determine which layers doesn't need a lot of bits to represent the data and some minor modification in the Verilog files to pass the arguments of the data width

and precision to each layers separately without overriding the values specified for each layer.

References

[1] O. I. Abiodun, A. Jantan, A. E. Omolara, K. V. Dada, N. A. Mohamed, and H. Arshad, “State-of-the-art in artificial neural network applications: A survey,” Heliyon, vol. 4, no. 11, 2018.

[2] A. Shawahna, S. M. Sait, and A. El-Maleh, “FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review,” IEEE Access, vol. 7, pp. 7823–7859, 2019.

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

[4] K. He, X. Zhang, S. Ren, and J. Sun, Deep residual learning for image recognition, in Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

[5] V. Nair and G. E. Hinton, Rectified linear units improve restricted Boltzmann machines, in International Conference on Machine Learning (ICML), 2010.

[6] "CNN Architectures: LeNet, AlexNet, VGG, GoogLeNet, ResNet and more…", Medium, 2017. [Online]. Available: https://medium.com/analytics vidhya/cnnsarchitectures-lenet-alexnet-vgg-googlenet-resnet-and-more666091488df5 [Accessed: 12-Oct- 2019]

[7] A. Krizhevsky, I. Sutskever and G. Hinton, "ImageNet classification with deep convolutional neural networks", 2012.

[8] Sheppard, Doug (October 16, 2000). "Beginner's Introduction to Perl". dev.perl.org. Archived from the original on June 5, 2011. Retrieved January 8, 2011