IMPLEMENTATION OF CONVOLUTIONAL NEURAL NETWORKS IN FPGA FOR IMAGE CLASSIFICATION

A Thesis

Presented to the

Faculty of

California State Polytechnic University, Pomona

In Partial Fulfillment

Of the Requirements for the Degree

Master of Science

In

Electrical Engineering

By

Mark A. Espinosa

2019

SIGNATURE PAGE

THESIS:	IMPLEMENTATION OF CONVOLUTIONAL NEURAL NETWORKS IN FPGA FOR IMAGE CLASSIFICATION
AUTHOR:	Mark A. Espinosa
DATE SUBMITTED:	Spring 2019
	Department of Electrical and Computer Engineering

Dr. Zekeriya Aliyazicioglu
Thesis Committee Chair
Electrical & Computer Engineering

Dr. James Kang
Electrical & Computer Engineering

Dr. Tim Lin
Electrical & Computer Engineering

ACKNOWLEDGEMENTS

To my mom and dad who have always believed in me in all my endeavors all my life. To my wife who has kept me going and cheered me on every step of the way. Thank you and love you all.

ABSTRACT

Machine Learning and Deep Learning its sub discipline are gaining popularity quickly. Machine Learning algorithms have been successfully deployed in a variety of applications such as Natural Language Processing, Optical Character Recognition, and Speech Recognition. Deep Learning particularly is suited to Computer Vision and Image Recognition tasks. The Convolutional Neural Networks employed in Deep Learning Neural Networks train a set of weights and biases which with each layer of the network learn to recognize key features in an image. This work set out to develop a scalable and modular FPGA implementation for Convolutional Neural Networks. It was the objective of this work to attempt to develop a system which could be configured to run as many layers as desired and test it using a currently defined CNN configuration, AlexNet. This type of system would allow a developer to scale a design to fit any size of FPGA from the most inexpensive to the costliest cutting-edge chip on the market. The objective of this work was achieved, and all layers were accelerated including Convolution, Affine, ReLu, Max Pool, and Softmax layers. The performance of the design was assessed, and it was determined its maximum achievable performance was approximately 3 GFLOPS.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 CURRENT WORK AND PRACTICE

In order to understand the overall motivation for this work, we must look at the Deep Learning and its pervasiveness in our everyday lives, how Convolutional Neural Networks are being implemented in FPGA, and how FPGAs compare with other platforms such as CPUs and GPUs. This section will explore these areas and will lend better insight as to how this work hopes to contribute to the field.

1.1 Machine Learning

The best definition of what Machine Learning is comes from Professor Tom Mitchell of Carnegie Mellon University. Machine learning can be described as when a computer program is said to learn from experience **E** with respect to some task **T** and some performance measure **P**, if its performance on **T**, as measured by **P**, improves with experience **E**. (Mitchell, 1997) As this definition pertains to image classification, the experience **E** would be the experience of having the program classify tens of thousands of images. The task **T** would be the task of classifying images and the performance measure **P** would be the probability that it correctly classifies the image.

Learning algorithms have been successfully deployed in a variety of applications, including:

a. Text or document classification

b. Natural language processing

c. Speech recognition, speech synthesis, speaker verification

d. Optical character recognition (OCR)

e. Computational biology applications

f. Computer vision tasks,

g. Games

h. Unassisted vehicle control (robots, navigation)

i. Medical diagnosis

j. Recommendation systems, search engines, information extraction systems.

This list is by no means comprehensive, and learning algorithms are applied to new applications every day. (Mohri et.al., 2014) Moreover, such applications correspond to a wide variety of learning problems. Some major classes of learning problems are:

a. Classification: Assign a category to each item. For example, document classification may assign items with categories such as politics, business, sports, or weather while image classification may assign items with categories such as landscape, portrait, or animal. The number of categories in such tasks is often relatively small but can be large in some difficult tasks and even unbounded as in OCR, text classification, or speech recognition. (Mohri et.al., 2014)

b. Regression: Predict a real value for each item. Examples of regression include prediction of stock values or variations of economic variables. In this problem, the penalty for an incorrect prediction depends on the magnitude of the difference between the true and predicted values, in contrast with the classification problem, where there is typically no notion of closeness between various categories. (Mohri et.al., 2014)

c. Ranking: Order items according to some criterion. Web search, e.g., returning web pages relevant to a search query, is the canonical ranking example. Many other similar ranking problems arise in the context of the design of information extraction or natural language processing systems. (Mohri et.al., 2014)

d. Clustering: Partition items into homogeneous regions. Clustering is often performed to analyze very large data sets. For example, in the context of social network analysis, clustering algorithms attempt to identify “communities” within large groups of people.

e. Dimensionality reduction or manifold learning: Transform an initial representation of items into a lower-dimensional representation of these items while preserving some properties of the initial representation. A common example involves preprocessing digital images in computer vision tasks. (Mohri et.al., 2014)

The two most used types of Machine Learning Algorithms are Supervised Learning and Unsupervised Learning. (Mohri et.al., 2014)

a. Supervised learning: The learner receives a set of labeled examples as training data and makes predictions for all unseen points. This is the most common scenario associated with classification, regression, and ranking problems. (Mohri et.al., 2014)

b. Unsupervised learning: The learner exclusively receives unlabeled training data and makes predictions for all unseen points. Since in general no labeled example is available in that setting, it can be difficult to quantitatively evaluate the performance of a learner. Clustering and dimensionality reduction are example of unsupervised learning problems. (Mohri et.al., 2014)

As we can see from the preceding explanation, Machine Learning is used in modern Image Classification. Since Image Classification is essentially the subject of this work, let's now look at a specific example. Machine learning programs can be trained in several different ways. In one type of training, the program is shown a lot of pictures of different animals and each picture is labeled with the name of the animal; the cats are all labeled "cat". The program will eventually learn that the animals that look like cats are called "cats" without ever being programmed to call a picture of a cat a "cat". (Murnane, 2016)

The program does this by learning combinations of features that tend to appear together. Cats have visual features, such as their body shape, long whiskers, and the way their faces look. These features make them visually different from other animals. The program learns to associate this distinctive combination of features with the word "cat". This learning process is usually called constructing a model of a cat. (Murnane, 2016)

Once it has constructed the cat model, a machine learning program tests the model by trying to identify the cats in a set of pictures it hasn't seen before. The program measures how well it did at identifying the new cats and uses this information to adjust the model so it will do a better job of picking out cats the next time it tries. The new model is then tested, its performance is evaluated, and it receives another adjustment. This iterative process continues until the program has built a model that can identify cats with a high level of accuracy. (Murnane, 2016)

1.2 Deep Learning

Deep learning carries out the machine learning process using an artificial neural net that is composed of a number of levels arranged in a hierarchy. The network learns something simple at the initial level in the hierarchy and then sends this information to the next level. The next level takes this simple information, combines it into something that is a bit more complex, and passes it on the third level. This process continues as each level in the hierarchy builds something more complex from the input it received from the previous level. (Murnane, 2016)

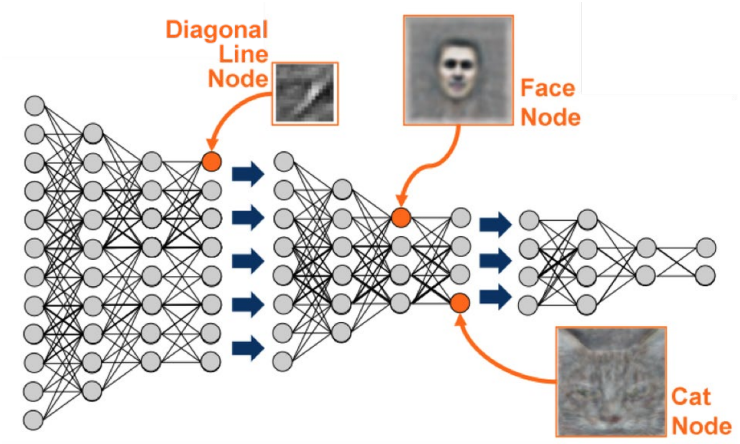

Figure 1: *Illustration of Deep Learning Neural Network Layers and how it learns specific features. (Murnane, 2016)*

Continuing the cat example, the initial level of a deep learning network might use differences in the light and dark areas of an image to learn where edges or lines are in a picture of a cat. The initial level passes this information about edges to the second level which combines the edges into simple shapes like a diagonal line or a right angle. The third level combines the simple shapes into more complex objects like ovals or rectangles. The next level might combine the ovals and rectangles into rudimentary whiskers, paws and tails. The process continues until it reaches the top level in the hierarchy where the network has learned to identify cats. While it was learning about cats, the network also learned to identify all the other animals it saw along with the cats. (Murnane, 2016)

1.3 Core Deep Learning Architectures

As we have seen in the above explanations of what Machine Learning and Deep Learning are, the most critical aspect of performing this kind of Artificial Intelligence work is the data set being used to train the neural network model. For the task of Image Classification, many researchers have been using the well-known database "ImageNet".

ImageNet is an image dataset organized according to the WordNet hierarchy. WordNet is a large lexical database of English. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. The resulting network of meaningfully related words and concepts can be navigated with a browser. (Fellbaum, 1998) Each meaningful concept in WordNet, possibly described by multiple words or word phrases, is called a "synonym set" or "sysnet". There are more than 100,000 sysnets in WordNet with the majority of them, more than 80,000, as nouns. (Fellbaum, 1998)

In 2010 ImageNet announced the first ever ImageNet Challenge which was a competition of which Deep Learning Algorithm could best estimate the content of photographs. The validation and test data from ImageNet and for this competition consisted of 200,000 photographs, which were collected from flickr and other search engines. These photographs were hand labeled with the presence or absence of 1000 distinct object categories. The same competition has been held every year since then and announces the winning algorithms in tasks such as Object Localization and Object Detection. (Li, F., et. al, 2015)

1.4 AlexNet

In 2012 Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton used the ImageNet database in order to develop the Deep Learning architecture which came to be known as AlexNet. Using Alexnet they trained a large, deep convolutional neural network to classify the 1.2 million high-resolution images in the ImageNet LSVRC-2010 contest into the 1000 different classes for the 2012 competition. (Li, F., et. al, 2017) The images used as input to their network were 224x224 RGB images.

The worst error rate they achieved was 37.5% and the best average error rate of 17.0%. The neural network consisted of 60 million parameters, 650,000 neurons, five Convolutional Layers followed by ReLu and Max Pool Layers, three Fully Connected Layers, and a 1000-way Softmax Classifier. (Krizhevsky,

2012) Below is an illustration on the overall architecture of the AlexNet model. Their work was done on two GTX 580 GPUs. This implementation was the first to use a Rectified Linear Unit as an activation layer rather than a Sigmoid Activation function. This implementation also won the ImageNet LSVRC-2012 competition.

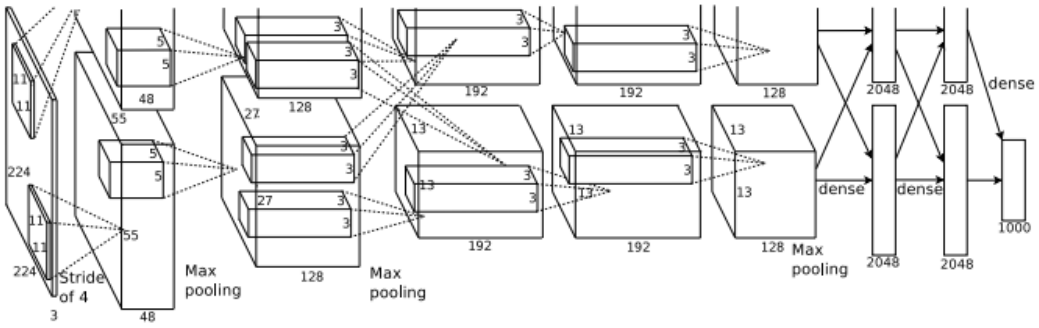

Figure 2: *Visual representation of AlexNet architecture. Illustration shows the layers used and their interconnectivity. (Krizhevsky, 2012)*

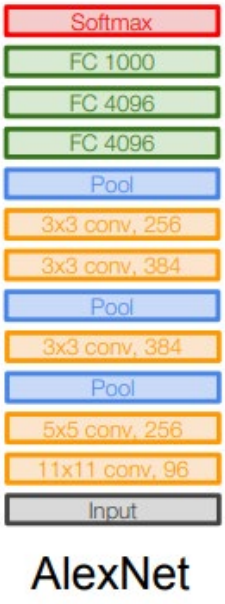

Figure 3: *Visual representation of AlexNet architecture. Illustration shows the layers used and their interconnectivity. (Li, F., et. al, 2017)*

1.5 VGGnet

In 2014 Karen Simonyan and Andrew Zisserman working for the Visual Geometry Group at the University of Oxford created VGGnet. Their work expanded on the Alexnet architecture by adding more convolutional layers and using small receptive fields such as 3x3 or 1x1. Different configurations of their network were tested with each configuration following a generic design and distinct only in the depth from 11, to 13, then to 19 weight layers. Each of these configurations had sub configurations as well. (Zisserman,2014)

During their experimentation, two VGGnet configurations VGG16 and VGG19 performed the best. Error rates for VGG16 were a max of 27.3% and an average of 8.1%. Error rates for VGG19 were a max of 25.5% and an average of 8.0%. (Zisserman,2014) By adding more layers to the network the number of parameters in both VGG16 and VGG19 increased to 138 and 144 million respectively. (Zisserman,2014) ***Figure 4*** shows the two VGGnet configurations as well as to how they compare to their predecessor AlexNet. This VGGnet architecture won the 2014 ImageNet LSRVC Challenge. (Li, F., et. al, 2017)

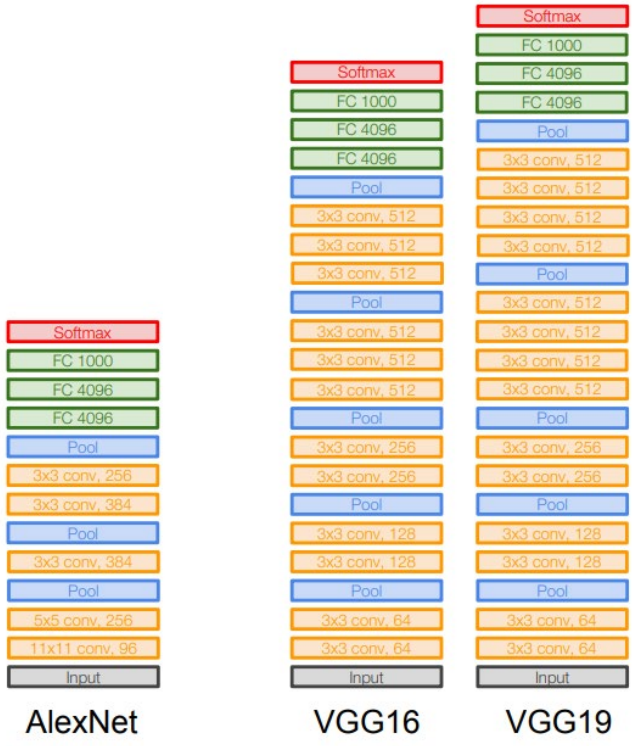

Figure 4: *Visual representation of VGGnet architecture. Illustration shows the layers used and their interconnectivity. (Li, F., et. al, 2017)*

1.6 ResNet

In 2015, Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun of Microsoft created the ResNet architecture which sought to improve on what VGGnet achieved by recognizing that network depth was crucial to a more accurate architecture. (He et. al., 2015) They also recognized that attempting to train a very deep network was plagued by the "vanishing gradient" problem which occurred during the back-propagation phase of the neural network training. They proposed "deep residual learning" which is a new framework that introduces "Shortcut Connections". (He et. al., 2015) They hypothesized that this structure would allow training and optimization to be much easier and solve the "vanishing gradient" problem. (Li, F., et. al, 2017)

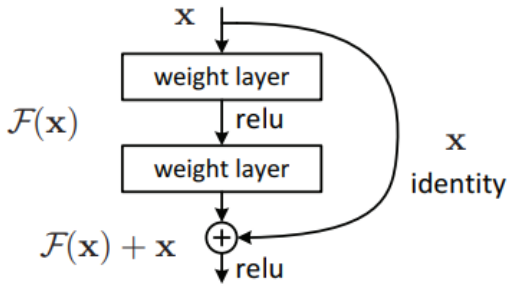

Figure 3: *Residual Learning: a building block of the ResNet architecture. (He et. al., 2015)*

The baseline for the Resnet architecture is VGGnet however the convolutional layers use only 3x3 filter weight kernels. For their experimentation they constructed a 34-layer plain network with no Shortcut connections and a 34 Layer network with shortcut connections, a Resnet. They also configured several networks with incrementally increasing layer count from 34 layers to 152 layers. Overall, 34-layer ResNet outperformed the 34-layer plain network and the average error rate achieved on the 152-layer network for the 2015 ImageNet LSVRC competition was 3.57%. This network architecture won the 2015 ImageNet LSVRC challenge. (Li, F., et. al, 2017)

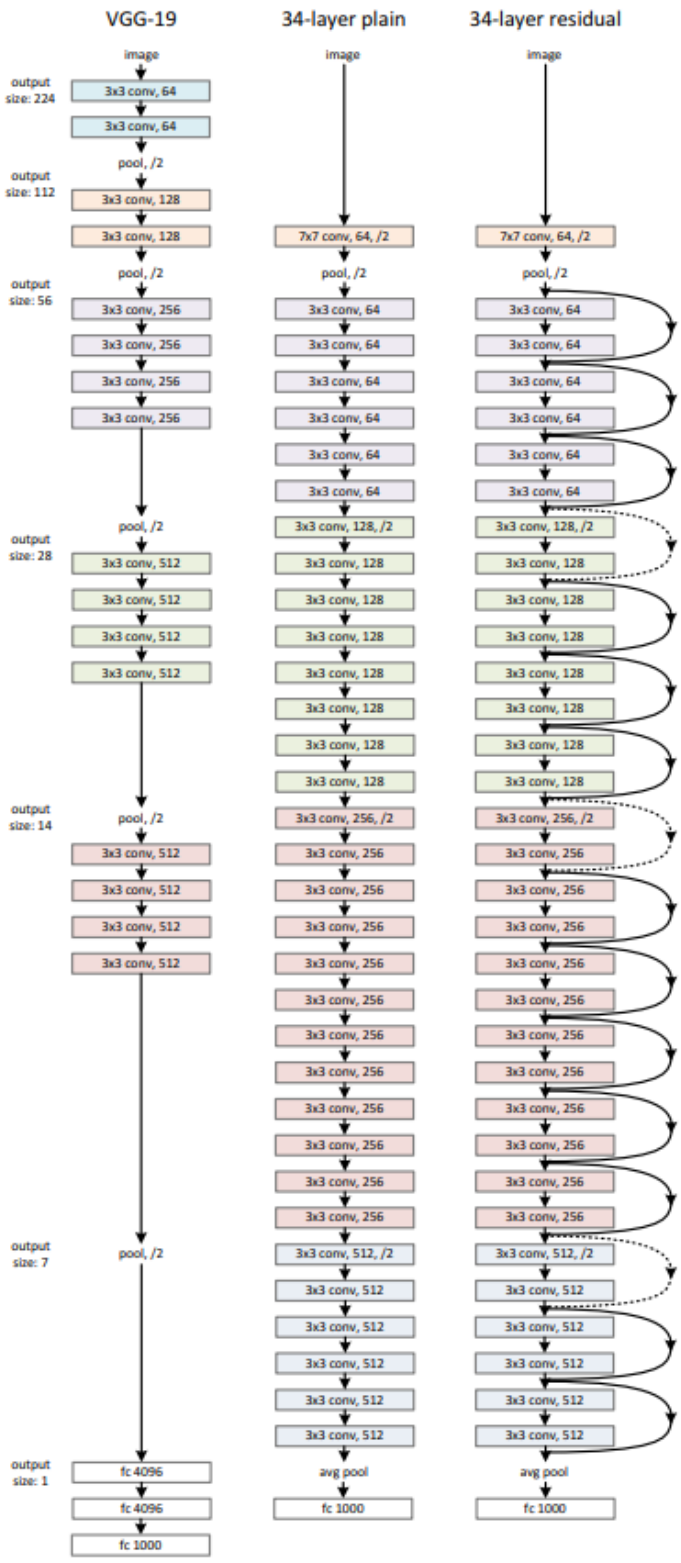

Figure 5: *Residual Learning: a building block of the ResNet architecture. (Li, F., et. al, 2017)*

1.7 Convolutional Neural Networks in FPGA: Literature Review

Now that we have introduced the world of Deep Learning and highlighted the most celebrated Convolutional Deep Neural Networks, lets now shift focus onto how Convolutional Neural Networks have been implemented into FPGA. We must keep in mind that the architectures such as AlexNet, VGGnet, and ResNet have been developed with only software development in mind. Therefore, these research groups are composed primarily of computer scientists and mathematicians. A few works have implemented Convolutional Neural Networks into FPGAs and this section presents a few.

1.7.1 Up to Now

One of the most limiting hardware realizations for Deep Learning techniques on FPGAs is design size. The trade-off between design reconfigurability and density means that FPGA circuits are often considerably less dense than hard-ware alternatives, and so implementing large neural networks has not always been possible. However, as modern FPGAs continue to exploit smaller feature sizes to increase density and incorporate hardened computational units along-side generic FPGA fabric, deep networks have started to be implemented on single FPGA systems. A brief timeline of important events in FPGA deep learning research is seen in ***Figure 6***. (Lacey, G. et. al.,2016)

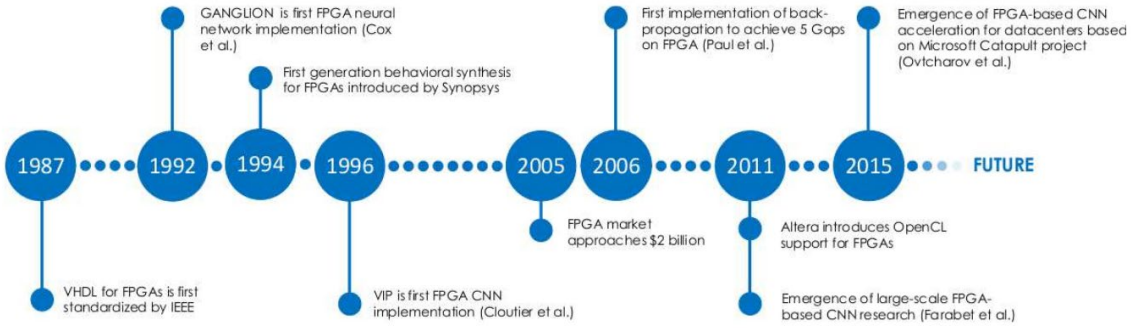

Figure 6: *Timeline of important events in FPGA deep learning research (Lacey, G. et. al.,2016)*

1.7.2 Other Works

A group out of Neurocoms in South Korea published and presented their work at the International Joint Conference on Neural Networks in 2015 where they implemented a Real-Time Video Object Recognition System using Convolutional Neural Networks. They used a custom 5 Layer neural network architecture developed first in Matlab. The system used grey-scale input images of 28x28 and each layer of the network was instantiated as logic all at once. The output of each layer would feed into the input of the next layer. The design employed the use of logic blocks they named synapses and receptors which formed the layers for the neural network. (***Figure 7***) The input images were classified from at 10 class list. This group used the Xilinx KC705 evaluations board and their logic operated at a frequency of 250MHz. Their measured power consumption was 3.1 watts. They utilized 42,616 Look Up Tables, 32 Block RAMs, and 326 DSP48 Blocks. The data format used was 16-bit Fixed Point and the group focused their work on the number of images they could classify in a second. (Ahn, B,2015)

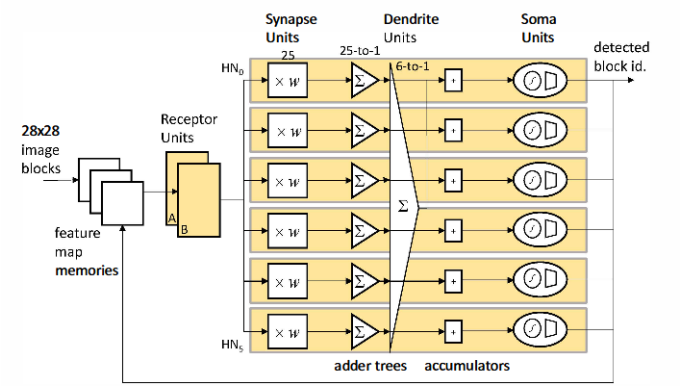

Figure 7: *Neurocoms work using 6 neurons and 2 receptor units (Ahn, B,2015)*

Another group from the Institute of Semiconductors from the Chinese Academy of Sciences in Beijing China attempted a small implementation in 2015. Their implementation ran on an Altera Arria V FPGA board operating at 50Mhz. The input images were 32x32, used an 8-bit fixed point data format, and used a 3 Convolution Layers with Activation, 2 Pooling Layers, and 1 Softmax Classifier. This work implemented custom processing elements which could be reconfigured when needed. (***Figure 8***) They measured their performance based on how many images could be processed. (Li, H et.al., 2015)

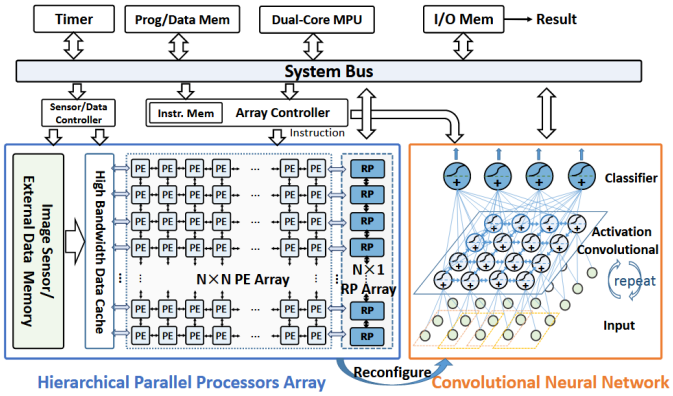

Figure 8: *Chinese Academy logic architecture*
(Li, H et.al., 2015)

A joint effort between Tsinghua University and Stanford university in 2016 yielded the "Angel-Eye" system. To accelerate the Convolution operation this system runs on the Xilinx Zynq XC7Z045 platform and uses an array of processing elements as shown in ***Figure 9***. The logic on the FPGA runs at a clock frequency of 150 MHz, uses 16bit fixed data format, uses 9.63 watts of power, and achieved 187.80 GFLOPS performance while running the VGG16 ConvNet. The resource utilization was not specified in their published work. The project that produced Angel-Eye also created a custom compiler which attempted to minimize external memory access and thereby reduce the inherent latency with memory transactions. **(**Guo, K. et. al., 2016)

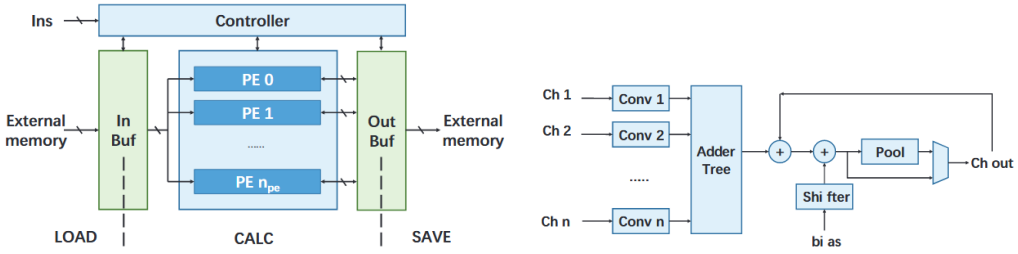

Figure 9: *Angel-Eye*
(Left) Angel-Eye architecture. (Right) Processing Element
(Guo, K. et. al., 2016)

Another work from a group at Purdue University in 2016 developed a Convolutional Neural Network accelerator which employed the use of custom software tools. These software tools were essentially compilers which explored optimization methods to improve the performance of the neural network. This work used the Xilinx Kintex-7 XC7K325T FPGA, achieved 58-115 GFLOPS performance, and ran a ConvNet with the same number of layers as AlexNet but with smaller channels. (Dundar, A. et. al., 2016) The resource utilization for their FPGA implementation was not specified.

The last work we will look at comes from the School of Computing, Informatics, and Decision Systems Engineering at Arizona State University. In 2016, one of their research groups efforted to create a scalable FPGA implementation of a Convolutional Neural Network. This group realized that as ever newer Deep Learning CNN configurations increase in layer count, and FPGA implementation would need to keep pace. This group also created a CNN compiler to analyze the input CNN model's structure and sizes, as well as the degree of computing parallelism set by users, to generate and integrate parametrized CNN modules. This implementation ran on a Stratix-V GXA7 FPGA operating with a clock frequency of 100MHz, uses a 16bit fixed data format, consumes 19.5 watts of power, and achieves 114.5 GFLOPS performance. Their FPGA resource utilization is 256 DSPS, 112K Look Up Tables, and 2,330 Block RAM. Their design employs the use of a shared multiplier bank for use with all the multiplication operations as shown in ***Figure 10***. (Ma, Y et. al., 2016)

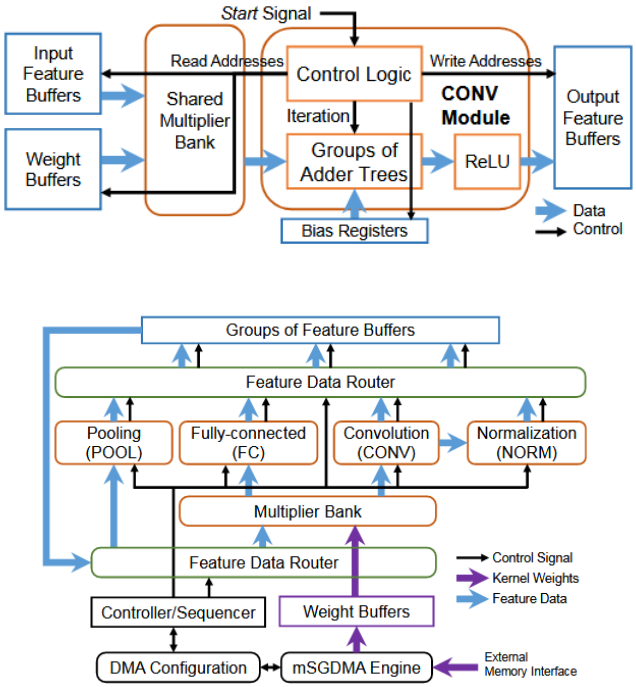

Figure 10: *Arizona State Implementation*
(Top) Convolution acceleration module block diagram. (Bottom) Integration of Scalable CNN Modules. (Ma, Y et. al., 2016)

2.0 THESIS STATEMENT

As we have seen by reviewing the current state of the art, Machine Learning is a fast-growing field of practice and research for Computer Scientists as well as Computer Engineers. This work efforts to design and develop a scalable, reusable, and user-friendly FPGA implementation of Convolutional Neural Networks for Image Classification tasks. This design would cater to the smaller more inexpensive Embedded Designs which have strict power and computation constraints.

As a road map for the rest of this work let's look at what will follow in the body of this work. First, we survey the Fundamental Theory behind Convolutional Neural Networks in Section 3.0. Section 4.0 explores the overall goals of this works FPGA implementation of Convolutional Neural Networks. Section 5.0 discusses the top-level architectural FPGA implementation involving multiple sub designs communicating on an AXI Bus. Section 6 through Section 9 discuss how each layer of a Convolutional Neural Network was accelerated on FPGA. Section 10 and Section 11 discuss how this implementation was verified in simulation and on hardware respectively. Section 12 reviews the overall results of the work and offers insight as to how to improve the overall performance. Section 13 closes this work with a brief summary. The source code and all files pertaining to this work are too numerous to include in the Appendix of this document. Therefore, links to the Github repositories containing the source files are provided.

3.0 FUNDAMENAL CONCEPTS

In order to properly delve into the proposed design, we should first review some fundamental theory behind Convolutional Neural Networks. This section will review the mathematical properties of CNNs as well as how they are implemented into working systems and will be more of the research paper portion of this work.

3.1 Convolutional Neural Networks

Most of the texts available on the subject of Neural Networks delve more deeply into the genesis of Neural Network research. Most texts cover primarily the Perceptron, Recurrent Neural Network, and LSTM Neural Network models which have been used in a variety of applications in the decades of Neural Network research. Finding comprehensive content and coverage of how Convolutional Neural Networks are designed and function was very challenging. However, the latest information on the current state of Deep Learning was found in a Stanford University course CS231n which Stanford made open to the public to view the lectures and do the assignments. This avenue of research enabled this work to be accomplished and not encountering this content would have placed this work at an extreme disadvantage.

Therefore, the section describing the operation of Convolutional Neural Networks is taken from the course notes and the lectures on the subject. The mathematical formulations behind Convolutional Neural Networks are taken from the newest text in Deep Learning from Goodfellow et. al. of the Massachusetts Institute of Technology (Goodfellow, et. al, 2016). Goodfellow's treatment of the topic is very thorough and excerpts from the work are provided here.

3.1.1 Convolutional Neural Networks: What are they

Convolutional networks also known as convolutional neural networks, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. Examples include time-series data, which can be thought of as a 1-D grid taking samples at regular time intervals, and image data, which can be thought of as a 2-D grid of pixels. Convolutional networks have been tremendously successful in practical applications. The name "convolutional neural network" indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation.

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers. (Goodfellow, et. al, 2016)

3.1.2 Convolution Operation: Theory, Properties, and Mathematical Representation

In its most general form, convolution is an operation on two functions of a real-valued argument. To motivate the definition of convolution, we start with examples of two functions we might use.

Suppose we are tracking the location of a spaceship with a laser sensor. Our laser sensor provides a single output *x(t)*, the position of the spaceship at time *t*. Both *x* and *t* are real valued, that is, we can get a different reading from the laser sensor at any instant in time. (Goodfellow, et. al, 2016)

Now suppose that our laser sensor is somewhat noisy. To obtain a less noisy estimate of the spaceship's position, we would like to average several measurements. Of course, more recent measurements are more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function *w(a)*, where *a* is the age of a measurement. (Goodfellow, et. al, 2016)

If we apply such a weighted average operation at every moment, we obtain a new function *s* providing a smoothed estimate of the position of the spaceship:

$$s(t) = \int x(a)w(t-a)da$$

This operation is called convolution. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)(t)$$

In our example, *w* needs to be a valid probability density function, or the output will not be a weighted average. Also, *w* needs to be 0 for all negative arguments, or it will look into the future, which is presumably beyond our capabilities. These limitations are particular to our example, though. In general, convolution is defined for any functions for which the above integral is defined and may be used for other purposes besides taking weighted averages. (Goodfellow, et. al, 2016)

In convolutional network terminology, the first argument (in this example, the function *x*) to the convolution is often referred to as the input, and the second argument (in this example, the function *w*) as the kernel. The output is sometimes referred to as the feature map. (Goodfellow, et. al, 2016)

In our example, the idea of a laser sensor that can provide measurements at every instant is not realistic. Usually, when we work with data on a computer, time will be discretized, and our sensor will provide data at regular intervals. In our example, it might be more realistic to assume that our laser provides a measurement once per second. The time index *t* can then take on only integer values. If we now assume that *x* and *w* are defined only on integer *t*, we can define the discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. We will refer to these multidimensional arrays as tensors. Because each element of the input and kernel must be explicitly stored separately, we usually assume that these functions are zero everywhere but in the finite set of points for which we store the values. This means that in practice, we can implement the infinite summation as a summation over a finite number of array elements. (Goodfellow, et. al, 2016)

Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image *I* as our input, we probably also want to use a two-dimensional kernel *K*:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Convolution is commutative, meaning we can equivalently write

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, i-n)K(m,n)$$

Usually the latter formula is more straightforward to implement in a machine learning library, because there is less variation in the range of valid values of m and n. (Goodfellow, et. al, 2016)

The commutative property of convolution arises because we have flipped the kernel relative to the input, in the sense that as m increases, the index into the input increases, but the index into the kernel decreases. The only reason to flip the kernel is to obtain the commutative property. While the commutative property is useful for writing proofs, it is not usually an important property of a neural network implementation. Instead, many neural network libraries implement a related function called the cross-correlation, which is the same as convolution but without flipping the kernel:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, i+n)K(m,n)$$

Many machine learning libraries implement cross-correlation but call it convolution. In this text we follow this convention of calling both operations convolution and specify whether we mean to flip the kernel or not in contexts where kernel flipping is relevant. In the context of machine learning, the learning algorithm will learn the appropriate values of the kernel in the appropriate place, so an algorithm based on convolution with kernel flipping will learn a kernel that is flipped relative to the kernel learned by an algorithm without the flipping. It is also rare for convolution to be used alone in machine learning; instead convolution is used simultaneously with other functions, and the combination of these functions does not commute regardless of whether the convolution operation flips its kernel or not. (Goodfellow, et. al, 2016)

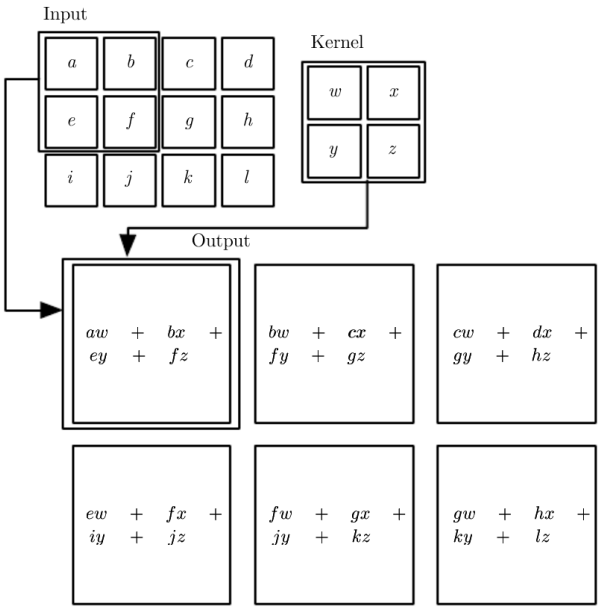

Figure 11: *An example of 2-D convolution without kernel flipping.*
We restrict the output to only positions where the kernel lies entirely within the image, called "valid" convolution in some contexts. We draw boxes with arrows to indicate how the upper-left element of the output tensor is formed by applying the kernel to the corresponding upper-left region of the input tensor. (Goodfellow, et. al, 2016)

Convolution leverages three important ideas that can help improve a machine learning system: sparse interactions, parameter sharing, and equivariant representations. Moreover, convolution provides a means for working with inputs of variable size. We now describe each of these ideas in turn. (Goodfellow, et. al, 2016)

3.1.2.1 Sparse Interactions

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. Convolutional networks, however, typically have sparse interactions (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means that

computing the output requires fewer operations. These improvements in efficiency are usually quite large. (Goodfellow, et. al, 2016)

If there are m inputs and n outputs, then matrix multiplication requires $m \times n$ parameters, and the algorithms used in practice have $O(m \times n)$ runtime (per example). If we limit the number of connections each output may have to k, then the sparsely connected approach requires only $k \times n$ parameters and $O(k \times n)$ runtime. For many practical applications, it is possible to obtain good performance on the machine learning task while keeping k several orders of magnitude smaller than m. For graphical demonstrations of sparse connectivity, see ***Figure 12*** and ***Figure 13***. In a deep convolutional network, units in the deeper layers may indirectly interact with a larger portion of the input, as shown in ***Figure 14***. This allows the network to efficiently describe complicated interactions between many variables by constructing such interactions from simple building blocks that each describe only sparse interactions. (Goodfellow, et. al, 2016)

3.1.2.2 Parameter Sharing

Parameter sharing refers to using the same parameter for more than one function in a model. In a traditional neural net, each element of the weight matrix is used exactly once when computing the output of a layer. It is multiplied by one element of the input and then never revisited. As a synonym for parameter sharing, one can say that a network has tied weights, because the value of the weight applied to one input is tied to the value of a weight applied elsewhere. In a convolutional neural net, each member of the kernel is used at every position of the input (except perhaps some of the boundary pixels, depending on the design decisions regarding the boundary). The parameter sharing used by the convolution operation means that rather than learning a separate set of parameters for every location, we learn only one set. This does not affect the runtime of forward propagation—it is still $O(k \times n)$—but it does further reduce the storage requirements of the model to k parameters. Recall that k is usually several orders of magnitude smaller than m. Since m and n are usually roughly the same size, k is practically insignificant compared to $m \times n$. Convolution is thus dramatically more efficient than dense matrix multiplication in terms of the memory requirements and statistical efficiency. For a graphical depiction of how parameter sharing works, see ***Figure 15***. (Goodfellow, et. al, 2016)

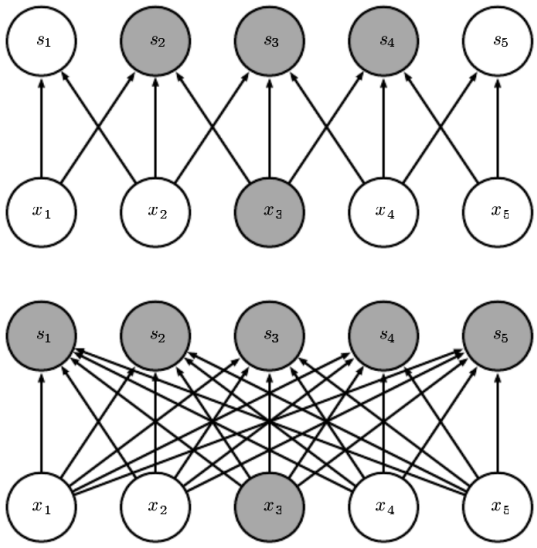

Figure 12: *Sparse connectivity, viewed from below.*
We highlight one input unit, x3, and highlight the output units in s that are affected by this unit. (Top)When s is formed by convolution with a kernel of width 3, only three outputs are affected by x. (Bottom) When s is formed by matrix multiplication, connectivity is no longer sparse, so all the outputs are affected by x3. (Goodfellow, et. al, 2016)

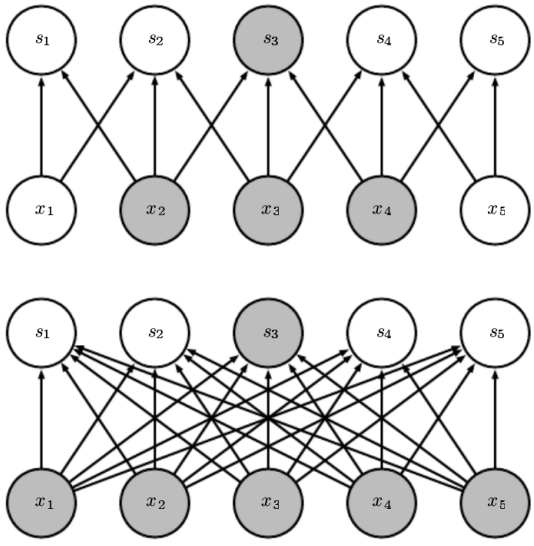

Figure 13: *Sparse connectivity, viewed from above.*
We highlight one output unit, s3, and highlight the input units in x that affect this unit. These units are known as the receptive field of s3. (Top) When s is formed by convolution with a kernel of width 3, only three inputs affect s3. (Bottom)When s is formed by matrix multiplication, connectivity is no longer sparse, so all the inputs affect s3. (Goodfellow, et. al, 2016)

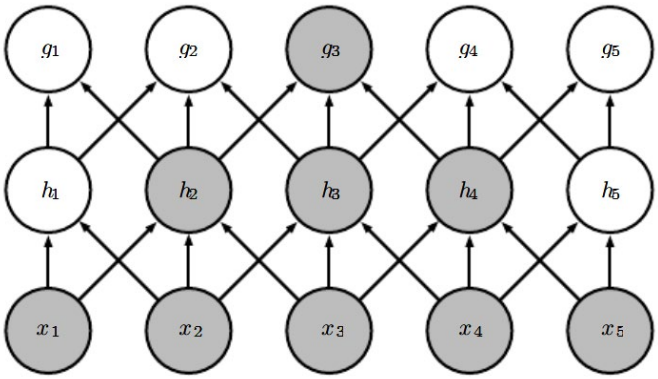

Figure 14: *Receptive Field.*
*The receptive field of the units in the deeper layers of a convolutional network is larger than the receptive field of the units in the shallow layers. This effect increases if the network includes architectural features like strided convolution (**Figure 17**) or pooling. This means that even though direct connections in a convolutional net are very sparse, units in the deeper layers can be indirectly connected to all or most of the input image. (Goodfellow, et. al, 2016)*

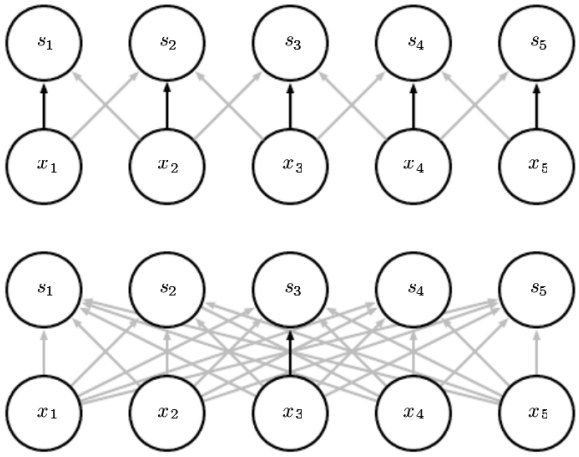

Figure 15: *Parameter sharing.*
Black arrows indicate the connections that use a particular parameter in two different models. (Top)The black arrows indicate uses of the central element of a 3-element kernel in a convolutional model. Because of parameter sharing, this single parameter is used at all input locations. (Bottom)The single black arrow indicates the use of the central element of the weight matrix in a fully connected model. This model has no parameter sharing, so the parameter is used only once. (Goodfellow, et. al, 2016)

3.1.2.3 Sparse Interactions

In the case of convolution, the particular form of parameter sharing causes the layer to have a property called equivariance to translation. To say a function is equivariant means that if the input changes, the output changes in the same way. (Goodfellow, et. al, 2016)

Specifically, a function $f(x)$ is equivariant to a function g if $f(g(x)) = g(f(x))$. In the case of convolution, if we let g be any function that translates the input, that is, shifts it, then the convolution function is equivariant to g. For example, let I be a function giving image brightness at integer coordinates. Let g be a function mapping one image function to another image function, such that $I'=g(I)$ is the image function with $I'(x, y) = I(x - 1, y)$. This shifts every pixel of I one unit to the right. If we apply this transformation to I, then apply convolution, the result will be the same as if we applied convolution to I', then applied the transformation g to the output. When processing time-series data, this means that convolution produces a sort of timeline that shows when different features appear in the input. (Goodfellow, et. al, 2016)

If we move an event later in time in the input, the exact same representation of it will appear in the output, just later. Similarly, with images, convolution creates a 2-D map of where certain features appear in the input. If we move the object in the input, its representation will move the same amount in the output. This is useful for when we know that some function of a small number of neighboring pixels is useful when applied to multiple input locations. For example, when processing images, it is useful to detect edges in the first layer of a convolutional network. (Goodfellow, et. al, 2016)

The same edges appear more or less everywhere in the image, so it is practical to share parameters across the entire image. In some cases, we may not wish to share parameters across the entire image. For example, if we are processing images that are cropped to be centered on an individual's face, we probably want to extract different features at different locations—the part of the network processing the top of the face needs to look for eyebrows, while the part of the network processing the bottom of the face needs to look for a chin. Convolution is not naturally equivariant to some other transformations, such as changes in the scale or rotation of an image. Other mechanisms are necessary for handling these kinds of transformations. (Goodfellow, et. al, 2016)

Finally, some kinds of data cannot be processed by neural networks defined by matrix multiplication with a fixed-shape matrix. Convolution enables processing of some of these kinds of data. (Goodfellow, et. al, 2016)

3.1.3 Activation and Pooling

A typical layer of a convolutional network consists of three stages (see ***Figure 16***). In the first stage, the layer performs several convolutions in parallel to produce a set of linear activations. In the second stage, each linear activation is run through a nonlinear activation function, such as the rectified linear activation function. This stage is sometimes called the detector stage. In the third stage, we use a pooling function to modify the output of the layer further. (Goodfellow, et. al, 2016)

A pooling function replaces the output of the net at a certain location with a summary statistic of the nearby outputs. For example, the max pooling operation reports the maximum output within a rectangular neighborhood. Other popular pooling functions include the average of a rectangular neighborhood, the L2 norm of a rectangular neighborhood, or a weighted average based on the distance from the central pixel. (Goodfellow, et. al, 2016)

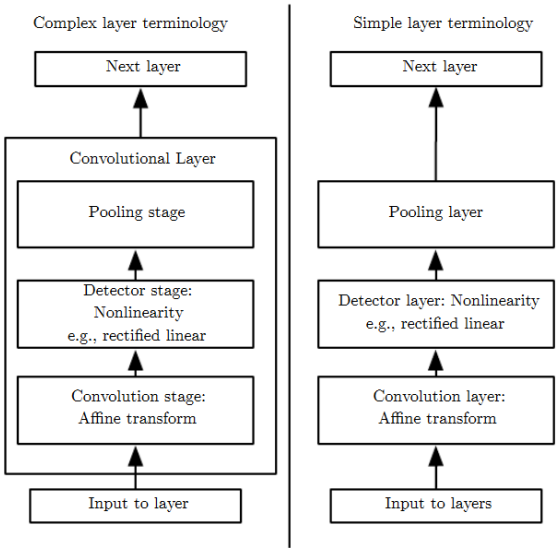

Figure 16: *The components of a typical convolutional neural network layer. There are two commonly used sets of terminology for describing these layers. (Left)In this terminology, the convolutional net is viewed as a small number of relatively complex layers, with each layer having many "stages." In this terminology, there is a one-to-one mapping between kernel tensors and network layers. (Right)In this terminology, the convolutional net is viewed as a larger number of simple layers; every step of processing is regarded as a layer in its own right. This means that not every "layer" has parameters. (Goodfellow, et. al, 2016)*

3.2 Convolutional Neural Network in Practice

When discussing convolution neural networks, we usually do not refer exactly to the standard discrete convolution operation as it is usually understood in the mathematical literature and as it was detailed in the previous section. The functions used in practice differ slightly. Here we describe these differences in detail and highlight some useful properties of the functions used in neural networks. (Goodfellow, et. al, 2016) We also look more closely as to how Convolutional Neural Networks are structured and what layers constitute a full system.

3.2.1 Practical Convolution

First, when we refer to convolution in the context of neural networks, we usually mean an operation that consists of many applications of convolution in parallel. This is because convolution with a single kernel can extract only one kind of feature, albeit at many spatial locations. Usually we want each layer of our network to extract many kinds of features, at many locations. (Goodfellow, et. al, 2016)

Additionally, the input is usually not just a grid of real values. Rather, it is a grid of vector-valued observations. For example, a color image has a red, green and blue intensity at each pixel. In a multilayer convolutional network, the input to the second layer is the output of the first layer, which usually has the output of many different convolutions at each position. When working with images, we usually think of the input and output of the convolution as being 3-D tensors, with one index into the different channels and two indices into the spatial coordinates of each channel. Software implementations usually work in batch mode, so they will actually use 4-D tensors, with the fourth axis indexing different examples in the batch, but we will omit the batch axis in our description here for simplicity. (Goodfellow, et. al, 2016)

Because convolutional networks usually use multichannel convolution, the linear operations they are based on are not guaranteed to be commutative, even if kernel flipping is used. These multichannel operations are only commutative if each operation has the same number of output channels as input channels. (Goodfellow, et. al, 2016)

Assume we have a 4-D kernel tensor $\boldsymbol{K}$ with element $K_{i,j,k,l}$ giving the connection strength between a unit in channel i of the output and a unit in channel j of the input, with an offset of k rows and l columns between the output unit and the input unit. Assume our input consists of observed data $\mathbf{V}$ with element $V_{i,j,k}$ giving the value of the input unit within channel i at row j and column k. Assume our output consists of $\mathbf{Z}$ with the same format as $\mathbf{V}$. If $\mathbf{Z}$ is produced by convolving $\mathbf{K}$ across $\mathbf{V}$ without flipping $\mathbf{K}$, then

$$Z_{i,j,k} = \sum_{l,m,n} V_{l,j+m-1,k+n-1} K_{i,l,m,n}$$

where the summation over l, m and n is over all values for which the tensor indexing operations inside the summation are valid. In linear algebra notation we index into arrays using a 1 for the first entry. This necessitates the −1 in the above formula. Programming languages such as C and Python index starting from 0 which renders the above expression even simpler. (Goodfellow, et. al, 2016)

3.2.2 Stride

We may want to skip over some positions of the kernel to reduce the computational cost (at the expense of not extracting our features as finely). We can think of this as down sampling the output of the full convolution function. If we want to sample only every s pixels in each direction in the output, then we can define a down sampled convolution function c such that

$$Z_{i,j,k} = c(\boldsymbol{K}, \boldsymbol{V}, s)_{i,j,k} = \sum_{l,m,n} \left[V_{l,(j-1)\times s+m,(k-1)\times s+n} K_{i,l,m,n} \right]$$

We refer to s as the stride of this down sampled convolution. It is also possible to define a separate stride for each direction of motion. See ***Figure 17*** for an illustration. (Goodfellow, et. al, 2016)

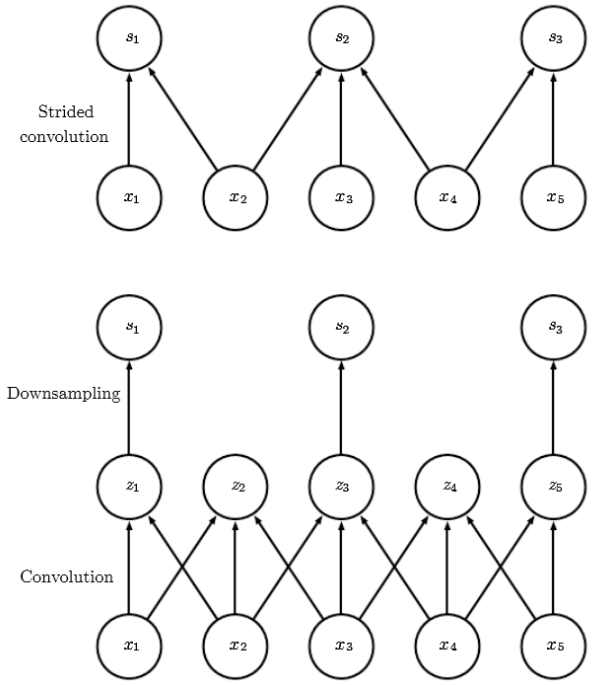

Figure 17: *Convolution with a stride.*
In this example, we use a stride of two. (Top)Convolution with a stride length of two implemented in a single operation. (Bottom) Convolution with a stride greater than one pixel is mathematically equivalent to convolution with unit stride followed by down sampling. Obviously, the two-step approach involving down sampling is computationally wasteful, because it computes many values that are then discarded. (Goodfellow, et. al, 2016)

3.2.3 Padding

One essential feature of any convolutional network implementation is the ability to implicitly zero pad the input **V** to make it wider. Without this feature, the width of the representation shrinks by one pixel less than the kernel width at each layer. Zero padding the input allows us to control the kernel width and the size of the output independently. Without zero padding, we are forced to choose between shrinking the spatial extent of the network rapidly and using small kernels—both scenarios that significantly limit the expressive power of the network. See ***Figure 18*** for an example. (Goodfellow, et. al, 2016)

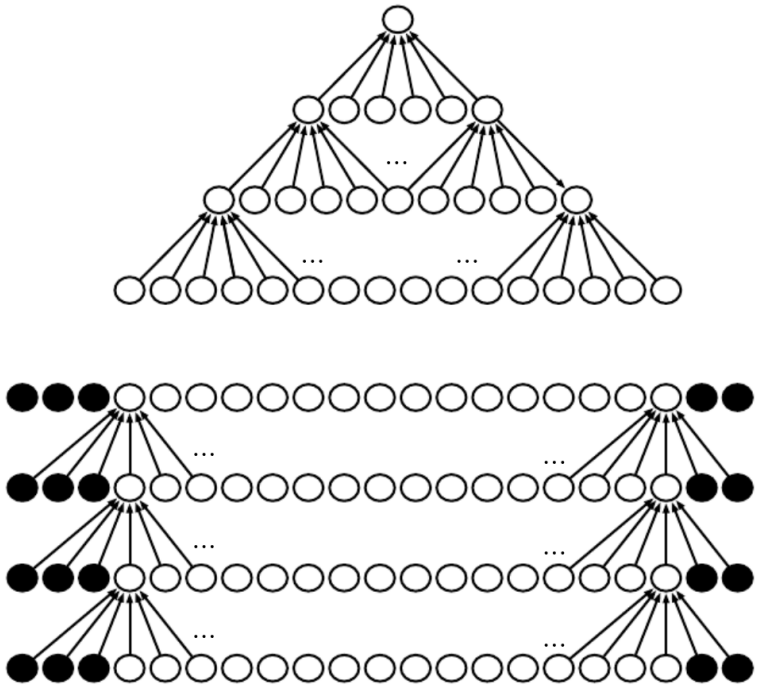

Figure 18: *The effect of zero padding on network size.*
Consider a convolutional network with a kernel of width six at every layer. In this example, we do not use any pooling, so only the convolution operation itself shrinks the network size. (Top)In this convolutional network, we do not use any implicit zero padding. This causes the representation to shrink by five pixels at each layer. Starting from an input of sixteen pixels, we are only able to have three convolutional layers, and the last layer does not ever move the kernel, so arguably only two of the layers are truly convolutional. The rate of shrinking can be mitigated by using smaller kernels, but smaller kernels are less expressive, and some shrinking is inevitable in this kind of architecture. (Bottom)By adding five implicit zeros to each layer, we prevent the representation from shrinking with depth. This allows us to make an arbitrarily deep convolutional network. (Goodfellow, et. al, 2016)

3.2.4 Putting It All Together

So now that we have reviewed the fundamental concepts of Convolutional Neural Networks, we now look more closely as to how a system employing CNNs uses specific layers. As described in the previous section, a simple Convolutional Network is a sequence of layers, and every layer of the network transforms one volume of activations to another through a differentiable function. (Li, F., et. al, 2017) We use four main types of layers to build ConvNet architectures:

a. Convolutional Layer

b. Rectified Linear Activation Layer

c. Pooling Layer

d. Fully Connected Layer (exactly as seen in regular Neural Networks)

A full Convolutional Neural Network Architecture stacks these layers. Let's briefly look at an example architecture in order to understand how practical Convolutional Neural Networks are constructed. Let's say we have an input image from the CIFAR-10 data set of a car with dimensions 32x32x3. These dimensions are 32 pixels high, 32 pixels wide, and 3 channels or colors in this case. Like the ImageNet database, the CIFAR-10 dataset is a much smaller database of images with each image 32x32x3 and from only 10 classes. (Li, F., et. al, 2017) A simple ConvNet for CIFAR-10 classification could have the layer architecture shown in ***Figure 19***.

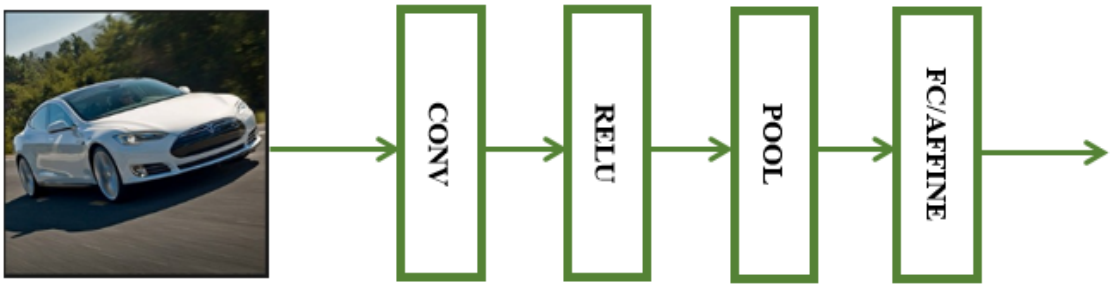

Figure 19: *Example ConvNet Architecture*

Looking at these layers in more detail we that:

a. Input Image: The example image is of dimension 32x32x3 and will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R, G, B. (Li, F., et. al, 2017)

b. CONV Layer: The CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as [32x32x12] if we decided to use 12 filters. (Li, F., et. al, 2017)

c. ReLu Layer: The RELU layer will apply an elementwise activation function, such as the max(0, x) thresholding at zero. This leaves the size of the volume unchanged ([32x32x12]). (Li, F., et. al, 2017)

d. Maxpool Layer: The POOL layer will perform a down sampling operation along the spatial dimensions (width, height), resulting in volume such as [16x16x12]. (Li, F., et. al, 2017)

e. Affine / Fully Connected Layer: The FC (i.e. fully connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score, such as among the 10 categories of CIFAR-10. As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume. (Li, F., et. al, 2017)

In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores. Note that some layers contain parameters and other don't. In particular, the CONV/FC layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers are trained with gradient descent so that the class scores that the ConvNet computes are consistent with the labels in the training set for each image. (Li, F., et. al, 2017)

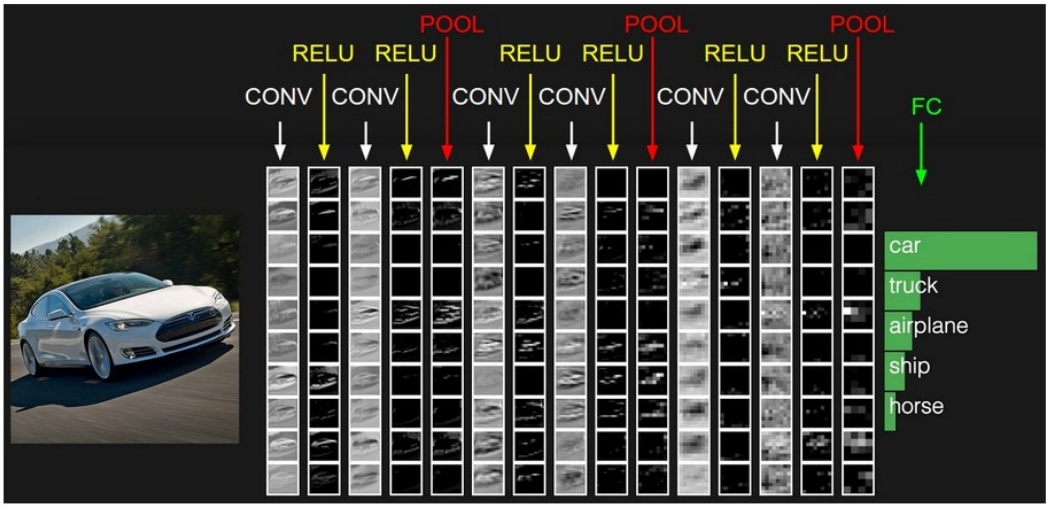

Figure 20: *Example Image Classified using Example Architecture.*
The activations of an example ConvNet architecture. The initial volume stores the raw image pixels (left) and the last volume stores the class scores (right). Each volume of activations along the processing path is shown as a column. Since it's difficult to visualize 3D volumes, we lay out each volume's slices in rows. The last layer volume holds the scores for each class, but here we only visualize the sorted top 5 scores, and print the labels of each one.
(Li, F., et. al, 2017)

4.0 DCNN DESIGN AND IMPLEMENTATION

Up to this point we have reviewed the necessary background behind Deep Learning, reviewed several celebrated Deep Convolutional Neural Network architectures, and reviewed how Deep Convolutional Neural networks have been implemented into FPGA. Now that the necessary background has been established in this subject, let's now look at the particulars of this design.

This section will describe how this work implements Deep Convolutional Neural Networks in FPGA. We will start with an overview of the design's similarities and distinctions with previous works, the goal of the design, and the tools which will be used. We will then move onto describing the overall architecture to be implemented onto the FPGA. Finally, we will review the three major sub-designs in detail: Convolutional/Affine Layer, ReLu Layer, Max Pooling Layer, and Softmax Layer.

4.1 Similarities

In reviewing the previous works by other groups who have implemented DCNNs on FPGAs we can see that there are many similarities in their implementations and this work. Of course, there are a few aspects of DCNNs which will need to be common across any design which undertakes the acceleration of DCNNs. Therefore, necessary aspects such as the needed layers (i.e. convolution, ReLu, max pool, etc.…) and adder trees to sum the products of all channels will not be discussed in this section.

4.1.1 Bus Protocol

First, in previous works we see designs using sub-module intercommunication. Many of the previous works which utilized separate sub-modules in their overall design used a communication bus protocol. This approach has the benefit of allowing already designed intellectual property from a FPGA manufacturer such as Intel or Xilinx to be used. This allows the design to be focused on the DCNN portion of the overall task rather than the infrastructure of the design itself. Another benefit is hardware microprocessors or implemented co-processors can communicate with the submodules themselves. This provides a software developer as well as a hardware developer great insight into the state of the design and is invaluable when debugging and verifying the design functionality. The only drawback is that a bus protocol can add to the overall overhead of a design. This is due to the handshaking between sub-modules

in order to establish reliable communication. The presence of the bus protocol also requires more signal routing which does use overall FPGA resources and can lead to more dwell time with no task being executed. The drawbacks of a bus protocol can be successfully managed by carefully planning out the overall design's concept of operations.

4.1.2 DSP Slices

Another common feature found in most of the previous works and in this work is the use of Digital Signal Processing Slices. These slices are dedicated hardware which can perform multiply and add operations for both floating point and fixed precision numbers. These slices are usually much faster than any custom design implemented by a designer which would need to be described in an HDL language. Previous designs have leveraged the parallel nature of FPGAs by using as many if not all the DSP slices available on the FPGA being used. It is in the designs of DCNNs as it is in many other designs the case where the more DSP slices available the faster a design will be.

4.1.3 Scalability

An important feature found in other works and in this work is the notion of stepping through the DCNN. As was seen in other works, software implementations of DCNNs are getting ever bigger year after year, with the first ResNet design utilizing 152 Layers. Implementing every single layer simultaneously in an FPGA would be so resource intensive it would very quickly not fit into many FPGAs on the market. However, other works as well as this one have taken the approach of implementing reusable designs which can be programmed to perform the function of all the layers necessary in the DCNN architecture.

4.1.4 Data Format

In the software environment, Deep Learning research is conducted with weight data as 64-bit double precision floating point signed numbers. Although there have been works which have implemented DCNNs with 32-bit Single numbers, there is a growing body of evidence to prove that reducing the bit size and format of these numbers can significantly impact overall performance. A common alteration of the number format is to use 16-bit fixed precision numbers. However, a more interesting and arguably better

design would be to truncate the 32-bit Single number down to a 16-bit "Half" precision number.

4.2 Distinctions

With the similarities between other works and this one fully described; we can now look at how this design is distinct among the others.

4.2.1 Simple Interface

First, many previous works made a significant amount of effort in creating a custom compiler in order to fully describe a Deep Convolutional Neural Network. This compiled network description would then be loaded into the design. This was done in an effort to tease out the most amount of performance possible since the compiler would analyze the proposed CNN configuration and generate scripts for the FPGA. However, in this design the desire is to make the DCNN as useful as possible to a software designer and hardware designer alike. This would amount to the FPGA hardware being programmable and able to be commanded by function calls made in the microprocessor of the design which essentially perform register writes to the FPGA implementation in this work. Therefore, software could be written to command the DCNN FPGA Layers and trigger their execution all while avoiding a custom compiler. This method is the one attempted in this work.

4.2.2 RTL Instead of High-Level Synthesis

Another attempt made by many groups is to use High Level Synthesis Tools which are a bridge in FPGA between Software C++ code and Hardware Description Language (HDL). These HLS tools attempt to create synthesizable HDL code from user C code. The use of these HLS tools, while a valuable aide in development, can lead to inefficiencies in the FPGA's resource utilization. Therefore, for this work the design is made using VHDL instead of Verilog or HLS tools. There is the point of view that further DCNN in FPGA research should focus on Deep Learning algorithms which cater mainly to hardware design. Although this viewpoint is understandable and would be ideal, it ignores the very likely fact that much of the Deep Learning algorithm development is and will continue to be conducted by Computer Scientists and Mathematicians. Therefore, there is a good case that can be made to attempt to accelerate the current Deep

Learning algorithms which are the state of the art. Doing this and providing a user-friendly interface between the FPGA design and the software designer is an ideal scenario.

4.2.3 Flexible Design

In previous works the design of a DCNN is largely tailored to the specific hardware board which the research group is using. The specific number of DSPs needed for their design was a set value. This work attempts to allow the number of DSPs in the design to be configurable depending on the FPGA being used. This would allow a developer the ability to use a wide variety of FPGA chips instead of tailoring their design to only one.

Another feature of this work is the fact that each layer in the DCNN is modular and can be used in conjunction with a bus protocol. This allows a developer to insert multiple instances of the Convolutional Layer, Affine Layer, and Max Pooling Layer. The functions of each of these layers is broken up into discrete sub-modules on the AXI Bus.

4.3 Goals

The primary goal of this design is to develop a working Deep Convolutional Neural Network FPGA accelerator for image classification. Several features were desired in the design in order allow for maximum flexibility and reusability.

1. All Layers Accelerated: For this design all the Layers in Deep Convolutional Neural Networks were to be accelerated. These layers are the Convolutional Layer, ReLu Layer, Max Pooling Layer, Affine Layer, and Softmax Layer. Therefore, an FPGA VHDL design is needed for each layer functioning in the design.
2. Modularity: In order to allow future code reuse, all layers in the design would need to be modular and able to communicate through the AXI Bus Protocol
3. Scalability: In order to allow multiple FPGA platforms to be used, the design would have to accommodate for a variable number of DSPs.

4. Programmability: This design would need to allow for software to perform register writes to configure the layers of the neural network, thereby, allowing any size network to be run on the FPGA.

4.4 Tools

Several tools were used throughout the development effort of implementing a Convolutional Neural Network onto an FPGA. The Xilinx line of chips were selected due to their widespread use and this author's previous familiarity with Xilinx products. Therefore, the tools used for this development effort were from the wide array of tools Xilinx offers. The primary design environment was the Xilinx Vivado 2016.4 package (***Figure 21***) which served as the main design hub throughout development. From here each layer type in the neural network was created as an AXI capable submodule. Vivado also provided the integration with already made Xilinx IP such as the Microblaze Co-Processor and Memory Interface Generator (MIG). Lastly, Vivado also provided a springboard for the software development which developed the simple software running on the Microblaze.

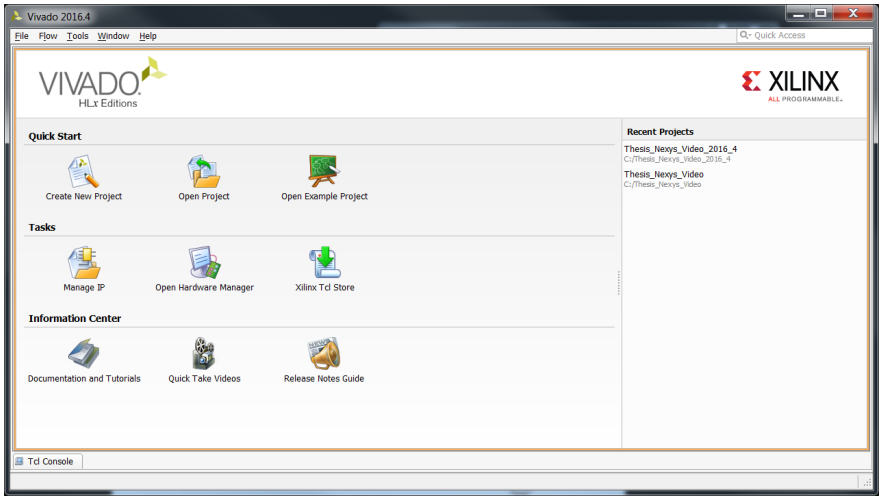

Figure 21: *Xilinx Vivado 2016.4 IDE*

Another tool which allowed for hardware debugging and software design was the Xilinx Software Design Kit (SDK) 2016.4 as shown in ***Figure 22***. This tool provided the use of the XSCT Console tool which

enables a user to directly command and read any memory location or register in the FPGA design from a PC workstation.

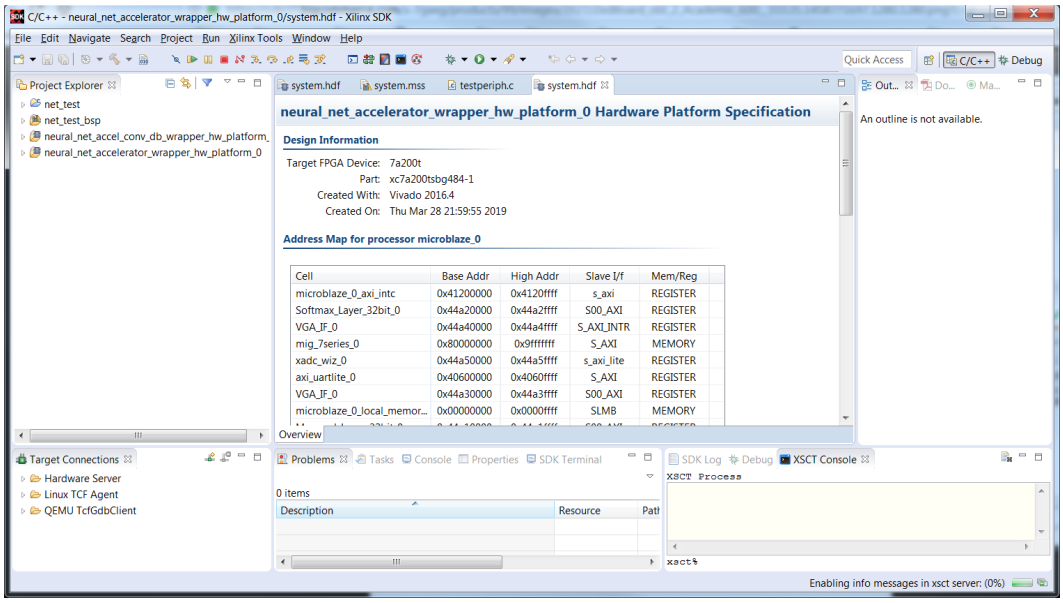

Figure 22: *Xilinx Software Design Kit 2016.4*

4.5 Hardware

For this work, two potential FPGA boards were selected due to their availability and multitude of features allowing for easier integration and testing. The Zedboard shown in ***Figure 23***(left) utilizes the Xilinx Zynq XC7Z020 System-On-Chip which hosts an ARM Processor as well as an Artix-7 logic fabric. The Nexys Video is a more powerful board which utilizes the Xilinx Artix-7 XC7A200T FPGA and is shown in ***Figure 23***(right).

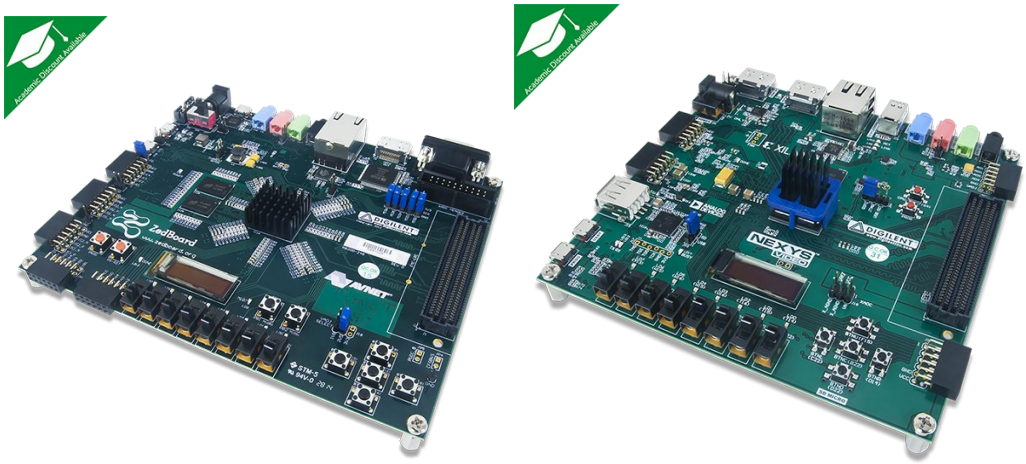

Figure 23: *(Left) Zedboard (Right) Nexys Video*

The specifications for each FPGA are shown in ***Figure 24*** and ***Figure 25.*** Looking at these figures we can see that the XC7A200T is quite a bit more powerful when compared to the XC7Z020. The XC7A200T has 215K logic cells, 13.1MB of Block RAM, 270K Flip Flops, and 740 DSP slices. The XC7Z020's has 85K logic cells, 4.9MB of Block RAM, 106K Flip Flops, and 220 DSP slices. Therefore, depending on how many resources this design takes up, there are two boards available to fit the design.

		XC7A200T
	Part Number	215,360
Logic Resources	Logic Cells	33,650
	Slices	269,200
	CLB Flip-Flops	2,888
Memory Resources	Maximum Distributed RAM (Kb)	365
	Block RAM/FIFO w/ ECC (36 Kb each)	13,140
	Total Block RAM (Kb)	10
Clock Resources	CMTs (1 MMCM + 1 PLL)	500
I/O Resources	Maximum Single-Ended I/O	240
	Maximum Differential I/O Pairs	740
Embedded Hard IP Resources	DSP Slices	1
	PCIe® Gen2[1]	1
	Analog Mixed Signal (AMS) / XADC	1
	Configuration AES / HMAC Blocks	16
	GTP Transceivers (6.6 Gb/s Max Rate)[2]	-1, -2
Speed Grades	Commercial Temp (C)	-2L, -3
	Extended Temp (E)	-1, -2, -1L
	Industrial Temp (I)	

Figure 24: *Specifications for the Artix-7 XC7A200T FPGA (Xilinx, 2018)*

Programmable Logic (PL)	7 Series PL Equivalent		Artix-7
	Logic Cells		85K
	Look-Up Tables (LUTs)		53,200
	Flip-Flops		106,400
	Total Block RAM (# 36Kb Blocks)		4.9Mb (140)
	DSP Slices		220
	PCI Express®		—
	Analog Mixed Signal (AMS) / XADC[2]		s with up to
	Security[3]		entication f
	Speed Grades	Commercial	-1
		Extended	-2,-3
		Industrial	-1, -2, -1L

Figure 25: *Specifications for the Zynq XC7Z020 Artix-7 FPGA (Xilinx, 2017)*

5.0 FPGA TOP-LEVEL ARCHITECURAL DESIGN AND CONCEPT

In order to implement the Forward Pass of a Convolutional Neural Network, the multiple layers involved need to be included in a top-level FPGA design. As was seen in Section 3.2.4, the layers which comprise a Convolutional Neural Network for Deep Learning are:

a. Convolution Layer

b. Rectified Linear Unit (ReLu) Layer

c. Max Pooling Layer

d. Affine (Fully Connected) Layer

e. Classification Layer

Each of these layers needs to pull data from the memory available on the FGPA board in order to execute their designed tasks. The overall architecture of the design is shown in

Figure 30. However, before jumping straight to the details of the CNN Layers, we must first address several significant structural design decisions. Important considerations such as Bus Protocol, Data Format, and a method of performing Floating Point arithmetic need to be considered.

5.1 AXI BUS Overview

The selection of a bus protocol allows for the system to be modular and flexible. When a design deploys a Bus for command and data transfers, any custom designed logic block with bus interface can attach to the bus with no impact to other logic blocks in the system. Let's briefly describe how the AXI Bus protocol works.

The AXI Bus protocol, which stands for Advanced eXtensible Interface, is part of ARM AMBA, a family of micro controller buses first introduced in 1996. The first version of AXI was first included in AMBA 3.0, released in 2003. AMBA 4.0, released in 2010, includes the second version of AXI, AXI4. A robust collection of third-party AXI tool vendors is available that provide a variety of verification, system development, and performance characterization tools. (ARM, 2018)

There are three types of AXI4 interfaces:

a. AXI4—for high-performance memory-mapped requirements. This is for memory mapped interfaces and allows burst of up to 256 data transfer cycles with just a single address phase. (ARM, 2018)
b. AXI4-Lite—for simple, low-throughput memory-mapped communication (for example, to and from control and status registers). This is a lightweight, single transaction memory mapped interface. It has a small logic footprint and is a simple interface to work with both in design and usage. (ARM, 2018)
c. AXI4-Stream—for high-speed streaming data. This removes the requirement for an address phase altogether and allows unlimited data burst size. AXI4-Stream interfaces and transfers do not have address phases and are therefore not considered to be memory-mapped. (ARM, 2018)

The AXI specifications describe an interface between a single AXI master and a single AXI slave, representing IP cores that exchange information with each other. Memory mapped AXI masters and slaves can be connected using a structure called an Interconnect block. The Xilinx AXI Interconnect IP contains AXI-compliant master and slave interfaces and can be used to route transactions between one or more AXI masters and slaves. Both AXI4 and AXI4-Lite interfaces consist of five different channels:

1. Read Address Channel
2. Write Address Channel
3. Read Data Channel
4. Write Data Channel
5. Write Response Channel

Data can move in both directions between the master and slave simultaneously, and data transfer sizes can vary. The limit in AXI4 is a burst transaction of up to 256 data transfers. AXI4-Lite allows only 1 data transfer per transaction. ***Figure 26*** shows how an AXI4 Read transaction uses the Read address and Read

data channels. ***Figure 27*** shows how an AXI4 Write transaction uses the Write address and Write data channels. (ARM, 2018)

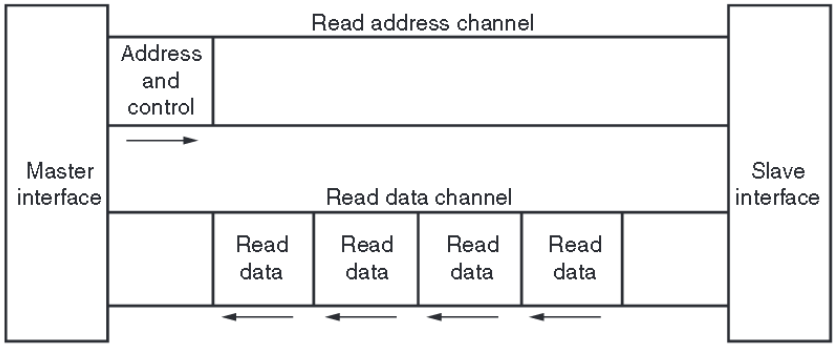

Figure 26: *Channel Architecture of Reads (ARM, 2018)*

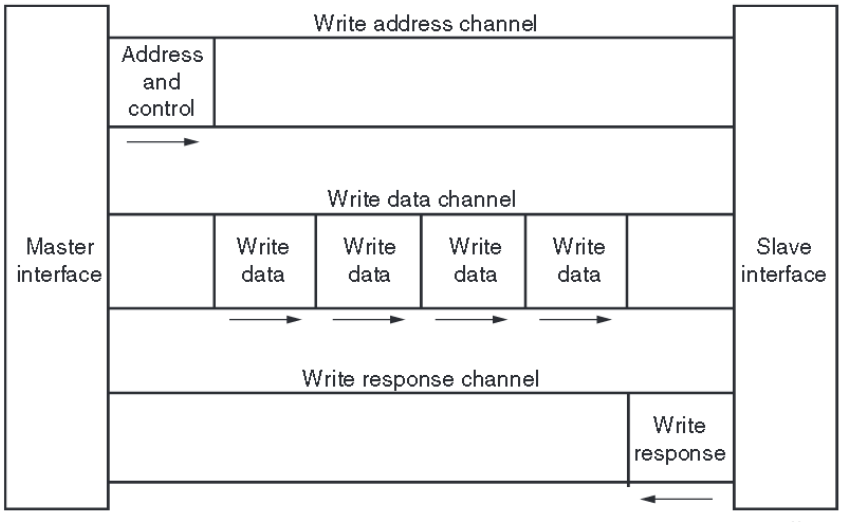

Figure 27: *Channel Architecture of Writes (ARM, 2018)*

As shown in ***Figure 26*** and ***Figure 27***, AXI4 provides separate data and address connections for Reads and Writes, which allows simultaneous, bidirectional data transfer. AXI4 requires a single address and then bursts up to 256 words of data. At a hardware level, AXI4 allows a different clock for each AXI master-slave pair. In addition, the AXI protocol allows the insertion of register slices (often called pipeline stages) to aid in timing closure. (ARM, 2018)

AXI4-Lite is similar to AXI4 with some exceptions, the most notable of which is that bursting, is not supported. The AXI4-Stream protocol defines a single channel for transmission of streaming data. The AXI4-Stream channel is modeled after the Write Data channel of the AXI4. Unlike AXI4, AXI4-Stream interfaces can burst an unlimited amount of data. (ARM, 2018)

As mentioned earlier, memory mapped AXI masters and slaves can be connected using a structure called an Interconnect block. Therefore, each master and slave in this design has a designated address which is used by the Microblaze to address each master and slave. The Address Map is shown in **Table 1**.

Table 1: *AXI Address Map*

AXI Module	AXI Address Start	AXI Address End
Microblaze Local Memory	0x00000000	0x0000FFFF
Microblaze Slave	0x41200000	0x4120FFFF
Memory Interface	0x800000000	0x9FFFFFFF
UART	0x40600000	0x4060FFFF
Convolution Layer	0x44A00000	0x44A0FFFF
Max Pooling Layer	0x44A10000	0x44A1FFFF
Softmax Layer	0x44A20000	0x44A2FFFF
VGA Interface	0x44A30000	0x44A3FFFF

5.2 Data Format

The decision of which data format to choose has significant impacts on an FPGA's resource utilization. The data format is a consideration since the data values throughout a Convolutional Neural Network are not integer values but can have a wide range of fractional precision. As was seen in the research, while using a 32-bit floating point single precision format may lead to increased accuracy, it inevitably leads to increased FPGA resource utilization. Therefore, it can be said that using a large format such as single precision floating point may be a bit overkill. This can be especially true if Convolutional Neural Networks regularly normalize the input data and convolved data. Looking briefly at the data formats to choose from we can see that we have several options.

5.2.1 Fixed Point

The fixed-point number consists of integer and fraction parts. (Brown et. al., 2002) It can be written in the positional number representation as:

$$B = b_{n-1}b_{n-2} \cdots b_1 b_0 . b_{-1}b_{-2} \cdots b_{-k}$$

The value of the number is

$$V(B) = \sum_{i=-k}^{n-1} b_i \times 2^i$$

The position of the radix point is assumed to be fixed; hence the name fixed-point number. If the radix point is not shown, then it is assumed to be to the right of the least significant digit, which means that the number is an integer. (Brown et. al., 2002)

If we look at an example, let's say we have the value 11.3125 and we want to convert this number to a 16bit Fixed Point representation. The integer part of the number is:

$$(11)_{10} = (00001011)_2$$

The fractional part of the number is:

$$(0.3125)_{10} = (?)_2$$

Whole Part	Fractional Part
	0.3125
0	0.625
1	0.25

0	0.5
1	0

$$(0.3125)_{10} = (.010100000\ )_2$$

Therefore the 16-bit Fixed Point representation is:

$$(11.3125)_{10} = (00001011.010100000\ )_2$$

The 8bit integer part of the 16-bit Fixed Point data format has a range from 0 to 255. The 8bit fractional portion of the 16-bit Fixed Point data format has a precision of 3.9E-3. (Brown et. al., 2002)

5.2.2 Floating Point Single and Double Precision

The Floating-Point Single Precision format is depicted below in ***Figure 28***(a). The left-most bit is the sign bit, 0 for positive and 1 for negative numbers. There is an 8-bit exponent field, E, and a 23-bit mantissa field, M. The exponent is with respect to the radix 2. Because it is necessary to be able to represent both very large and very small number, the exponent can be either positive or negative. Instead of simply using an 8-bit signed number as the exponent., which would allow exponent values in the range -128 to 127, the IEEE standard specifies the exponent in the excess-127 format. (Brown et. al., 2002) In this format the value 127 is added to the value of the actual exponent so that:

$$Exponent = E - 127$$

In this way E becomes a positive integer. This format is convenient for adding and subtracting floating-point numbers because the first step in these operations involves comparing the exponents to determine whether the mantissas must be appropriately shifted to add/subtract the significant bits. The range of E is 0 to 255. The extreme values of E = 0 and E = 255 are taken to denote the exact zero and infinity,

respectively. Therefore, the normal range of the exponent is -126 to 127, which is represented by the values of E from 1 to 254. (Brown et. al., 2002)

The mantissa is represented using 23 bits. The IEEE standard calls for a normalized mantissa, which means that the most-significant bit is always equal to 1. Thus, it is not necessary to include this bit explicitly in the mantissa field. Therefore, if M is the bit vector in the mantissa field, the actual value of the mantissa is 1.M, which gives a 24-bit mantissa. (Brown et. al., 2002) Consequently, the floating-point format in ***Figure 28***(a) represents the number:

$$Value = \pm 1.M \times 2^{E-127}$$

The size of the mantissa field allows the representation of numbers that have the precision of about seven decimal digits. The exponent field range of 2^{-126} to 2^{-127} corresponds to about $10^{\pm 38}$. (Brown et. al., 2002)

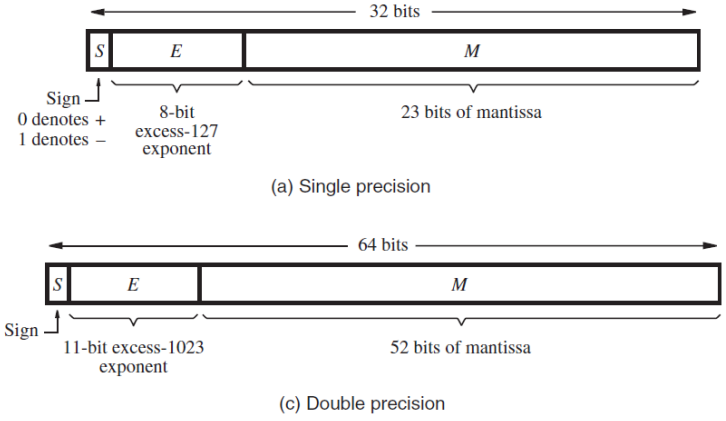

Figure 28: *IEEE Standard Floating-Point Formats*
(Brown et. al., 2002)

The Floating-Point Double Precision format is depicted in ***Figure 28***(b) and uses 64 bits. Both the exponent and mantissa fields are larger. (Brown et. al., 2002) This format allows greater range and precision of numbers. The exponent field has 11 bits, and it specifies the exponent in the excess-1023 format, where:

$$Exponent = E - 1023$$

The range of E is 0 to 2047, but again the values of E = 0 and E = 2047 are used to indicate the exact 0 and infinity, respectively. Thus, the normal range of the exponent is -1022 to 1023, which is represented by the values of E from 1 to 2046. (Brown et. al., 2002)

The mantissa field has 52 bits. Since the mantissa is assumed to be normalized, its actual value is again 1.M. Therefore, the value of a floating-point number is:

$$Value = \pm 1.M \times 2^{E-1023}$$

This format allows representation of numbers that have the precision of about 16 decimal digits and the range of approximately$10^{\pm 308}$. (Brown et. al., 2002)

5.2.3 Floating Point Half Precision

In 2008 the Institute of Electrical and Electronic Engineers released a revised Standard for Floating-Point Arithmetic. (IEEE, 2008) This newer standard included a new floating-point format designated as "binary16", where "binary32" is known as Single Precision and "binary64" is known as Double Precision. The binary16 format has recently come to be known colloquially as the "Half Precision" format. (IEEE, 2008)

The Floating-Point Half Precision format is depicted in ***Figure 28***(b) and uses only 16 bits. Both the exponent and mantissa fields are smaller than the Single and Double counterparts. This format allows for greater range and precision of numbers than the fixed format but less than that of the Single or Double Precision formats. The exponent field has 5 bits, and it specifies the exponent in the excess-15 format, where,

$$Exponent = E - 15$$

The range of E is 0 to 31, but again the values of E = 0 and E = 31 are used to indicate the exact 0 and infinity, respectively. Thus, the normal range of the exponent is -14 to 15, which is represented by the values of E from 1 to 32. (IEEE, 2008)

The mantissa field has 10 bits. Since the mantissa is assumed to be normalized, its actual value is again 1.M. Therefore, the value of a floating-point number is

$$Value = \pm 1.M \times 2^{E-15}$$

This format allows representation of numbers that have the precision of about 4 decimal digits and the range of approximately 10^4 to 10^{-5}. (IEEE, 2008)

5.2.4 Data Format Decision

Having looked at the various formats to choose from, we can see that the two which would be good candidates for use in this implementation of a Convolutional Neural Network would be the Single and Half Precision formats. While the Single Precision would give increased accuracy, the Half Precision would be best suited for an Embedded System Design. Although, a knee jerk reaction would be to immediately jump to implement the Half Precision, a few considerations would lend merit to the argument that a Single Precision system would first need to be implemented. The first of these considerations is another aspect of Floating-Point Arithmetic in Computer systems. Round off error deals with the rules in which a Computer system will round toward the limit of its computational precision. Different systems may incorporate a subtly different round off scheme which can generate large scale differences in a final answer when multiple arithmetic operations are being performed. Another consideration would be commonality between the outputs of a software model of the neural network and the outputs of an FPGA design implementing a neural network. Having a commonality between a software model and the FPGA design enables the inevitable debugging effort which takes place during the simulation and hardware phases of an FPGA design. Therefore, the Single Precision format is selected as the primary format for this work. Time permitted; the Half Precision format would be explored. This step by step approach allows for a baseline design to be developed first. Then once the baseline is fully functional, the newer format can be tested.

5.3 DSP48 Floating Point Logic

With the main objective of this work being the implementation of a Convolutional Neural Network, the task of designing the Floating-Point Arithmetic Logic was abstracted away by utilizing already made macros. The Xilinx DSP38E1 Slice located physically on the FPGA fabric allows for the efficient implementation of various mathematical operations.

All Xilinx 7 Series FPGAs have many dedicated, full-custom, low-power DSP slices, combining high speed with small size while retaining system design flexibility. The DSP slices enhance the speed and efficiency of many applications beyond digital signal processing, such as wide dynamic bus shifters, memory address generators, wide bus multiplexers, and memory mapped I/O registers. The basic functionality of the DSP48E1 slice is shown in ***Figure 29***. (Xilinx UG479, 2018)

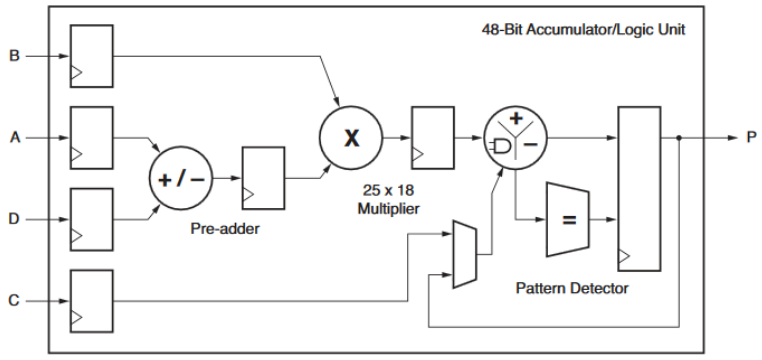

Figure 29: *Basic DSP48E1 Slice Functionality (Xilinx UG479*

5.4 Top Level Architecture and Concept

With solutions for the Bus Protocol, Data Format, and Floating-Point logic decided, now we can look at the overall top-level design of this implementation of Convolutional Neural Networks. As shown in ***Figure 30***, we can see the top-level block diagram of the top-level FPGA Design. With the goal of modularity in mind, each CNN layer type is instantiated in the design. A Xilinx generated memory interface is instantiated in order to interface with the board's DDR Memory. The DDR Memory is the main receptacle for all input, weight, bias, and output data sets being generated by this design. A Xilinx Microblaze Co-Processor is also included which coordinates the operations and flow of data between the layers of the neural network. A Xilinx UART core is instantiated to allow debugging and commanding from an external computer. Each of the blocks of logic communicates through a central AXI Bus. This AXI Bus forms the essential back bone in the system for all data transfer and commanding.

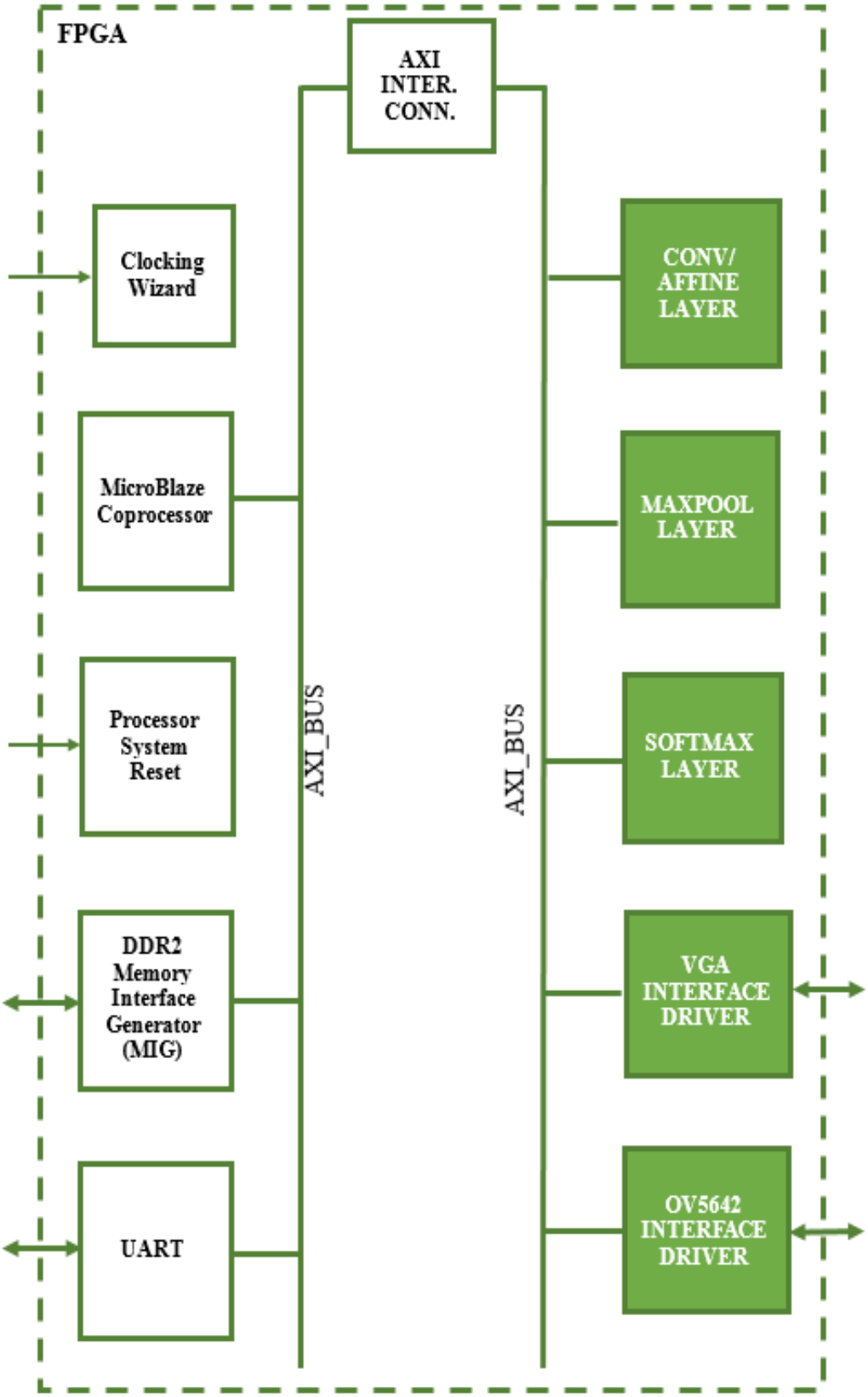

Figure 30: *FPGA Top Level Architecture*

5.5 Top Level Module Descriptions

The table below contains a brief description of each module in the top-level FPGA design. The benefit of using the AXI Bus becomes clear when considering the significant amount of infrastructure which has already been designed. The AXI Interconnect, Clocking Wizard, MicroBlaze Co-Processor, Processor System Reset, Memory Interface Generator, and UART are all modules from existing Xilinx IP. This allowed this work to focus on the design and development of the Deep Convolutional Neural Network aspects of the design.

Table 2: *Top Level Modules*

MODULE NAME	SUMMARY DESCRIPTION	ORIGIN
AXI Interconnect	Part of the LogiCORE IP from Xilinx, this module allows any mixture of AXI master and slave devices to be connected to it.	Xilinx LogiCORE IP AXI Interconnect v2.1
Clocking Wizard	Part of the LogiCORE IP from Xilinx, this module automates the process of creating a clocking network.	Xilinx LogiCORE IP Clocking Wizard v5.1
MicroBlaze Coprocessor	The Microblaze embedded soft core is a reduced instruction set computer (RISC) optimized for implementation on Xilinx FPGAs.	Xilinx MicroBlaze v9.6
Processor System Reset	Part of the LogiCORE IP from Xilinx, this module provides a mechanism to handle the reset conditions for a given system.	Xilinx LogiCORE IP Processor System Reset Module v5.0
Memory Interface Generator	Part of the LogiCORE IP from Xilinx, this module	Xilinx MIG IP
UART	Part of the LogiCORE IP from Xilinx, this module provides the controller interface for asynchronous serial data transfer.	Xilinx LogiCORE IP AXI UARTlite v2.0

CONV/AFFINE LAYER	Convolution Layer which executes the convolution operation on image data. This module also performs the Affine Layer function.	Custom Module
MAXPOOL LAYER	Max Pooling module which performs the max pooling operation on image data.	Custom Module
SOFTMAX LAYER	Softmax Layer which performs the classification of the image content.	Custom Module
VGA INTERFACE DRIVER	Module to interface with the VGA Monitor and display image sensor images.	Custom Module
OV5642 INTERFACE DRIVER	Module to interface with the Omnivision OV5642 image sensor.	Custom Module

5.6 Top Level Port Descriptions

The table below shows the top-level ports of the FPGA design. These ports are routed to physical pins on the FPGA chip and interface with the DDR memory.

Table 3: *Top Level Port Description*

PORT NAME	TYPE	DIR.	DESCRIPTION
i_sys_clk	std_logic	in	System clock from PCB board oscillator
i_reset_rtl	std_logic	in	Off chip reset from PCB board
i_HREF	std_logic	in	Image Sensor Horizontal Reference Signal
i_VSYNC	std_logic	in	Image Sensor Vertical Sync Signal
i_PCLK	std_logic	in	Image Sensor Pixel Clock
i_btn_config	std_logic	in	Button signal to configure the image sensor
i_config_bypass	std_logic	in	Switch signal which will allow logic to ignore config step and continue
i_SDATA	std_logic_vector (15 downto 0)	in	Data from image sensor
o_XCLK	std_logic	out	Image sensor processor clock
o_hsync	std_logic	out	VGA Horizontal sync
o_vsync	std_logic	out	VGA Vertical sync
o_vga_data	std_logic_vector (15 downto 0)	out	Data to VGA Monitor
o_PWDN	std_logic	out	Power down signal to image sensor processor
o_SIOC	std_logic	out	Image sensor serial input/ouput clock
o_SIOD	std_logic	out	Image sensor serial input/output data
uart_rx	std_logic	in	Universal Asynchronous Receive and Transmit (UART) Receive Port
uart_tx	std_logic	out	Universal Asynchronous Receive and Transmit (UART) Transmit Port
ddr2_dq	std_logic_vector (15 downto 0)	Inout	DDR2 Data Input
ddr2_dqs_p	std_logic_vector (1 downto 0)	Inout	DDR2 Data Strobe Positive Differential
ddr2_dqs_n	std_logic_vector (1 downto 0)	Inout	DDR2 Data Strobe Negative Differential
ddr2_addr	std_logic_vector (12 downto 0)	out	DDR2 Address Inputs
ddr2_ba	std_logic_vector (2 downto 0)	out	DDR2 Bank Address Inputs
ddr2_ras_n	std_logic	out	DDR2 Command Inputs Negative Differential
ddr2_cas_n	std_logic	out	DDR2 Command Input
ddr2_we_n	std_logic	out	DDR2 Write Enable
ddr2_ck_p	std_logic_vector (0 downto 0)	out	DDR2 Clock Positive Differential
ddr2_ck_n	std_logic_vector (0 downto 0)	out	DDR2 Clock Negative Differential
ddr2_cke	std_logic_vector (0 downto 0)	out	DDR2 Clock Enable
ddr2_cs_n	std_logic_vector (0 downto 0)	out	DDR2 Chip Select
ddr2_dm	std_logic_vector (1 downto 0)	out	DDR2 Input Data Mask
ddr2_odt	std_logic_vector (0 downto 0)	out	DDR2 On-die Termination

5.7 Top Level Memory Map

In order to execute the AlexNet Convolutional Neural Network, each layer's input, output, weights, and bias data sets need to be saved to DDR Memory on the FPGA Board. With the 32bit Single Precision data format selected, we can calculate the systems memory requirements which are shown in ***Table 4***. With the memory requirements determined, we can derive the memory map for the 32bit system which is shown in ***Table 5***.

Table 4: *Memory Requirements for AlexNet CNN*

Layer Data Set	Layer Specifications	Memory in Bytes 32 bit
Input Image	Height: 227 Width: 227 Channels: 3	309174 (16bit)
Weight 1	Height: 11 Width: 11 Channels: 3 Filters: 96	139392
Conv1	Height: 55 Width: 55 Channels: 96	1161600
Max Pool 1	Height: 27 Width: 27 Channels: 96	279936
Weight 2	Height: 5 Width: 5 Channels: 96 Filters: 256	2457600
Conv2	Height: 27 Width: 27 Channels: 256	746496
Max Pool 2	Height: 13 Width: 13 Channels: 256	173056
Weight 3	Height: 3 Width: 3 Channels: 256 Filters: 384	3538944
Conv3	Height: 13 Width: 13 Channels: 384	259584
Weight 4	Height: 3 Width: 3 Channels: 384 Filters: 384	5308416
Conv4	Height: 13 Width: 13 Channels: 384	259584
Weight 5	Height: 3 Width: 3 Channels: 384	3538944

	Filters: 256	
Conv5	Height: 13 Width: 13 Channels: 256	173056
Max Pool 3	Height: 6 Width: 6 Channels: 256	36864
Weight 6	Height: 6 Width: 6 Channels: 256 Filters: 4096	150994944
Fully Conn. 6	Height: 1 Width: 1 Channels: 4096	16384
Weight 7	Height: 1 Width: 1 Channels: 4096 Filters: 4096	67108864
Fully Conn. 7	Height: 1 Width: 1 Channels: 4096	16384
Weight 8	Height: 1 Width: 1 Channels: 4096 Filters: 10	163840
Fully Conn. 8	Height: 1 Width: 1 Channels: 10	40
Bias 1	96	384
Bias 2	256	1024
Bias 3	384	1536
Bias 4	384	1536
Bias 5	256	1024
Bias 6	4096	16384
Bias 7	4096	16384
Bias 8	10	40
TOTAL		236721414

Table 5: *Top Level Memory Map*

Data Set	Offset Address Start	Offset Address End
Input Image	0x00000000	0x0004B7FF
Weight 1	0x0004B800	0x0006D8FF
Conv1	0x0006D900	0x001892FF
Max Pool 1	0x00189300	0x001CD8FF
Weight 2	0x001CD900	0x00425BFF
Conv2	0x00425A00	0x004DBEFF
Max Pool 2	0x004DBF00	0x005063FF
Weight 3	0x00506400	0x008664FF
Conv3	0x00866500	0x008A5AFF
Weight 4	0x008A5B00	0x00DB5BFF
Conv4	0x00DB5A00	0x00DF52FF
Weight 5	0x00DF5300	0x011553FF
Conv5	0x01155400	0x0117F8FF
Max Pool 3	0x0117F900	0x011888FF
Weight 6	0x40000000	0x0D0000FF
Fully Conn. 6	0x0D000100	0x0D0041FF
Weight 7	0x10000000	0x140000FF
Fully Conn. 7	0x14000100	0x140041FF
Weight 8	0x18000000	0x18FA00FF
Fully Conn. 8	0x18FA0100	0x18FB00FF
Bias 1	0x1C000000	0x1C0001FF
Bias 2	0x1C000200	0x1C00040F
Bias 3	0x1C000410	0x1C000A1F
Bias 4	0x1C000A20	0x1C00102F
Bias 5	0x1C001030	0x1C00113F
Bias 6	0x1C001140	0x1C00214F
Bias 7	0x1C002150	0x1C00315F
Bias 8	0x1C003160	0x1C00355F

6.0 CONVOLUTION / AFFINE LAYER

6.1 Algorithm

Using the background covered thus far, FPGA hardware can be described which would allow the FPGA circuitry to perform the Convolution operation. The Convolution Operation involves applying a weight filter kernel to an input image in order to generate an output volume. These output volumes in effect filter the image for certain features which are determined during training of the weight set of the Neural Network. (Li, F., et. al, 2017) In order to describe how a CNN can be implemented into an FPGA we look at the example in ***Figure 31*** shown below

.

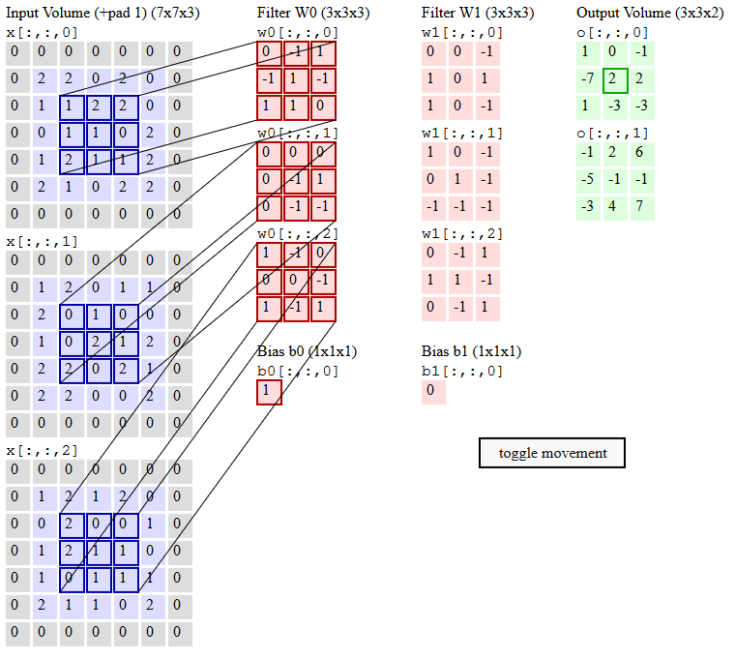

Figure 31: *Convolution Example.*
(Li, F., et. al, 2017)

Here we can see that each element in the filter weight kernel is multiplied by its corresponding element in the image data. So, for the case with a 3x3 filter weight kernel, 9 filter weight values are multiplied by 9 image data values, to generate 9 products. These 9 products are then summed together in order to produce the output volume. This is done for each color of the input image and in subsequent volumes/filter maps, each channel. This operation is again repeated for each subsequent weight filter set. So, if a 227x227x3

image has an 11x11x3x96 weight filter kernel set applied to it, with a stride and pad of 2, the resulting output volume would be 55x55x96.

6.2 Conv/Affine Layer – Top Level Architecture and Concept

The Convolutional / Affine Layer Module architecture is shown below in ***Figure 32***. The module utilizes an AXI4-Lite Slave Interface which provides a command and status interface between the MicroBlaze processor and the internal logic of the module. The module also utilizes an AXI4 Master Interface which reads and writes data to and from the DDR memory on the FPGA board. The AXI Master Interface retrieves the input data, weights, and biases from specified DDR memory locations for the convolution and fully connected operations then outputs the output volume to a specified region of memory.

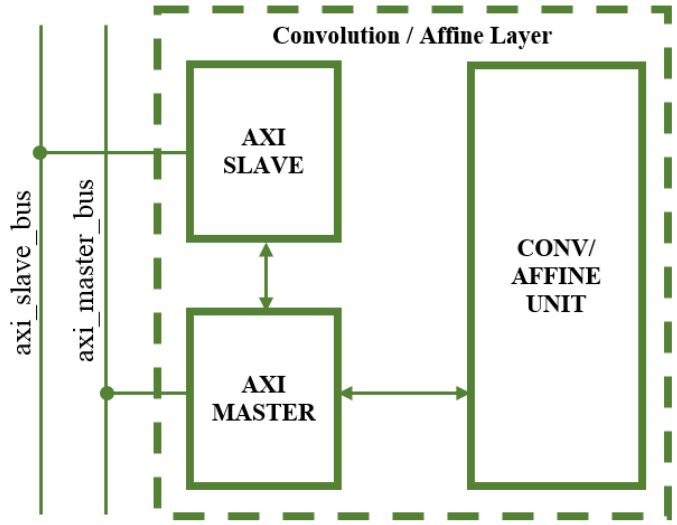

Figure 32: *Top level Architecture of the Convolution/Affine Layer*

6.3 Conv/Affine Layer - Register Set

The AXI Slave provides the following register set for the Convolution / Affine Layer shown in ***Table 6***. This layer design is very large and required quite a few registers to allow for reliable RTL operation and to meet timing when generating the bit file to put on the actual FPGA board.

Table 6: *Register List for the Conv/Affine Layer Design*

Register Name	Offset Address
Control Register	0x0
Status Register	0x4
Input Data Address Register	0x8
Output Data Address Register	0xC
Weights Address Register	0x10
Input Volume Parameters Register	0x14
Output Volume Parameters Register	0x18
Weight Parameters Register	0x1C
Convolution Parameters Register	0x20
Reserved	0x24
Reserved	0x28
Reserved	0x2C
Bias Address Register	0x30
Bias Parameters Register	0x34
Weight Multiple 0 Register	0x38
Weight Multiple 1 Register	0x3C
Input Multiple 0 Register	0x40
Input Multiple 1 Register	0x44
Output Multiple 0 Register	0x48
Output Multiple 1 Register	0x4C
Affine Parameters 0 Register	0x50

6.3.1 Control Register

This register contains essential settings which determine the mode in which the Conv/Affine layer operates.

Table 7: *Control Register Bit Map*

Output Data Address Register				**(Base Address + 0x00C)**			
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
Unassigned							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Unassigned					**Input_size**		
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Unassigned		**Affine_en**	**ReLu_en**	Unassigned			**Reset_all**
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Unassigned			**Debug_mode**	Unassigned			**Start**

Table 8: *Control Register Description*

Bit Field	Name	Initial Value	Description
31:19	unassigned	0x0	
18:16	Input_size	0x0	Specifies if the input data is 8,16, or 32 bits wide.
15:14	Unassigned	0x0	
13	Affine_en	'0'	Enables the Affine Layer Logic
12	ReLu_en	'0'	Enables the Relu Layer Logic
11:9	Unassigned	0x0	
8	Reset_all	'0'	Resets the design logic.
7:5	Unassigned	0x0	
4	Debug_mode	'0'	Allows for each channel iteration output to be written at a different memory location for hardware verification testing
3:1	unassigned	0x0	
0	Start	'0'	Starts the Affine and Convolution Layer execution

6.3.2 Status Register

The status register breaks out important logic signals which may be used for debugging the design once hardware testing has begun.

Table 9: *Status Register Bit Map*

Status Register				**(Base Address + 0x004)**			
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
Unassigned							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
unassigned		**Inbuff almost full**	**Inbuff full**	unassigned		**Inbuff almost empty**	**Inbuff empty**
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
unassigned		**Outbuff almost full**	**Outbuff full**	unassigned		**Outbuff almost empty**	**Outbuff empty**
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Conv complete	**Weights loaded**	**more dsps**	**Iteration complete**	unassigned			**busy**

Table 10: *Status Register Description*

Bit Field	Name	Initial Value	Description
31:22	Unassigned	0x0	Register bits for unassigned in current stage of development
21	Inbuff_almost_full	'0'	Indicates the input buffer is almost full and is about to fill up
20	Inbuff_full	'0'	Indicates the input buffer is full and will not accept more data
19:18	Unassigned	0x0	Register bits for unassigned in current stage of development
17	Inbuff_almost_empty	'0'	Indicates the input buffer contains only one piece of valid data
16	Inbuff_empty	'0'	Indicates the input buffer is empty with no valid data.
15:14	Unassigned	0x0	Register bits for unassigned in current stage of development
13	Outbuff_almost_full	'0'	Indicates the output buffer is almost full and is about to fill up
12	Outbuff_full	'0'	Indicates the output buffer is full and will not accept more data
11:10	Unassigned	0x0	Register bits for unassigned in current stage of development
9	Outbuff_almost_empty	'0'	Indicates the output buffer contains only one piece of valid data
8	outbuff_empty	'0'	Indicates the output buffer is empty with no valid data.
7	Conv_complete	'0'	Indicates convolution is complete
6	Weights_loaded	'0'	Indicates the new weight filter kernel is loaded and ready
5	More_dsps	'0'	Indicates not enough dsps present to execute all channels
4	done	'0'	Indicates that the Softmax classifier has completed the current operation
3:1	Unassigned	0x0	Register bits for unassigned in current stage of development
0	busy	'0'	Indicates the Softmax classifier is currently busy executing the current operation

6.3.3 Input Data Address Register

The value contained in this register points the location in the DDR memory were the layer input data begins.

Table 11: *Input Data Address Register Bit Map*

Input Data Address Register				(Base Address + 0x008)			
31	30	29	28	27	26	25	24
Input_data_addr							
23	22	21	20	19	18	17	16
Input_data_addr							
15	14	13	12	11	10	9	8
Input_data_addr							
7	6	5	4	3	2	1	0
Input_data_addr							

Table 12: *Input Data Address Register Description*

Bit Field	Name	Initial Value	Description
31:0	Input_data_address	0x0	Address where design will pull data from

6.3.4 Output Data Address Register

The value contained in this register points the location in the DDR memory were the layer will begin outputting the resulting data from the Convolution of Fully connected operations.

Table 13: *Output Data Address Register Bit Map*

Output Data Address Register				(Base Address + 0x00C)			
31	30	29	28	27	26	25	24
Output_data_addr							
23	22	21	20	19	18	17	16
Output_data_addr							
15	14	13	12	11	10	9	8
Output_data_addr							
7	6	5	4	3	2	1	0
Output_data_addr							

Table 14: *Output Data Address Register Description*

Bit Field	Name	Initial Value	Description
31:0	Output_data_address	0x0	Address where Max Pooler will save the output of the max pooling operation

6.3.5 Weights Address Register

The value contained in this register points the location in the DDR memory were the layer weight kernel data begins.

Table 15: *Filter Weights Address Register Bit Map*

Weights Address Register				(Base Address + 0x010)			
31	30	29	28	27	26	25	24
Filter_weights_addr							
23	22	21	20	19	18	17	16
Filter_weights_addr							
15	14	13	12	11	10	9	8
Filter_weights_addr							
7	6	5	4	3	2	1	0
Filter_weights_addr							

Table 16: *Filter Weights Address Register Description*

Bit Field	Name	Initial Value	Description
31:0	Filter_weights_address	0x0	Address where design will save the output of the conv/affine operation

6.3.6 Input Volume Parameters Register

This register contains information on the size of the input image such as Height, Width, and number of channels.

Table 17: *Input Volume Parameters Register Bit Map*

Input Volume Parameters Register				(Base Address + 0x014)			
31	30	29	28	27	26	25	24
image_height							
23	22	21	20	19	18	17	16
image_width							
15	14	13	12	11	10	9	8
image_channels							
7	6	5	4	3	2	1	0
image_channels							

Table 18: *Input Volume Parameters Register Description*

Bit Field	Name	Initial Value	Description
31:24	Image_height	0x0	Address where Max Pooler will save the output of the max pooling operation
23:16	Image_width	0x0	Address where Max Pooler will save the output of the max pooling operation
15:0	Image_channels	0x0	Address where Max Pooler will save the output of the max pooling operation

6.3.7 Output Volume Parameters Register

This register contains information on the size of the output image such as Height, Width, and number of channels.

Table 19: *Output Volume Parameters Register Bit Map*

Output Data Address Register				**(Base Address + 0x018)**			
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
output_volume_height							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
output_volume_width							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
output_volume_channels							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
output_volume_channels							

Table 20: *Output Volume Parameters Register Description*

Bit Field	Name	Initial Value	Description
31:24	Output_volume_height	0x0	Number of Rows in the Output Image
23:16	Output_volume_width	0x0	Number of Columns in the Output image
15:0	Output_volume_channels	0x0	Number of Channels in the output image

6.3.8 Weight Parameters Register

This register contains information on the size of the weight filter kernel such as Height, Width, number of channels, and number of filters.

Table 21: *Weight Parameters Register Bit Map*

Output Data Address Register				(Base Address + 0x01C)			
31	30	29	28	27	26	25	24
weight_filter_height				weight_filter_width			
23	22	21	20	19	18	17	16
number_of_channels							
15	14	13	12	11	10	9	8
number_of_channels				number_of_filters			
7	6	5	4	3	2	1	0
number_of_filters							

Table 22: *Weight Parameters Register Description*

Bit Field	Name	Initial Value	Description
31:28	Weight_filter_height	0x0	Number of rows of the kernel
27:24	Weight_filter_width	0x0	Number of columns of the kernel
23:12	Number_of_channels	0x0	Number of channels of the kernel
11:0	Number_of_filters	0x0	Number of filters of the kernel

6.3.9 Convolution Parameters Register

This register contains information on the size of the input image such as Height, Width, and number of channels.

Table 23: *Convolution Parameters Register Bit Map*

Output Data Address Register				(Base Address + 0x020)			
31	30	29	28	27	26	25	24
unassigned							
23	22	21	20	19	18	17	16
unassigned							
15	14	13	12	11	10	9	8
Number of DSPs							
7	6	5	4	3	2	1	0
Stride				Pad			

Table 24: *Convolution Parameters Register Description*

Bit Field	Name	Initial Value	Description
31:16	unassigned	0x0	
15:8	Number of DSPs	0x0	
7:4	Stride	0x0	Number of pixels to stride by
3:0	Pad	0x0	Number of rows and columns to zero pad the image

6.3.10 Bias Address Register

The value contained in this register points the location in the DDR memory were the layer bias data begins.

Table 25: *Bias Data Address Register Bit Map*

Bias Data Address Register				(Base Address + 0x030)			
31	30	29	28	27	26	25	24
Bias_data_addr							
23	22	21	20	19	18	17	16
Bias_data_addr							
15	14	13	12	11	10	9	8
Bias_data_addr							
7	6	5	4	3	2	1	0
Bias_data_addr							

Table 26: *Bias Data Address Register Description*

Bit Field	Name	Initial Value	Description
31:0	Bias_data_address	0x0	Address where Conv/Affine Layer will obtain the Bias values.

6.3.11 Bias Parameters Register

This register contains information on the size of the bias data.

Table 27: *Bias Parameters Register Bit Map*

Bias Parameters Register				(Base Address + 0x034)			
31	30	29	28	27	26	25	24
unassigned							
23	22	21	20	19	18	17	16
unassigned							
15	14	13	12	11	10	9	8
Bias_length							
7	6	5	4	3	2	1	0
Bias_length							

Table 28: *Bias Parameters Register Description*

Bit Field	Name	Initial Value	Description
31:16	unassigned	0x0	
15:0	Bias_length	0x0	Number of bias values to read into the design.

6.3.12 Weight Multiple 0 Register

This register contains a precalculated value which is essential for the design to read the correct weight filter kernel data into the design.

$$weight\_multiple\_0 = kernel\ height * kernel\ width * channels\ allowed$$

Table 29: *Weight Multiple 0 Register Bit Map*

Weight Multiple 0 Register				(Base Address + 0x038)			
31	30	29	28	27	26	25	24
Weight_multiple_0							
23	22	21	20	19	18	17	16
Weight_multiple_0							
15	14	13	12	11	10	9	8
Weight_multiple_0							
7	6	5	4	3	2	1	0
Weight_multiple_0							

Table 30: *Weight Multiple 0 Register Description*

Bit Field	Name	Initial Value	Description
31:0	Weight_multiple_0	0x0	Precalculated value to determine proper weight memory address to read.

6.3.13 Weight Multiple 1 Register

This register contains a precalculated value which is essential for the design to read the correct weight filter kernel data into the design.

$$weight\_multiple\_1 = kernel\ height * kernel\ width * Input\ channels$$

Table 31: *Weight Multiple 0 Register Bit Map*

Weight Multiple 1 Register				(Base Address + 0x03C)			
31	30	29	28	27	26	25	24
Weight_multiple_1							
23	22	21	20	19	18	17	16
Weight_multiple_1							
15	14	13	12	11	10	9	8
Weight_multiple_1							
7	6	5	4	3	2	1	0
Weight_multiple_1							

Table 32: *Weight Multiple 1 Register Description*

Bit Field	Name	Initial Value	Description
31:0	Weight_multiple_1	0x0	Precalculated value to determine proper weight memory address to read.

6.3.14 Input Multiple 0 Register

This register contains a precalculated value which is essential for the design to read the correct Input image data into the design.

$$input\_multiple\_0 = Input\ Height * Input\ Width$$

Table 33: *Input Multiple 0 Register Bit Map*

Input Multiple 0 Register				(Base Address + 0x040)			
31	30	29	28	27	26	25	24
input_multiple_0							
23	22	21	20	19	18	17	16
input_multiple_0							
15	14	13	12	11	10	9	8
input_multiple_0							
7	6	5	4	3	2	1	0
input_multiple_0							

Table 34: *Input Multiple 0 Register Description*

Bit Field	Name	Initial Value	Description
31:0	input_multiple_0	0x0	Precalculated value to determine proper input data memory address to read.

6.3.15 Input Multiple 1 Register

This register contains a precalculated value which is essential for the design to read the correct Input image data into the design.

$$input\_multiple\_1 = (Weight\ Height - Pad) * Input\ Width$$

Table 35: *Input Multiple 1 Register Bit Map*

Input Multiple 1 Register				(Base Address + 0x044)			
31	30	29	28	27	26	25	24
input_multiple_1							
23	22	21	20	19	18	17	16
input_multiple_1							
15	14	13	12	11	10	9	8
input_multiple_1							
7	6	5	4	3	2	1	0
input_multiple_1							

Table 36: *Input Multiple 1 Register Description*

Bit Field	Name	Initial Value	Description
31:0	input_multiple_1	0x0	Precalculated value to determine proper input data memory address to read.

6.3.16 Output Multiple 0 Register

This register contains a precalculated value which is essential for the design to write the resulting output data to the correct memory address.

$$Output\_multiple\_0 = Output\ Height * Output\ Width$$

Table 37: *Output Multiple 0 Register Bit Map*

Output Multiple 0 Register				(Base Address + 0x048)			
31	30	29	28	27	26	25	24
output_multiple_0							
23	22	21	20	19	18	17	16
output_multiple_0							
15	14	13	12	11	10	9	8
output_multiple_0							
7	6	5	4	3	2	1	0
output_multiple_0							

Table 38: *Output Multiple 0 Register Description*

Bit Field	Name	Initial Value	Description
31:0	output_multiple_0	0x0	Precalculated value to determine proper output data memory write address.

6.3.17 Output Multiple 1 Register

This register contains a precalculated value which is essential for the design to write the resulting output data to the correct memory address.

$$Output\_multiple\_1 = Input\ Width * Output\ Channels$$

Table 39: *Output Multiple 1 Register Bit Map*

Output Multiple 1 Register				(Base Address + 0x04C)			
31	30	29	28	27	26	25	24
output_multiple_1							
23	22	21	20	19	18	17	16
output_multiple_1							
15	14	13	12	11	10	9	8
output_multiple_1							
7	6	5	4	3	2	1	0
output_multiple_1							

Table 40: *Output Multiple 1 Register Description*

Bit Field	Name	Initial Value	Description
31:0	output_multiple_1	0x0	Precalculated value to determine proper output data memory write address.

6.3.18 Affine Parameters 0 Register

This register contains information on the number of channels and filters in the current set being calculated.

Table 41: *Affine Parameters 0 Register Bit Map*

Affine Parameters 0 Register				(Base Address + 0x050)			
31	30	29	28	27	26	25	24
Affine_fitlers_in_set							
23	22	21	20	19	18	17	16
Affine_filters_in_set							
15	14	13	12	11	10	9	8
Affine_channels_in_set							
7	6	5	4	3	2	1	0
Affine_channels_in_set							

Table 42: *Affine Parameters 0 Register Description*

Bit Field	Name	Initial Value	Description
31:0	affine_filters_in_set	0x0	Number of filters to group together per execution of the Affine Layer
15:0	affine_channels_in_set	0x0	Number of channels to group together per execution of the Affine Layer

6.4 Conv/Affine Layer - AXI Master

While the AXI Slave interface receives commands via the AXI Slave Bus, the AXI Master is responsible for reading into the design all relevant data from memory and then writing the resulting volume of data back to memory for the next layer in the Neural Network to use. The AXI Master logic is a large state machine shown in ***Figure 33*** with states specifically designed to retrieve the input data, weight filter kernel data, and bias data from memory regions specified by the AXI Slave Register settings. This module starts the Convolution and Affine operations and collects the results of these operations.

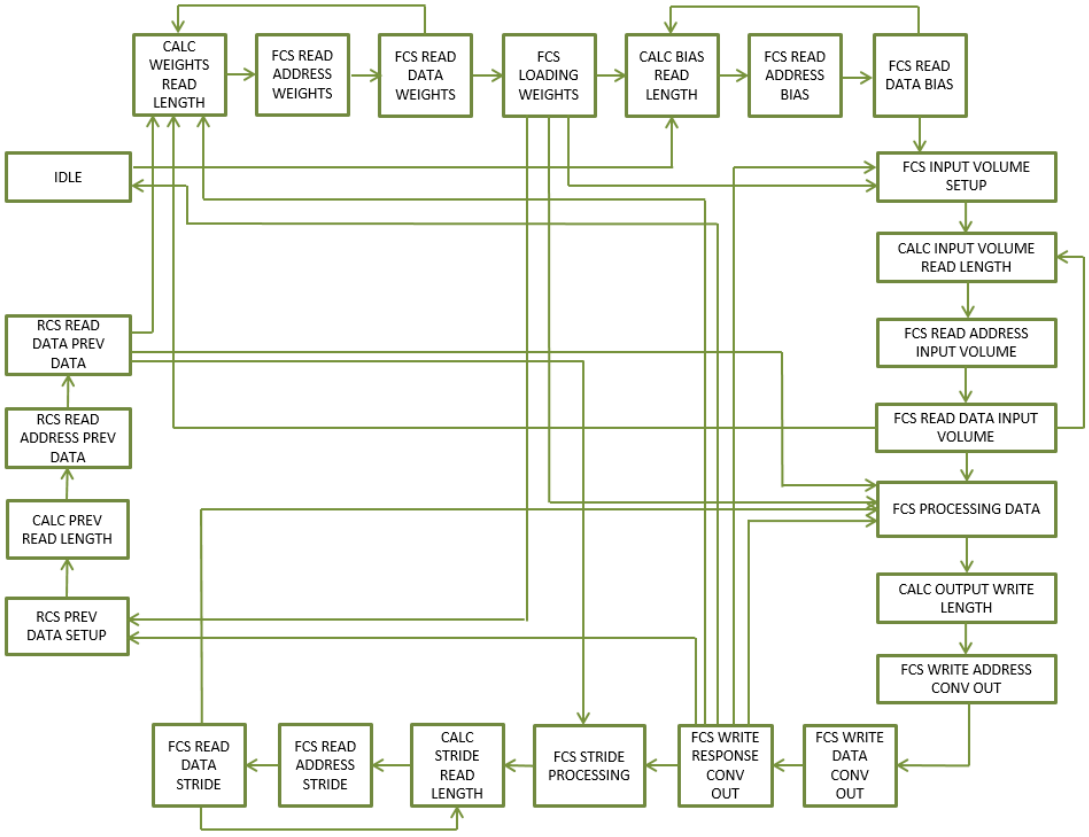

Figure 33: *Finite State Machine for the AXI Master Module.*
Due to the complexity, the state machine has been simplified for readability.

Due to the complexity of this large state machine the figure was simplified in order to increase its readability. Therefore, the states are briefly detailed below in text.

IDLE: The FSM checks various control signals in order to ensure that the logic downstream is ready to receive data and begin its operation. This state also waits for a start command from the Control Register.

CAL_WEIGHTS_READ_LENGTH, FCS_ADDRESS_WEIGHTS,

FCS_READ_DATA_WEIGHTS: These states calculate the appropriate size of AXI read transfer to perform based on the Weight data configuration registers in the AXI Slave interface. This is to ensure that the correct weights are being read from memory for the Convolution and Affine operations to yield a correct answer. Once all the appropriate weights have been read in the state machine moves on.

FCS LOADING WEIGHTS: This state holds the state machine until the logic downstream signals that all the weights have been loaded into the Channel Unit Weight FIFOs. For the convolution operation, on the first iteration of the channel set, the State Machine moves onto obtaining the biases for the Neural Network. On subsequent iterations the state machine will move onto load new input data. For the Affine operation, on the first iteration of the channel set, the state machine moves onto hold for the operation complete. On subsequent iterations the state machine moves onto fetch the layer results from the previous iteration.

BIAS_READ_LENGTH, FCS_READ_ADDRESS_BIAS,

FCS_READ_DATA_BIAS: These states read in the Bias data for the current layer in the Neural Network. The correct number of bias values is specified in the configuration register: Bias Parameters Register.

FCS_INPUT_VOLUME_SETUP, CALC_INPUT_VOLUME_LENGTH,

FCS_READ_ADDRESS_INPUT_VOLUME, FCS_READ_DATA_INPUT_VOLUME: These states calculate the appropriate size of AXI write transfer to perform based on the input data configuration registers in the AXI Slave interface. This is to ensure that the correct input data is being read from memory for the Convolution and Affine operations to yield a correct answer. Once all the appropriate input data have been read in the state machine moves on.

FCS_PROCESSING_DATA: This state holds the state machine until the output FIFO buffer indicates that there is data in the buffer. Once data is detected, the state machine moves on.

CALC_OUTPUT_WRITE_LENGTH, FCS_WRITE_ADDRESS_CONV_OUT, FCS_WRITE_DATA_CONV_OUT, FCS_WRITE_RESPONSE_CONV_OUT:
These states calculate the appropriate size of AXI write transfers to perform based on the output volume configuration registers in the AXI Slave interface. The state machine reads out all the data from the previous Convolution or Affine operation and writes the data to the address specified in the Output Address configuration register. This state is the ultimately the last state in the whole state machine design. The state machine will move on if there are more channel iterations to complete and/or the full Convolution or Affine output has not yet been calculated.

FCS_STRIDE_PROCESSING, CALC_STRIDE_READ_LENGTH, FCS_READ_ADDRESS_STRIDE, FCS_READ_DATA_STRIDE: This state is uniquely a Convolutional operation state. The Affine operation does not stride across the input volume. If the full input volume has not yet been completed, additional rows of input data will be read from memory. The number of rows read is specified by the Convolution Parameters configuration register. Once all the additional stride rows are complete read in for each of the channel set being calculated, the design moves onto the FCS_PROCESSING_DATA state to wait for output data from the Convolution operation to be generated.

RCS_PREV_DATA_SETUP, CALC_PREV_READ_LENGTH, RCS_READ_ADDRESS_PREV_DATA, RCS_READ_DATA_PREV_DATA: If the number of input channels exceeds the number of DSPs available for use, these previous data state will be used. These states are used in both the Convolutional and Affine situations. After the Convolution or Affine operation has completed for the input data volume, the whole state machine starts again to work on the next set of channel data. However, this time before each operation, the previous data that was output is read back into the design so that it can be summed with the current operation output.

6.5 Conv/Affine Unit – Design and Ports

A look inside the Convolution / Affine Unit is shown in ***Figure 34*** below. The unit is comprised of four major functional blocks. The input buffer is a FIFO buffer which receives data from the AXI Master and reads it out to the Filter logic. Once the Filter logic has finished performing Convolution or an Affine operation the data is passed through the ReLu Unit and output to the Output FIFO buffer. The contents of the output buffer are then read out by the AXI Master interface and written to DDR Memory.

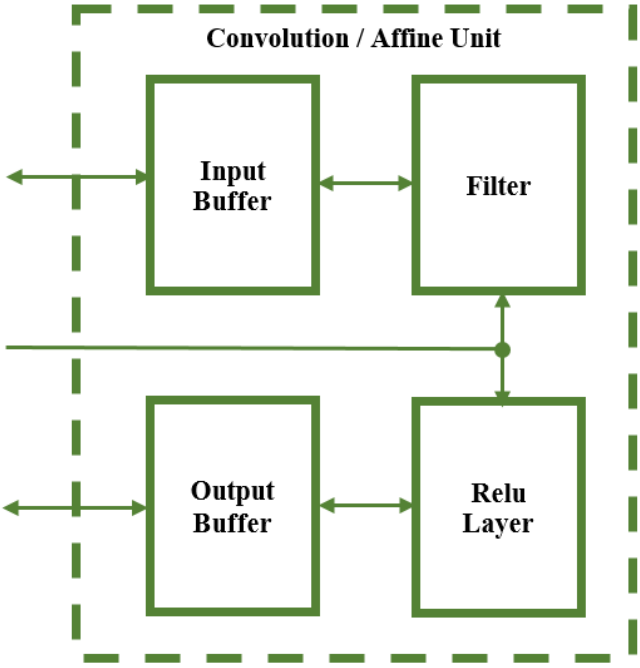

Figure 34: *Architecture of the Convolution / Affine Unit*

The ports for this submodule are shown in ***Table 43*** below.

Table 43: *Convolution/Affine Unit Port Descriptions*

PORT NAME	TYPE	DIR.	DESCRIPTION
i_clk	std_logic	In	Clock signal to for sequential logic
i_reset_n	std_logic	In	Active Low Reset
i_start	std_logic	In	Start Pulse to begin Convolution Operation
i_output_volume_size	std_logic_vector (7 downto 0)	In	Height and Width of Output Volume
i_input_volume_channels	std_logic_vector (11 downto 0)	In	Number of channels in output volume
i_input_volume_size	std_logic_vector (7 downto 0)	In	Height and Width of Input Volume
i_number_of_filters	std_logic_vector (11 downto 0)	In	Number of filters in weight kernel set
i_weight_filter_channels	std_logic_vector (11 downto 0)	In	Number of channels in weight kernel set
i_weight_filter_size	std_logic_vector (3 downto 0)	In	Height and Width of weight kernel set
i_pad	std_logic_vector (3 downto 0)	In	Number of rows and columns to pad input data
i_stride	std_logic_vector (3 downto 0)	In	Number of pixels to skip
i_ReLu_en	std_logic	In	Signal to enable the ReLu activation function
i_inbuff_din	std_logic_vector (15 downto 0)	In	Input data to the convolution layer
i_inbuff_wr_en	std_logic	In	Write enable signal for input buffer
i_outbuff_rd_en	std_logic	In	Read enable signal for output buffer
o_weights_loaded	std_logic	Out	Signal indicated that the new weight kernel is loaded and ready
o_conv_complete	std_logic	Out	Signal indicating convolution is complete
o_more_dsps	std_logic	Out	Signal indicating more dsps needed to finish channel set
o_iteration_complete	std_logic	Out	Signal indicating partial channels executed
o_channels_allowed	std_logic_vector (7 downto 0)	Out	Number of channels allowed based off dsps used
o_dsps_used	std_logic_vector (7 downto 0)	Out	Number of dsps circuits used in this design
o_inbuff_empty	std_logic	Out	Input Buffer has no valid data
o_inbuff_almost_empty	std_logic	Out	Input Buffer has one more

			piece of valid data
o_inbuff_full	std_logic	Out	Input Buffer cannot accept any more data
o_inbuff_almost_full	std_logic	Out	Input Buffer can accept one more piece of valid data
o_inbuff_valid	std_logic	Out	Input Buffer contains valid data
o_outbuff_dout	std_logic_vector (15 downto 0)	Out	Output Buffer data to AXI Master Interface
o_outbuff_empty	std_logic	Out	Output Buffer does not contain valid data
o_outbuff_almost_empty	std_logic	Out	Output Buffer contains only one piece of valid data
o_outbuff_full	std_logic	Out	Output Buffer cannot accept any more data
o_outbuff_almost_full	std_logic	Out	Output Buffer can only accept one more piece of data
o_outbuff_valid	std_logic	Out	Output Buffer contains valid data

6.6 Convolution Layer - Submodule Definitions and Operations

The following section will describe the design of each of the submodules involved in the design of the Convolution / Affine Layer Unit.

6.6.1 Filter Sub-module

The inner workings of the Filter submodule are shown below in ***Figure 35***. The Filter submodule takes in data from the input buffer FIFO and distributes the data across all the Channel Units in the Volume Weight Multiply Logic group. This group takes the input data and weight data and uses the Xilinx DSP48 hardware embedded into the FPGA itself to perform a floating-point single precision multiplication. The product results of the Floating-Point Multiplication operations are passed to the Accumulator Logic group. There the product data is summed together in order to get the final summed kernel data that both Convolution and Affine operations require. The Accumulator Logic group also sums the kernel data together with the bias data.

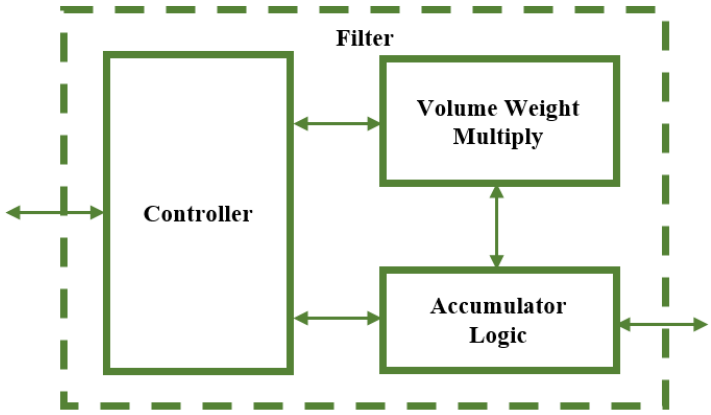

Figure 35: *Filter Submodule Architecture*

6.6.2 Channel Unit Sub-modules

The heart of this design is the Channel Unit which involves the use of a FIFO buffer and glue logic to perform vital operations of the Convolution and Affine layer in the Convolutional Neural Network. Essentially this design utilizes heavily the available DSP resources on the FGPA chip. The number of available DSPs dictates the maximum number of Channel Units instantiated into the design. Therefore, as

is shown in ***Figure 36*** the maximum number of Channel Units *n* instantiated in the design is directly influenced by the number of usable DSPs on the FPGA.

Operationally each Channel Unit corresponds to one row of data of the input image or one row of data for all the filters for the weight data set. Weight filter kernels in Convolutional Neural Networks typically have a size of 3x3, 5x5, 7x7, or 11x11. Therefore, the height of the weight filter kernel will dictate the number of input volume channels that can be operated on at any given time. So, the configuration of the Channel Units and the overall Filter is directed by the available number of DSPs and the number channels the input data has.

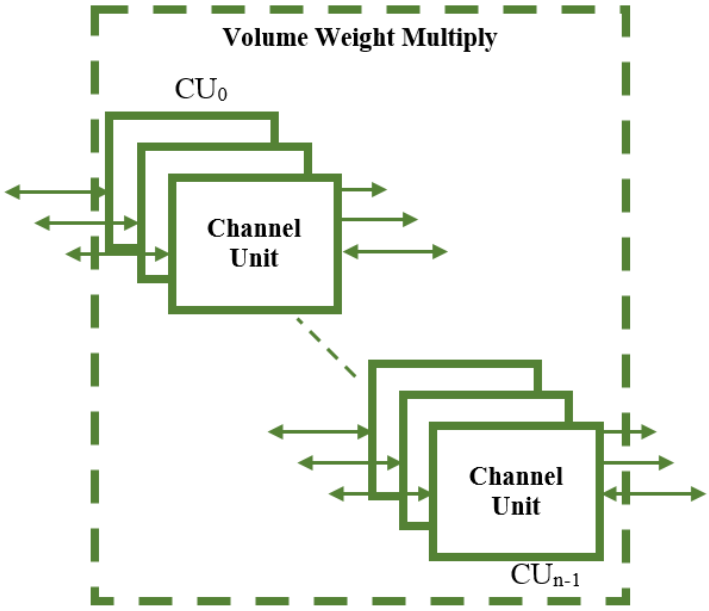

Figure 36: *Number of Channel Units able to be used in the design.*

The inner workings of the Channel Unit itself are shown below in ***Figure 37***. Here we can see what was described earlier. The Channel Unit block takes input data and weight data and uses the Xilinx DSP48 hardware embedded into the FPGA itself to perform a floating-point single precision multiplication.

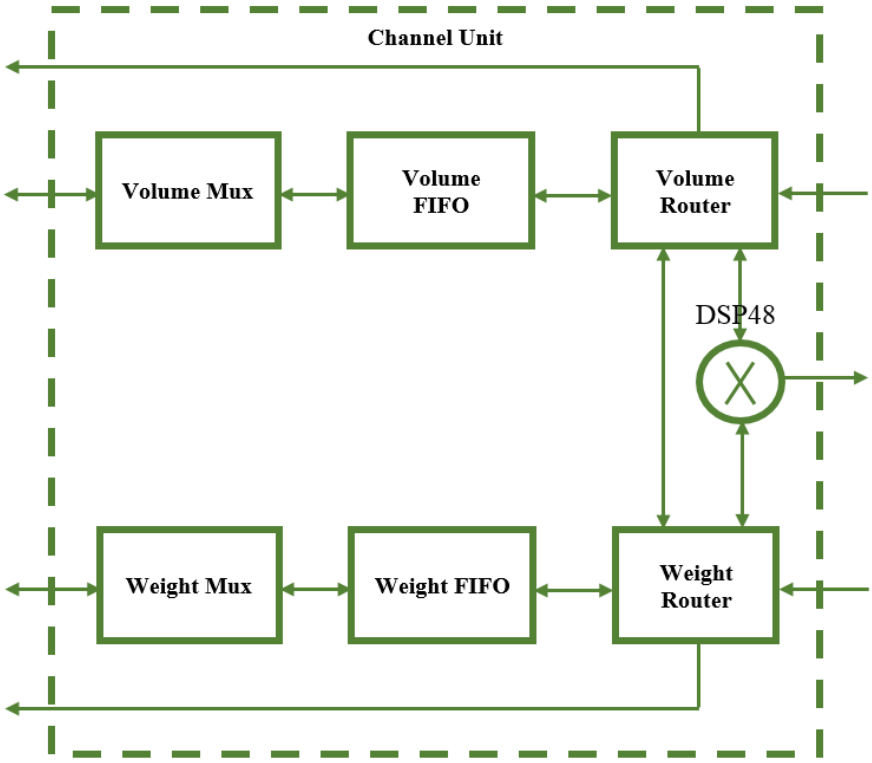

Figure 37: *Architecture of the Channel Unit*

In order to more thoroughly explain the concept of operation of this design let's look an example case. Let's say we have an input data volume of dimension 227x227x3 which is 227 pixels high, 227 pixels wide, and with 3 channels. Let's also assume for this example we have a corresponding weight set of dimensions 5x5x3x16. As shown in ***Figure 38*** and ***Figure 39***, both input volume data and weight data are loaded into the channel units available. Since the weight filter kernel is 5 pixels high, only 5 rows of input volume data of the 227 total rows are loaded into the FIFOs. The corresponding weight filter data for all the weight filters is loaded into the weight FIFOs as well. In ***Figure 38*** and ***Figure 39*** it is important to note that the data FIFOs are surrounded by logic blocks. These logic blocks are the Mux and the Router logic. The Mux serves to multiplex the various data streams being written into the data FIFO. The Router serves to direct where the flow of data is going between FIFOs in the Channel Unit.

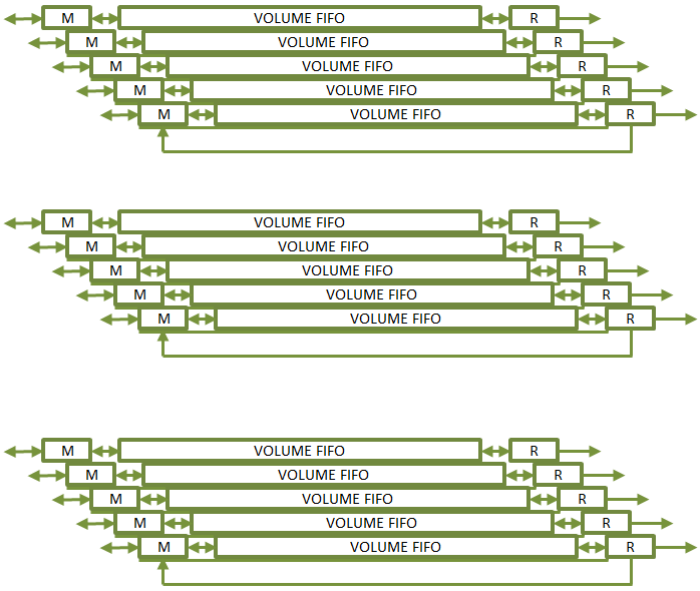

Figure 38: *Volume FIFOs with Mux and Router for the Input Volume FIFOs*

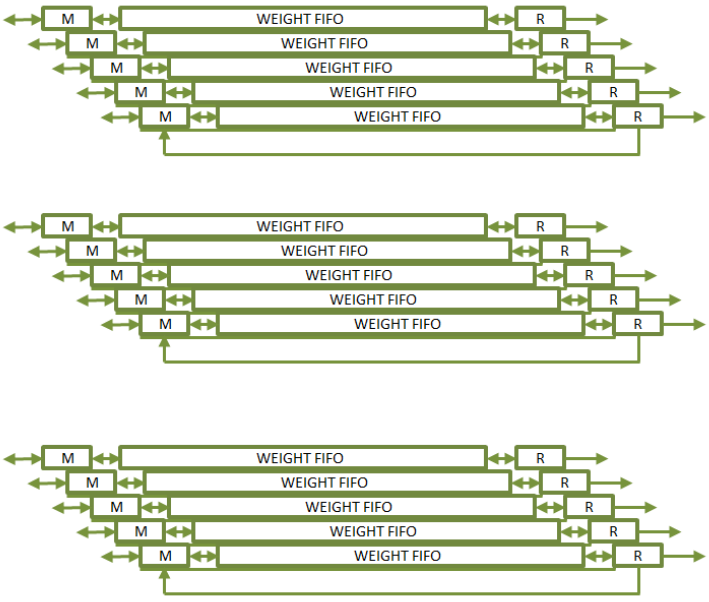

Figure 39: *Weight FIFOs with Mux and Router for the Weight Data FIFOs*

For Convolutional Layers in a Neural Network the input data volume may be zero padded around the boarders of the input data volume. Therefore, the input data image will have top, bottom, right, and left boarders with zero value data. This practice of padding is also accomplished in this design as shown below in ***Figure 40***. Maintaining the earlier input and weight data example we add the additional configuration that this example will have a padding of 2. We see in ***Figure 40***(Top) that in this case the first two rows of data going to the input volume FIFOs are zero filled. For the remaining 3 input volume FIFOs the first three rows of input data are fed to the input volume data FIFOs. Note the first three rows of input data are padded on the right and left sides with 2 zero data values. Similarly, the padding for the bottom border of the input data volume is shown in ***Figure 40***(Bottom).

Figure 40: *Padding of the Input Data Volume.*
(Top) Shows the padding of the top and sides of the input data volume. (Bottom) Shows the padding of the bottom and sides of the input data volume.

When the design is given a new input data volume to use in the execution of a Convolution or Affine operation it proceeds as illustrated below in ***Figure 41***. The Controller loads full rows of both input and weight data into the appropriate Channel Unit FIFO.

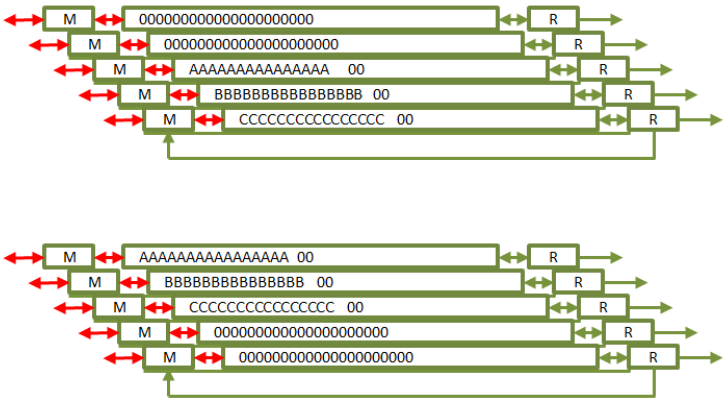

Figure 41: *Shows how new data is read into the Channel Unit FIFOs*

After the rows have been loaded into both Input Volume and Weight Data FIFOs, the Routers for the Weight Data read out the first Weight Filter Kernel and hold it in registers. Once the kernel is ready, the Routers on the Volume FIFOs begin to read out the input volume data while also accounting for stride across the row itself. As both input data and weight kernel are read out simultaneously, they are sent to the Channel Units DSP48 multiplier block. Also, simultaneously the input volume data is written back into the input data FIFOs and is shown below in ***Figure 42***. Writing back the input data allows the same data to be reused for execution with the next weight filter kernel, thereby reducing the number of AXI transactions necessary in the design.

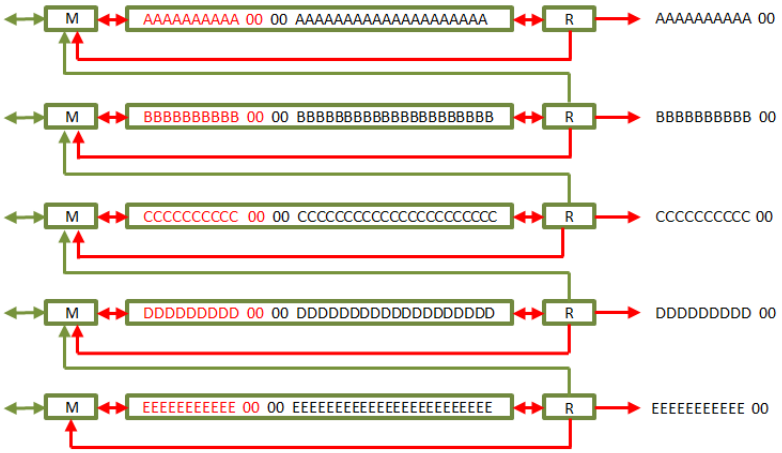

Figure 42: *FIFO Data being Recycled.*
Shows how during normal operation, the volume data is recycled back into the FIFO for future use with additional Wight Filter Kernels.

After all the weight filter kernels have been used with the input data volume rows, the stride operation begins. This is to satisfy the vertical stride required for the Convolution operation. To accomplish this function, the Router for each Channel Unit reads out its FIFOs data and sends the data to the Mux of the Channel Unit neighboring it. Looking at ***Figure 43***, if we were to designate each Channel Unit as Rows 0 through 4 we can see that during the stride operation, new data enters the Row 4 FIFO. The data currently in the Row 4 FIFO is redirected and written into the Row 3 FIFO. The data currently in the Row 3 FIFO is redirected and written into the Row 2 FIFO. The data currently in the Row 2 FIFO is redirected and written into the Row 1 FIFO. The data currently in the Row 1 FIFO is redirected and written into the Row 0 FIFO. Lastly, the data currently in the Row 0 FIFO is read out by its Router and allowed to be lost since that data is no longer needed.

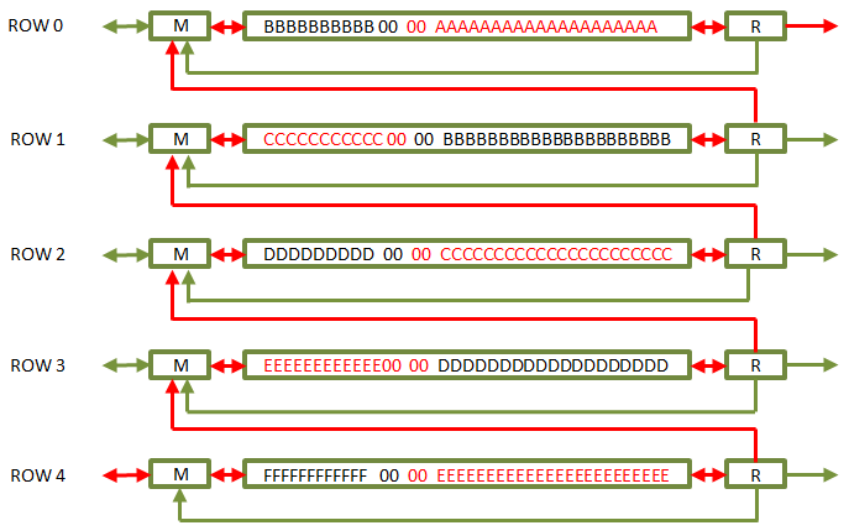

Figure 43: *Vertical Stride Operation.*
Shows how a vertical stride operation reads out data from one FIFO and writes it to the neighboring FIFO.

After all the stride rows have been read in for each channel being processed, the Routers resume reading out data to the DSP48 Floating Point multipliers. This stride and convolve pattern is repeated until all the rows of the input image have been read into the FIFOs.

For the Affine Layer operation, the logic executes in much the same manner as with the Convolution operation. The only key difference is that there is no striding across the image with a weight filter kernel. The Affine Layer operation uses a unique weight for each input data pixel.

6.6.3 Volume and Weight Muxes

The Volume and Weight Mux blocks, as described earlier essentially mux between the various input data streams to the FIFOs of the Channel Units. Depending on the operation the controller will specify an enable signal to active the new data stream, previous data stream, or recycled data stream. The new data stream is used for new data entering the Channel Unit FIFO. The previous data stream is used during the stride operation to allow for the neighboring Router to write its FIFOs content to this FIFO. The recycled data stream is used during the convolution operation while the Volume Data Router is reading out input volume data. As shown in ***Figure 44*** and ***Figure 45***, the structure is a straight forward mux with three select lines. The inputs are the i_new_data, i_prev_data, and i_recycled_data streams with all other input possibilities tied to ground. The flip flop illustrates the clocked nature of this process. Adding the flip flop after the mux helps in closing timing later in the design process.

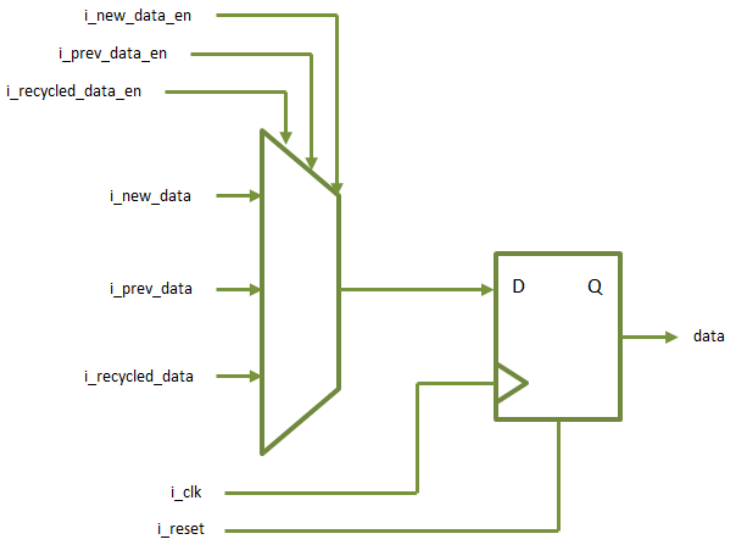

Figure 44: *Shows the design of the Volume and Weight Mux blocks to select the data stream.*

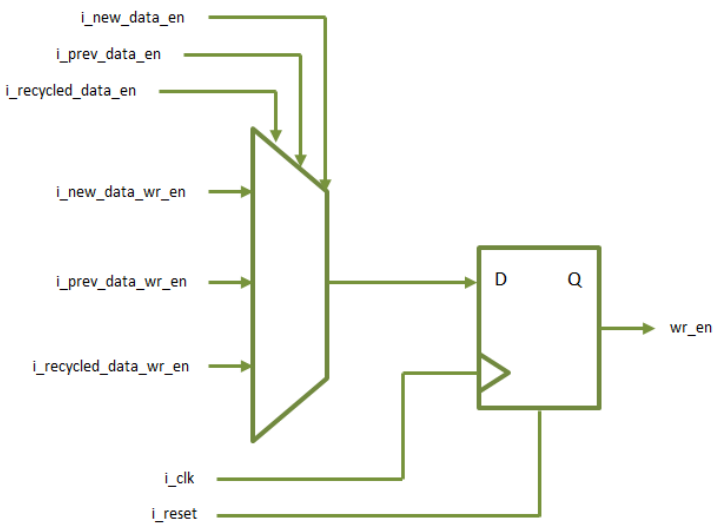

Figure 45: *Shows the design of the Volume and Weight Mux blocks to select the data stream enable signals.*

6.6.4 Volume Router Submodule

As described earlier, in each Channel Unit the Volume Router will read out the data from the Volume FIFO. During operation it reads out input data as wide as the weight filter kernel width. For instance, if the weight filter kernel has a size of 3x3, then the volume router will only read out 3 pieces of input volume data from the Volume FIFO. According to the stride setting from the Convolution Parameters register, the Volume Router will read out the next pieces of data to perform the Horizontal stride required by the Convolution operation. Throughout the process of striding and convolving data, the Router always checks to see if the downstream logic is ready to receive data. The state machine for the submodule is shown in ***Figure 46*** and the states are described in text for simplicity.

IDLE: The FSM checks various control signals coming from the Controller in order to determine which mode it will operate in. If given an enable signal to begin the vertical stride, it will move to the Snake Fill state. If given a signal from the Volume FIFO is not empty and has data, then the FSM will go to the Fill state. If given the enable signal to empty the Volume FIFO it will go to the Top Row Empty state.

FILL: This state will read out the same number of input volume data pixels as dictated by the weight filter kernel width. The data is stored in a register array for use in the Convolution and Affine operations.

SEND: Once the data array is filled according to the weight filter kernel width size, the input volume data is sent out to the DSP48 Floating Point Multipliers.

ACC_HOLD: After sending the data to the Multipliers the finite state machine waits until the Accumulator Logic indicates it is ready to receive more data. For the Affine operation the cycle repeats by going back to the Idle state. For the Convolution Operation a horizontal stride is required and the FSM therefore moves to the Shift state.

SHIFT: This state moves all the input volume data down in the register array down by the number of strides indicated in the Convolution Parameters Register. The input volume data shifted out of the array is lost since it no longer needed.

SNAKE_FILL: If the Controller indicates that the vertical stride is to be performed, this state reads out the contents of the Volume FIFO and sends it back to the Controller for distribution to the correct Volume Mux.

TOP ROW EMPTY: Once the vertical stride operation has completed, the old data in the top row needs to be read out since it is no longer needed. This state reads out one row of data from the Volume FIFO.

EMPTY RESET: This state informs the Controller that the row empty operation has completed, and the Volume Router is ready for the next operation.

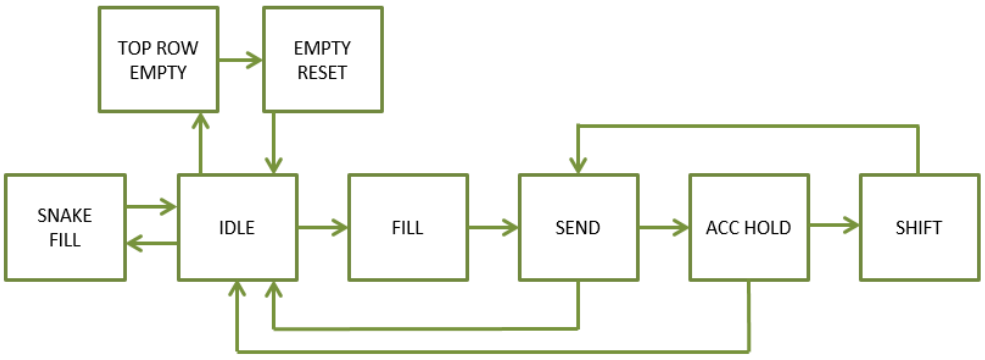

Figure 46: *Finite State Machine for the Volume Router.*

6.6.5 Weight Router Submodule

As described earlier, the Weight Router reads in weight filter kernel data from the Weight FIFO as wide as the Weight Parameters Register dictates. This data is also written into a register array within the Weight Router. Once the data from the weight filter kernel is read into the register array, the FSM signals to the Controller and Volume Router that the filter is loaded and ready. At this point the Convolution or Affine operation begins. After the Convolution or Affine operation has completed for all rows in the Channel Units, the next filter kernel is read out from the Weight FIFOs and used to execute the next round of operations. The state machine for the submodule is shown in ***Figure 47*** and the states are described in text for simplicity.

IDLE: The FSM checks various control signals coming from the Controller which will kick off the rest of the state machine operation.

PRIMING: This state holds the FSM meanwhile data begins to be written into the Weight FIFOs. Once there exists data in the Weight FIFOs the state machine will move on to load the weight filter kernel into the data register array.

LOAD_KERNEL: This state reads in weight filter kernel data from the Weight FIFO as wide as the Weight Parameters Register dictates. Once the data from the weight filter kernel is read into the register array, the FSM signals to the Controller and Volume Router that the filter is loaded and ready.

PROCESSING: As the Convolution or Affine operation executes, the weight filter kernel data is sent to the DSP48 Multipliers. This continues until the entire row of input volume data has been processes with the weight filter kernel. This cycle continues for all weight filter kernels.

EMPTY_WEIGHTS: When the Convolution or Affine operations have completed for the current weight filters and current rows, the Weights FIFOs are emptied in order to clear them before writing in the next set of weight filter kernel data.

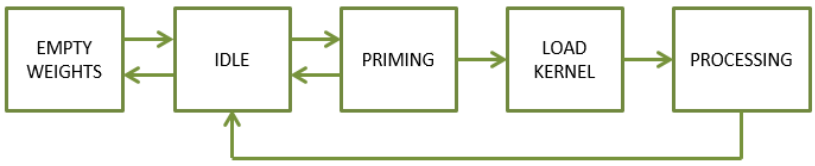

Figure 47: *Shows the design of the Volume and Weight Mux blocks.*

6.6.6 Controller Sub-module

For all the operations described thus far, the Controller unit has been the logic orchestrating the entire data flow. There is such a large amount of data to keep track of, it seemed to be a good idea to have one central controller to direct the operations of all the smaller submodules already discussed. Just like the AXI Master module, the Controller finite state machine is massive with many states. Describing every single signal and state in a traditional ASM chart manner would be far too tedious, would burden the reader's attention, and would be better described succinctly as a general flow of states with a text brief of what each state does. As shown in ***Figure 48*** and ***Figure 49***, the finite state machine for the controller can take one of two major paths. One path sets the Controller to perform Convolution while the other sets the Controller to

perform the Affine Fully Connected operation. The state machine which performs Convolution is shown in ***Figure 48*** and the states are described in text for simplicity below.

IDLE: The FSM checks for a start command from the Control Register as well as the affine enable signal to determine if the layer will operate as a Convolution Layer or as an Affine Fully Connected Layer.

IS_NET_READY: This state will hold the FSM until it receives signals from all the Channel Units that they are ready to process data. Once all the Channel Units are ready, the state machine moves on.

CALC_PARAMETERS: This state determines if there will be more channel iterations necessary to completely process the data. As previously described, the availability of the DSP blocks in hardware put a limit as to how many input channels can be processed at any given time. If the input data volume contains more channels than there are available DSPs, then this state calculates how man iterations are required. The size of the input image with padding is also determined here.

FETCH_AND_LOAD_WEIGHTS: This state reads in weight filter kernel data from the input buffer and distributes it into the Weight FIFOs of each of the Channel Units.

PAD_VOLUME_TOP: If the contents of the Convolution Parameters Register indicates that the input data volume should be padded and none of the input volume rows have been processed yet, the FSM will begin to zero fill Volume FIFOs of the first rows. After the Volume FIFOs for the number of rows indicated by the pad field of the Convolution Parameters Register have been zero filled, the state machine moves on.

PAD_VOLUME_LEFT: If the contents of the Convolution Parameters Register indicates that the input data volume should be padded, padding is added to every Volume FIFO before the actual image data is written.

FETCH_VOLUME: During this state, the data contained in the input buffer FIFO is distributed to the Volume FIFOs across all the Channel Units. If a 0-padding value is given in the Convolution Parameters Register, the state machine would bypass the padding states.

PAD_VOLUME_RIGHT: If the contents of the Convolution Parameters Register indicates that the input data volume should be padded, padding is added to every Volume FIFO after the actual image data is written.

WAIT_ONE_PAD: This state holds the state machine for one clock cycle to allow the data to finish being written into the Volume FIFOs of the Channel Units.

CONVOLUTION: This state holds the state machine until the Convolution operation is completed on the current rows of data loaded in the Channel Units. Once the Controller receives a signal from the downstream logic that the rows have been processed, the Controller FSM moves onto the Stride operations.

PVL_SINGLE, FV_SINGLE, WAIT_ONE_PREPVR_SINGLE, PVR_SINGLE, WAIT_ONE_SINGLE, PAD_SINGLE: These states perform the same function as the PAD VOLUME TOP, PAD VOLUME LEFT, FETCH VOLUME, PAD VOLUME RIGHT, and WAIT ONE PAD states. The only difference is that in these new states, only one row of new data is loaded into the Channel Units versus several. These states are separate from the other states since after each chain of single row states is executed, the vertical stride operation is performed by the SNAKE_FILL and EMPTY_TOP_ROW states.

SNAKE_FILL: After each single row is fetched in the states PVL_SINGLE, FV_SINGLE, WAIT_ONE_PREPVR_SINGLE, PVR_SINGLE, WAIT_ONE_SINGLE, and PAD_SINGLE, the vertical stride operation is allowed to occur one row at a time. This operation was covered earlier in ***Figure 43***.

EMPTY_TOP_ROW: After the vertical stride operation occurs, the last operation to complete the vertical stride is to read out the old data in the top channel unit Volume FIFO. This data is no longer needed and is discarded by the Volume Router downstream. This operation was covered earlier in ***Figure 43***.

The state machine as shown in ***Figure 49*** for the Affine Fully Connected Layer operation runs much like the state machine for the Convolution Layer operation. The operation still requires input volume data, weight filter kernel data, and bias data. The operation does not need, however, to stride across the input volume with the weight filter kernel. The nature of the Affine Fully Connected Layer is such that every item of input volume data has a unique weight associated with it. Therefore, all the needed data is read into the Channel Units just as in the Convolution data. However, since the number of filters is greater than that of the Convolution layer, this requires the weights to be loaded a chunk at a time rather than at one time. For instance, if the largest filter count for a Convolutional Layer is around 256 or 384, the Affine Layer can have filter counts of 4096. This design will feed in only the number of filters dictated by the Affine Parameters Register and execute the Affine operation on those filters. Once those are complete, another chunk of filter data will be read into the Channel Units and processed. This cycle will continue until all filters have been processed for the current input volume data loaded into the Volume FIFOs. With the rest of the state machine being the same as with the Convolution FSM, the only different states are detailed.

EMPTY_WEIGHTS: After the Affine Fully Connected operation completes, this state will empty the Weight FIFOs in the Channel Units. This is to clear the Weight FIFOs for the next chunk of filter data. This state does not empty the Volume FIFOs since they are still be used for processing.

EMPTY_VOL_WEIGHTS: After the Affine Fully Connected operation completes for all filters, this state will empty the Weight FIFOs and Volume FIFOs both in the Channel Units. This is to clear all the Channel Unit FIFOs for the next set of filter and input volume data.

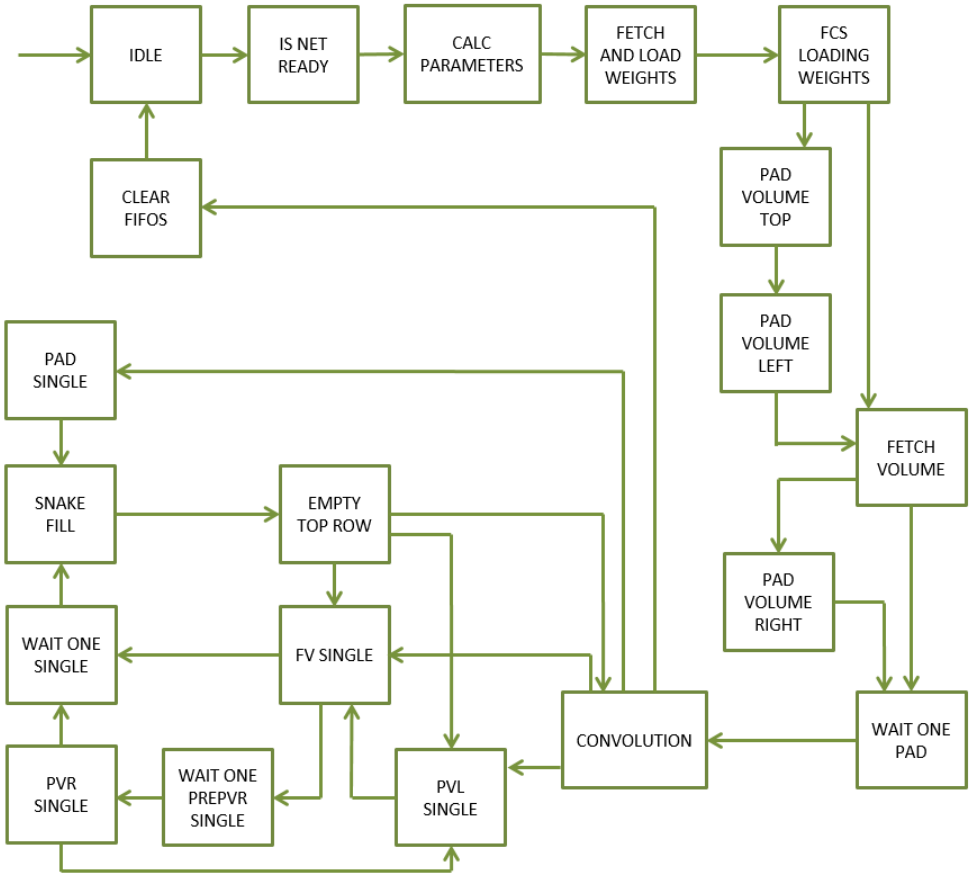

Figure 48: *Finite State Machine for the Convolution Layer Operation*

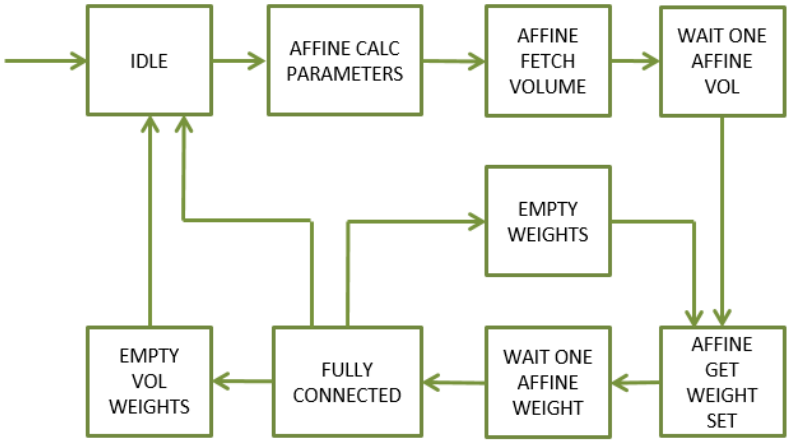

Figure 49: *Finite State Machine for the Affine Layer Operation*

6.6.7 Accumulator Logic

The inner workings of the Accumulator Logic group are shown below in ***Figure 50***. As the DSP48 Multipliers begin to output valid product data and their accompanying valid signals, the DSP Accumulator uses an adder tree to sum the results of the Convolution or Affine operation together. The results of the adder tree are summed together to arrive at the Kernel Sum. This sum is written to the Accumulator FIFO. The Accumulator Router reads the Accumulator FIFO data and uses a DSP48 Adder to perform a Floating-Point Single Precision Addition between the Accumulator FIFO data and the Bias data or the Previous data. As described earlier, the design is only able to process a limited number of input volume channels at a time. This requires the design to process groups of channel s at a time creating an iterative processing scheme. Therefore, the first iteration will add the Bias Data and subsequent iterations will add the Previous Data together with the data from the Accumulator FIFO. The Previous Data refers to the previous results to the Convolution or Affine operation.

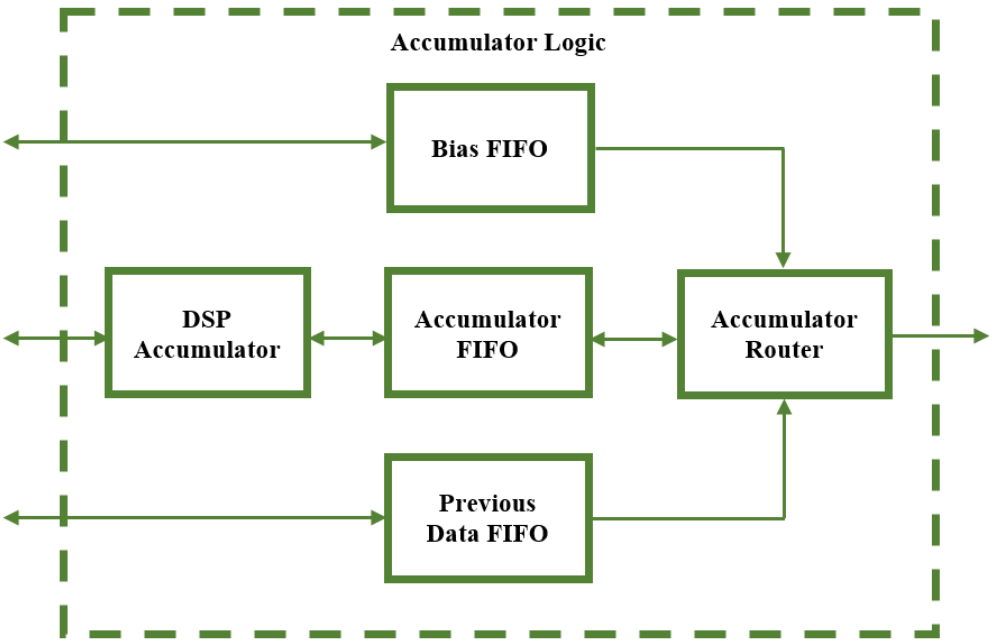

Figure 50: *Accumulator Logic Submodule Architecture*

6.6.8 DSP Accumulator Sub-module

As previously stated, the DSP Accumulator uses an adder tree to sum the results of the Convolution or Affine operation together. The adder tree can be seen in ***Figure 52***. Since the upstream logic in the Channel Units moves across the Input Volume row by row, the product results received by the DSP Accumulator will be the size of the Weight Filter Kernel by as many channels being processed. Therefore, the result of the adder tree will be the summation of the columns as shown in ***Figure 51***. As we see in the figure if we have input volume data **X** with 3 channels and Weight Filter Kernel **W** with a size 5x5, the result of the Adder Tree is 5 wide and is the summation of all the product data down the columns.

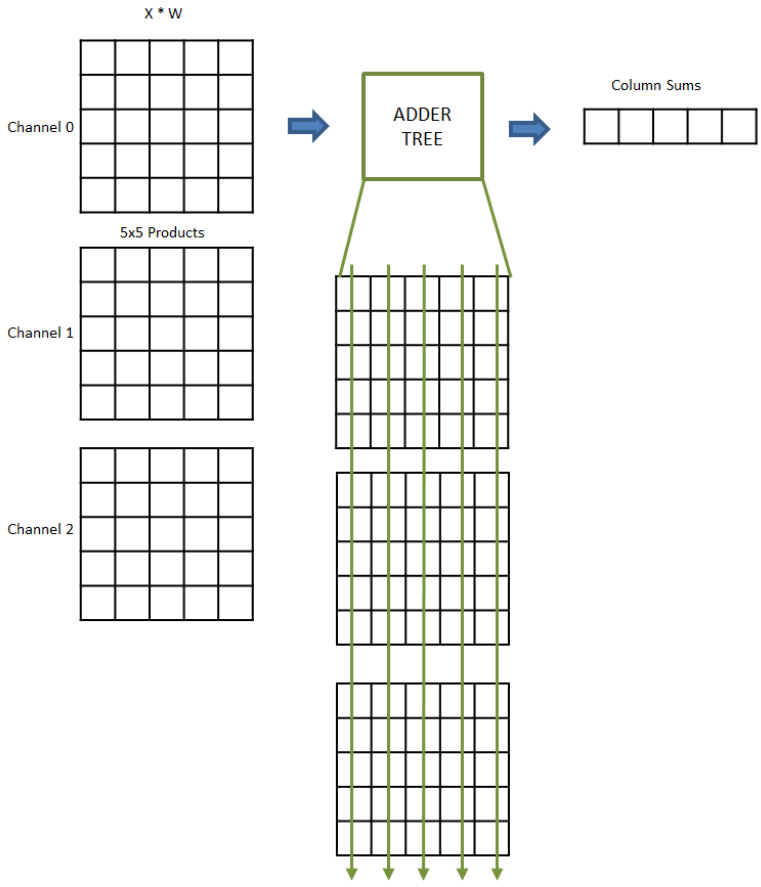

Figure 51: *Adder Tree Column Sums.*
Illustrates how the Adder tree sums the product down the columns of all channel data to produce a column sum.

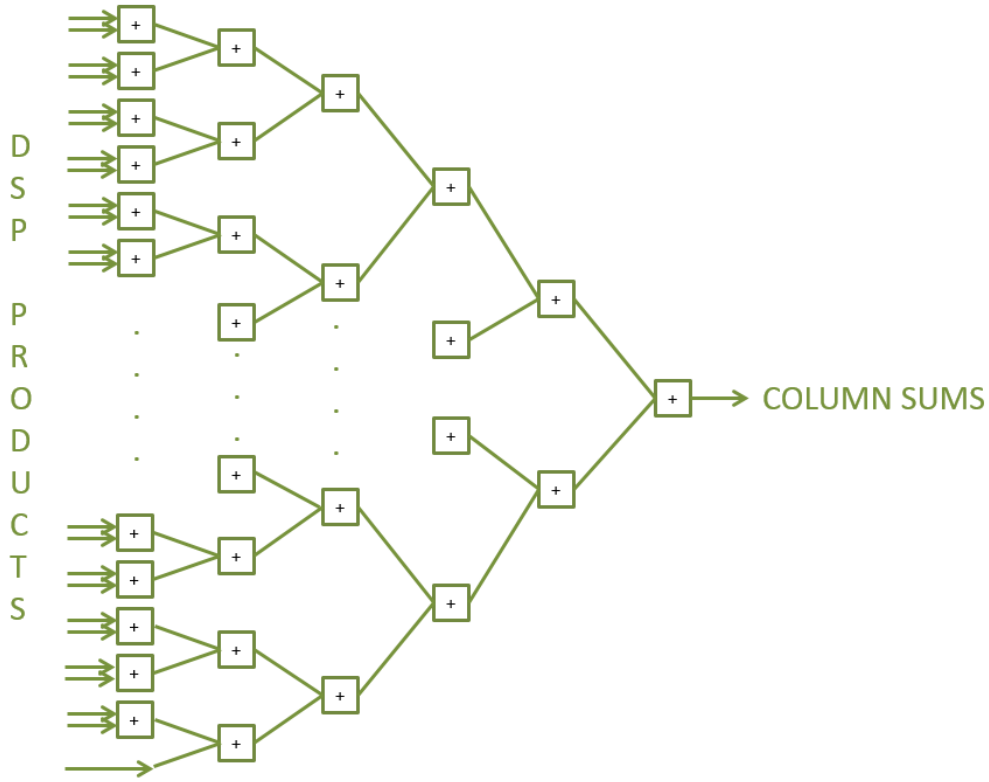

Figure 52: *Shows the adder tree employed to sum the product data.*

The column sum data is saved in a register array to be used in the next operation. It is important to note that the implementation of the adder tree registers the results of each adder layer. This was done in order to pipeline the adder tree and help with timing closure later. Not using the pipelined registers would cause there to be very long combinational paths in the design and would make it very hard for the Place and Route tool to establish FPGA logic routes which would meet the 100 MHz timing requirement.

Once the column sum data is obtained, the column sums are again summed by a smaller adder tree to obtain the single Kernel Sum data value. This is shown in ***Figure 53*** where we see that register array data being used as inputs to the small adder tree to obtain the single Kernel Sum. This process repeats as the Channel Units produce data for the rest of the filters in the Weight set.

The process of putting the products from the Channel Units through adder trees and registers to obtain the Kernel Sum requires the design to keep track of the valid signal as it propagates through the design. Therefore, a shift register is employed which is sized to accommodate the predetermined clock cycle delay between input to the adder tree and output. The input to the shift register is the product valid signal.

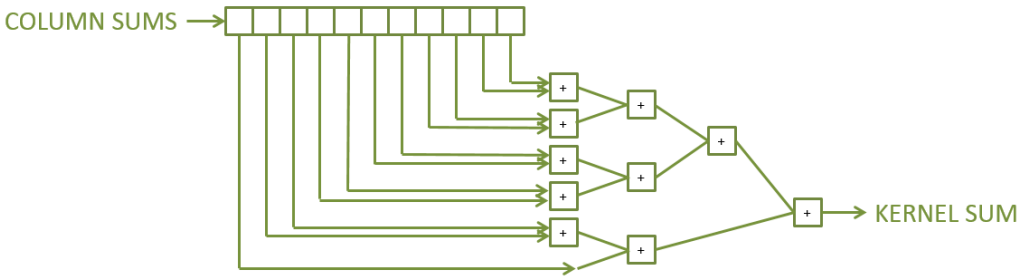

Figure 53: *Shows the adder tree employed to sum the column data.*

The state machine shown in ***Figure 54*** for the DSP Accumulator is very straight forward since the DSP48 adders do all of the heavy lifting. Each state is briefed below.

IDLE: The FSM checks for the product valid signals in order to kick off the accumulator. Once product valid signals are received, the state machine moves on.

RUN: This state allows the column adder tree to continue adding the product data being input from the Channel Units. Once the valid signal in the data valid shift register is detected after the delay of the adders has lapsed, the state machine registers the resulting column sum value into a register array. This register array is then used to feed the second smaller adder tree to obtain the kernel sum.

QUE: This state is precautionary and is used when the Accumulator FIFO is full and cannot accept new Kernel Sum values. This state places the data in a very shallow buffer and signals to the upstream logic that the DSP Accumulator is no longer read to receive new data. This lack of readiness informs the Routers in the Channel Units to continue to hold until the Accumulator is ready again.

EMPTY_QUE: If the Accumulator FIFO is no longer full and can accept data, this state first reads out the contents of its small buffer before signaling to the Channel Unit Routers that the DSP Accumulator is ready to receive new data.

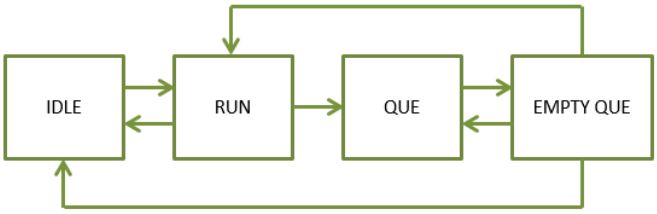

Figure 54: *Shows the Finite State Machine for the DSP Accumulator logic.*

6.6.9 Accumulator Router Sub-module

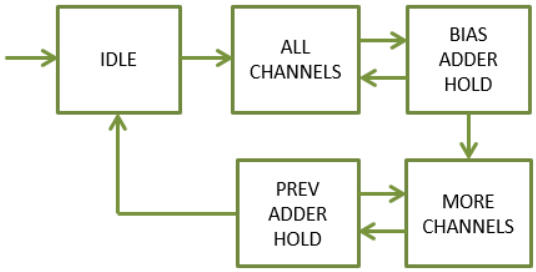

Figure 55: *Shows the Finite State Machine for the Accumulator Relay.*

As described earlier, the design is only able to process a limited number of input volume channels at a time. This requires the design to process groups of channels at a time creating an iterative processing scheme. Therefore, the first iteration will add the Bias Data and subsequent iterations will add the Previous Data together with the data from the Accumulator FIFO. The Previous Data refers to the previous results to the Convolution or Affine operation.

The state machine shown in ***Figure 55*** for the Accumulator is also very straight forward. Each state is briefed below.

IDLE: The FSM checks for the existence of data in the Accumulator FIFO and the Bias FIFO. If there exists data in the Accumulator FIFO and Bias FIFO, the state machine will move on.

ALL_CHANNELS: This state reads out one piece of data from each of the Bias and Accumulator FIFOs and passes these values to the DSP48 Adder block. Once this occurs, the state machine moves on to hold and wait for the Adder block results.

BIAS_ADDER_HOLD: This state holds the state machine until the Adder has had time to fully process the Floating-Point Single Precision addition operation. This state also keeps track of which row, filter, and pixel it is processing by maintaining counters. These counters are important since the Accumulator Relay is the last logic block in the whole Convolution or Affine Operation and signals to the AXI Master and Controller units when it has completed.

MORE_CHANNELS, PREV_ADDER_HOLD: These states perform much of the same functions as the first two, except for in this case they add the Previous Data from the last Convolution / Affine execution to the current Kernel Sum from the Accumulator FIFO.

7.0 RELU LAYER

Implementing the ReLu Layer is actually very simple. This is because the Rectified Linear Unit only checks to see if a value is less than zero or greater than or equal to zero. Therefore, any value coming into the ReLu unit will be forced to zero if it is negative and passed if it is not. This operation can be placed upstream of the Convolutional Layer's output buffer with each value being written into the output buffer checked and modified as necessary. Of course, this option of using a ReLu unit is not utilized in the Fully Connected or Affine layer and therefore needs to be deactivated once the Convolution Layer is performing this function. Deactivation of this layer is simply done by an AXI register write from the Microblaze Processor.

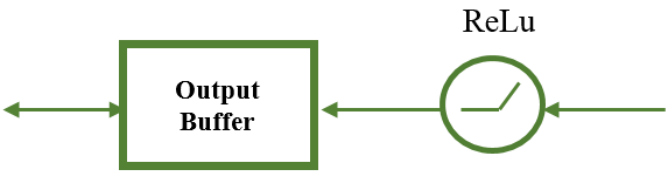

Figure 56: *ReLu Implementation.*
ReLu Layer is value check before output buffer of Convolution/Affine Layer

8.0 MAX POOLING LAYER

8.1 Algorithm

In order to detail the Max Pooling operation let's take a specific example. Let's say we have an image of size 13x13x256, a filter kernel with size 3x3, and a stride of 2. The Max Pooling operation involves sliding a 3x3 kernel across 3 rows by 3 columns at a time as shown in ***Figure 57***. The stride involves moving the kernel across the image in increments of 2 pixels in both vertical and horizontal directions as shown in ***Figure 57*** and ***Figure 58***. At each increment the kernel considers all the values in the 3x3 neighborhood and chooses the greatest value as shown in ***Figure 59***. This value is passed by the kernel to be the output pixel value for the scaled down Max Pooled image. This process is repeated across the height, width, and channels of the input image to generate a full Max Pooled image. Given an input image of 13x13x356, the output is a Max Pooled image of size 6x6x256

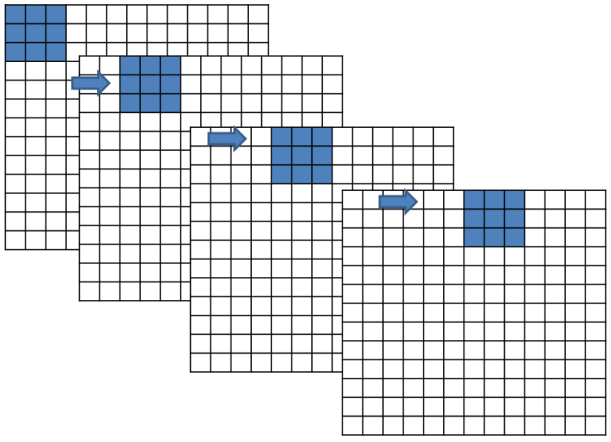

Figure 57: *Horizontal stride across input image.*

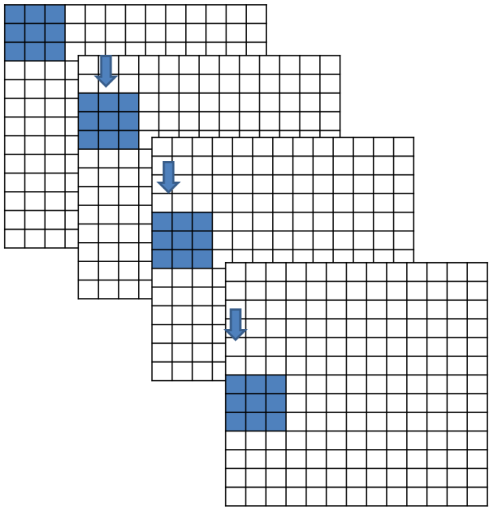

Figure 58: *Vertical stride across input image.*

0.74	0.23	0.56
0.03	0.11	0.23
0.52	0.63	0.85

Figure 59: *Max Pooling operation.*
Max Pooling selects the maximum value of the 3x3 or 2x2 neighborhood

8.2 Max Pooling Layer - Architecture

The Max Pooling Layer Module architecture is shown below in ***Figure 60***. The module utilizes an AXI4-Lite Slave Interface which provides a command and status interface between the MicroBlaze processor and the internal logic of the module. The module also utilizes an AXI4 Master Interface which reads and writes data to and from the DDR memory on the FPGA board. The AXI Master Interface retrieves input data from specified DDR memory locations for the Max Pooling operation then outputs the Max Pooled result to a specified region of memory.

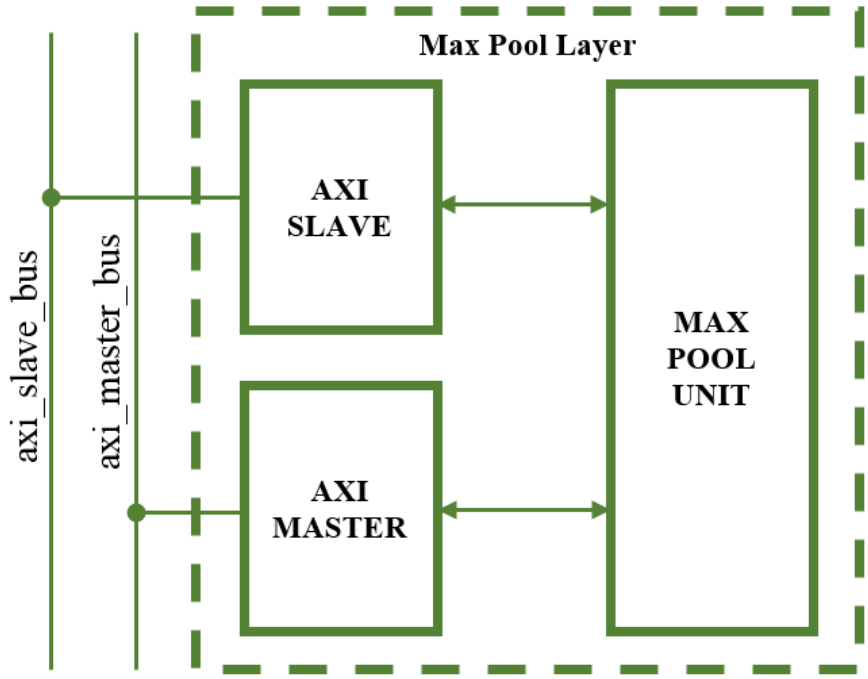

Figure 60: *Top level Architecture of the Max Pooling Layer*

8.3 Max Pooling Layer - Register Set

The AXI Slave provides the following register set for the Max Pooling Layer.

Table 44: *Register List for the Max Pooling Layer Design*

Register Name	Offset Address
Control Register	0x0
Status Register	0x4
Input Data Address Register	0x8
Output Data Address Register	0xC
Input Volume Parameters Register	0x10
Output Volume Parameters Register	0x14
Kernel Parameters Register	0x18
Address 1 Parameters Register	0x1C
Address 2 Parameters Register	0x20
Debug Register	0x24

8.3.1 Control Register

This register allows for controlling the Max Pool Layer using a few essential signals.

Table 45: *Control Register Bit Map*

Control Register				**(Base Address + 0x020)**			
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
Unassigned							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
Unassigned							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
Unassigned							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
Unassigned							**start**

Table 46: *Control Register Description*

Bit Field	Name	Initial Value	Description
31:1	Unassigned	0x0	Register bits for unassigned in current stage of development
0	start	'0'	Kicks off the Max Pooler AXI Master State Machine which begins reading in volume data.

8.3.2 Status Register

The status register breaks out important logic signals which may be used for debugging the design once hardware testing has begun.

Table 47: *Status Register Bit Map*

Status Register				**(Base Address + 0x004)**			
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
unassigned							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
unassigned		**Inbuff almost full**	**Inbuff full**	unassigned		**Inbuff almost empty**	**Inbuff empty**
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
unassigned		**Outbuff almost full**	**Outbuff full**	unassigned		**Outbuff almost empty**	**Outbuff empty**
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
unassigned			**done**	unassigned			**busy**

Table 48: *Status Register Description*

Bit Field	Name	Initial Value	Description
31:22	Unassigned	0x0	Register bits for unassigned in current stage of development
21	Inbuff_almost_full	'0'	Indicates the input buffer is almost full and is about to fill up
20	Inbuff_full	'0'	Indicates the input buffer is full and will not accept more data
19:18	Unassigned	0x0	Register bits for unassigned in current stage of development
17	Inbuff_almost_empty	'0'	Indicates the input buffer contains only one piece of valid data
16	Inbuff_empty	'0'	Indicates the input buffer is empty with no valid data.
15:14	Unassigned	0x0	Register bits for unassigned in current stage of development
13	Outbuff_almost_full	'0'	Indicates the output buffer is almost full and is about to fill up
12	Outbuff_full	'0'	Indicates the output buffer is full and will not accept more data
11:10	Unassigned	0x0	Register bits for unassigned in current stage of development
9	Outbuff_almost_empty	'0'	Indicates the output buffer contains only one piece of valid data
8	outbuff_empty	'0'	Indicates the output buffer is empty with no valid data.
7:5	Unassigned	0x0	Register bits for unassigned in current stage of development
4	done	'0'	Indicates that the Max Pool has completed the current operation
3:1	Unassigned	0x0	Register bits for unassigned in current stage of development
0	busy	'0'	Indicates the Max Pool is currently busy executing the current operation

8.3.3 Input Data Address Register

The value contained in this register points the location in the DDR memory were the layer input data begins.

Table 49: *Input Data Address Register Bit Map*

Input Data Address Register				(Base Address + 0x008)			
31	30	29	28	27	26	25	24
Input_data_addr							
23	22	21	20	19	18	17	16
Input_data_addr							
15	14	13	12	11	10	9	8
Input_data_addr							
7	6	5	4	3	2	1	0
Input_data_addr							

Table 50: *Input Data Address Register Description*

Bit Field	Name	Initial Value	Description
31:0	Input_data_address	0x0	Address where Max Pooler will pull data from

8.3.4 Output Data Address Register

The value contained in this register points the location in the DDR memory were the layer will begin outputting the resulting data from the Max Pool operation.

Table 51: *Output Data Address Register Bit Map*

Output Data Address Register				(Base Address + 0x00C)			
31	30	29	28	27	26	25	24
Output_data_addr							
23	22	21	20	19	18	17	16
Output_data_addr							
15	14	13	12	11	10	9	8
Output_data_addr							
7	6	5	4	3	2	1	0
Output_data_addr							

Table 52: *Output Data Address Register Description*

Bit Field	Name	Initial Value	Description
31:0	Output_data_address	0x0	Address where Max Pooler will save the output of the max pooling operation

8.3.5 Input Parameters Register

This register contains information on the size of the input image such as Height, Width, and number of channels.

Table 53: *Input Parameters Register Bit Map*

Input Parameters Register				(Base Address + 0x010)			
31	30	29	28	27	26	25	24
Input_height[31:24]							
23	22	21	20	19	18	17	16
Input_width[23:16]							
15	14	13	12	11	10	9	8
Input_channels[15:8]							
7	6	5	4	3	2	1	0
Input_channels[7:0]							

Table 54: *Input Parameters Register Description*

Bit Field	Name	Initial Value	Description
31:24	Input_height	0x0	Value which specifies the height of the input volume
23:16	Input_width	0x0	Value which specifies the width of the input volume
15:0	Input_channels	0x0	Value which specifies the number of channels that need to be processed

8.3.6 Output Parameters Register

This register contains information on the size of the output image such as Height, Width, and number of channels.

Table 55: *Output Parameters Register Bit Map*

Output Parameters Register				(Base Address + 0x014)			
31	30	29	28	27	26	25	24
Output_height[31:24]							
23	22	21	20	19	18	17	16
Output_width[23:16]							
15	14	13	12	11	10	9	8
Output_channels[15:8]							
7	6	5	4	3	2	1	0
Output_channels[7:0]							

Table 56: *Output Parameters Register Description*

Bit Field	Name	Initial Value	Description
31:24	Output_height	0x0	Value which specifies the height of the output volume
23:16	Output_width	0x0	Value which specifies the width of the output volume
15:0	Output_channels	0x0	Value which specifies the number of channels that will be produced by the max pooling operation

8.3.7 Kernel Parameters Register

This register contains information on the size of the Max Pooling kernel such as Height, Width, number of channels, and number of filters.

Table 57: *Kernel Parameters Register Bit Map*

Kernel Parameters Register				(Base Address + 0x018)			
31	30	29	28	27	26	25	24
kernel_height[31:24]							
23	22	21	20	19	18	17	16
kernel_width[23:16]							
15	14	13	12	11	10	9	8
Kernel_stride[15:8]							
7	6	5	4	3	2	1	0
unassigned[7:0]							

Table 58: *Kernel Parameters Register Description*

Bit Field	Name	Initial Value	Description
31:24	kernel_height	0x0	Value which specifies the height of the filter weight kernel
23:16	kernel_width	0x0	Value which specifies the width of the filter weight kernel
15:8	kernel_stride	0x0	Value which specifies the stride to be used when performing the max pooling operation

8.4 Max Pooling Layer - AXI Master

While the AXI Slave interface receives commands via the AXI Slave Bus, the AXI Master is responsible for reading into the design all relevant data from memory and then writing the resulting volume of data back to memory for the next layer in the Neural Network to use. The AXI Master logic is shown in ***Figure 61*** with states specifically designed to retrieve the input data from memory regions specified by the AXI Slave Register settings. This module starts the Max Pooling operations and collects the results.

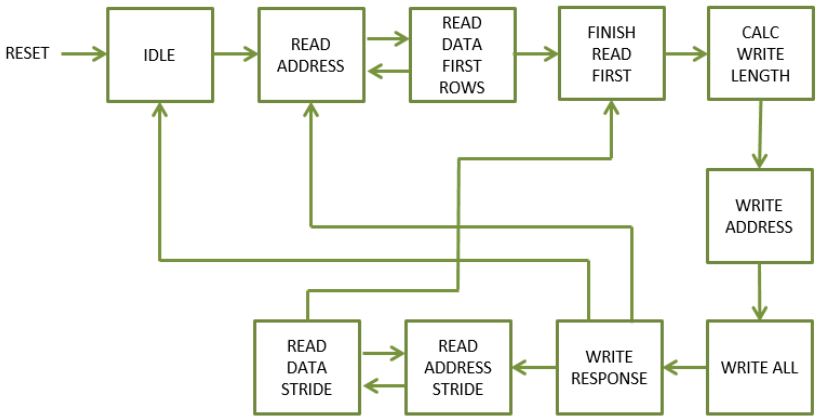

Figure 61: *AXI Master FSM.*
Finite State Machine for the Max Pooling Layer AXI Master Module.

The state machine is shown in ***Figure 61*** and the states are described in text for simplicity below.

IDLE: The FSM checks for a start command from the Control Register as well as the signal indicating that the Max Pool operation has already completed.

READ_ADDRESS, READ_DATA_FIRST_ROWS: These states perform AXI Read transfers to read in rows of input image data. Since the Max Pool Kernel is only 2x2 or 3x3 the kernel size in Kernel Parameters Register dictates how many rows are read into the Input Buffer. Once the first few rows have been read into the design, the state machine moves on.

FINISH_READ_FIRST: This state holds the state machine until the max pooling operation has completed for the rows currently loaded into the design. Once the signal which indicates that the rows have been processed is received, the state machine moves on.

CALC_WRITE_LENGTH, WRITE_ADDRESS, WRITE_ALL, WRITE_RESPONSE: These states calculate the appropriate size of AXI write transfers to perform based on the Output Parameters configuration register in the AXI Slave interface. The state machine reads out all the data from the previous Max Pooling operation and writes the data to the address specified in the Output Address configuration register. This state is the ultimately the last state in the whole state machine design. The state machine will continue to process data if all the rows for the input image data have not yet been processed.

8.5 Max Pooling Layer – Design and Ports

A look inside the Max Pool Unit is shown in ***Figure 62***. The unit is comprised of five major functional blocks. The input buffer is a FIFO buffer which receives data from the AXI Master and reads it out to the Row Controller. The Row Controller sends the input data to the FIFO Network so that correct pixels are being used in the Max Pool operation. Once the FIFO Network is fully primed, the max pooling filter kernel is applied across the input image. This is achieved by using the Heap Sorter submodule to obtain the max value in the 3x3 or 2x2 neighborhood. The resulting max value is written out to the Output FIFO buffer. The same processed is repeated until the max pooling kernel has covered all columns and rows of the input data image.

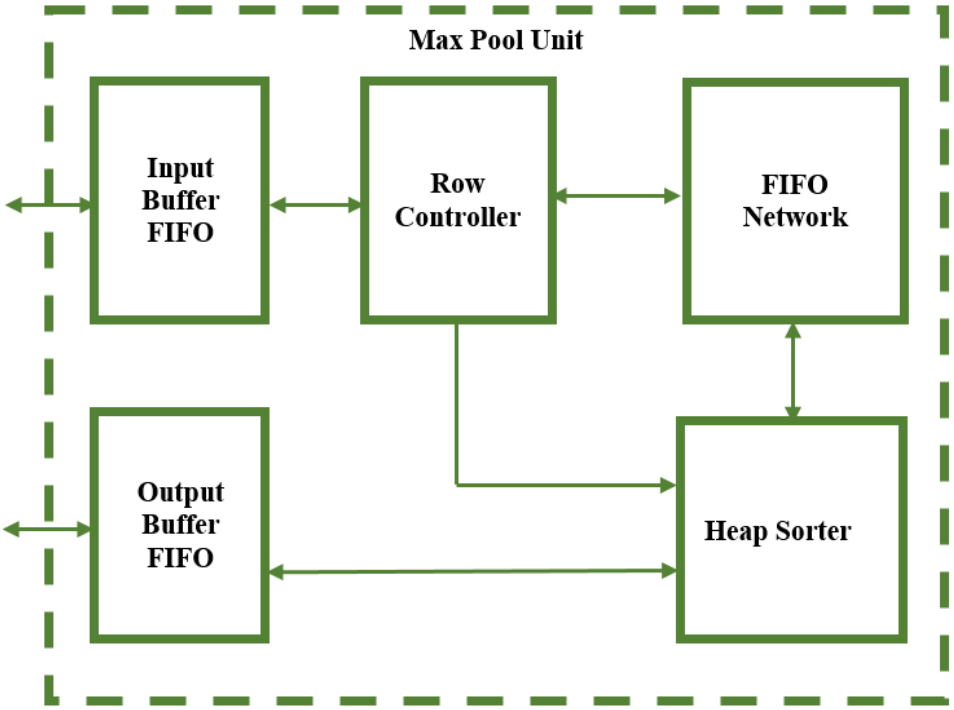

Figure 62: *Architecture of the Max Pool Unit*

The ports for this submodule are shown in ***Table 59*** below.

Table 59: *Max Pooling Unit Port Descriptions*

PORT NAME	TYPE	DIR.	DESCRIPTION
i_clk	std_logic	In	Clock signal to for sequential logic
i_reset_n	std_logic	In	Active Low Reset
i_input_volume_size	std_logic_vector (7 downto 0)	In	Value specifying the size of the input volume
i_output_volume_size	std_logic_vector (7 downto 0)	In	Value specifying the size of the output volume after max pooling
i_pool_kernel_size	std_logic_vector (3 downto 0)	In	Value specifying what size of kernel to apply to the volume data
i_stride	std_logic_vector (3 downto 0)	In	Value specifying how many pixels to skip when selecting center pixel
i_outbuff_rd_en	std_logic	In	Read Enable of the Output buffer
i_inbuff_din	std_logic_vector (15 downto 0)	In	Volume Data to write to the input buffer
i_inbuff_wr_en	std_logic	In	Write Enable of the Input Buffer

o_inbuff_empty	std_logic	Out	Indicates Input buffer is empty with no valid data
o_inbuff_almost_empty	std_logic	Out	Indicates input buffer is almost empty with one more valid piece of data
o_inbuff_full	std_logic	Out	Indicates input buffer is full
o_inbuff_almost_full	std_logic	Out	Indicates input buffer is about to fill up
o_outbuff_dout	std_logic_vector (15 downto 0)	Out	Output buffer Data out to AXI Master
o_outbuff_empty	std_logic	Out	Indicates output buffer is empty with no valid data
o_outbuff_almost_empty	std_logic	Out	Indicates output buffer is almost empty with one more valid piece of data
o_outbuff_full	std_logic	Out	Indicates output buffer is full
o_outbuff_almost_full	std_logic	Out	Indicates output buffer is about to fill up
o_outbuff_valid	std_logic	Out	Indicates output buffer contains valid data
o_channel_complete	std_logic	Out	Indicates that the Max Pooling Operation on the current volume has completed.

8.6 Max Pooling Layer - Submodule Definitions and Operations

The following section will describe the design of each of the submodules involved in the design of the Max Pooling Layer Unit.

8.6.1 Row Controller and FIFO Network

The Row Controller receives input data from the AXI Master via the Input Buffer as shown in ***Figure 63***. The AXI Master will read the input image information from the DDR Memory in rows. Looking at ***Figure 63***, let's take an example case of a 13x13x256 input image with a Max Pool Kernel of size 3x3 and stride of 2. With this example case in mind, three rows of input image information are read by the AXI Master and passed down to the Row Controller. Once the Row Controller is signaled that the AXI Master has loaded information into the Input Buffer FIFO, the Controller will load the three rows of image data into the Row FIFOs. After the three rows of data are loaded, the Row Controller will send one 3x3 kernels worth of data to the Heap Sorter as shown in ***Figure 67***. The Row Controller reads out 3 pixels from the Row FIFOs each, for a 3x3 kernel to be sent to the Heap Sorter. It is important to note that the data read out of each Row FIFO is also written back into the same Row FIFO, so the row data can be reused in the striding process. The three Row FIFOs constitute the FIFO Network in this design. Once the Heap Sorter has determined the Maximum Value from the 3x3 Kernel, that value is written to the Output Buffer.

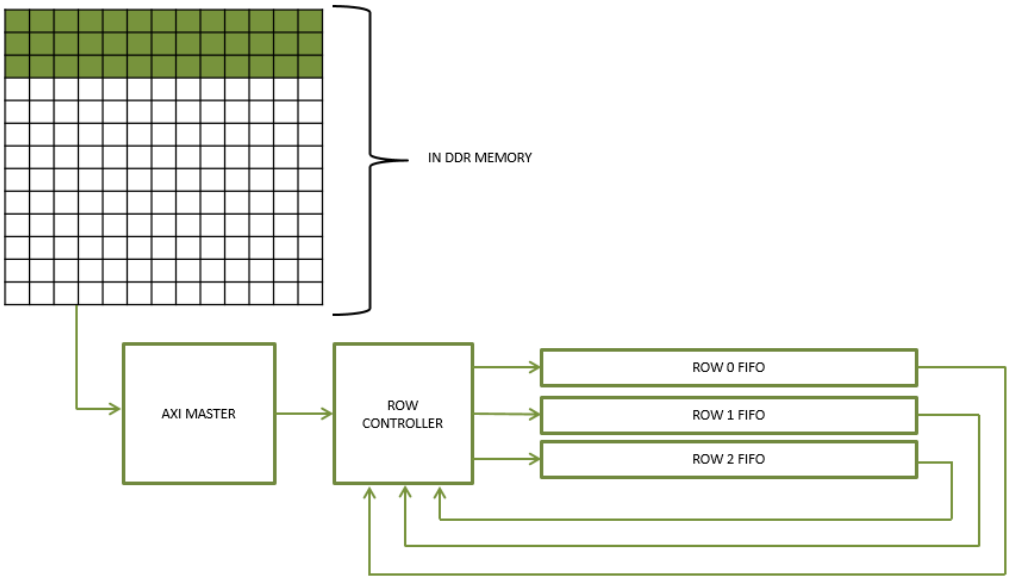

Figure 63: *Row data loaded into FIFO Network*

This pattern is repeated each time the Kernel is moved down the input image by 2 pixels. The kernel is shifted down the image both in the horizontal direction and the vertical direction.

The state machine for the submodule is shown in ***Figure 64*** and the states are described in text for simplicity.

IDLE: The FSM checks for a start signal from the Control Register before moving on. This state also checks for the existence of data in the input buffer as well as the signal indicating that a channel was completed. Once data is available in the input buffer or a channel was completed, the state machine will move on.

PRIME ROW 0, P01 WAIT, PRIME ROW 1, P12 WAIT, PRIME ROW 2, P2S WAIT: These states read out the input image row data in the input buffer and write the rows to the Row FIFOs in the FIFO Network. If the kernel is of dimension 2x2, only two FIFOs will be loaded with row data instead of three. Therefore, in a 2x2 kernel case, the state machine will bypass the PRIME ROW 2 state entirely and move on with the sequence.

PRIME SORTER: This state reads out a piece of row data from the FIFO Network until the data read out is as wide as the Max Pool Kernel. For instance, in a 2x2 kernel case, this state will read out twice from each Row FIFO. As the data is being read out of each Row FIFO, the same data is also being written back into their respective FIFO to be used later during striding.

COLUME STRIDE SHIFT: After the Heap Sorter has sorted out the previous kernel of data it had been sent, this state strides across the rows loaded in the Row FIFOs. This is done by reading and discarding n-1 pieces of information each where n is the stride value. After the striding has completed for the rows currently loaded in the Row FIFOs, the state machine will move on to load the next three rows of input data.

ROW STRIDE SHIFT: If all the input image rows of data have yet to be processed through the Max Pooling Layer, this state will perform a vertical stride down the image. This is accomplished by the Row Controller connecting the output of the Row 0 FIFO to the input of the Row 1 FIFO and connecting the output of the Row 1 FIFO to the input of the Row 2 FIFO. The input of the Row 0 FIFO receives the data from the input buffer and the old data in the Row 2 FIFO is read out and allowed to be lost. One row of data is moved at a time until the correct number rows have been shifted according to the stride setting in the Kernel Parameters Register.

CLEAR_COUNTERS: This state performs the function in the state name. The state clears all counters and output control signals in preparation of the next state.

CHANNEL CLEAR: Once all the rows of the input image have been processed for a channel, the Row FIFOs are read out until empty. This allows the next round of data to be loaded into fresh Row FIFOs that do not have stale data.

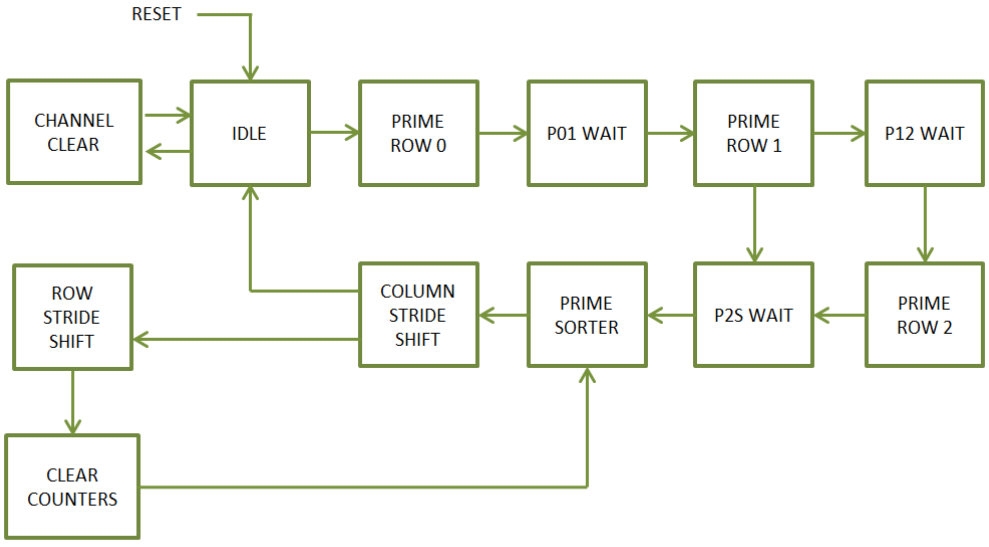

Figure 64: *Finite State Machine for the Max Pool Layer Row Controller*

8.6.2 Heap Sorter Sub-module

In order to explain the Heap Sorter module first let's look at the Heapsort algorithm. The (binary) heap data structure is an array object that we can view as a nearly complete binary tree as shown in ***Figure 65***. Each node of the tree corresponds to an element of the array. The tree is filled on all levels except possibly the lowest, which is filled from the left up to a point. (Cormen et.al, 2009)

There are two kinds of binary heaps: max-heaps and min-heaps. In both kinds, the values in the nodes satisfy a heap property, the specifics of which depend on the kind of heap. In a max-heap, the max-heap property is that for every node other than the root a parent nodes value is greater than the child nodes value. Thus, the largest element in a max-heap is stored at the root, and the subtree rooted at a node contains values no longer than that contained at the node itself. A min-heap is organized in the opposite way; the min-heap property is that for every node other than the root, a parent nodes value is less than the child nodes value. The smallest element in a min-heap is at the root. (Cormen et.al, 2009)

As shown in ***Figure 65***, a max-heap can be viewed as a binary tree and an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. (Cormen et.al, 2009)

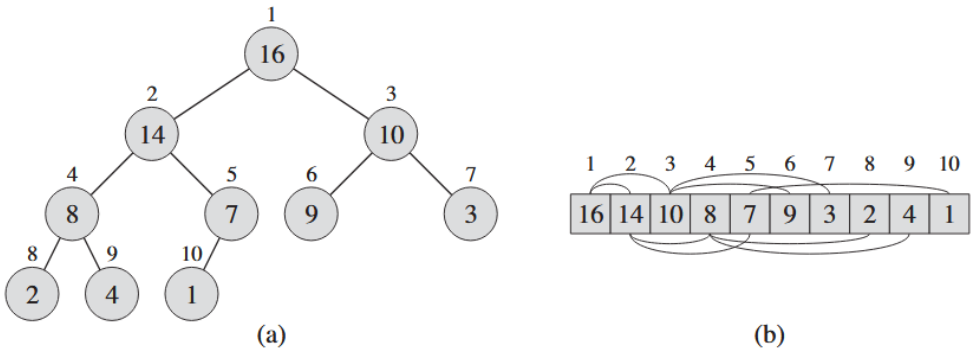

Figure 65: *A max-hep viewed as (a) a binary tree and (b) an array. (Cormen et.al, 2009)*

In order to maintain the max-heap property, we call the procedure Max-Heapify. This procedure lets the value in a parent node "float down" in the max-heap so that the subtrees obey the max-heap property. (Cormen et.al, 2009)

In order to fully understand the action of Max-Heapify let's look at an example shown in ***Figure 66***. In Max-Heapify at each step the largest of the elements between parent or right and left children is determined. If the root node is the largest, then the subtree is already a max-heap and the action is complete. Otherwise, one of the two children have the largest element, and is swapped with the parent node. This may cause one of the subtrees to violate the max-heap property. Consequently, we perform the Max-Heapify action recursively on that subtree. As shown in ***Figure 66***, the initial configuration at node 2 violates the max-heap property since it is not larger than both children. The max-heap property is restored for node 2 in ***Figure 66***(b) by exchanging the value at node 2 with the value at node 4. This destroys the max-heap property for node 4. The recursive execution of Max-Heapify swaps the value at node 4 with the value at node 9. As shown in ***Figure 66***(c) node 4 is fixed up, and the recursive execution of Max-Heapify yields no further change to the data structure.

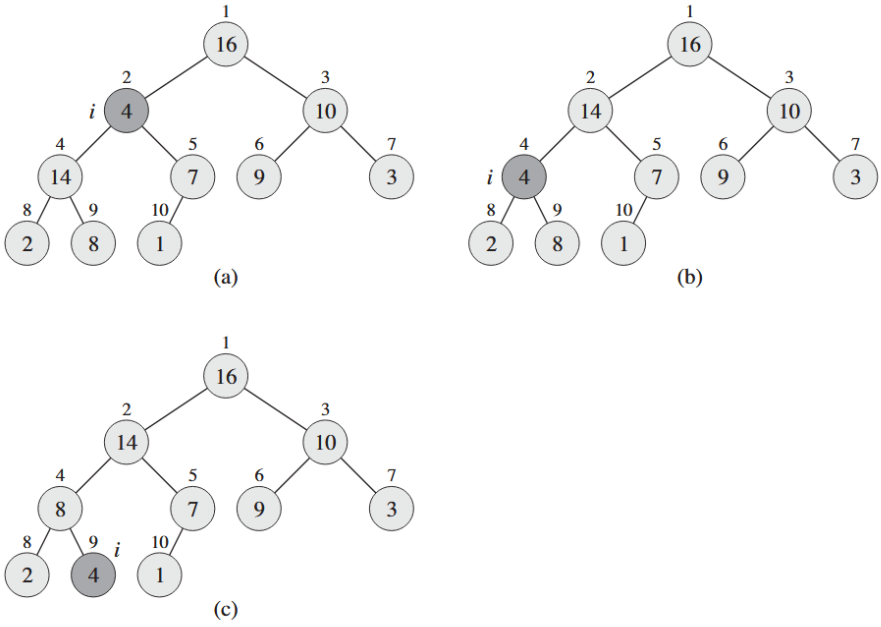

Figure 66: *Max-Heapify in action.*
(Cormen et.al, 2009)

Now having explained the Heapsort Algorithm, we can now look at how it was implemented in this design. After the Row Controller has loaded the image row data into the Row FIFOs, the Row Controller then sends one kernels worth of image data to the Heap Sorter Submodule as illustrated in ***Figure 67***. Once the data is handed over to the Heap Sorter Submodule, the maximum value of that 3x3 or 2x2 neighborhood is found and written to the Output Buffer.

The state machine for the submodule is shown in ***Figure 68*** and it performs the Max-Heapify action as described above. The state machine executes recursively on the kernel data and swaps any node value that violates the max-heap.

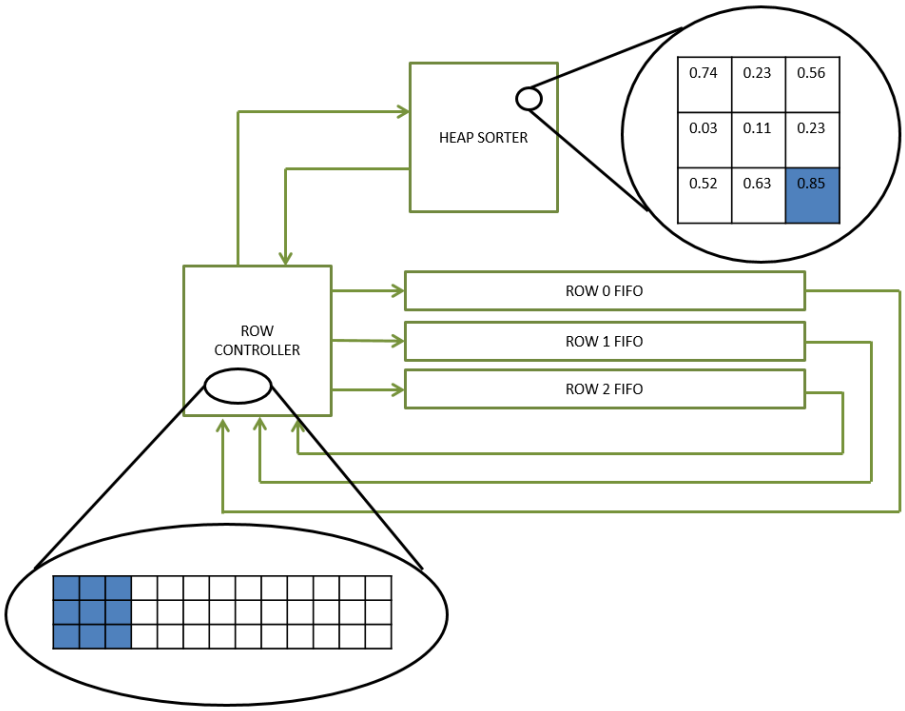

Figure 67: *Loaded row information is processes through the Heap Sorter*

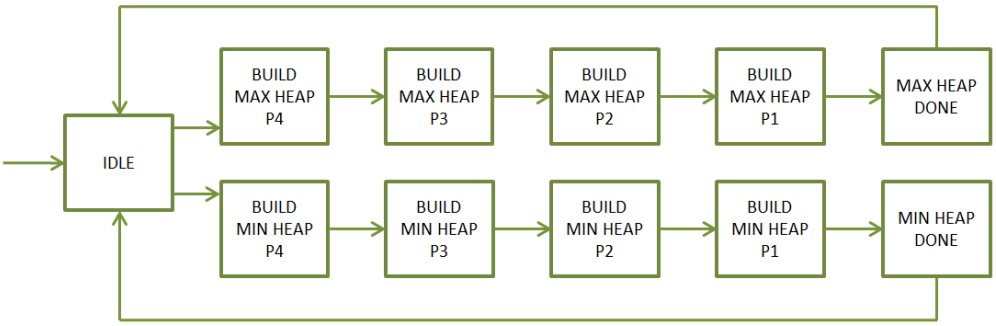

Figure 68: *Finite State Machine for the Heap Sorter*

9.0 SOFTMAX LAYER

9.1 Algorithm

As seen in Section 3.0, a Convolutional Neural network is configured to arrive at several class scores and thereby classify the image it is given. These scores are then passed through a loss or cost function and used to arrive at a loss value as well as the probabilities. This is to measure how well the neural network has classified the image it was given. Therefore, the loss will be high if we are doing a poor job of classifying the image and low if we are doing well. (Li, F., et. al, 2017)

A popular classifier, which will compute the loss, is the Softmax Classifier. The Softmax classifier outputs the normalized class probabilities. (Li, F., et. al, 2017) The Softmax Function is:

$$L_i = -\log\left(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}\right) \qquad P_i = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

Where we are using the notation f_j to mean the j-th element of the vector of class scores f. The Softmax Function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one. (Li, F., et. al, 2017)

Due to mathematical complexity in the FPGA implementation, the Probabilities are output by the Softmax Layer. In order to implement the Softmax Function and arrive at the Probability, each of the mathematical operations must be performed.

9.2 Softmax Layer - Architecture

The Softmax Layer Module architecture is shown below in ***Figure 69***. The module utilizes an AXI4-Lite Slave Interface which provides a command and status interface between the MicroBlaze processor and the internal logic of the module. The module also utilizes an AXI4 Master Interface which reads and writes data to and from the DDR memory on the FPGA board. The AXI Master Interface retrieves input data from specified DDR memory locations for the Softmax operation then outputs the result to a specified region of memory.

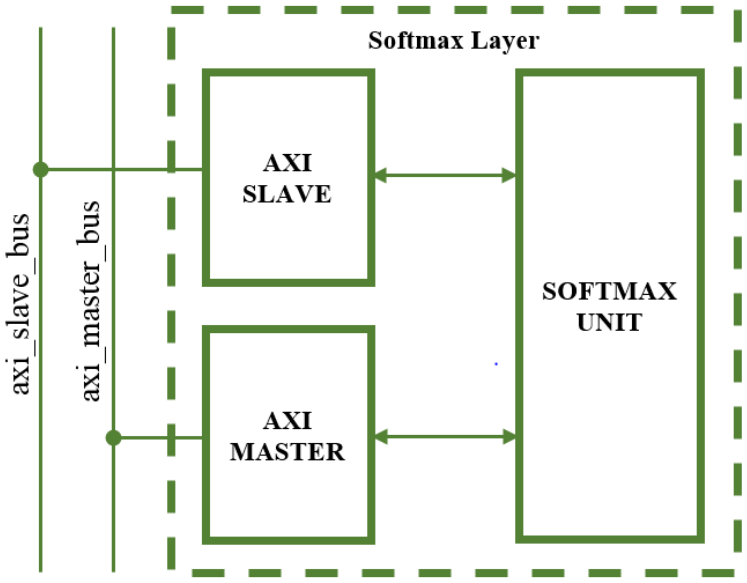

Figure 69: *Top level Architecture of the Softmax Layer*

9.3 Softmax Layer - Register Set

The AXI Slave provides the following register set for the Softmax Layer.

Table 60: *Register List for the Softmax Layer Design*

Register Name	Offset Address
Control Register	0x0
Status Register	0x4
Input Data Address Register	0x8
Output Data Address Register	0xC
Probability 0 Register	0x10
Probability 1 Register	0x14
Probability 2 Register	0x18
Probability 3 Register	0x1C
Probability 4 Register	0x20
Debug Register	0x24

9.3.1 Control Register

This register allows for controlling the Softmax Layer using a few essential signals.

Table 61: *Control Register Bit Map*

Control Register				(Base Address + 0x024)			
31	30	29	28	27	26	25	24
Unassigned							
23	22	21	20	19	18	17	16
Unassigned							
15	14	13	12	11	10	9	8
Unassigned							
7	6	5	4	3	2	1	0
Unassigned							**start**

Table 62: *Control Register Description*

Bit Field	Name	Initial Value	Description
31:1	Unassigned	0x0	Register bits for unassigned in current stage of development
0	start	'0'	Kicks off the Softmax Classifier AXI Master State Machine which begins reading in data.

9.3.2 Status Register

The status register breaks out important logic signals which may be used for debugging the design once hardware testing has begun.

Table 63: *Status Register Bit Map*

Status Register				(Base Address + 0x004)			
31	30	29	28	27	26	25	24
unassigned		**Expbuff almost full**	**Expbuff full**	unassigned		**Expbuff almost empty**	**Expbuff empty**
23	22	21	20	19	18	17	16
unassigned		**Inbuff almost full**	**Inbuff full**	unassigned		**Inbuff almost empty**	**Inbuff empty**
15	14	13	12	11	10	9	8
unassigned		**Outbuff almost full**	**Outbuff full**	unassigned		**Outbuff almost empty**	**Outbuff empty**
7	6	5	4	3	2	1	0
unassigned			**done**	unassigned			**busy**

Table 64: *Status Register Description*

Bit Field	Name	Initial Value	Description
31:30	Unassigned	0x0	Register bits for unassigned in current stage of development
29	expbuff_almost_full	'0'	Indicates the input buffer is almost full and is about to fill up
28	expbuff_full	'0'	Indicates the input buffer is full and will not accept more data
27:26	Unassigned	0x0	Register bits for unassigned in current stage of development
25	expbuff_almost_empty	'0'	Indicates the input buffer contains only one piece of valid data
24	expbuff_empty	'0'	Indicates the input buffer is empty with no valid data.
23:22	Unassigned	0x0	Register bits for unassigned in current stage of development
21	Inbuff_almost_full	'0'	Indicates the input buffer is almost full and is about to fill up
20	Inbuff_full	'0'	Indicates the input buffer is full and will not accept more data
19:18	Unassigned	0x0	Register bits for unassigned in current stage of development
17	Inbuff_almost_empty	'0'	Indicates the input buffer contains only one piece of valid data
16	Inbuff_empty	'0'	Indicates the input buffer is empty with no valid data.
15:14	Unassigned	0x0	Register bits for unassigned in current stage of development
13	Outbuff_almost_full	'0'	Indicates the output buffer is almost full and is about to fill up
12	Outbuff_full	'0'	Indicates the output buffer is full and will not accept more data
11:10	Unassigned	0x0	Register bits for unassigned in current stage of development
9	Outbuff_almost_empty	'0'	Indicates the output buffer contains only one piece of valid data
8	outbuff_empty	'0'	Indicates the output buffer is empty with no valid data.
7:5	Unassigned	0x0	Register bits for unassigned in current stage of development
4	done	'0'	Indicates that the Softmax classifier has completed the current operation
3:1	Unassigned	0x0	Register bits for unassigned in current stage of development
0	busy	'0'	Indicates the Softmax classifier is currently busy executing the current operation

9.3.3 Input Data Address Register

The value contained in this register points the location in the DDR memory were the layer input data begins.

Table 65: *Input Data Address Register Bit Map*

Input Data Address Register				(Base Address + 0x008)			
31	30	29	28	27	26	25	24
Input_data_addr							
23	22	21	20	19	18	17	16
Input_data_addr							
15	14	13	12	11	10	9	8
Input_data_addr							
7	6	5	4	3	2	1	0
Input_data_addr							

Table 66: *Input Data Address Register Description*

Bit Field	Name	Initial Value	Description
31:0	Input_data_address	0x0	Address where Softmax Layer will pull data from

9.3.4 Output Data Address Register

The value contained in this register points the location in the DDR memory were the layer will begin outputting the resulting data from the Softmax operation.

Table 67: *Output Data Address Register Bit Map*

Output Data Address Register				(Base Address + 0x00C)			
31	30	29	28	27	26	25	24
Output_data_addr							
23	22	21	20	19	18	17	16
Output_data_addr							
15	14	13	12	11	10	9	8
Output_data_addr							
7	6	5	4	3	2	1	0
Output_data_addr							

Table 68: *Output Data Address Register Description*

Bit Field	Name	Initial Value	Description
31:0	Output_data_address	0x0	Address where the Softmax Layer will save the output of the classification operation

9.3.5 Probability 1 Register

The register contains the highest probability as well its associated class after the softmax classifier has finished classifying the image.

Table 69: *Prediction 1 Register Bit Map*

Prediction 1 Register				(Base Address + 0x010)			
31	30	29	28	27	26	25	24
probability [31:24]							
23	22	21	20	19	18	17	16
probability [23:16]							
15	14	13	12	11	10	9	8
class [15:8]							
7	6	5	4	3	2	1	0
class [7:0]							

Table 70: *Prediction 1 Register Description*

Bit Field	Name	Initial Value	Description
31:16	Probability	0x0	Probability value calculated by the Softmax classifier
15:0	Class	0x0	Class number associated with probability

9.3.6 Probability 2 Register

The register contains the second highest probability as well its associated class after the softmax classifier has finished classifying the image.

Table 71: *Prediction 2 Register Bit Map*

Prediction 2 Register				(Base Address + 0x014)			
31	30	29	28	27	26	25	24
probability [31:24]							
23	22	21	20	19	18	17	16
probability [23:16]							
15	14	13	12	11	10	9	8
class [15:8]							
7	6	5	4	3	2	1	0
class [7:0]							

Table 72: *Prediction 2 Register Description*

Bit Field	Name	Initial Value	Description
31:16	Probability	0x0	Probability value calculated by the Softmax classifier
15:0	Class	0x0	Class number associated with probability

9.3.7 Probability 3 Register

The register contains the third highest probability as well its associated class after the softmax classifier has finished classifying the image.

Table 73: *Prediction 3 Register Bit Map*

Prediction 3 Register				**(Base Address + 0x018)**			
31	**30**	**29**	**28**	**27**	**26**	**25**	**24**
probability [31:24]							
23	**22**	**21**	**20**	**19**	**18**	**17**	**16**
probability [23:16]							
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**
class [15:8]							
7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
class [7:0]							

Table 74: *Prediction 3 Register Description*

Bit Field	Name	Initial Value	Description
31:16	Probability	0x0	Probability value calculated by the Softmax classifier
15:0	Class	0x0	Class number associated with probability

9.3.8 Probability 4 Register

The register contains the fourth highest probability as well its associated class after the softmax classifier has finished classifying the image.

Table 75: *Prediction 4 Register Bit Map*

Prediction 4 Register				(Base Address + 0x01C)			
31	30	29	28	27	26	25	24
probability [31:24]							
23	22	21	20	19	18	17	16
probability [23:16]							
15	14	13	12	11	10	9	8
class [15:8]							
7	6	5	4	3	2	1	0
class [7:0]							

Table 76: *Prediction 4 Register Description*

Bit Field	Name	Initial Value	Description
31:16	Probability	0x0	Probability value calculated by the Softmax classifier
15:0	Class	0x0	Class number associated with probability

9.3.9 Probability 5 Register

The register contains the fifth highest probability as well its associated class after the softmax classifier has finished classifying the image.

Table 77: *Prediction 5 Register Bit Map*

Prediction 5 Register				(Base Address + 0x020)			
31	30	29	28	27	26	25	24
probability [31:24]							
23	22	21	20	19	18	17	16
probability [23:16]							
15	14	13	12	11	10	9	8
class [15:8]							
7	6	5	4	3	2	1	0
class [7:0]							

Table 78: *Prediction 5 Register Description*

Bit Field	Name	Initial Value	Description
31:16	Probability	0x0	Probability value calculated by the Softmax classifier
15:0	Class	0x0	Class number associated with probability

9.4 Softmax Layer – AXI Master

While the AXI Slave interface receives commands via the AXI Slave Bus, the AXI Master is responsible for reading into the design the scores data from memory and then writing the resulting probability and class number to registers on the AXI Slave interface as well as to DDR Memory. The AXI Master logic is shown in ***Figure 70*** with states specifically designed to retrieve the scores data from memory regions specified by the AXI Slave Register settings. This module starts the Softmax Function operation and collects the results.

The state machine is shown in ***Figure 70*** and the states are described in text for simplicity below.

IDLE: The FSM checks for a start command from the Control Register before moving to start the Softmax Function operation.

CALC_READ_ADDRESS_LENGTH, READ_ADDRESS, READ_DATA: These states perform AXI Read transfers to read in the scores data from the neural network. The scores data is written into the Input Buffer. Once the scores for the classes have been read into the design, the state machine moves on.

SOFTMAX_EXECUTING: This state holds the state machine until the softmax function has completed executing.

CALC_WRITE_ADDRESS_LENGTH, WRITE_ADDRESS, WRITE_ALL, WRITE_RESPONSE: These states calculate the appropriate size of AXI write transfers to perform based on the number of classes specified in the Control register in the AXI Slave interface. Once data is detected in the Output Buffer, these states will read out the contents of the Output Buffer, sort the data from largest to smallest, and finally write the results to both memory and registers.

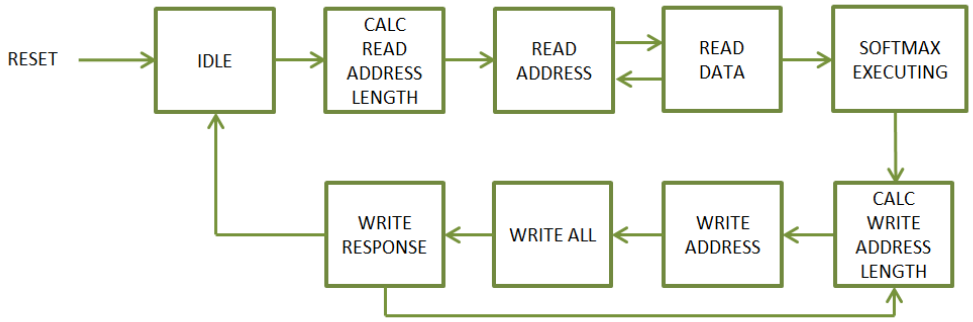

Figure 70: *Finite State Machine for the Softmax Layers AXI Master*

9.5 Softmax Layer – Design and Ports

A look inside the Softmax Unit is shown in ***Figure 71*** below. The unit is comprised of seven major functional blocks. The input buffer is a FIFO buffer which receives scores data from the AXI Master and reads it out to the Softmax Controller. The Softmax Controller sends the scores data to the Exponential Function logic so that the exponential of each of the scores is calculated. As the exponential of the scores is calculated, the results of exponentiation are written into the Exponential Buffer FIFO. Once the exponential of each score is calculated, the Softmax Controller signals the Softmax Adder Wrapper logic to begin reading out the Exponential Buffer data and summing it all together. As the summation is executing the data in the Exponential Buffer is written back into itself to be used during the Division operation. After the summation is complete, the Softmax Adder Wrapper signals to the Softmax Divider Wrapper to begin the division process. The results of the division process are sent back to the Softmax Controller where they are quickly sorted as they come in.

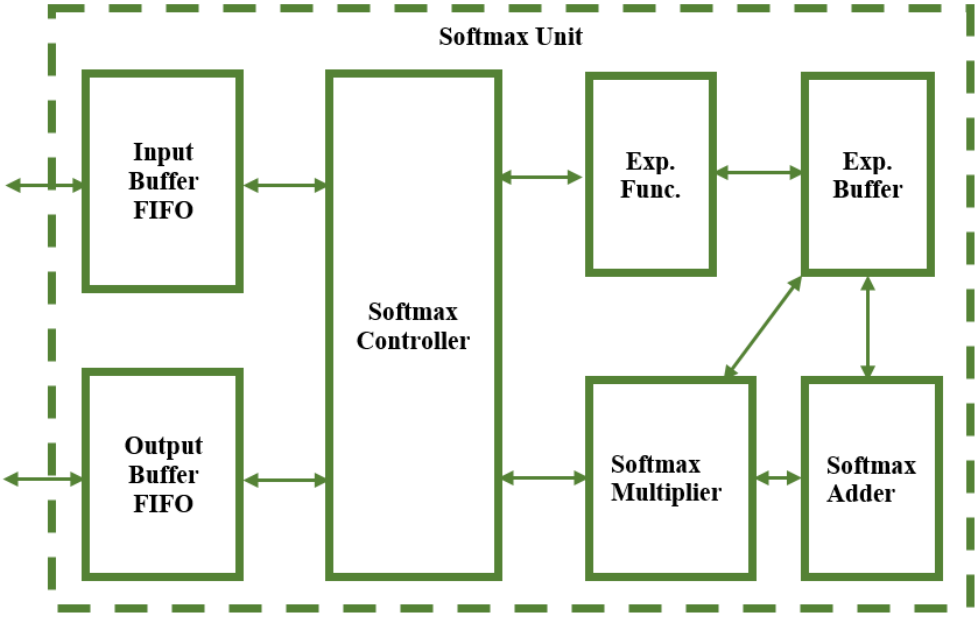

Figure 71: *Architecture of the Softmax Unit*

Table 79: *Softmax Unit Port Descriptions*

PORT NAME	TYPE	DIR.	DESCRIPTION
i_clk	std_logic	In	Clock signal to for sequential logic
i_reset_n	std_logic	In	Active Low Reset
i_num_elements	std_logic_vector (15 downto 0)	In	Value specifies the number of classes which will be used in classification
i_inbuff_din	std_logic_vector (15 downto 0)	In	Data to write to the input buffer
i_inbuff_wr_en	std_logic	In	Write Enable of the Input Buffer
i_outbuff_rd_en	std_logic	In	Read Enable of the Output buffer
o_inbuff_empty	std_logic	Out	Indicates Input buffer is empty with no valid data
o_inbuff_almost_empty	std_logic	Out	Indicates input buffer is almost empty with one more valid piece of data
o_inbuff_full	std_logic	Out	Indicates input buffer is full
o_inbuff_almost_full	std_logic	Out	Indicates input buffer is about to fill up
o_outbuff_dout	std_logic_vector (15 downto 0)	Out	Output buffer Data out to AXI Master
o_outbuff_empty	std_logic	Out	Indicates output buffer is empty with no valid data
o_outbuff_almost_empty	std_logic	Out	Indicates output buffer is

			almost empty with one more valid piece of data
o_outbuff_full	std_logic	Out	Indicates output buffer is full
o_outbuff_almost_full	std_logic	Out	Indicates output buffer is about to fill up
o_outbuff_valid	std_logic	Out	Indicates output buffer contains valid data
o_expbuff_empty	std_logic	Out	Indicates exponential buffer is empty with no valid data
o_expbuff_almost_empty	std_logic	Out	Indicates exponential buffer is almost empty with one more valid piece of data
o_expbuff_full	std_logic	Out	Indicates exponential buffer is full
o_expbuff_almost_full	std_logic	Out	Indicates exponential buffer is about to fill up
o_expbuff_valid	std_logic	Out	Indicates exponential buffer contains valid data
o_softmax_complete	std_logic	Out	Indicates that the Softmax classifier has completed.
o_busy	std_logic	Out	Indicates that the Softmax Layer is still busy executing

9.6 Softmax Layer - Submodule Definitions and Operations

The following section will describe the design of each of the submodules involved in the design of the Convolution / Affine Layer Unit.

9.6.1 Exponential Function

In order to implement the Softmax Function into the FPGA design, custom Exponential Function logic was developed. This logic performs the Maclaurin Series Taylor Expansion on the input data to calculate, through successive iterations, the exponential.(Weisstein, 2019) Being that the input data are scores and are positive, there is no need to consider the possibility that the input data may be a negative value. (Weisstein, 2019)

The Maclaurin Series Taylor Expansion of an exponential function is an expansion about 0,

$$f(x) = f(0) = f'(0)x + \frac{f''(0)}{2!}x^2 + \frac{f^{(3)}(0)}{3!}x^3 + \cdots + \frac{f^{(n)}(0)}{n!}x^n$$

Where n is the order up to which the expansion will calculate the answer. Refining the above equation and applying it to the exponential function we can rewrite the equation as. (Weisstein, 2019)

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$$

Therefore, to implement these mathematical functions, we must pre-calculate the factorials out to an order *n*. As a starting point for this design the order n chosen was *n=24*. Once the factorials are pre-calculated and their values preloaded into registers in the design, any input to the exponential function can be calculated with iterations of addition and multiplication. This design employs the use of the DSP48 Floating Point Single Precision Adder and Multiplier logic blocks. The pre-calculated factorials are shown in ***Table 80***.

The state machine for the submodule is shown in ***Figure 72*** and the states are described in text for simplicity.

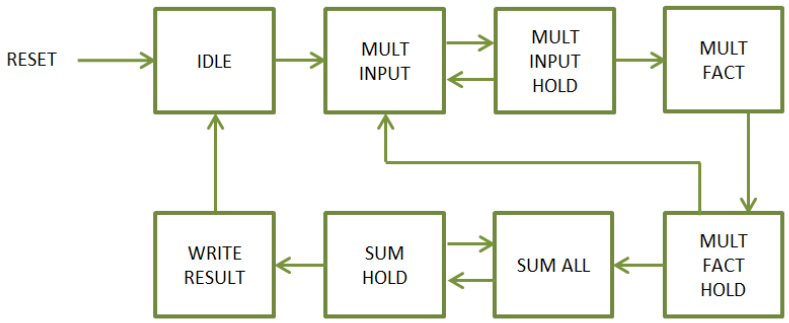

Figure 72: *Finite State Machine for the Exponential Function Logic*

IDLE: The FSM checks for a signal from the Softmax Controller that it is ready for the Exponential Function to begin processing data. This state also checks that it is receiving valid data from the Softmax Controller.

Table 80: *Factorials Pre-calculated for use in Design*

n	1/n! Decimal	1/n! Hexadecimal
0	1	3F800000
1	1	3F800000
2	0.5	3F000000
3	0.166667	3E2A0000
4	0.041667	3D2A0000
5	0.008333	3C080000
6	0.001389	3AB60000
7	0.000198	39500000
8	2.48E-05	37D00000
9	2.76E-06	36380000
10	2.76E-07	34930000
11	2.51E-08	32D70000
12	2.09E-09	310F0000
13	1.61E-10	2F300000
14	1.15E-11	2D490000
15	7.65E-13	2B570000
16	4.78E-14	29570000
17	2.81E-15	274A0000
18	1.56E-16	25340000
19	8.22E-18	23170000
20	4.11E-19	20F20000
21	1.96E-20	1EB80000
22	8.9E-22	1C860000
23	3.87E-23	1A3B0000

MULT_INPUT: This state loads the input data value and the current value in the Multiplication register to the DSP Multiplier. This is done to calculate the x^n where x is the input data value and n is the order.

MULT_HOLD: This state holds the state machine for the amount delay needed for the Floating-Point Multiplier to generate a valid output.

MULT_FACT: This state sends the product from the multiplier along with the pre-calculated factorial to the Floating-Point Multiplier. This is done to calculate the $\frac{x^n}{n!}$ value where x is the input data value and n is

the order.

MULT_FACT_HOLD: This state holds the state machine for the amount delay needed for the Floating-Point Multiplier to generate a valid output. The states MULT_INPUT, MULT_HOLD, MULT_FACT, and MULT_FACT_HOLD are repeated iteratively 24 times in order to calculate the full list of $\frac{x^n}{n!}$ values, where *n* is in the range 0 to 23. Each product is placed into an array for the later step of summing them all together.

SUM_ALL, SUM_HOLD: These states sum the entire contents of the array containing the $\frac{x^n}{n!}$ values as well as the 1 and *x* value individually in the equation.

WRITE_RESULT: After the Exponential Function has completed its execution, the result is written to the Exponential Buffer.

9.6.2 Softmax Adder Wrapper

Once the exponential of the scores is calculated by the Exponential Function and stored in the Exponential Buffer, the Softmax Controller signals the Softmax Adder Wrapper to begin the summation process. The summation employs the use of a DSP48 Floating Point Addition Logic block to perform the individual additions.

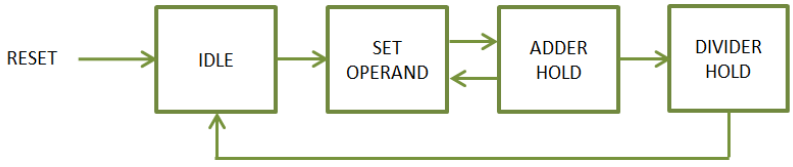

Figure 73: *Finite State Machine for the Softmax Adder Wrapper*

The state machine for the submodule is shown in ***Figure 73*** and the states are described in text for simplicity.

IDLE: The FSM checks for a signal from the Softmax Controller that the exponentiation process is complete for all scores. It also checks for the existence of data in the Exponential Buffer.

SET_OPERANDS: This state sets the addend and augend for the DSP Adder.

ADDER_HOLD: This state holds the state machine to account for the adder delay between valid input to valid output. The result is registered for use in the next summation iteration. Both this state and SET OPERANDS are executed iteratively until all the data in the Exponential Buffer has been summed together.

DIVIDER_HOLD: This state holds the state machine until the division operation has fully completed.

9.6.3 Softmax Divider Wrapper

Once the summation process has completed, the Softmax Adder Wrapper signals the Softmax Divider Wrapper to begin the division process. The division employs the use of a DSP48 Floating Point Divider Logic block to perform the individual divisions. This logic divides the data in the Exponential Buffer by the summation result found by the Softmax Adder Wrapper.

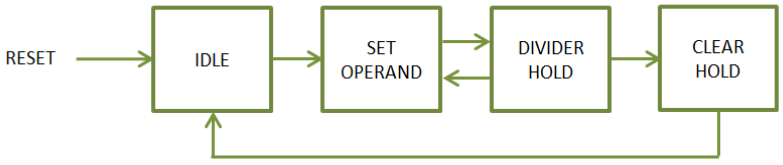

Figure 74: *Finite State Machine for the Softmax Divider Wrapper*

The state machine for the submodule is shown in ***Figure 74*** and the states are described in text for simplicity.

IDLE: The FSM checks for a signal from the Softmax Adder Wrapper indicating that the summation process has completed. This state also checks for the existence of data in the Exponential Buffer FIFO.

SET_OPERANDS: This state reads out one piece of data from the Exponential Buffer FIFO and sets the dividend and divisor for the DSP Adder.

DIVIDER_HOLD: This state holds the state machine to account for the divider delay between valid input data and valid output data.

CLEAR_HOLD: Once the division process completes, the state machine waits for a clear signal from the Softmax Controller in order clear all counters and signals.

9.6.4 Softmax Controller

For all the operations described thus far, the Softmax Controller has been the logic orchestrating the entire data flow. With several logic blocks accessing one FIFO buffer in the design, it seemed to be a better design to have one logic block control which submodule gets access to the buffer.

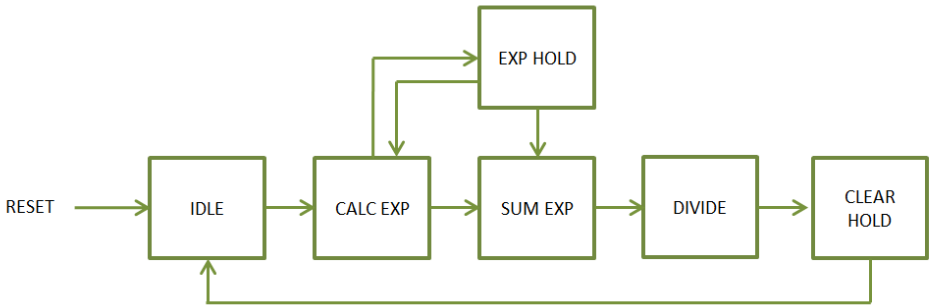

Figure 75: *Finite State Machine for the Softmax Controller*

The state machine for the submodule is shown in ***Figure 75*** and the states are described in text for simplicity.

IDLE: The FSM checks for a signal from the Softmax Adder Wrapper indicating that the summation process has completed. This state also checks for the existence of data in the Exponential Buffer FIFO.

CALC_EXP: This state checks for the existence of data in the Input Buffer as well as if the Exponential Function is ready to receive data. Once those conditions are satisfied, this state reads out a piece of data from the Input Buffer and sends it to the Exponential Function.

EXP_HOLD: This state holds the state machine until the Exponential Function signals that it is ready for another piece of data. Both CALC_EXP and EXP_HOLD execute iteratively until all the input score data is sent to the Exponential Function.

SUM_EXP: This state signals the Softmax Adder Wrapper that the exponentiation process is complete and to begin the addition process. This state holds the state machine until the summation has completed. Also, the Softmax Controller gives the Softmax Adder Wrapper access to the Exponential Buffer by sending a select signal to mux controlling the Exponential Buffer lines.

DIVIDE: This state gives the Softmax Divider Wrapper access to the Exponential Buffer and holds the state machine until the division process has completed.

CLEAR_HOLD: Once the Softmax Function has been allowed to fully execute for the given class scores, this state clears the downstream logic in preparation for the next image classification.

10.0 SIMULATION VERIFICATION TESTING

The most challenging and time-consuming aspect of the development process is the verification of the design in question. In order to properly verify any FPGA design, the design must be tested against its various operating conditions. This kind of rigorous testing must be performed both in simulation and in the hardware itself. Testing must incorporate running data through the design and analyzing the resulting output to verify that the design is in fact performing as intended. To be able to definitively determine that a design is operating as intended, a model of the design must be generated which mimics the designs operations and output. After the truth model is generated, the FPGA design can be simulated using the known inputs and comparing the output of the design with that of the truth model. After the design is fully functioning in a simulated environment, the hardware testing can begin.

For this project there existed absolutely no tools prior to commencing this work nor were there peers available which had prior experience. Therefore, a considerable amount of time was invested into developing an array of tools in various software environments that would be capable of testing the design to completion. Of course, the processes of discovering and learning these tools meant that the verification effort was extremely time consuming and involved more total time to complete than the design itself.

10.1 Python Tools

Before moving forward with any work on a hardware design, the inner workings of Convolutional Neural Networks themselves had to be understood. This involved researching for the most comprehensive treatment of the Convolutional Neural Network material. The most comprehensive coverage, at the time this work was researched, rested with the creators of the ImageNet Challenge out of Stanford University. Their open source coursework from their Computer Science course cs231n enabled this work and their class tools laid the foundation in gaining understanding of the subject. The Stanford tools utilized the Anaconda software package which uses Python Notebooks. These tools aided in the fundamental understanding of the subject as well as aided in developing a model. Using these tools other custom Python tools were created which ran and trained an AlexNet Convolutional Neural Network. The output of each layer, the trained Weights, and trained Biases of this AlexNet model were then used in the verification effort of the design.

10.2 Trained Model

To develop a trained model of a Convolutional Neural Network, two major avenues were pursued in parallel. One avenue was expanding on the Stanford tools and developing a model using the Anaconda with Jupyter Notebooks in Python. The other avenue was to use the much vaunted Tensorflow Software Package developed by Google which utilizes the GPU on a Desktop or Laptop computer to perform the plethora of calculations required for a neural network.

Of these two, the Tensorflow option seemed to be the most promising since it could train a Convolutional Neural Network very quickly using the GPU and leverage the extremely large data set provided by the ImageNet Challenge database. The goal of developing a model in Tensorflow was achieved. However, although developing a CNN model in Tensorflow was successful, diving into its inner workings in order to see the data pixel by pixel proved to be much more tedious and difficult to work with.

Therefore, expanding on the Stanford tools was used to train a CNN with a very small training set. This was done since the objective of this work was to prove a CNN can be deployed efficiently on a small FPGA and not to develop a brand-new CNN framework itself. For this work proving that the layers are operational, and they yield correct results is all that is required.

Using the Stanford tools, a small data set of 10 images was used to train an AlexNet Convolutional Neural Network. These images were selected randomly from a list of 10 Classes shown in ***Table 81***.

Table 81: *10 Classes Used to Train the AlexNet Model*

Class Number	ImageNet ID	Class Description
1	n04548362	wallet, billfold, notecase, pocketbook
2	n03642806	laptop, laptop computer
3	n03063599	coffee mug
4	n02690373	airliner
5	n03930630	pickup, pickup truck
6	n01614925	bald eagle, American eagle, Haliaeetus leucocephalus
7	n03085013	computer keyboard, keypad
8	n02708093	analog clock
9	n03272010	electric guitar
10	n02992529	cellular telephone, cellular phone, cellphone, cell, mobile phone

It is important to note that the ImageNet Challenge Database contains more than 100,000 individual images spanning 1000 or more classes with each image differing in size. Therefore, additional tools Python tools were created to search the database for specific files, determine their size, obtain their bounding box boundaries, and finally resize the image to the 227x227 AlexNet input image size. The randomly selected images which were used as the model's training set are shown in ***Figure 76***.

Figure 76: *10 Randomly Selected Images from 10 Classes*

Using these 10 images as the training set for the AlexNet CNN, the network was trained over 1000 iterations as shown in ***Figure 77***. The model was trained using the following parameters.

a. Regularization = 0.0 (No Regularization)

b. Learning Rate = $5x10^{-3}$

c. Weight Scale = $1.5x10^{-2}$

d. Iterations = 1000

e. Batch Size = 10

f. Update Rule = Stochastic Gradient Descent

g. Learning Rate Decay = 1.0 (No Decay)

From these parameters and from looking at ***Figure 77***, we can clearly see that the trained AlexNet model has overfit its training set with a Training Accuracy of 100% and a Validation Accuracy of 0%. For the purposes of this work, this situation is adequate since as stated before the objective is to prove that the CNN is functional on the FPGA given a weight set and input image. It would be a subject of another work entirely to train the neural network on the FPGA itself.

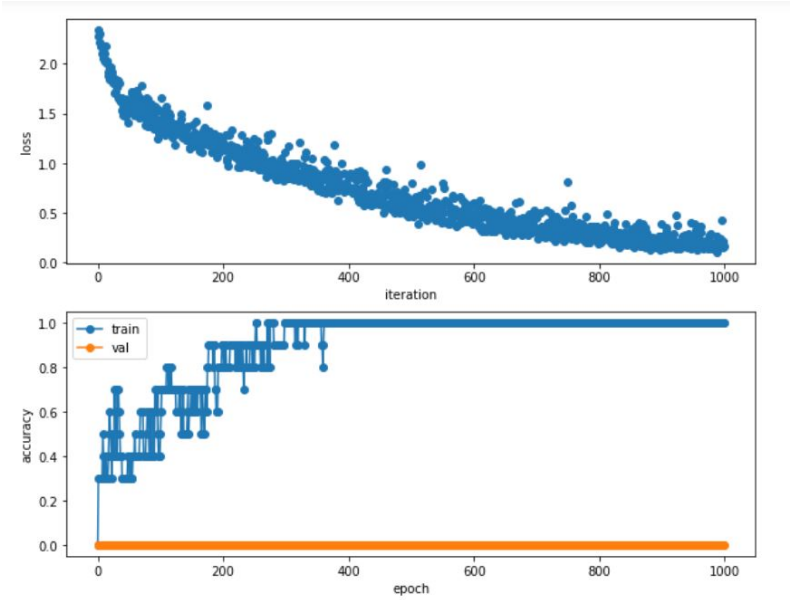

Figure 77: *Loss and Training Accuracy over 1000 iterations*

10.3 Matlab Model

After the custom Python tools were created and run for the AlexNet Convolutional Neural Network, the outputs, trained Weights, and trained Biases for each layer were used to create truth data. The truth data would later be used with the FPGA simulation to verify the design. This was accomplished by generating Matlab scripts which mimicked the operations of the hardware FPGA design. Therefore, given input data, weights, and biases the scripts would generate a comparable truth data to use in simulation as well as make available all the variables involved in the operation. Giving insight to the internal variables of the Convolution, Affine, Max Pool, and Softmax layers was a critical debugging tool and was used in tandem with the FPGA simulations in order to spot were the logic of the design needed work. These scripts can be seen at the link in the Appendix.

10.4 Simulation Testing

Using the files generated from the Matlab scripts, the FPGA designs for the layer types of the Convolutional Neural Network were tested in simulation. The choice of simulation environment was Modelsim PE Student Edition by Mentor Graphics which provided a great deal of flexibility and dependability in its operation. This software simulation package shares much in common with the professional level Mentor Graphics Questasim software package. Each layer type in the design was tested individually in simulation thereby reducing the complexity of the simulation as a whole. Various attempts at testing the layer types were made, however, the most useful method was to test the layer type in an actual AXI configured test bench. This method was the most useful and ultimately led to design completion and verification. Therefore, a Xilinx Vivado block diagram was created for each layer type in two different configurations; a virtual memory configuration, and a block ram configuration. The Vivado Block diagrams for each layer are shown in ***Figure 78*** through ***Figure 83***.

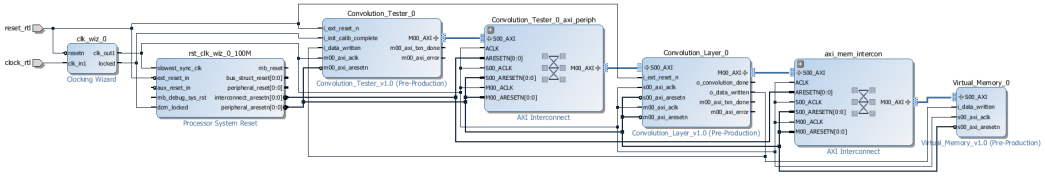

Figure 78: *Convolution/Affine Layer Virtual Memory Test Bench*

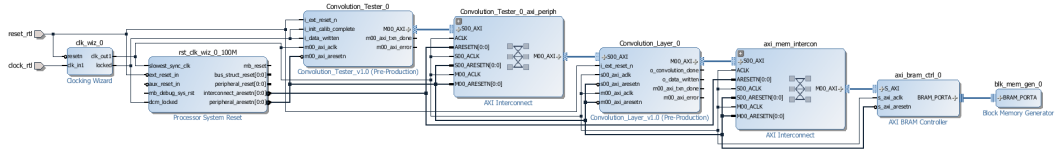

Figure 79: *Convolution/Affine Layer Block RAM Test Bench*

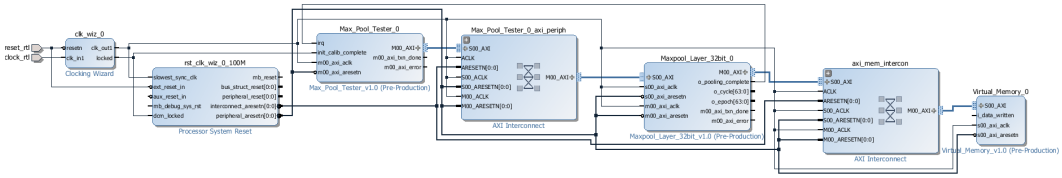

Figure 80: *Max Pool Layer Virtual Memory Test Bench*

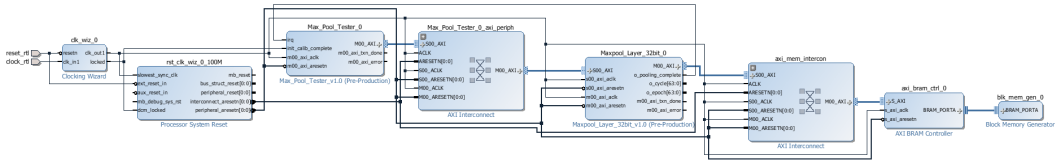

Figure 81: *Max Pool Layer Block RAM Test Bench*

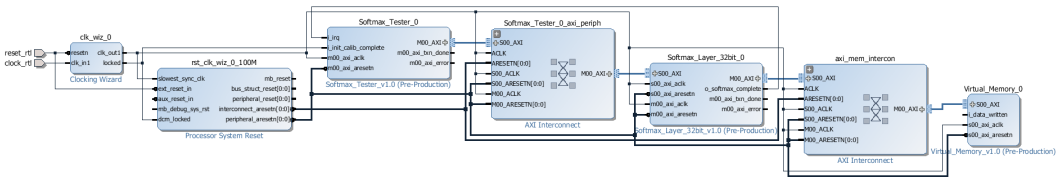

Figure 82: *Softmax Layer Virtual Memory Test Bench*

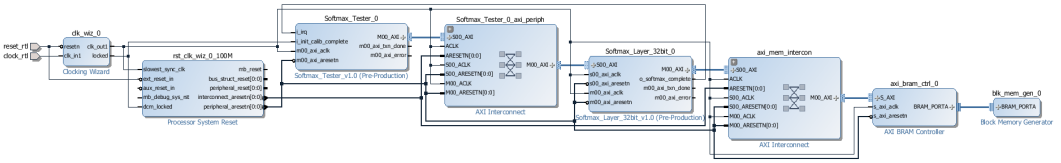

Figure 83: *Softmax Layer Block RAM Test Bench*

The virtual memory configuration required the design of a custom memory block which would read input, weight, and bias data into arrays and allow the AXI bus to read and write from and to those arrays. This virtual memory allowed for actual data to be used in the simulation and would check the resulting data against the truth files automatically.

The block RAM configuration operated much simpler than the Virtual Memory configuration in that it only required the use of a single BRAM instead of a custom memory block. Using a Xilinx block RAM for simulation allowed this type of simulation to execute very quickly whereas the virtual memory configuration would take much longer to complete.

Therefore, both simulation types were used in tandem. The block RAM configuration was used to test overall execution and flow of the design and the Virtual Memory configuration was used to test the correctness of the Neural Network layer results against a truth file.

During the simulation testing, the Microblaze Co-Processor was not used since doing so would have made the testing overly complicated. Therefore, a custom tester module was written to take the place of the Microblaze. This custom tester configures each of the layer's configuration registers and initiates the execution of the layer. Three identical testers were developed which would test the Convolution/Affine Layer, Maxpool Layer, and Softmax Layer. The source code for all the test-benches can be found at the link in the Appendix.

Simulation testing was deemed complete when each of the Convolutional Neural Network layer types were tested with simulated input data, weights, and biases and their output matched that of the truth file generated by the Matlab scripts.

11.0 HARDWARE VERIFICATION AND TESTING

Once the simulation testing had completed, the Hardware Verification testing phase of the design development proceeded. With many Embedded Systems projects, which comes first, the design or the required hardware, is a bit of a chick or the egg problem. Some projects in industry force upon a project a required hardware platform which engineers are required to design their system to run on. Other projects in industry allow for the design itself to drive the requirements that hardware components would need to meet. For this work the scenario would be the latter, where after the design was somewhat finalized, only then were the full hardware requirements known. This is consistent with a research and development type situation where the process of developing a system for the first time fully reveals its needed resources.

11.1 FPGA Board

After the design was mostly finalized, the design was run through the Xillinx Synthesis and Implementation tools in order to generate a bit file for hardware integration. Mostly finalized, meaning that no major structural changes would be made to the logic of the design. At this point it became clear that the 32-bit data configuration of the design would take too many resources and would not fit onto the Zedboard as was previously hoped. Therefore, the alternative board, the Nexys Video, was used which does have enough resources to fit the Convolutional Neural Network design.

11.2 Testing

Once the Convolutional Neural Network FPGA design met the timing requirements of 100MHz clock rate, hardware testing moved along. As with the simulation testing, the hardware testing first tested each layer type individually in a full Microblaze-AXI configuration. Now that these hardware configuration tests included the Microblaze processor, the Xilinx SDK XCST tool was used. This tool allows for direct commanding from the PC computer through the Microblaze and onto the AXI bus running on the FPGA. To aide in this effort, custom scripts written in the TCL language were made which would configure the registers of each layer, load the input data to DDR Memory from the PC hard drive, and execute the layer once all configuration and data loads were complete.

Of course, as it is with many Embedded Systems designs, the first integration with hardware often shows the design does not function and kicks off the hardware debugging effort. At this level in development there isn't as much access to the individual signals of the design as there is with simulation. This lack of observability necessitates the use of yet another tool to allow for the hardware debugging to take place at all. The tool in question would be some sort of Logic Analyzer to scope the internal signals of the design and bring them out for human eyes to analyze. Therefore, the Xilinx Integrate Logic Analyzer tool was used extensively in order to peer inside the design during its operation and see where the logic needed to be adjusted.

Each layer by design writes its output data to a specified place in the DDR memory. Again, using TCL scripts this output data was read out of the FPGA board memory and written to a bin file for comparison with truth data. The Beyond Compare software was used in order to perform a bit by bit comparison and ensure that the output data of each layer matched the expected truth data.

The use of simulation testing and these hardware testing methods were used iteratively in order to test logic changes in simulation first before deploying the changes on hardware.

12.0 RESULTS AND DISCUSSION

As stated earlier, the objective of this work was to successfully implement a scalable, modular, and programmable Convolutional Neural Network on a small FPGA. This would allow a cost effective and versatile tool for Embedded Applications utilizing image recognition/classification. This work was successful in this endeavor and this section will characterize the designs performance both with Simulation and Hardware results.

12.1 Floating-Point Operation Calculations

We know that each Convolution and Affine layer will perform a certain number of Floating-Point Operations. Therefore, we can say that the total number of Floating-Point Operations divided by the time of execution yields the Floating-Point Operations per Second. The calculations arriving at the total number of Floating-Point Operations is as follows. Let's use the first Convolutional Layer as an example.

The input image is 227x227x3 with a weight filter kernel of size 11x11x3x96 and an output volume of size 55x55x96. The design currently allows for 33 Channel Units. So, the total number of Floating-Point Operations for one output volume rows worth of the Channel Units is

$$CHU_{FLOP} = weight\ width \times Filters \times Strides \times Channel\ Units$$

$$CHU_{FLOP} = 11 \times 96 \times 55 \times 33$$

$$CHU_{FLOP} = 1916640$$

The adder tree employs the use of 6 adder layers with the following number of adders per layer

Layer	Number of Adders
Layer1	16
Layer2	8
Layer3	4
Layer4	2

Layer5	1
Layer6	1

$$ACCU\_TREE_{FLOP} = \ weight\ width\ \times Filters\ \times Strides\ \times Tree\ Adders$$

$$ACCU\_TREE_{FLOP} = \ 11\ \times 96\ \times 55\ \times 32$$

$$ACCU\_TREE_{FLOP} = \ 1858560$$

The kernel adder tree adds an additional 3 adder layers with the following number of adders per layer.

Layer	Number of Adders
Layer1	5
Layer2	3
Layer3	1

$$KERNAL\_TREE_{FLOP} = \ output\ width\ \times Filters\ \times Kernal\ Tree\ Adders$$

$$KERNAL\_TREE_{FLOP} = \ 55\ \times 96\ \times 9$$

$$KERNAL\_TREE_{FLOP} = \ 47520$$

The total number of Floating-Point Operations for one output volume rows worth of the Accumulator is.

$$ACCU_{FLOP} = \ ACCU\_TREE_{FLOP} + KERNAL\_TREE_{FLOP}$$

$$ACCU_{FLOP} = 1906080$$

The design adds the Bias values

$$BIAS_{FLOP} = 96$$

Since there are 55 output volume rows, the number of Floating-Point operations for the Channel Unit and Accumulator will be repeated 55 times.

$$VOLUME_{FLOP} = 55 * (ACCU_{FLOP} + CHU_{FLOP}) + BIAS_{FLOP}$$

$$VOLUME_{FLOP} = 207636096$$

For the Convolution and Affine Layers, if an input image contains more channels than there are available resources, the design will process channels in groups. Therefore, after each iteration of channel groups, the output data calculated for the previous channel group must be summed with the output data calculated for the current channel group. Continuing with the example,

$$PREV_{FLOP} = Out\ Height * Out\ Width\ * Out\ Channels * (Channel\ Iterations - 1)$$

$$PREV_{FLOP} = 55 * 55\ * 96 * (1 - 1)$$

$$PREV_{FLOP} = 0$$

For the current example the input image contains a number of channels within the available resources and would therefore not require previous output data to be summed with current output data. The same calculations for each Convolution and Affine Layer are performed and shown in ***Table 82*** to determine the total number of Floating-Point Operations being performed by the layers.

Table 82: *Floating Point Operations Per AlexNet Layer*

	CHU_FLOP	ACC_FLOP	BIAS_FLOP	ROW_FLOP	VOL_FLOP
CONV1	1916640	1906080	96	3822720	210249696
CONV2	1036800	1168128	256	2204928	62332672
CONV3	359424	524160	384	883584	13498752
CONV4	359424	524160	384	883584	14537088
CONV5	239616	349440	256	589056	9691392
AFFINE1	589824	823296	4096	1413120	90701824
AFFINE2	131072	167936	4096	299008	38797312
AFFINE3	320	410	10	730	94720

12.2 Memory Transactions

Another important performance characteristic is the number of memory transactions being initiated in order to successfully read in all the relevant data and write out the correct result. This becomes important since each transaction with off chip memory adds delay in the overall execution of the Convolution or Affine Layer. Therefore, keeping the number of memory transactions to a minimum will aid the performance of any design. Continuing with the example of the first Convolution layer we can estimate the number of memory transactions for both the read and write cases. The Convolution Layer inputs four kinds of data; input image data, weight filter kernel data, bias data, and previous output volume data.

The input image data is read into the design in a row by row basis. The first transaction will prime the Volume FIFOs in the Channel Units by reading in the first few rows of each channel. So, in this case if we have 3 channels of the input image, three transactions will load the first few rows.

$$Prime\_FIFO_{MT} = Channels\ Allowed$$

$$Prime\_FIFO_{MT} = 3$$

For the stride operation, each row for each channel is read from memory individually and loaded into the appropriate FIFO. This is done once as many times as the stride value dictates. Strider operations are performed until the entire input image is loaded into the design. The first Convolution layer specifies a stride of 4.

$$Stride_{MT} = Channels\ Allowed\ \times Stride\ \times (Output\ Height - 1)$$

$$Stride_{MT} = 3\ \times 4\ \times (55 - 1)$$

$$Stride_{MT} = 3\ \times 4\ \times (55 - 1)$$

$$Stride_{MT} = 648$$

Both the memory transactions to load the first rows into the design as well as the stride memory transactions are performed as many times as needed to read all the input image channels into the design.

$$Input\ Read_{MT} = Channel\ Iterations\ \times (Stride_{MT} + Prime\_FIFO_{MT})$$

$$Input\ Read_{MT} = 1\ \times (648 + 3)$$

$$Input\ Read_{MT} = 651$$

The weight filter kernels are read into the design as one large transaction spanning the allowable channels. So, in example with a weight filter kernel set of dimensions 11x11x3x96, the 11x11 kernel for all three channels is read in as one transaction. This is repeated for each filter and in the case where there are more channels than available resources.

$$Weights_{MT} = Filters\ \times Channel\ Iterations$$

$$Weights_{MT} = 96\ \times 1$$

$$Weights_{MT} = 96$$

The bias data is read in as one single transaction.

$$Bias_{MT} = 96$$

In the case where the number of input image channels exceeds the number of available resources, the previous output volume results are read back into the design to sum these values with those of the current output volume data. The previous output volume data is read into the design one row at a time.

$$Previous\ Data_{MT} = Output\ Channels\ \times Output\ Rows\ \times (Channel\ Iterations - 1)$$

$$Previous\ Data_{MT} = 96\ \times 55\ \times (1 - 1)$$

$$Previous\ Data_{MT} = 0$$

For the current example the input image contains a number of channels within the available resources and would therefore not require previous output data to be summed with current output data. The same

calculations for each Convolution and Affine Layer are performed and shown in ***Table 83*** to determine the total number of memory transactions being performed by the layers.

Table 83: *Memory Read and Write Transactions Per AlexNet Layer*

Layer	Input Read Trans.	Weight Read Trans.	Bias Read Trans.	Prev Read Trans.	Write Trans.	Total Mem Trans.
CONV1	651	96	1	0	5280	6028
CONV2	2592	256	1	103680	110592	217121
CONV3	3328	384	1	154752	159744	318209
CONV4	4992	384	1	234624	239616	479617
CONV5	4992	256	1	156416	159744	321409
AFFINE1	256	262144	1	258048	262144	782593
AFFINE2	4096	524288	1	520192	524288	1572865
AFFINE3	4096	1280	1	1270	1280	7927

12.3 Simulation Performance

We can estimate the performance of each layer of the design by analyzing the simulations used during simulation testing. During the operation of the design, each layer is configured and then allowed to start its execution after a start signal has been given by command to the layer's AXI Slave interface. After each layer has completed its execution, a done or complete signal is set high by the layer's logic. We can get an idea of the total time of execution for each layer by comparing the start time to the complete time. These results are shown in ***Table 84***.

Table 84: *Simulation Execution time of each AlexNet Layer*

Layer	Start Time	End Time	Total Time	FLOPs
CONV1	2.725us	71733.655 us	71.73093 ms	2.931 G
CONV2	2.725us	485199.955 us	485.19723 ms	128.468 M
CONV3	2.725us	363293.625 us	363.2909 ms	37.156 M
CONV4	2.725us	545573.945 us	545.57122 ms	26.645 M
CONV5	2.725us	364695.804598us	364.693079598ms	26.574 M
AFFINE1	2.725us	514410.525 us	514407.8 ms	176.322 M
AFFINE2	2.725us	442502.215 us	442499.49 ms	87.677 M
AFFINE3	2.725us	27927.805 us	27925.08 ms	33.919 M

12.4 Synthesized and Implemented Design

The finalized design implementing a Convolutional Neural Network can be seen in ***Figure 85*** and is reflective of the top level block diagram in

Figure 30. The design was synthesized and implemented in the Xilinx Vivado development environment. The pertinent specifications for the synthesized design can be summed up in ***Figure 88*** which shows the design summary of the generated bit file.

12.4.1 Timing Summary

As shown in ***Figure 88*** as well as in ***Figure 84***, the implementation of a Convolutional Neural Network on FPGA met its timing requirements with the constraint of a 100MHz input clock frequency.

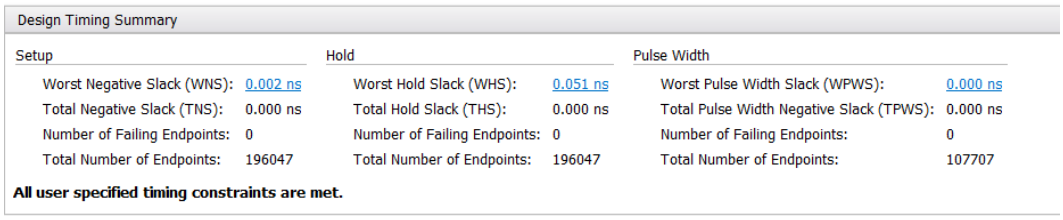

Figure 84: *Timing Report for Final Design*

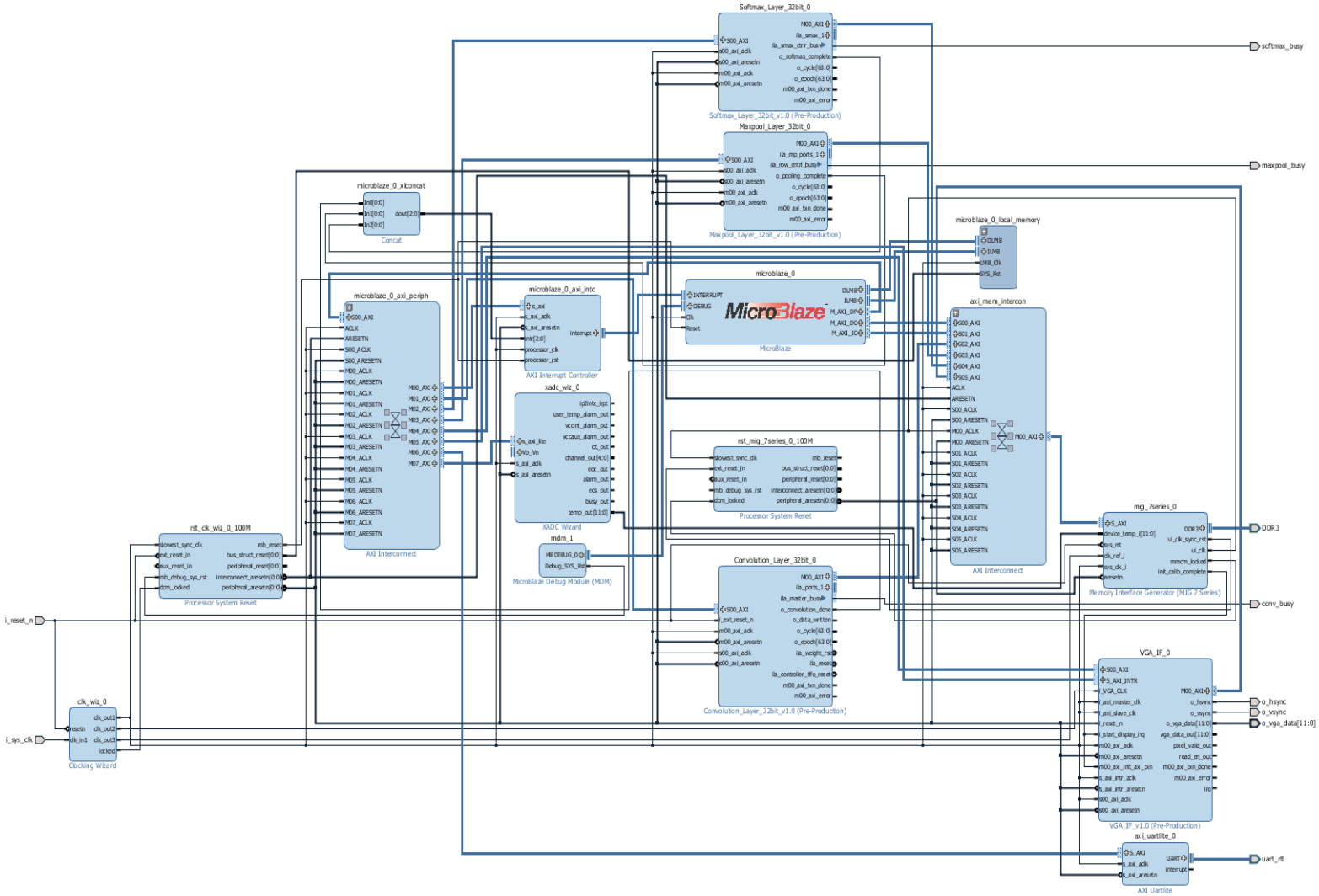

Figure 85: *Final Top-Level FPGA Design*

12.4.2 Resource Utilization

As shown in ***Figure 88*** as well as in ***Figure 86*****,** the design currently is using ~70% of the FPGA Look-up Tables, ~40% of the FPGA Flip Flops, and ~40% of the FPGA Block RAM. It is worth making special mention that the design currently only uses ~16% of the FPGA DSPs. ***Figure 89*** illustrates the FPGA resource utilization.

Resource	Utilization	Available	Utilization %
LUT	91865	133800	68.66
LUTRAM	2594	46200	5.61
FF	103610	267600	38.72
BRAM	139.50	365	38.22
DSP	119	740	16.08
IO	69	285	24.21
BUFG	7	32	21.88
MMCM	2	10	20.00
PLL	1	10	10.00

Figure 86: *Resource Utilization of Final Design*

12.4.3 Power Usage

As shown in ***Figure 88*** as well as in ***Figure 87*****,** the design currently consumes 1.374 watts of dynamic power and 0.179 watts of static power with a total power consumption of 1.553 watts.

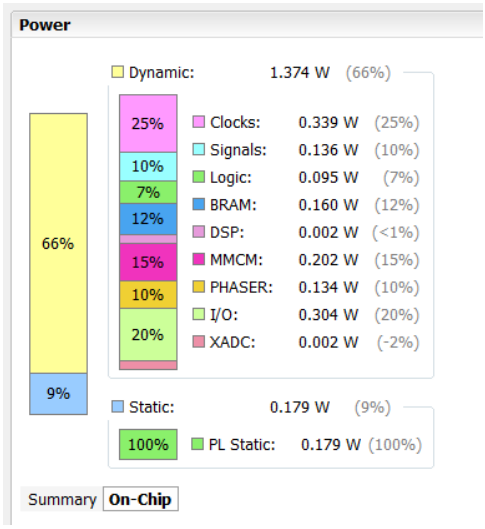

Figure 87: *Power Consumption of Design*

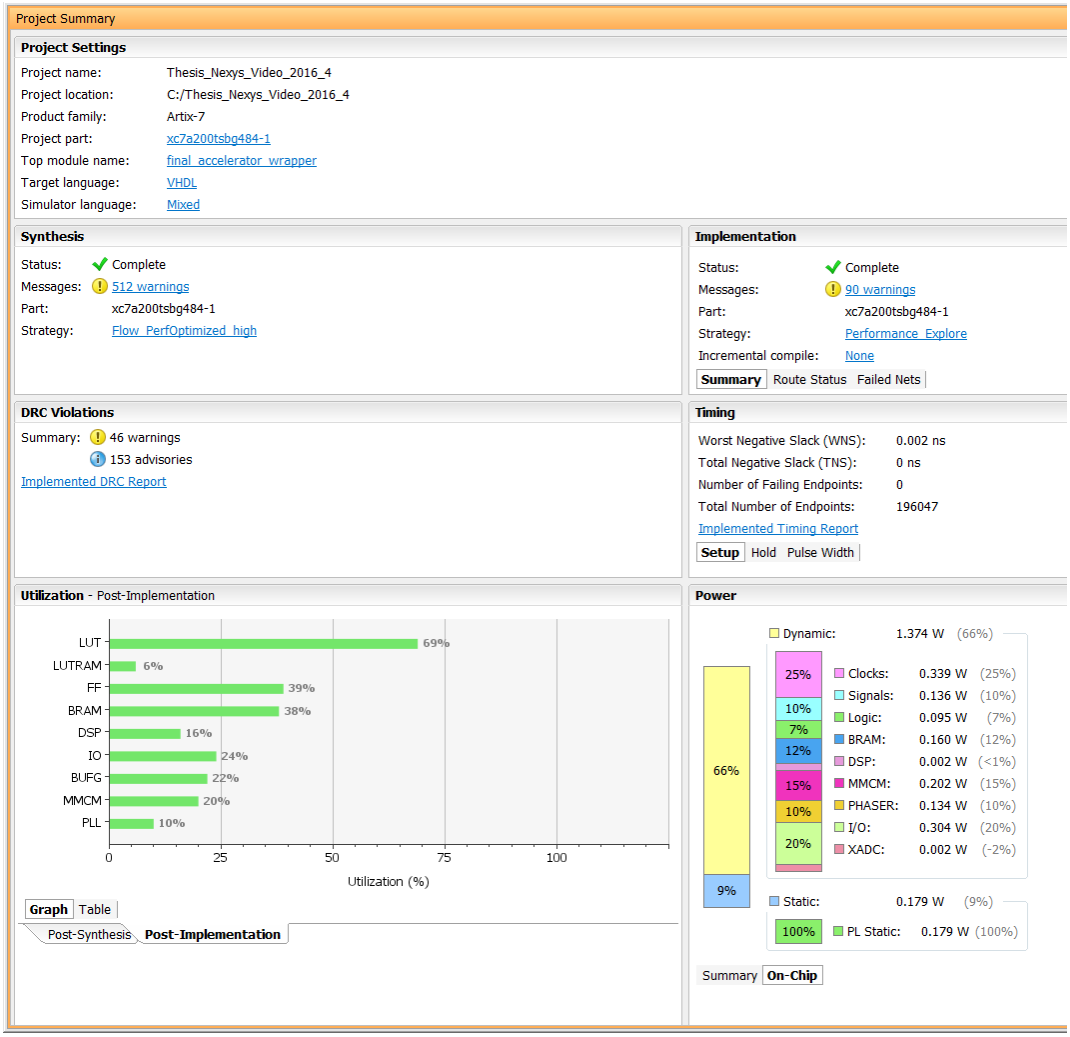

Figure 88: *Synthesized and Implemented Design*

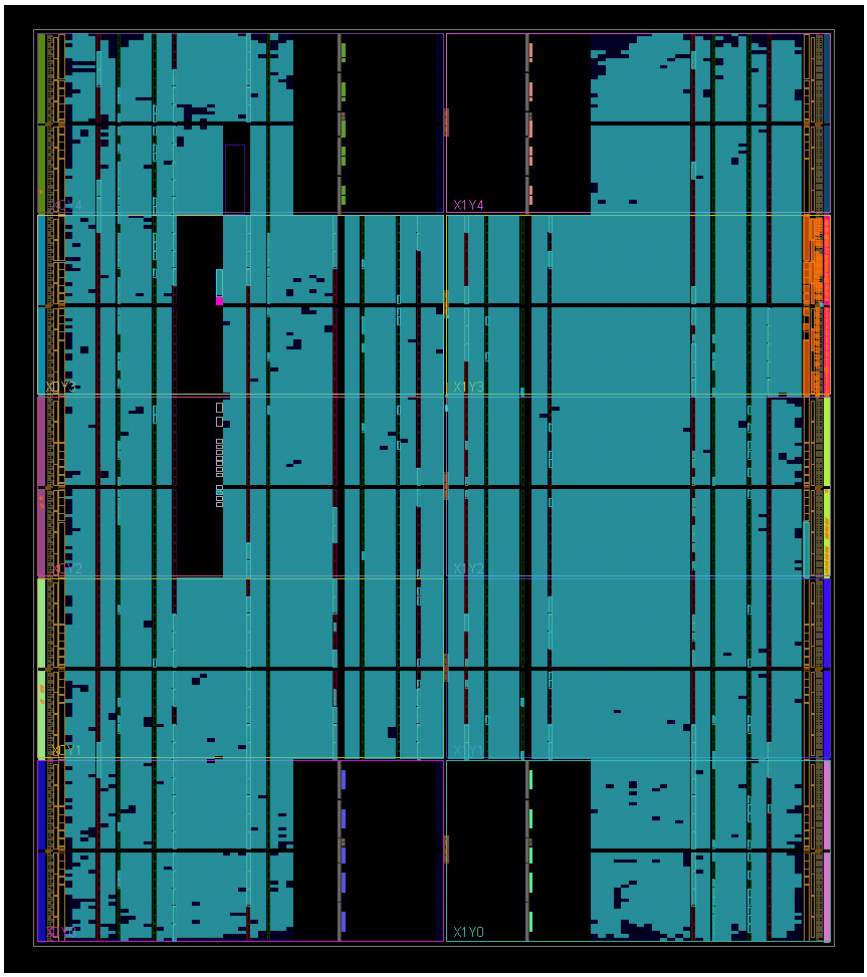

Figure 89: *Graphic showing usage of FPGA resources.*

12.5 Hardware Performance

Just as how the Floating-Point Operations per second were calculated for the simulations, we can also calculate the FLOPs in a similar fashion. In this case however the Integrated Logic Analyzer was used along with an epoch counter running on the FPGA. The combination of the Vivado ILA and the epoch counter aided greatly in the hardware debugging of the design. The epoch counter increments an epoch count every time a cycle counter counts 100 cycles. With a 100MHz clock frequency, the epoch counter increments every 1us. Both Cycle Counter and Epoch Counter are broken out of the design and fed to the logic analyzer for display and analysis. The counters start when the start signal is given by command and stop when the done signal is flagged by the logic signaling that the Convolution or Affine operation has completed. These results are shown in ***Table 85***.

Table 85: *Hardware execution times of each AlexNet Layer*

Layer	Start Time	End Time	Total Time	FLOPs
CONV1	0	71198.67 us Epoch = 0x1161e Cycle = 0x43	71.19867 ms	2.9530 G
CONV2	0	547753.71 us Epoch = 0x85BA9 Cycle = 0x47	547.75371 ms	108.806 M
CONV3	0	463776.90 us Epoch = 0x713A0 Cycle = 0x5A	463.77690 ms	24.858 M
CONV4	0	697862.14 us Epoch = 0xaa606 Cycle = 0x0E	697.86214 ms	16.551 M
CONV5	0	466757.25 us Epoch = 0x71f45 Cycle = 0x19	466.75725 ms	16.543 M
AFFINE1	0	796440.32 us Epoch = 0xc2718 Cycle = 0x20	796.44032 ms	110.922 M
AFFINE2	0	1018890.52 us Epoch = 0xf8c0a Cycle = 0x34	1018.89052 ms	33.446 M
AFFINE3	0	4682.26 us Epoch = 0x124A Cycle = 0x1A	4.68226 ms	17.769 M

12.5.1 Output Results

The ten randomly selected images shown in ***Figure 76*** from the 10 class list shown in ***Table 81*** were run through this hardware FPGA implementation of a Convolutional Neural Network in the AlexNet configuration. All 10 images were classified correctly by the FPGA hardware with the image's class being correctly identified as being the one with the largest score out of 10 classes. As shown below in ***Table 86***, both the Python Model scores as well as the Hardware scores are shown for comparison. Each ImageNet ID for each input image is given along with the proper class number out of the 10-class set. As can be seen from the scores, both the Python and Hardware scores are virtually identical with the hardware achieving a great deal of precision. Also provided are the results of the Softmax Classifier hardware probabilities shown in ***Table 87***. Looking at ***Table 87***, we can see that the images are classified with 80-100% probability.

Table 86: *10 Images scored by Model vs Hardware*

Image #	Image ID	Model Class	Model Score	HW Class	HW Score Hex	HW Score Dec	%Diff
1	n02690373_3435	4	7.771	4	0x40F8778B	7.7646	0.0824
2	n02708093_1783	8	5.008	8	0x40A05553	5.0104	0.0479
3	n02992529_59933	10	4.844	10	0x409AF820	4.8428	0.0248
4	n02690373_6755	4	9.227	4	0x41138E41	9.2222	0.0520
5	n03930630_24834	5	6.361	5	0x40CB98F0	6.3624	0.0220
6	n02708093_3688	8	5.808	8	0x40BA17C8	5.8154	0.1272
7	n02690373_17389	4	9.753	4	0x411C0F50	9.7537	0.0072
8	n03063599_3650	3	8.181	3	0x41028B29	8.159	0.2696
9	n03063599_4149	3	5.665	3	0x40B4E50D	5.653	0.2123
10	n03063599_5006	3	7.136	3	0x40E41692	7.1278	0.1150

Table 87: *10 Images Softmax Probability Results*

Image #	Image ID	HW Class	HW Score Hex	HW Score Dec	HW Probs Hex	HW Probs %
1	n02690373_3435	4	0x40F8778B	7.7646	0x3F7D7487	99.01
2	n02708093_1783	8	0x40A05553	5.0104	0x3F53D255	82.74
3	n02992529_59933	10	0x409AF820	4.8428	0x3F55CC70	83.52
4	n02690373_6755	4	0x41138E41	9.2222	0x3F7C6A0D	98.60
5	n03930630_24834	5	0x40CB98F0	6.3624	0x3F576CE9	84.15
6	n02708093_3688	8	0x40BA17C8	5.8154	0x3F6FC1F4	93.66
7	n02690373_17389	4	0x411C0F50	9.7537	0x3F7ECA8E	99.53
8	n03063599_3650	3	0x41028B29	8.159	0x3F674FB1	90.36
9	n03063599_4149	3	0x40B4E50D	5.653	0x3F6A050F	91.41
10	n03063599_5006	3	0x40E41692	7.1278	0x3F7A8D36	97.87

To see how each layer of the neural network is processing the input image, the results of each layer in the neural network for an example image shown in ***Figure 90*** are read out of the hardware memory and converted to BIN file for display. This example image is run through the hardware and the results of the first Convolutional Layer with the ReLu activation is shown in ***Figure 91***. The results of each subsequent AlexNet Layer are shown in ***Figure 92*** through ***Figure 98***. Each of these output images was read from hardware memory and are results of the full AlexNet Convolutional Neural Network executing.

Figure 90: *Example Input Image*

The image shown in ***Figure 91*** is the result of Convolving the Input RGB image with the Weight filter and bias data sets for the first convolution layer, the CONV1 layer. The input image shown in ***Figure 90*** was of dimension 227x227x3, the Weight Filter Kernel was of dimension 11x11x3x96, and the Bias data was 96 values. The Convolution operation used a stride of 4 columns and/or rows as well as a pad of 0. The resulting image in ***Figure 91*** is a mosaic of the output image or feature map with dimensions 55x55x96. All 96 channels are displayed, and we can already see the weight filter kernel beginning to filter out and/or intensify certain features in the initial image.

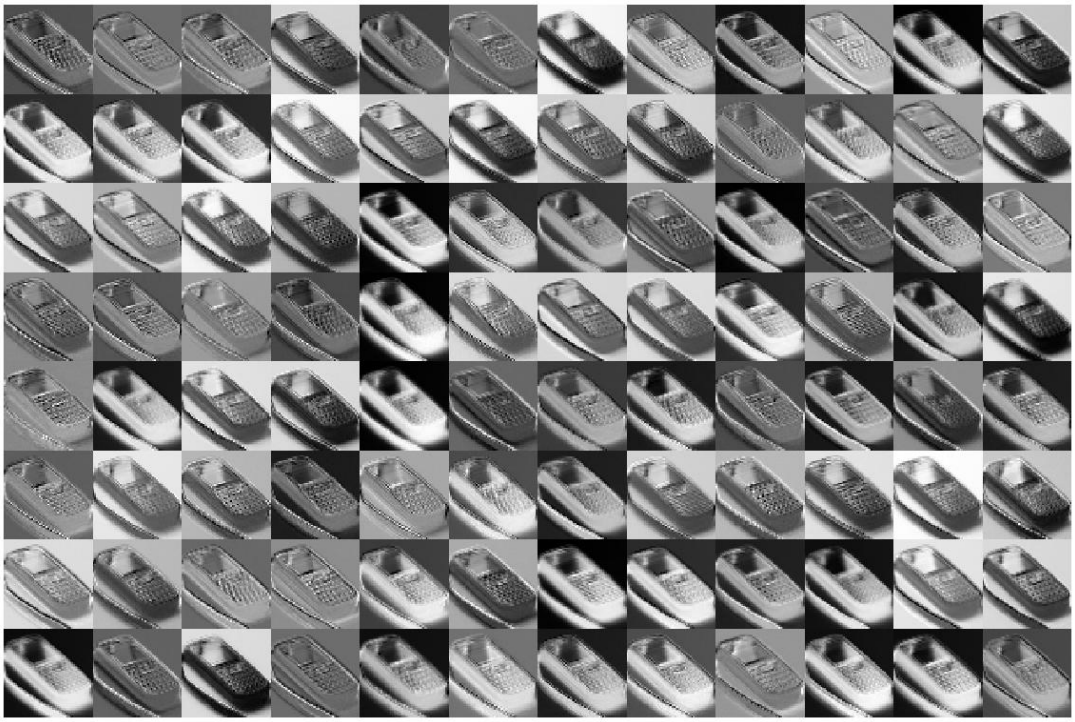

Figure 91: *Example result of Convolution/ReLu Layer 1.*
Output Feature Map or Image has dimensions of 55 pixels high, 55 pixels wide, and 96 channels deep.

After the image had been run though the CONV1 layer, the image is passed through the first Maxpooling Layer, MAXPOOL1. As was described in a previous section, the Maxpooling layer applies a max value kernel for a 3x3 neighborhood throughout the entire image data set. This kernel strides every 2 rows and/or columns. Therefore, the image shrinks in size from 55x55x96 to 27x27x96. The resulting image shown in ***Figure 92*** shows how this process already begins to eliminate some of the lesser intensity detail.

Figure 92: *Example result of Maxpool Layer 1*
Output Feature Map or Image has dimensions of 27 pixels high, 27 pixels wide, and 96 channels deep.

The image shown in ***Figure 93*** is the result of Convolving the output of the MAXPOOL1 layer with the Weight filter and bias data sets for the second convolution layer, the CONV2 layer. The MAXPOOL1 result shown in ***Figure 92*** was of dimension 27x27x96, the Weight Filter Kernel was of dimension 5x5x96x256, and the Bias data was 256 values. The Convolution operation used a stride of 1 column and/or row as well as a pad of 2. The resulting image in ***Figure 93*** is a mosaic of the output image or feature map with dimensions 27x27x256. All 256 channels are displayed, and we can especially see the difference when compared to ***Figure 91*** in that even more weight filters are beginning to suppress some features of the original image and intensify others.

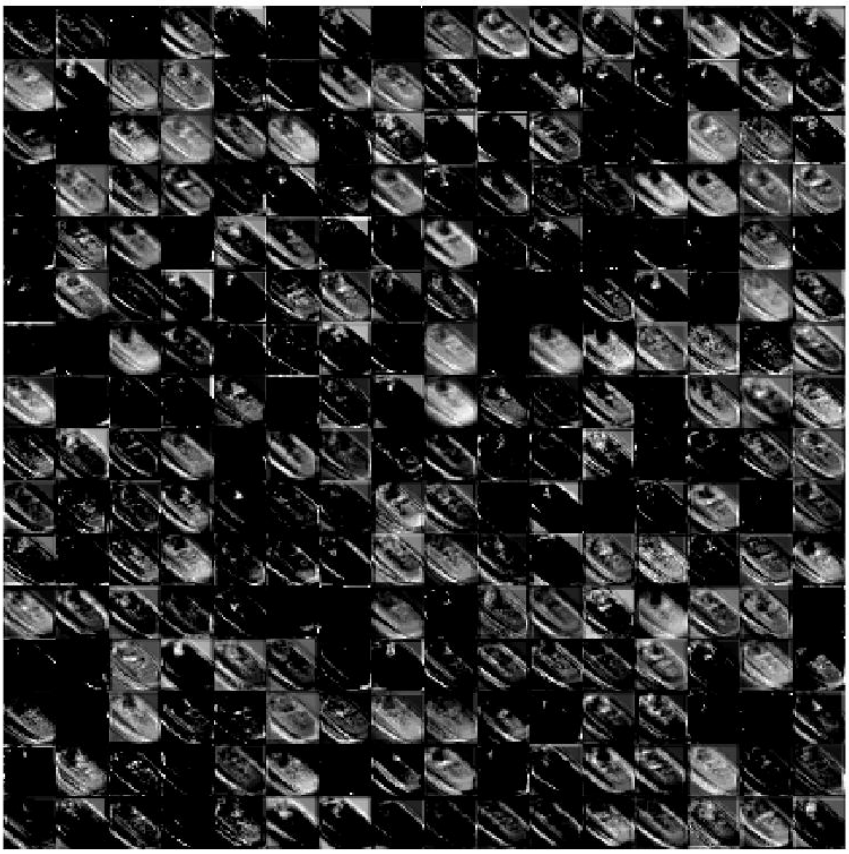

Figure 93: *Example result of Convolution/ReLu Layer 2.*
Output Feature Map or Image has dimensions of 27 pixels high, 27 pixels wide, and 256 channels deep.

Same as with the MAXPOOL1 layer, the resulting image from the CONV1 layer is passed through the second Maxpooling Layer, MAXPOOL2. This Maxpooling layer also strides every 2 rows and/or columns as well as uses a 3x3 max filter neighborhood. Therefore, the image shrinks in size from 27x27x256 to 13x13x256. The resulting image shown in ***Figure 94***.

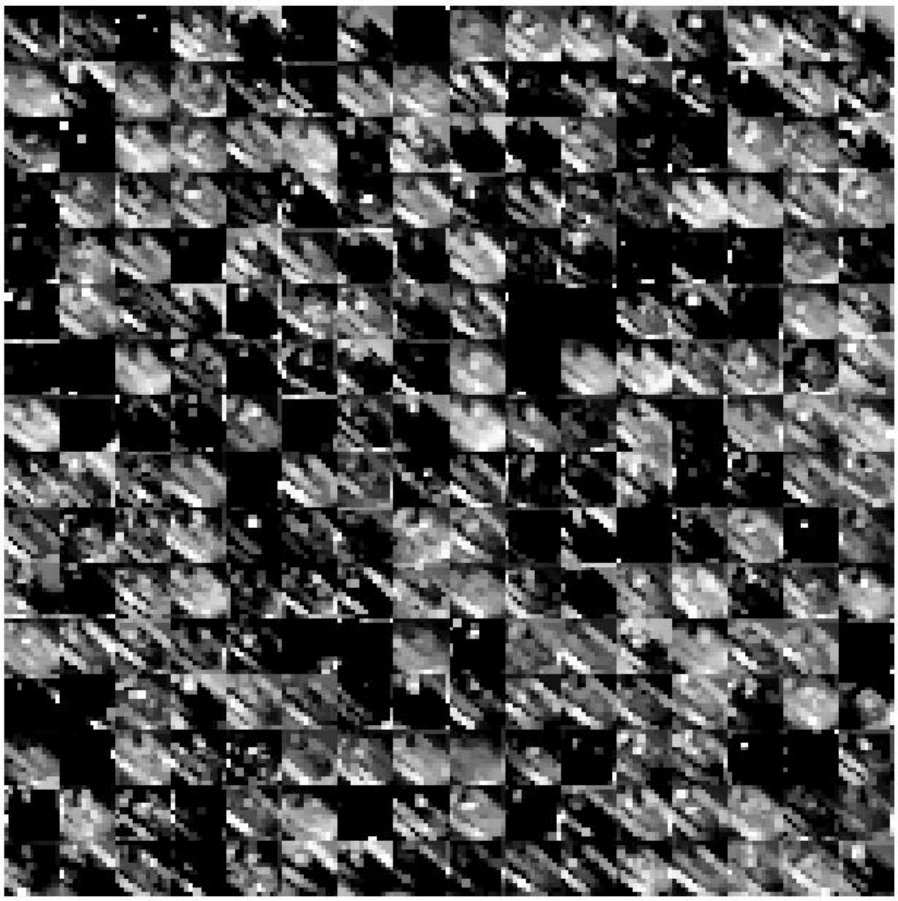

Figure 94: *Example result of Maxpool Layer 2*
Output Feature Map or Image has dimensions of 13 pixels high, 13 pixels wide, and 256 channels deep.

Just as with the CONV1 and CONV2 layers, the CONV3 (see ***Figure 95***), CONV4 (see ***Figure 96***), and CONV5 (see ***Figure 97***) layers execute in the same manner. The only real difference is that between the CONV3 through 5 layers, there is no Max pool operation. These three Convolution layers use a stride and pad of 1 with a 3x3 weight filter kernel. Therefore, the output images between these three layers does not shrink in size and stays 13x13 even though their channels change in size. As the input image is process through these layers, we can see the weight filter kernels further filtering out some image features while intensifying others.

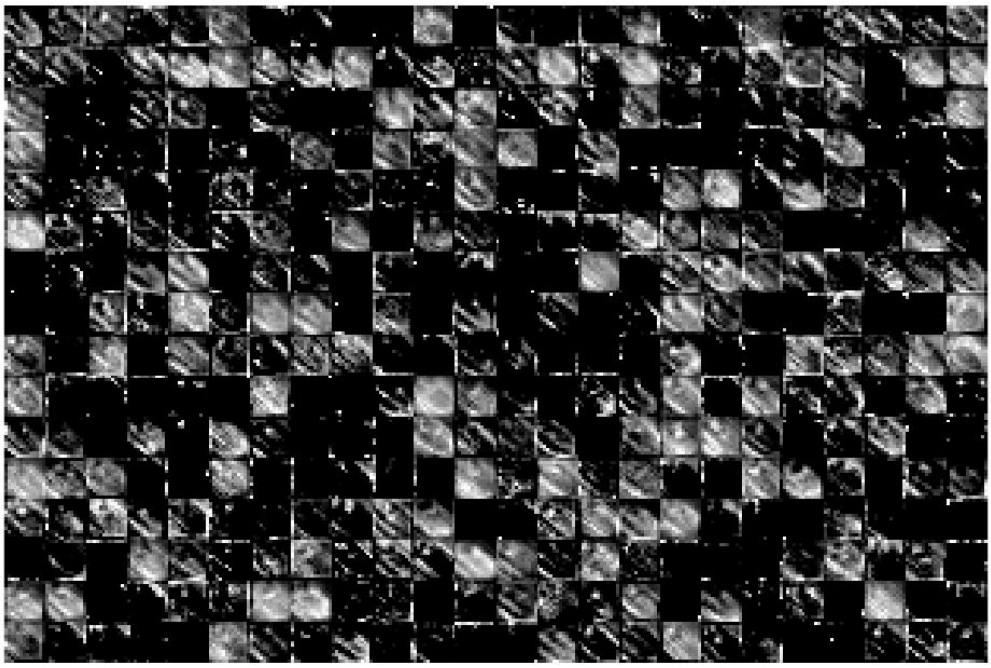

Figure 95: *Example result of Convolution/ReLu Layer 3.*
Output Feature Map or Image has dimensions of 13 pixels high, 13 pixels wide, and 384 channels deep.

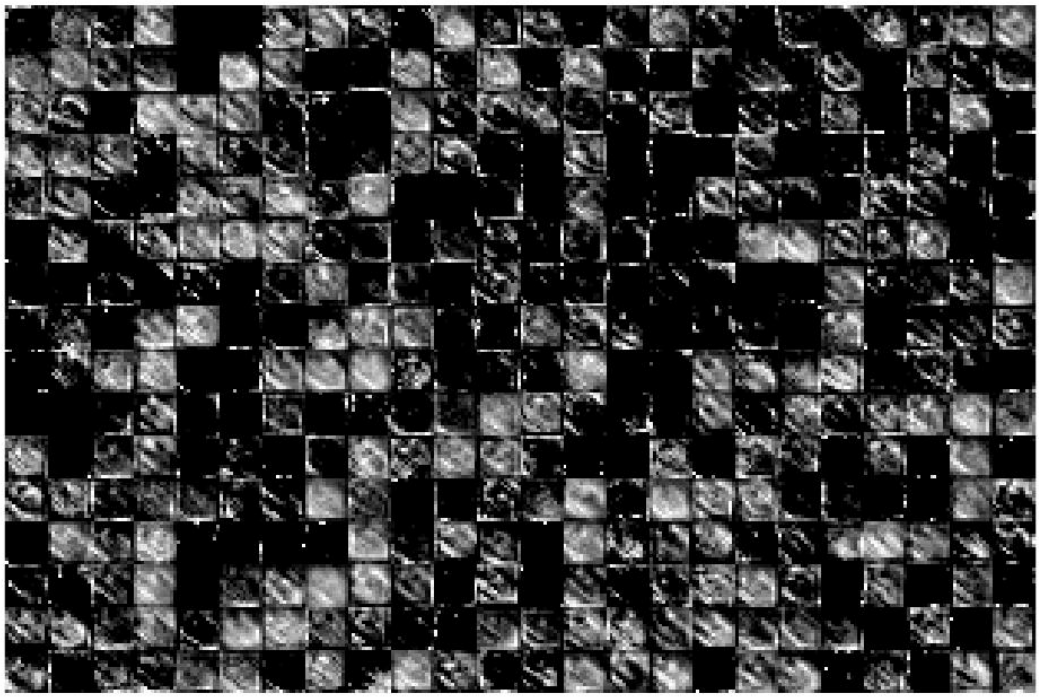

Figure 96: *Example result of Convolution/ReLu Layer 4.*
Output Feature Map or Image has dimensions of 13 pixels high, 13 pixels wide, and 384 channels deep.

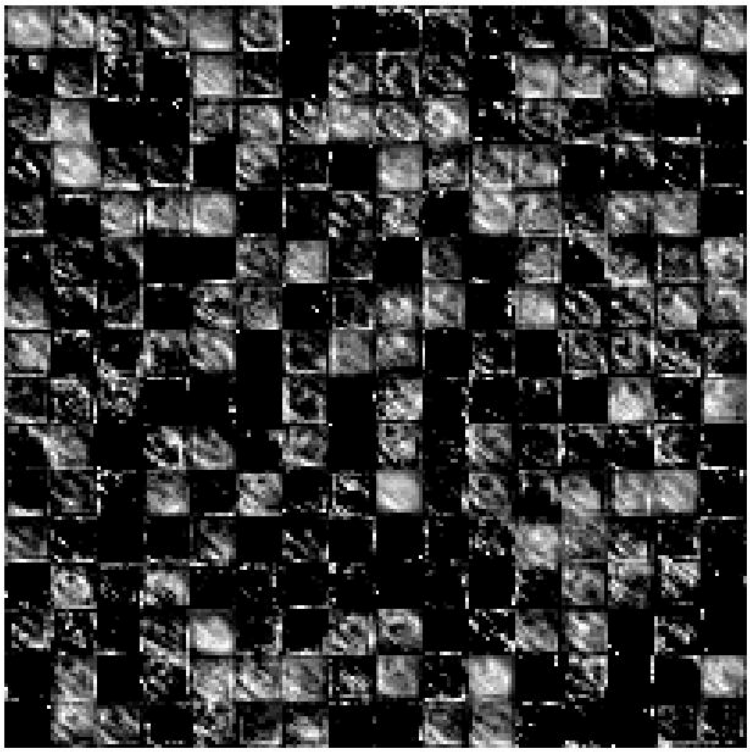

Figure 97: *Example result of Convolution/ReLu Layer 5.*
Output Feature Map or Image has dimensions of 13 pixels high, 13 pixels wide, and 256 channels deep.

Same as with the MAXPOOL1 and MAXPOOL2 layers, the resulting image from the CONV1 layer is passed through the third and last Maxpooling Layer, MAXPOOL3. This Maxpooling layer also strides every 2 rows and/or columns as well as uses a 3x3 max filter neighborhood. Therefore, the image shrinks in size from 13x13x256 to 6x6x256. The resulting image shown in ***Figure 98***.

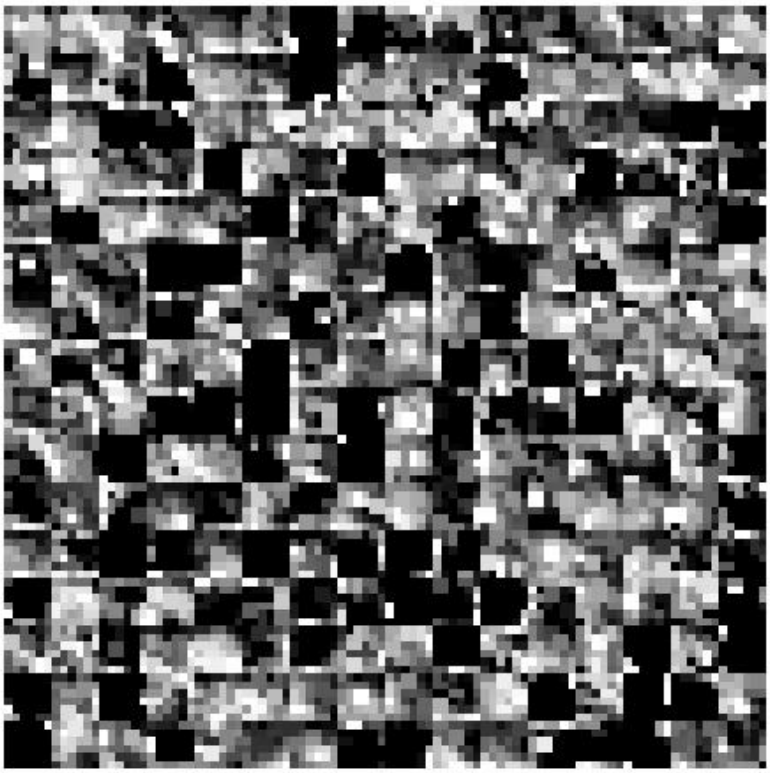

Figure 98: *Example result of Maxpool Layer 3*
Output Feature Map or Image has dimensions of 6 pixels high, 6 pixels wide, and 256 channels deep.

12.6 Performance Assessment

Now the that design's performance has been characterized and the image classification ability has been accessed to be accurate, a discussion should be had as to why the system performs as it does.

12.6.1 Sim vs. HW Flops

The floating-point operations per second for the layer simulations shown in ***Table 84*** and the FLOPs gathered from the hardware layers shown in ***Table 85*** are shown below in ***Table 88***. In the table although the simulation FLOPS results come very close to approximating the actual hardware FLOPS achievable by each layer, there is a growing percentage difference between the two performance assessments. This is likely due to the simulation estimating an ideal model of the actual hardware and does not and cannot consider real latencies present in the actual hardware. This difference has a compounding effect the more a layer requires channels to be processes in groups.

Table 88: *Simulation and Hardware FLOPs*

Layer	SIM FLOPs	HW FLOPs	%Diff	Channel Iterations
CONV1	2.931 G	2.9530 G	0.742%	1
CONV2	128.468 M	113.796 M	12.893%	16
CONV3	37.156 M	29.106 M	27.660%	32
CONV4	26.645 M	20.830 M	27.914%	48
CONV5	26.574 M	20.763 M	27.986%	48
AFFINE1	176.322 M	113.884 M	54.827%	64
AFFINE2	87.677 M	38.077 M	130.258%	128
AFFINE3	33.919 M	20.229 M	83.233%	128

12.6.2 Performance Breakdown

Also evident in ***Table 88*** is the fact that the majority of the AlexNet layers in this implementation perform at the MFLOPS range with the only GFLOPS performance seen by the first Convolutional Layer. With current research at large on GPUs yielding performance in the tens or hundreds of GFLOPS, an explanation as to why this design operates in the MFLOPS range predominantly is necessary.

As discussed earlier, in addition to the number of Floating-Point Operations needing to be performed, another aspect of the design which is of paramount importance is the number of Memory Transactions performed.

Table 89: *Simulation and Hardware FLOPs*

Layer	Ops To Perform	SIM FLOPS	HW FLOPS	Total Mem Trans.	Ratio
CONV1	210249696	2.931 G	2.9530 G	6028	34878.85
CONV2	62332672	128.468 M	113.796 M	217121	287.09
CONV3	13498752	37.156 M	29.106 M	318209	42.42
CONV4	14537088	26.645 M	20.830 M	479617	30.31
CONV5	9691392	26.574 M	20.763 M	321409	30.15
AFFINE1	90701824	176.322 M	113.884 M	782593	115.90
AFFINE2	38797312	87.677 M	38.077 M	1572865	24.67
AFFINE3	94720	33.919 M	20.229 M	7927	11.95

As shown in ***Table 89*** the relationship between the Floating Point Operations to be performed and the Memory Transactions needed can be seen when comparing the ratio between these two aspects and the measured Floating Point Operations Per Second achieved in the design. Despite having the greatest number of operations to perform, the first Convolution Layer was not slowed down by having to perform

too many memory transactions. Therefore, this balance between the operations needed and the memory transactions needed dictated the achievable performance. This relationship can be seen in both ***Figure 99*** and ***Figure 100***. Therefore, it can be stated that the best achievable performance with the design as it is currently configured is approximately 3 GFLOPs. Let's next discuss how this number can be increased.

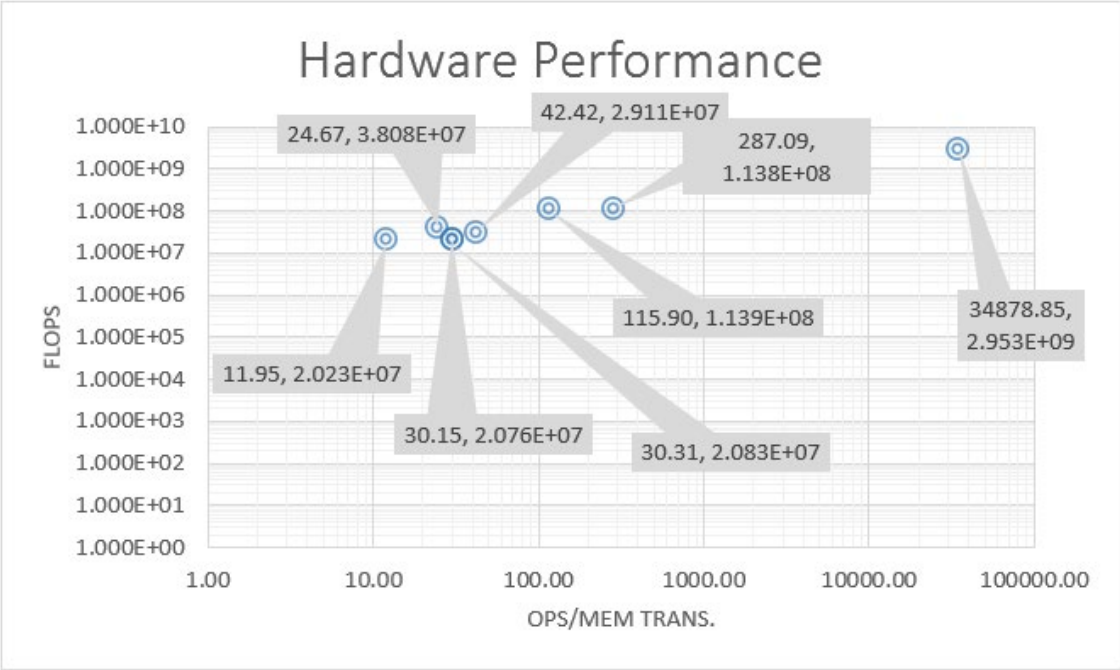

Figure 99: *Hardware Performance vs. Ops/Mem Trans. ratio*

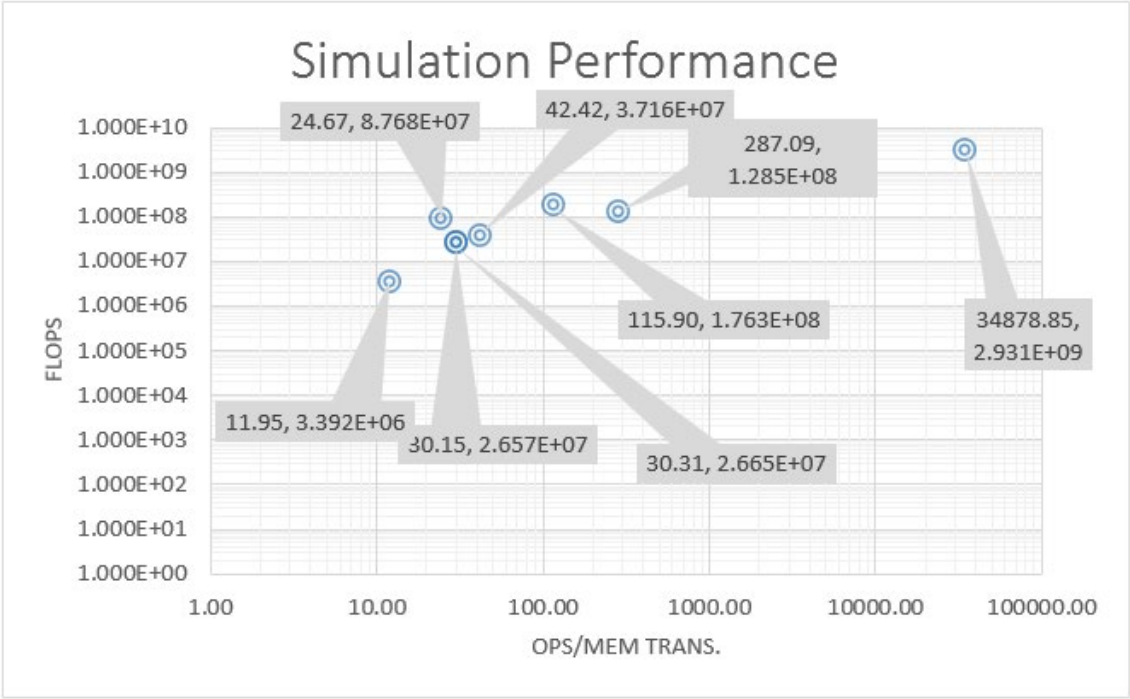

Figure 100: *Simulation Performance vs. Ops/Mem Trans. ratio*

These derived relationships from observations about this design exactly coincide with the reasoning of CNN system behavior from a joint research collaboration between UCLA and Peking University in China. This research group also saw in their FPGA CNN implementations the relationship between the Floating-Point Operations in the system needing to be done, the number of memory transactions, and the achieved FLOPs performance. They brought over an already existing modeling scheme from Multicore Processing and applied it to FPGA CNN systems. The Roofline Model, as its originators from UC Berkeley called it relates system performance to off-chip memory traffic and the peak performance provided by the hardware platform. (Zhang, C. et. al., 2015)

$$Attainable\ Perf. = min\begin{cases} Computational\ Roof \\ CTC\ Ratio\ \times BW \end{cases}$$

The equation formulates the attainable throughput of an application on a specific hardware platform. Floating-point performance (GFLOPS) is used as the metric of throughput. The actual floating-point performance of an application kernel can be no higher than the minimum value of two terms. The first term describes the peak floating-point throughput provided by all available computation resources in the system, or computational roof. Operations per DRAM traffic, or computation to communication (CTC) ratio, features the DRAM traffic needed by a kernel in a specific system implementation. The second term bounds the maximum floating-point performance that the memory system can support fora given computation to communication ratio. (Zhang, C. et. al., 2015)

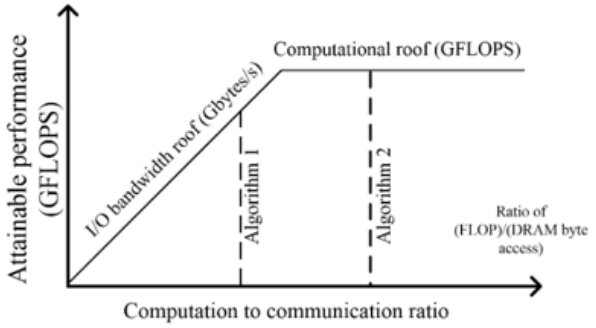

Figure 101: *Basis of the roofline model.*
(Zhang, C. et. al., 2015)

Figure 101 visualizes the roofline model with computational roof and I/O bandwidth roof. Algorithm 2 in the figure has higher computation to communication ratio, or better data reuse, compared to Algorithm 1. From the figure, we can see that by fully utilizing all hardware computation resources, Algorithm 2 can outperform Algorithm 1, in which computation resources are under-utilized because of the inefficient off-chip communication.

12.6.3 Methods of Improvement

This implementation of a Convolutional Neural Network in an AlexNet configuration is a first pass attempt and leave a lot room for improvement and optimization. There are a few ways the performance of this implementation can be increased which would be areas for future work. Looking at ***Table 90***, we can see the differences in resource utilization and performance between other recent works and this one. Although this implementation achieved a lower amount of GLOPs performance, the number of chip resources is far lower than any of the other implementations. Also, the estimated power consumption is far lower.

Table 90: *Comparison of other works to this work.*

	Guo, K. et. al., 2016	Ma, Y et. al., 2016	Zhang, C. et. al., 2015	Qiao, Y. et. al., 2016	This Work
FPGA	Zynq XC7Z045	Stratix-V GXA7 FPGA	Virtex7 VX485T	Zynq XC7Z045 x 2	**Artix7 XC7A200T**
Clock Freq	150 MHz	100MHz	100MHz	150MHz	**100MHz**
Data format	16-bit fixed	Fixed(8-16b)	32-bit Float	32-bit Float	**32-bit Float**
Power	9.63 W (measured)	19.5 W (measured)	18.61 W (measured)	14.4 W (measured)	**1.5 W (estimated)**
FF	127653	?	205704	218296	**103610**
LUT	182616	121000	186251	183190	**91865**
BRAM	486	1552	1024	402	**139.50**
DSP	780	256	2240	836	**119**
Performance	187.80 GFLOPS	114.5 GFLOPS	61.62 GFLOPS	77.8 GFLOPS	**2.93 GFLOPS**

12.6.3.1 Increase DSP usage

The current design depends heavily on the Channel Unit structure to perform the Convolution and Fully Connected operations. As it currently stands only 33 Channel Units which use 1 DPS each are instantiated in the design. Increasing the number of DSPs would allow for more channels to be processed at a given

time. More channels per Convolution/Affine operation mean the required number of memory transactions decreases and as can be seen in ***Figure 99*** and ***Figure 100*** this would increase the performance capability of each layer in the neural network. Increasing the DSP usage does not come without increasing other logic block instantiations as well. Therefore, even though the FPGA has around 700 DSPs, not all will likely be able to be used since other FPGA resources would reach 100% utilization before then.

12.6.3.2 Increase Clock Frequency

Another possible way of improving the achievable performance is to increase the clock frequency output by the Phase Locked Loop generating the FPGA system clock. The current design uses a 100MHz system clock frequency. Just doubling the clock frequency could possibly increase the max achievable system performance to 6 GFLOPs. Increasing the clock frequency may require some logic optimization since the logic may not be able to comply with new clock constraints timing requirements.

12.6.3.3 16bit Data Format

A method employed by some other works is to reduce the number of bits used to represent data values throughout the FPGA implementation of the Convolutional Neural Network. As discussed earlier, two other smaller data formats can be used instead of the one used in this work; 16-bit fixed and Half Precision.

12.6.3.4 Design Optimization

Another possible way of improving performance would be to optimize the already existing logic. As was stated earlier, this implementation is a first pass attempt and logic optimization for speed was sacrificed for functionality.

13.0 CONCLUSIONS

Machine Learning and Deep Learning its sub discipline are gaining popularity quickly as we have seen in reviewing the literature and overall state of the art. Machine Learning algorithms have been successfully deployed in a variety of applications such as Natural Language Processing, Optical Character Recognition, and Speech Recognition. Deep Learning particularly is best suited to Computer Vision and Image Recognition tasks. The Convolutional Neural Networks employed in Deep Learning Neural Networks train a set of weights and biases which with each layer of the network learn to recognize key features in an image. In recent years much of the Deep Learning Convolutional Neural Network has been firmly in the realm of Computer Science with much of the computation performed on large Graphical Processing Unit cards in desktop computer towers. However, as it was seen in the literature, GPUs while effective at processing large amounts of image data, are extremely power hungry. Current FPGA implementations have mostly seemed to concern themselves with acceleration of the Convolutional Layer only and specifically designed their structure to have a defined number of layers.

Therefore, this work set out to develop a scalable and modular FPGA implementation for Convolutional Neural Networks. It was the objective of this work to attempt to develop a system which could be configured to run as many layers as desired and test it using a currently defined CNN configuration, AlexNet. This type of system would allow a developer to scale a design to fit any size of FPGA from the most inexpensive to the costliest cutting-edge chip on the market.

Overall, the thesis of this work was proven to be correct that a system such as this can be developed for a small Embedded System. The objective of this work was achieved, and all layers were accelerated including Convolution, Affine, ReLu, Max Pool, and Softmax layers. The performance of the design was assessed, and it was determined its maximum achievable performance was approximately 3 GFLOPS. While a far cry from the most cutting-edge GPU implementation, this design was a first attempt and this design can be optimized using several approaches which could be the subject of a future work.

14.0 REFERENCES

1. Mitchell, T. (1997). Machine Learning (McGraw-Hill series in computer science).
2. Mohri, Mehryar, et al. Foundations of Machine Learning, MIT Press, 2014. ProQuest Ebook Central, http://ebookcentral.proquest.com/lib/csupomona/detail.action?docID=3339482. Created from csupomona on 2018-04-11 06:40:35.
3. Murnane, K. (Apr. 01, 2016). What Is Deep Learning And How Is It Useful?. Forbes.com. Retrieved from https://www.forbes.com/sites/kevinmurnane/2016/04/01/what-is-deep-learning-and-how-is-it-useful/#302bbf5ed547
4. Siegel, E. (Apr. 07, 2018). *Twelve Hot Deep Learning Applications Featured at Deep Learning World*. Retrieved from https://www.predictiveanalyticsworld.com/patimes/twelve-hot-deep-learning-applications-featured-at-deep-learning-world/9454/
5. Olga Russakovsky*, Jia Deng*, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei. (* = equal contribution) ImageNet Large Scale Visual Recognition Challenge. IJCV, 2015.
6. Christiane Fellbaum (1998, ed.) WordNet: An Electronic Lexical Database. Cambridge, MA: MIT Press.
7. Krizhevsky, A., Sutskever, I., & Hinton, G. (2012). ImageNet Classification with Deep Convolutional Neural Networks. Advances in Neural Information Processing Systems, 25(NIPS 2012). Retrieved from https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf
8. Zisserman, A. & Simonyan, K. (2014). Very Deep Convolutional Networks For Large-Scale Image Recognition. Retrieved from https://arxiv.org/pdf/1409.1556.pdf
9. He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. Retrieved from https://arxiv.org/pdf/1512.03385.pdf
10. Goodfellow, I., & Bengio, Y., & Courville, A., (2016). Convolutional Networks. In Dietterich, T., (Ed.), Deep Learning(326-339). Cambridge, Massachusetts: The MIT Press.
11. Ma, Y., & Suda, N., & Cao, Y., & Seo, J., & Vrudhula, S. (Sept. 29, 2016). Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA. International

Conference on Field Programmable Logic and Applications (FPL), 26th, Session S5b-Compilation. doi:10.1109/FPL.2016.7577356

12. Zhang, C., & Li, P., & Sun, G., & Guan, Y., & Xiao, B., & Cong, J. (Feb. 22, 2015). Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. Proceedings of the ACM/SIGDA International Symposium on field-programmable gate arrays, 2015, pp.161-170. doi: 10.1145/2684746.2689060

13. Ahn, B. (Oct. 01, 2015). Real-time video object recognition using convolutional neural network. International Joint Conference on Neural Networks (IJCNN), 2015. doi: 10.1109/IJCNN.2015.7280718

14. Li, H., & Zhang, Z., & Yang, J., & Liu, L., & Wu, N. (Nov. 6, 2015). A novel vision chip architecture for image recognition based on convolutional neural network. IEEE 2015 International Conference on ASIC (ASICON), 11th. doi: 10.1109/ASICON.2015.7516878

15. Motamedi, M., & Gysel, P., & Akella, V., & Ghiasi, S. (Jan. 28, 2016). Design space exploration of FPGA-based Deep Convolutional Neural Networks. 2016 Asia and South Pacific Design Automation Conference (ASP-DAC), 21st, pp.575-580. doi: 10.1109/ASPDAC.2016.7428073

16. Lacey, G., & Taylor, G., & Areibi, S., (Feb. 13, 2016). Deep Learning on FPGAs: Past, Present, and Future. Cornell University Library. https://arxiv.org/abs/1602.04283

17. Dundar, A., & Jin, J., & Martini, B., & Culurciello, E. (Apr. 08, 2016). Embedded Streaming Deep Neural Networks Accelerator With Applications. IEEE Transactions on Neural Networks and Learning Systems, Vol. 28, pp.1572-1583. doi: 10.1109/TNNLS.2016.2545298

18. Qiao, Y., & Shen, J., & Xiao, T., & Yang, Q., & Wen, M., & Zhang, C. (May 06, 2016). FPGA-accelerated deep convolutional neural networks for high throughput and energy efficiency. Concurrency and Computation Practice and Experience. John Wiley & Sons Ltd.

19. Guo, K., & Sui, L., & Qiu, J., & Yao, S., & Han, S., & Wang, Y., & Yang, H. (July. 13, 2016). Angel-Eye: A Complete Design Flow for Mapping CNN onto Customized Hardware. IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2016, pp.24-29. doi: 10.1109/ISVLSI.2016.129

20. Zhu, M., & Liu, L., & Wang, C., & Xie, Y. (Jun. 20, 2016). CNNLab: a Novel Parallel Framework for Neural Networks using GPU and FPGA-a Practical Study with Trade-off Analysis. Cornell University Library. https://arxiv.org/abs/1606.06234

21. Li, F., et. al. Image Classification Pipeline [PDF document]. Retrieved from Lecture Notes Online Website: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture2.pdf

22. Li, F., et. al. Linear Classification [HTML]. Retrieved from Lecture Notes Online Website: http://cs231n.github.io/linear-classify/

23. Li, F., et. al. Loss Functions and Optimization [PDF document]. Retrieved from Lecture Notes Online Website: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture3.pdf

24. Li, F., et. al. Backpropagation and Neural Networks [PDF document]. Retrieved from Lecture Notes Online Website: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture4.pdf

25. Li, F., et. al. Convolutional Neural Networks [PDF document]. Retrieved from Lecture Notes Online Website: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf

26. Li, F., et. al. Training Neural Networks [PDF document]. Retrieved from Lecture Notes Online Website: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture6.pdf

27. Li, F., et. al. Neural Networks [HTML]. Retrieved from Lecture Notes Online Website: http://cs231n.github.io/neural-networks-1/

28. Li, F., et. al. Deep Learning Software [PDF document]. Retrieved from Lecture Notes Online Website: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture8.pdf

29. Li, F., et. al. CNN Architectures [PDF document]. Retrieved from Lecture Notes Online Website: http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf

30. Xilinx (2018). 7 Series DSP48E1 Slice [User Guide UG479]. Retrieved from https://www.xilinx.com/support/documentation/user_guides/ug479_7Series_DSP48E1.pdf

31. Weisstein, Eric W. "Maclaurin Series." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/MaclaurinSeries.html

32. Cormen, et.al. (2009). *Introduction to Algorithms, 3rd ed.* Cambridge, Massachusetts: The MIT Press.

33. Brown, S. & Vranesic, Z. (2002). *Fundamentals of Digital Logic with Verilog Design, 1st ed.* New York, New York: McGraw-Hill Higher Education.

34. Xilinx (2017). Zynq-7000 All Programmable SoC Family Product Tables and Product Selection Guide. Retrieved from https://www.xilinx.com/support/documentation/selection-guides/zynq-7000-product-selection-guide.pdf

35. Xilinx (2018). All Programmable 7 Series Product Selection Guide. Retrieved from https://www.xilinx.com/support/documentation/selection-guides/7-series-product-selection-guide.pdf

36. ARM (2018). AMBA AXI and ACE Protocol Specification. Retrieved from http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf

37. IEEE Computer Society (2008). IEEE Standard for Floating –Point Aritmetic (IEEE Std 754) [Redlines]. Retrieved from https://ieeexplore-ieee-org.proxy.library.cpp.edu/stamp/stamp.jsp?tp=&arnumber=5976968

APPENDIX – Links to Repositories

The links to the Github repositories are provided here and serve as the appendix for this work due to large amount of content they embody.

https://github.com/maespinosa/Thesis_VHDL
https://github.com/maespinosa/Thesis_Python
https://github.com/maespinosa/Thesis_Documents
https://github.com/maespinosa/Thesis_Matlab_Models