2019-01-01

Dedicated Hardware for Machine/Deep Learning: Domain Specific Architectures

Angel Izael Solis
University of Texas at El Paso, aisolis@miners.utep.edu

DEDICATED HARDWARE FOR MACHINE/DEEP LEARNING: DOMAIN SPECIFIC ARCHITECTURES

ANGEL IZAEL SOLIS

Master's Program in Computer Engineering

APPROVED:

Patricia A. Nava, Ph.D.

Martine Ceberio, Ph.D.

John A. Moya, Ph.D.

Charles Ambler, Ph.D.
Dean of the Graduate School

DEDICATION

As a child my father was a truck driver and on his long trips on the road, he would play the radio often and listen to reflections. This is the one that has stuck with me my entire life…

La vida es en muchas maneras como un tren,
Está llena de experiencias y gente, como igual un tren está llena de estaciones,
a veces hay cambios de vía y accidentes.

Cuando nace unas de las primeras personas que conoce en su tren son sus padres.
Luego suben otros, amigos, conocidos, y si uno es tan afortunado, hasta el amor de su vida.
¡La cosa más loca es que a veces ni nos damos cuenta de cuando alguien se sube y baja!
Vienen con tanta prisa que ni nos damos cuenta de esto.

Pero la cosa que nos da más miedo a todos es que nunca sabemos con certeza cuando un querido se bajara de nuestro tren, o cuando nos toque bajar a nosotros.
Si seré sincero, yo en muchas ocasiones me eh querido bajar voluntariamente de este tren…

¡Pero aseguremos!
Que cuando ya nos toqué bajar, que dejemos memorias y recuerdos bonitos para los que continúen su pasaje…

Este proyecto es para todos aquellos que se han bajado de mi tren, todos aquellos que viajan conmigo ahorita, y todos aquellos todavía por subirse…

DEDICATED HARDWARE FOR MACHINE/DEEP LEARNING DOMAIN SPECIFIC ARCHITECTURES

by

ANGEL IZAEL SOLIS, B.S.E.E

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Department of Electrical and Computer Engineering

THE UNIVERSITY OF TEXAS AT EL PASO

MAY 2019

ACKNOWLEDGEMENTS

This project would have certainly not been possible without the tremendous help and support Dr. Patricia Nava has provided throught my entire professional career. She has helped me financially both directly and indirectly with countless letters of recommendation. She has not only helped me financially though; she has sat with me through countless hours of professional career advice, transferring years of so much knowledge and experience. Words cannot express how grateful both myself and my family is with Dr. Nava… Gracias.

This project would also certainly not have been completed without the support of Miroslava Barua. Admittedly going into this project, I was very unexperienced with using the Xilinx tool (used as the primary designing tool in this project!) but Miroslava has been so patient sitting with me for hours explaining how this tool works and behaves. Not only as a tutor has Miroslava helped the development of this project but she has helped me also when I have come to a design/implementation problem. I am truly grateful to have a peer, mentor and friend like Miroslava.

The following two acknowledgements are with the departments of Electrical and Computer Engineering and Computer Science here at the University of Texas at El Paso. The EE department has made resources available to me that have greatly heightened my experience here at UTEP and resources that without, this project would have been nearly impossible. I am very grateful to the Computer Science department for the software engineering background they instilled in me which extremely helped in the development of this project.

Lastly, I would like to thank all the organizations that helped me financially throughout my undergraduate and graduate career. The National Science Foundation, the STARS foundation, the Texas Space Grant Consortium, Microsoft Corporation, Texas Instruments and Lockheed Martin.

ABSTRACT

Artificial intelligence has come a very long way from being a mere spectacle on the silver screen in the 1920s [Hml18]. As artificial intelligence continues to evolve, and we begin to develop more sophisticated Artificial Neural Networks, the need for specialized and more efficient machines (less computational strain while maintaining the same performance results) becomes increasingly evident. Though these "new" techniques, such as Multilayer Perceptron's, Convolutional Neural Networks and Recurrent Neural Networks, may seem as if they are on the cutting edge of technology, many of these ideas are over 60 years old! However, many of these earlier models, at the time of their respective introduction, either lacked algorithmic sophistication (the very early McCulloch and Pitts, and Rosenblatt Perceptron), or suffered from insufficient training data to allow effective learning of a solution. Now, however, we are in the era of Big Data and the Internet of Things, where we have everything from autonomous vehicles to smart toilets that have Amazon's Alexa integration within them. This has given an incredible rise to sophistication of these new Artificial Neural Networks. This increase in sophistication has come at an expense of high computational complexity. Though traditional CPUs and GPUs have been the modern "go-to" processors for these types of applications, there has been an increasing interest in developing specialized hardware that not only speeds up these computations, but also does it in the most energy efficient manner.

The objective of this thesis is to provide the reader with a clear understanding of a subdiscipline in artificial intelligence, Artificial Neural Networks, also referenced as Multilayer Perceptron's or Deep Neural Networks; current challenges and opportunities within the Deep Learning field; and a coverage of proposed Domain Specific Architectures [Hen17] that aim at optimizing the type of computations being performed in the Artificial Neural Networks and the way data is moved throughout the processor in order to increase energy efficiency. The Domain

Specific Architecture guidelines utilized in this study are: investment of dedicated memories close to the processor, µ-architectural optimizations, leveraging the easiest form of parallelism, reduction of data size, and designing the Domain Specific Architecture to the domain specific language.

This study has managed to leverage four out of the five Domain Specific Architecture design guidelines. We have leveraged the use of dedicated memories and µ-architectural optimizations by building dedicated Functional Units which have their own dedicated memories and specialized multiplication hardware. We also leveraged the use of the easiest form of parallelism by using a Spatial Architecture, as opposed to the traditional Temporal Architecture. Specifically, the Temporal Architecture operates as a Systolic Array. We have also investigated the use of a newly proposed "mid-precision" floating point representation of data values, which consists of 12 bits in parallel, and is based on the IEEE 754 standard of "half-precision" which uses 16 bits.

The organization of this paper is as follows: first, we cover a brief history lesson of artificial intelligence, machine learning, and Artificial Neural Networks; next we go into a much more comprehensive background study of these algorithms, their origin, and some of their modern applications; a history of computer architecture and its different classifications; the Domain Specific Architecture guidelines; and the approach to the proposed DSA design. Next, we discuss the specific problems in the primary areas of study needed to build this Domain Specific Architecture, which includes a discussion of the test-bed design and results of the proposed design. A conclusion for the study, as well as discussion of future work are given in the final section.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF EQUATIONS

LIST OF TABLES

1 INTRODUCTION

"Artificial intelligence is not only the next big wave in computing — it's the next major turning point in human history… the Intelligence Revolution will be driven by data, neural networks and computing power." – Brian Krzanich (Intel CEO)

Although Machine Learning and Deep Learning are thought to be at the cutting edge of Artificial Intelligence techniques, the idea of these models and systems has been around since the 1940's with the McCulloch and Pitts model and Rosenblatt perceptron [Rea17]. Both models imitated the behavior of the brain, in a rudimentary fashion, using only hardware. These ideas and projects were, however, quickly attacked by other Artificial Intelligence researchers in the field. Minsky and Pappert argued that these models (McCulloch and Pitts, and Rosenblatt Perceptron) were oversimplifying the complicated tasks Artificial Intelligence was trying to solve. They advanced this attack on Machine Learning by verifying that these early models could not solve non-linearly separable problems, which are embodied by the XOR problem, as shown in Figure 1.1(b). Figure 1.1 illustrates the difference between a linearly separable problem and a non-linearly separable problem.

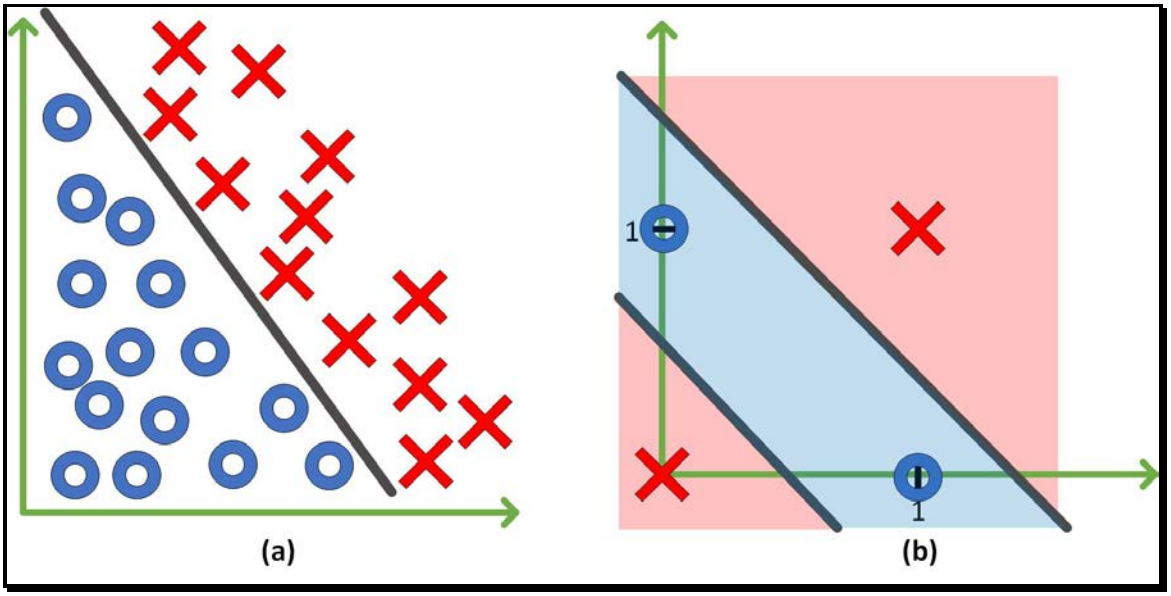

Figure 1.1: Linearly Separable Problem (a), XOR Problem (b)

Roughly 30 years later, parallel and distributed computing gave rise to the second wave of Machine Learning. The increase in computational resources made techniques like backpropagation feasible to implement. These newer and more sophisticated techniques allowed Machine Learning to tackle a wider variety of problems. Unfortunately, too many unmet promises due to the stagnant availability of datasets resulted in the second fall of Machine Learning. However, this was not all bad. This second historical decline in popularity was able to produce architectures like Convolutional Neural Networks that are now heavily used in day-to-day applications like image processing and facial recognition.

Fortunately, the era of Big Data and the Internet of Things (IoT) has given Machine Learning and Deep Learning a place in cutting edge technology again. The recent rise in data being produced from social media, online media (photos, videos, music), e-mails, online transactions, etc. has given birth to the third wave of Machine Learning and Deep Learning. This rise in available data has forced computer scientists to research and develop new techniques, models, and architectures to tackle both new and old problems. However, the development of these new algorithms has now shifted the pressure on computer engineers to develop or optimize existing

hardware to tackle these demanding new algorithms. Computer engineers are now pressured to optimize existing architectures such as Graphics Processing Units (GPUs) and develop Domain Specific Architectures (DSAs), like Google's Tensor Processing Unit (TPU), Microsoft's Catapult I and II, and Intel's Crest. This cycle has been named the "Virtuous Cycle" [Rea17] and is shown in Figure 1.2.

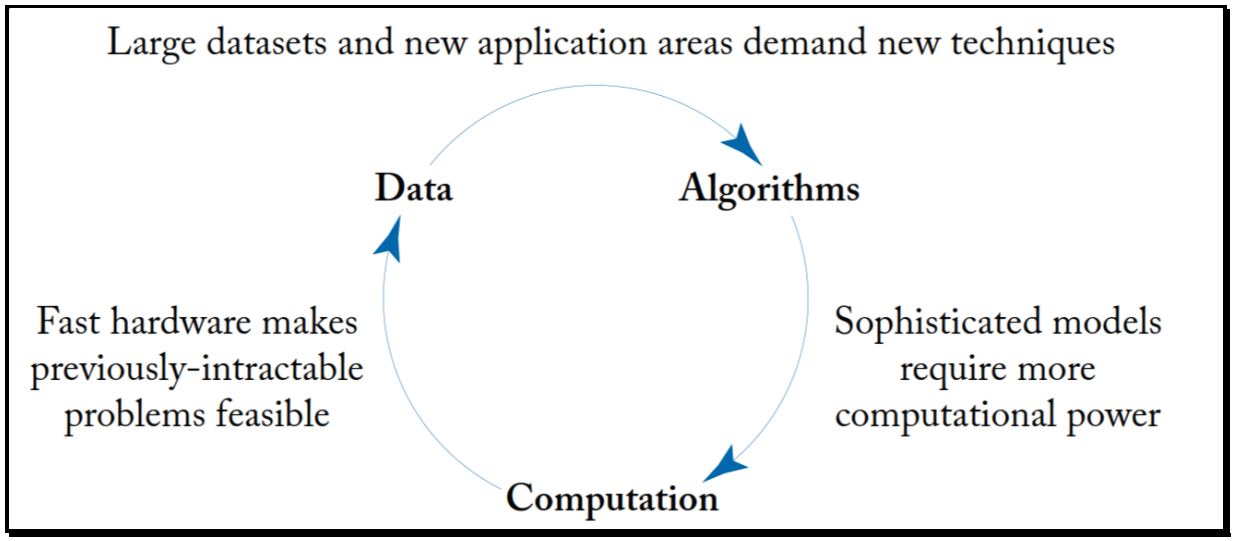

Figure 1.2: The "Virtuous Cycle"
Reagen, Brandon, et al. Deep Learning for Computer Architects [Rea17]

Those familiar with computational trends in Machine Learning and Deep Learning might question the necessity of these DSAs as GPUs become increasingly affordable and programmable. The straightforward answer is these specialized architectures can offer better overall performance by offering lower latency, shorter processing time, and faster decisions, at lower power consumption cost. Figure 1.3 [Rea17] illustrates how these specialized solutions via either Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuit (ASIC) offer solutions that have less prediction error with lower power consumption in comparison to traditional GPUs and CPUs.

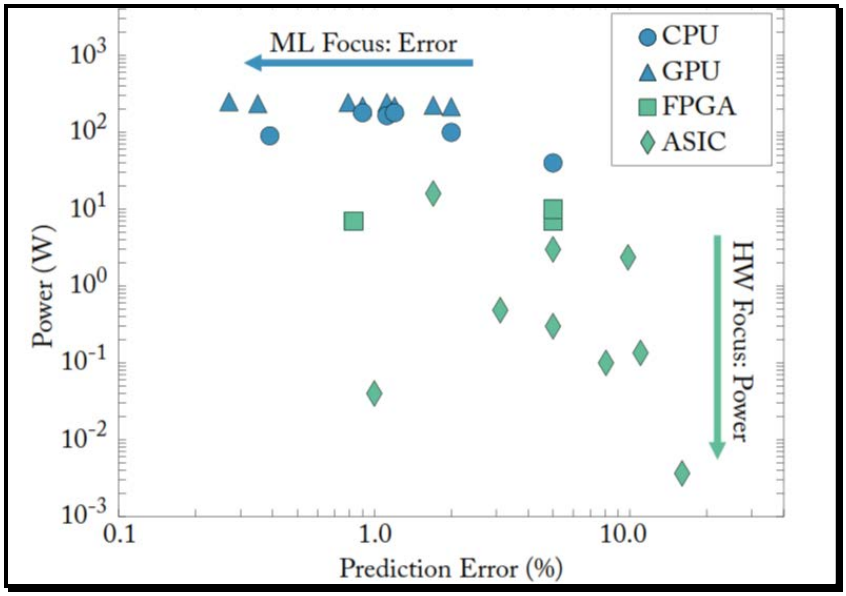

Figure 1.2: Literature survey illustrating Machine Learning focus on lower prediction error, while Hardware focus is on lower power consumption [Rea17].

The emphasis of this work is to develop a DSA that is able to accelerate the learning process for these Machine Learning and Deep Learning algorithms. The proposed architecture's performance is then compared with a traditional CPU and GPU. The organization of this work is as follows: background information will be provided on Artificial Neural Networks and Computer Architecture along with related work in the field; the problem, the work, and the results found during this investigation will be discussed; and finally, a conclusion will be presented and final recommendations for this project will be offered.

2 BACKGROUND

2.1 STUDY OF THE BRAIN

The Artificial Neural Network (ANN) was originally inspired by the intricate interconnections of our very own brains. These connections are sometimes also referred to as our "neural pathways." Although we still have a long way to go before, we fully understand the brain, we can, however, use what we do know in order model systems that can solve problems that may seem simple or intuitive to humans but are inherently difficult for a computer. Some of the problems that fall into this category can be, for example, natural language processing, speech recognition, object or pattern recognition and classification to name just a few. But the reader may wonder, why *are* these problems so difficult for a machine to solve? We will delve into this a little more later but to keep the explanation brief and simple, these types of problems have parameters that do not easily translate to values that a computer can understand, or represent, in terms of 0 or 1. In Figure 2.1, a biological neural network diagram is presented for observation, where in comparison to Figure 2.2, shows an example of an *artificial neuron*, which will be one of the two foci of this work.

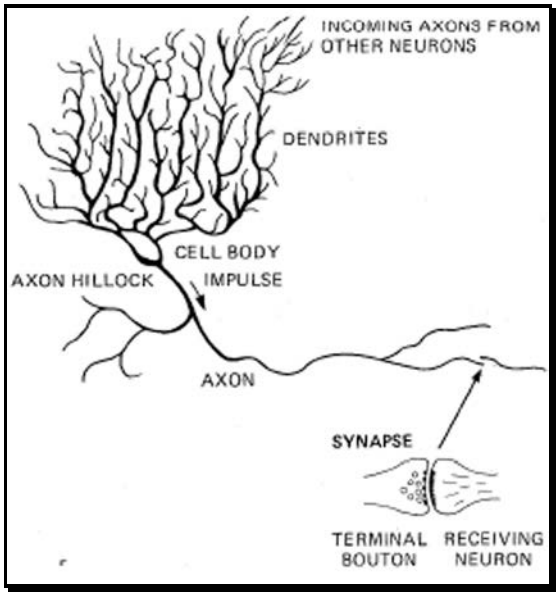

Figure 2.3: Biological Neuron,
Dayhoff, Neural Network Architectures: An Introduction [Day90]

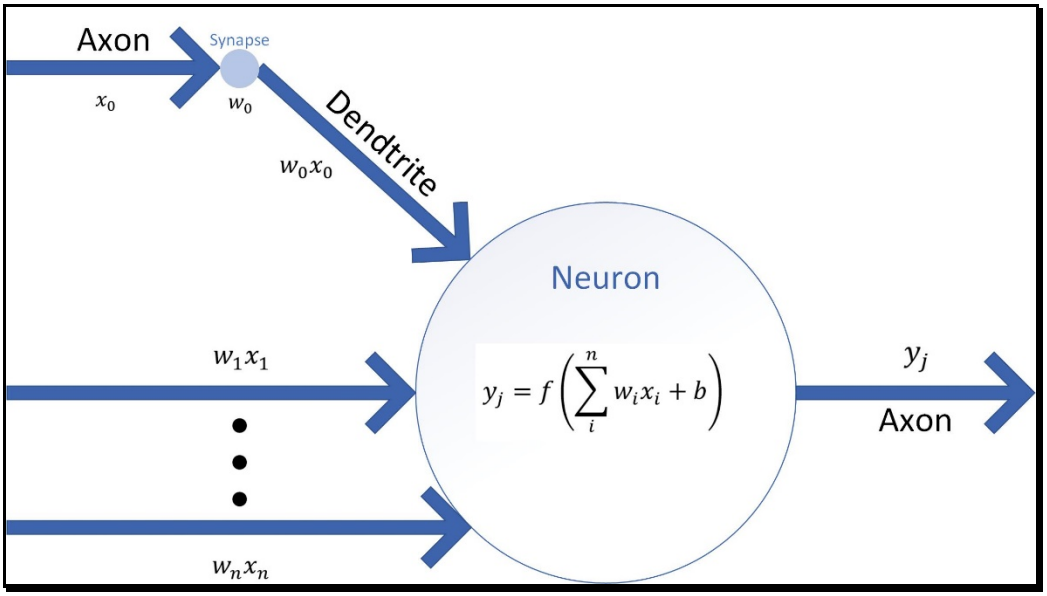

Figure 4.2: Artificial Neuron Architecture (ANN)

As illustrated in Figure 2.1, the organic nerve cell is observed to be composed of dendrites, axons, the cell body, and synapses connecting the dendrites and axons. The dendrites serve as transportation for incoming electrical impulses into a neuron. The axon is the output transportation

medium for a neuron while the point of contact for these two structures is called the synapse. The neuron is in charge of processing all incoming inputs from its dendrites and must decide whether to "fire" an output, based on the "strength" these inputs. This operation is translated to an Artificial Neural by representing each one of the previously mentioned structures in mathematical form. The cell body is simulated by a Multiply and Accumulate (MAC) function, where all inputs are multiplied by a weight and then summed, in a fashion imitating the "strength" of the biological neuron's inputs. The neuron decision of whether to "fire" is simulated by implementing a non-linear function, some of the most popular and traditional activation functions are shown in Figure 2.7. The dendrites and axons are represented by inputs and outputs, accordingly. The synapse or the "strength" of these electrical impulses is simulated by applying a weight to said inputs. Of course, the brain is not composed of a single neuron and our Machine Learning/ Deep Learning (ML/DL) models are not composed of a single Artificial Neuron. Figure 2.3 illustrates the similarities between the human brain and our developed ML/DL models.

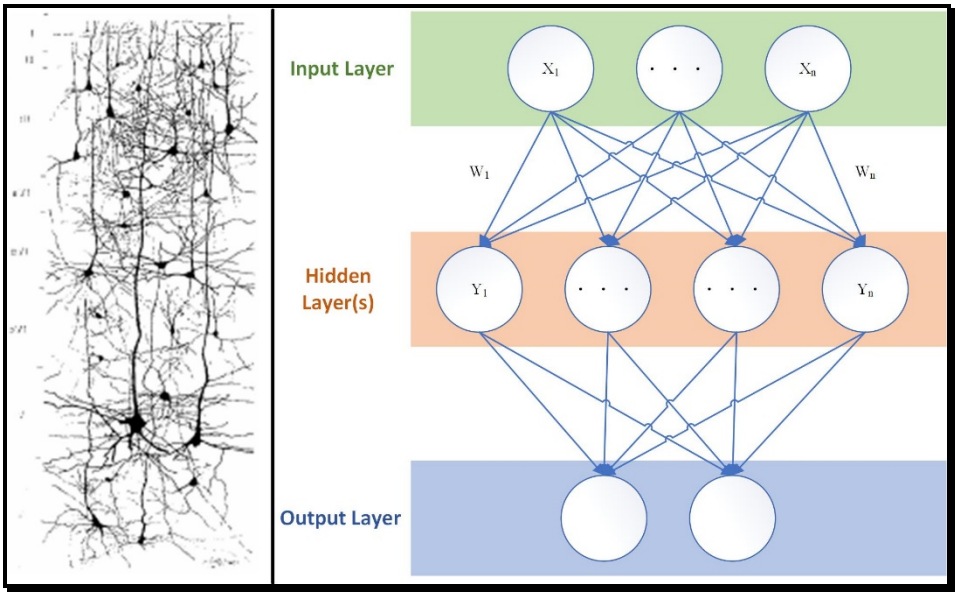

Figure 2.5: Comparison of Biological Neural Network (left figure) and Artificial Neural Network (right figure)

2.2 ARTIFICIAL INTELLIGENCE

Figure 2.4 illustrates the taxonomy of Artificial Intelligence (AI), Machine Learning (ML), Neural Networks (NN) and Deep Learning (DL). In the 1950s, a computer scientist named John McCarthy defined AI as the "science and engineering of creating intelligent machines that have the ability to achieve goals like humans do" [Sze17].

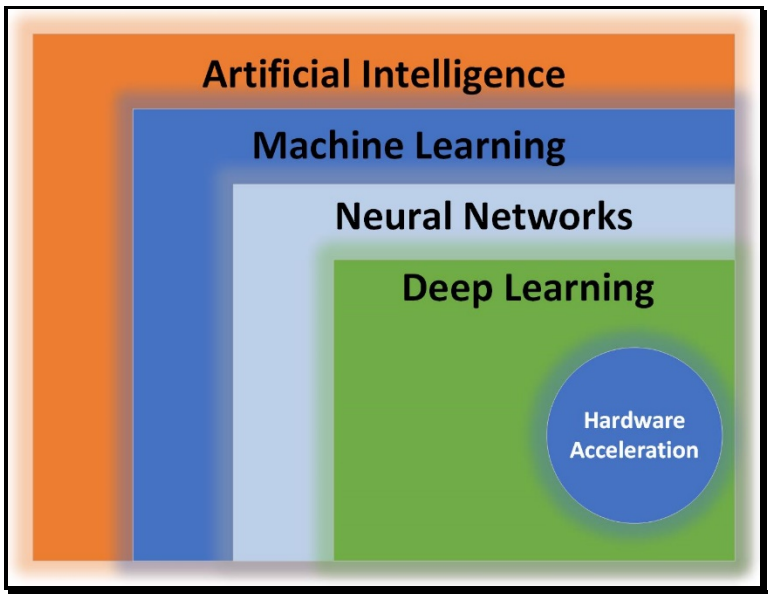

Figure 2.6: Deep Learning presented as a subfield of Artificial Intelligence

In its very early days, AI was good at solving computationally difficult problems that involved considerable logic, just as long as the problem could be modeled mathematically. The true challenge came from doing humanly intuitive tasks, such as facial recognition or language processing. It was in 1959 when Arthur Samuel defined ML as "the field of study that gives computers the ability to learn without being explicitly programmed to do so" [Sze17]. MD/DL is built upon the concept that the programmer does not have to explicitly program the machine to solve the problem, but the concept that if the machine is presented with enough knowledge and experience, it will be able to extract and learn the complicated features from simpler ones. At this point, the reader might ask, "what is the difference between Machine Learning and Deep

Learning?" The simplest answer is that Machine Learning is the art of designing and training a *computer* to do some task *intelligently*, using primarily statistical and probabilistic methods. Deep Learning is a subset of Artificial Neural Networks that employs the notion of a model that utilizes a very specific description of "neuron" behavior, and has more than one hidden layer composed of multiple neurons. These models are sometimes also referred to as Multilayer Perceptron's, or MLPs (but the marketing department thought Deep Learning sounded "cooler"). Now the reader might have a new question, "why would we want to use more than one hidden layer?" The reason for designing models with multiple hidden layers is to extract features that are more abstract than those that might be readily observable without the layered approach and its associated mathematical mapping, or transformation, from one multi-dimensional domain to another. For example, if a DL model were used to process image data, an image (of something that requires classification) would be provided as input to the first layer in the model. This layer might then use the pixel values to extract and report (output) lines and edges. The output from this first layer, then, would serve as the input to the next layer. The next layer might use the lines and edges to extract shapes, and report to the next layer, which might extract specific sets of shapes. The final layer might extract entire objects and scenes with this information. The composite of these layers would create a DL network capable of classifying, identifying, and reporting what object(s) are in the original image provided.

These tools have now enabled programmers and—with new services like AT&T Speech, IBM Watson, and Google Prediction—people with little technical knowledge in AI and programming to "program" a computer to learn how to do some intelligent task! The benefit of these tools is giving the user tremendous power without having to create a specialized skill set: instead of having to painfully tailor, adapt, fit, and alter a program to achieve a specific task or

"hard code," it the machine can, with generalized code, now be *trained.* Methods and different techniques for training will be presented later. These techniques will allow the machine to learn how to handle any new problem presented.

Attempts to "hard code" knowledge into the computer, typically referred to as a *knowledge base* approach, have proven to be unsuccessful. One of the most notable projects in this field is the Cyc project [Len90]. This ambitious project worked on the idea of having human operators try and devise formal rules that had just enough complexity to accurately describe the world. One notable failure in the project came when Cyc failed to understand the story of a man shaving in the morning. The man was using an electric razor but Cyc understood humans did not have electric parts, so Cyc asked if the man was still human [Ian17].

ML/DL models are able to make decisions based on some previous experience. These models are now capable enough to make recommendations using a mathematical model called logistic regression, these models are also able to classify items based on the simple Bayes theorem. The problem with these models lies in the fact that they rely heavily on data, and how that data is presented to the model. If the model is not presented with enough information, the system will then not learn properly. However, if the model is saturated with data, the system might "overfit" or not generalize the problem enough to account for any outliers in the data. Figure 2.5 is an illustration of underfitting, appropriate fitting, and overfitting. Note that underfitting, shown in Figure 2.5(a) is characterized by the case where the model has not learned the problem properly, and oversimplifies the decision surface, thereby misclassifying more samples than allowed or tolerated. Appropriate fitting, illustrated in Figure 2.5(b) is where the model has been well trained and produces good, if not perfect, results. Overfitting, shown in Figure 2.5(c), is where the model has been overly trained. Note that the decision surface is overly-complex, obtained by seeking

perfection in classifying the data provided. The recent rise in data availability, also referred to as the era of “big data,” has made ML/DL models increasingly useful and has revitalized this field in computing.

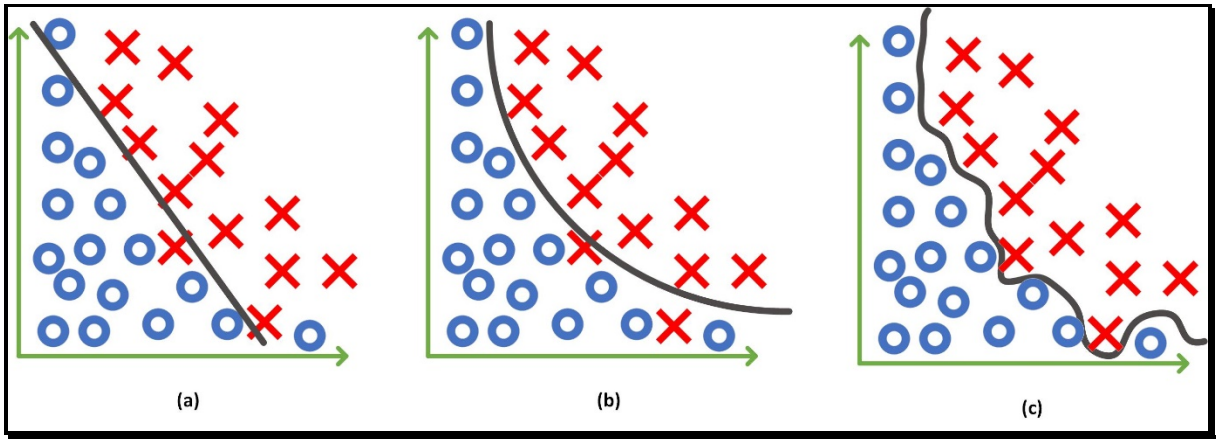

Figure 2.7: Model Training Outcomes: (a) Underfitting, (b) Appropriate Fitting, (c) Overfitting

2.3 ARTIFICIAL NEURAL NETWORKS AND DEEP NEURAL NETWORKS

Inside ML there is a subfield called brain-inspired computation [Sze17]. Because of the brain's ability to (relatively) quickly adapt and learn a new problem, it has been used as a blueprint for this field of study as previously discussed in section 2.1. Experts believe these neurons are the computational element that allows humans to learn, behave, and adapt. What enables this architecture to learn and adapt is the ability to "fine-tune" the weights (denoted as w_{ij} in Figure 2.2) to fit the importance of each individual input. This process of learning or *training*, as commonly known in ML and DL, is the process of iteratively changing and adapting these weights to fit the problem.

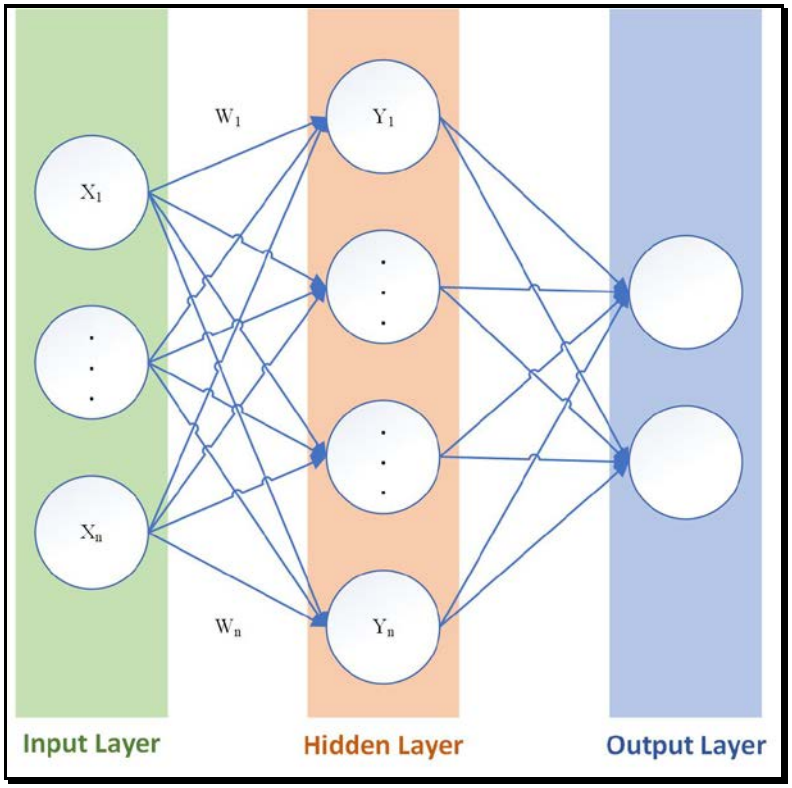

Figure 2.8: Basic Neural Network Topology

Figure 2.6 shows an example of a computational neural network. The neurons in the input layer propagate the values on to the fully interconnected neurons in the hidden layer and a neural

computation is applied to these inputs. In the case of a regular ML architecture, the results are propagated to the output layer. In the case of a DNN architecture, the results are propagated to the next hidden layer until reaching the output layer. In most cases, the number of layers in a DNN can range anywhere from five to more than a thousand layers [Sze17]. Equation 2.1 describes the computation performed at each neuron inside a basic neural network. Essentially the result of this equation is referred to the "output of the neuron." The operation is the result of the summation of each input-weight, multiplied by its corresponding input, or a Multiply and Accumulate (MAC), once that operation is complete, an arbitrary bias is added to each multiplication. The neuron shown in Figure 2.2 serves as an example of this computation performed with each input at each layer.

$$y_j = f\left(\sum_i^n w_i x_i + b\right) \tag{2.1}$$

Equation 2.1: Example Computation Performed at Each Neuron

Once the MAC is completed, this intermediate value goes through an activation function. A concept that is briefly mentioned in section 2.1 is the activation function. This is the function determines if a neuron should "fire." The reason for applying an activation function inside the neurons is to saturate or limit the output. Without these functions, the computation would be simply a weighted sum. These activation functions are nonlinear in nature, thereby able to produce an output only if the inputs surpass some threshold. Common non-linear functions are show in Figure 2.7.

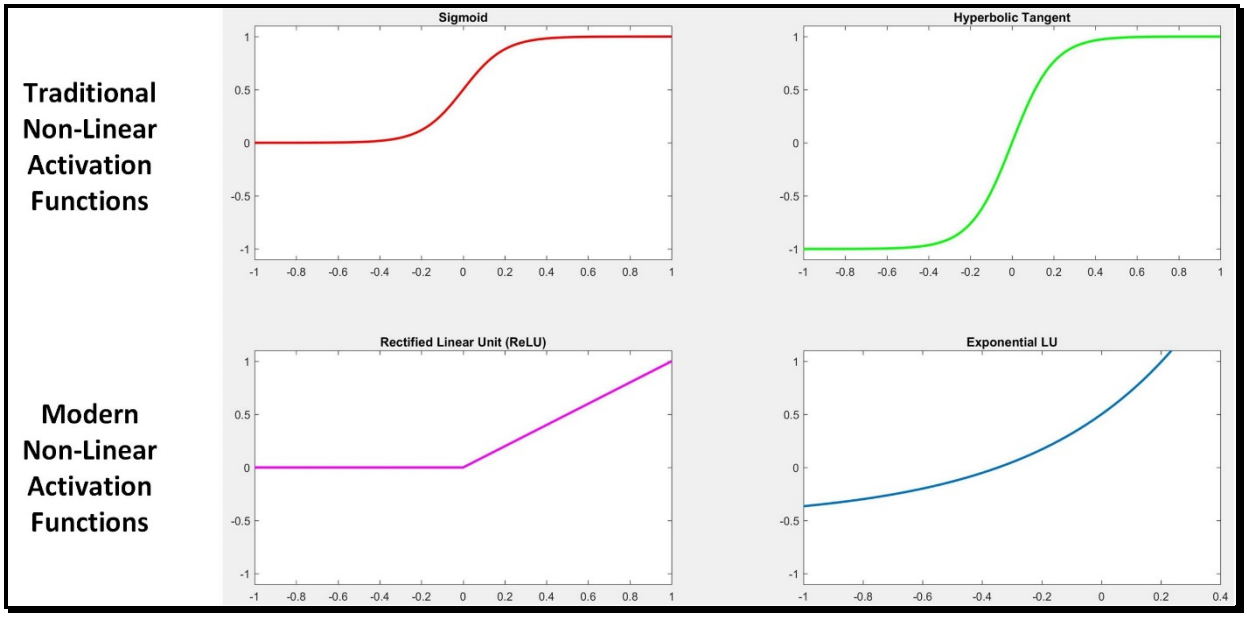

Figure 2.9: Traditional and Non-Traditional Activation Functions

In older ANN models, traditional non-linear activation functions are frequently used. Since then, however, interest has grown greatly in the need for activation functions that produce the same or similar results but are computationally less demanding. This need has led to the development of modern non-linear functions such as the Rectified Linear Unit (ReLU) and the Exponential Linear Unit.

2.4 INFERRING AND TRAINING

Once a neural network is trained, or has learned to perform its designed task, running the neural network, or program, on new and unseen inputs with the previously programmed weights is referred to as "inferring." In this section, image classification is used as an example (illustrated in Figure 2.8) to illustrate *training* and *inferring*.

When a DNN or ANN is used for *inference,* an input image must be supplied. The output of the network is the maximum value in a vector that indicates the probability that the object contained in the image belongs to one of the designated classes [Sze17]. The difference between the calculated probability and 1 is known as the loss (L). The objective of training this model is to modify and adjust the weights in the network to be able to classify a wide variety of objects into the appropriate categories and minimize the *L*.

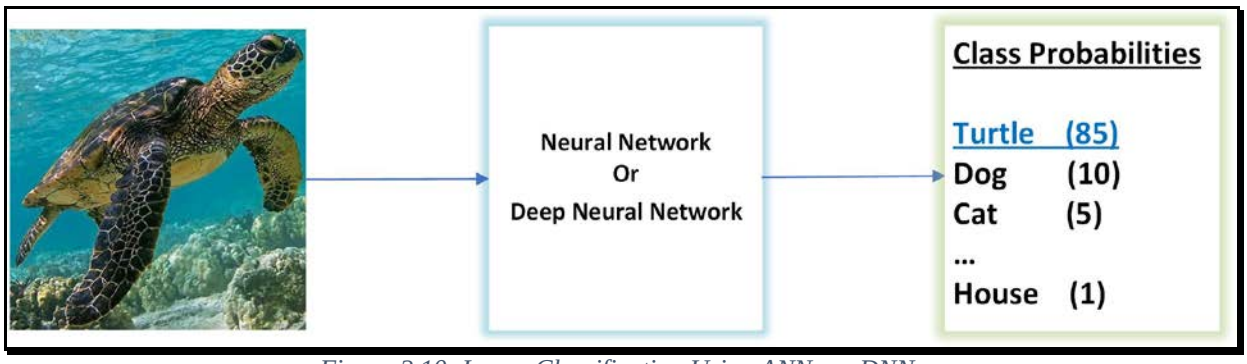

Figure 2.10: Image Classification Using ANNs or DNNs

The most common method to adjust the weights is using a technique called *backpropagation*. Note that the weights are designated as w_{ij}, where the subscript *ij* indicates the direction of the input, *i* denotes the neuron in question, and *j* designates the neuron providing input to the neuron in question. Backpropagation is a mathematical technique that calculates the gradient of the loss relative to each weight, which is the partial derivative of the loss with respect to the weight. This gradient, then, is used to update the weight. Equation 2.2 illustrates the calculations

for each weight, where α is commonly referred to as the learning rate. Alpha (α) is typically a small value in order to allow each individual weight to converge to some value, however not necessarily the same value. This process is repeated iteratively in order to minimize overall loss. Figure 2.9 provides a representation of the backpropagation process.

$$w_{ij}^{t+1} = w_{ij}^{t} - \alpha\left(\frac{\partial L}{\partial w_{ij}}\right) \tag{2.2}$$

Equation 2.2: Computing Gradient Loss with Respect to each Weight

Although backpropagation is one of the more popular techniques to train a neural network, there are two major demerits of the strategy. First, the inherent nature of backpropagation requires the intermediate outputs between layers to be stored in order to calculate the loss between layers. This increases the storage requirements drastically. Second, gradient descent requires a high level of precision, which also drastically increases storage demand alongside power consumption. To mitigate some of these drawbacks, a variety of strategies are employed at the software level in order to improve the efficiency of training. For example, oftentimes the loss from multiple sets of input data is collected before weights are updated (i.e. batch updates) [Sze17]. This is one of many strategies employed to speed up and stabilize the training process.

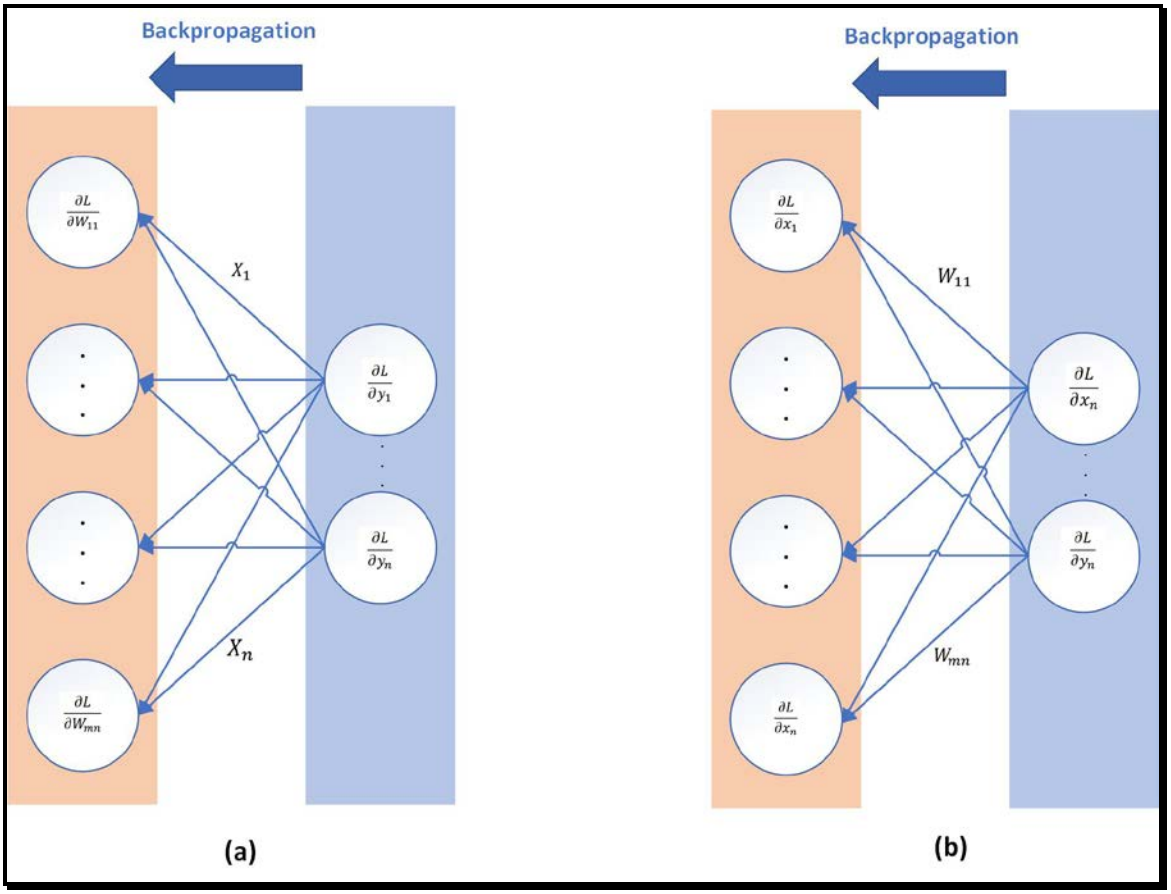

Figure 2.11: Neuron Backpropagation

2.5 LEARNING STRATEGIES

There are several popular strategies to train the network, i.e. determine the weights on inputs to the neurons. Many of the most common strategies fall into a category called *supervised learning*, where each training sample has an associated label (output). This is then compared with the neural network's output, which in turn generates adjusted weights. Many different methods for adjusting the weights exist, but one of the most popular is by using backpropagation. Strategies without labeled training samples fall into the category called unsupervised learning. This technique does not label samples, instead allows the model to create its own classes and classify samples accordingly. This technique is commonly used for pattern recognition, clustering and denoising data [Ian17]. *Semi-supervised learning* is a combination of both supervised and unsupervised learning, where only some or few samples are labeled and the model is left to create appropriate classes for the remaining ones. Another common approach to obtaining network parameters, or weights, is called *fine-tuning*, where the weights and architecture of a previous model are used as a start and are then adjusted to the new data set. This approach typically results in quicker training with higher precision.

2.6 ADVANCED NEURAL NETWORK ARCHITECTURES

The Deep Neural Networks discussed previously can theoretically learn any relationship presented [Rea17]. However, through research and development, more sophisticated models have been developed in order to target specific problems. These models typically offer a reduction in computational intensity by being easier to train and more robust to noise in the training data compared to traditional Deep Neural Networks.

2.6.1 Convolutional Neural Networks (CNNs)

"Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers" [Ian17]. These types of networks have been known to excel in areas that involve image processing such as facial or object recognition and detection. The reason they excel at this task is due to their excellent capability of recognizing relationships between adjacent pixels in an image. In a traditional DNN, the network would treat each pixel as exactly that, an individual pixel having initially little to no relationship with its neighboring pixels. However, from basic intuition we know that is false: pixels in an image or video generally have a heavy relationship with its neighboring pixels. CNNs are great at recognizing these types of relationships because of the very nature of the *convolution* operation, a mathematical operation that is essentially a moving weighted average across the provided data.

Inferring in this type of network is computed by creating, training then applying several small filters across the image, the result of each convolutional layer is typically referred to as a *feature map* (fmap). The way these networks are trained is similar to the training process in DNNs, gradient descent is applied for each location where the filter is used and followed by summing the result into a single value. An illustration of one convolutional layer is shown in Figure 2.10, which

illustrates how multiple filters are applied according the dimensionality of the input image. For example, images are typically represented by their Red, Green, Blue (RGB) value, so in this case we would apply one filter per dimension; one for the Red pixel values, one for the Green pixel values and another for the blue pixel values. We would find the weighted average for the black square, shift over according to a certain stride size and repeat this process until we have reached the end of the image.

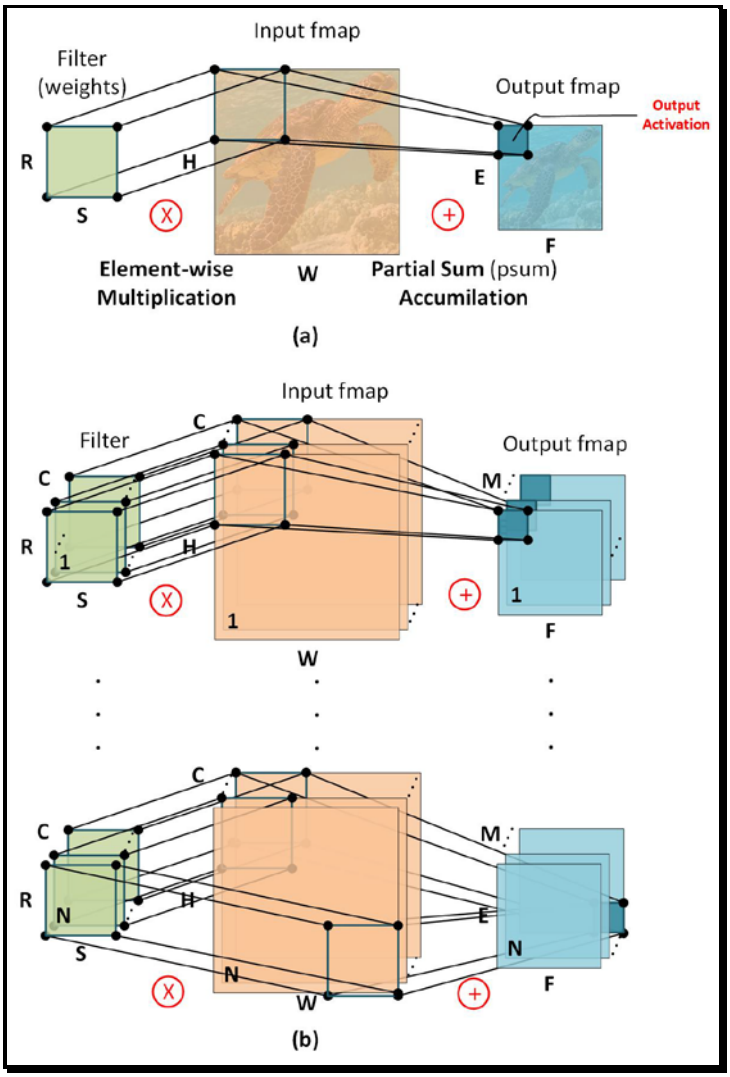

Figure 2.12: Representation of Convolutional Neural Networks

A common type of optimization seen in these types of networks is applying intermediate layers that "normalize" the output from these convolutional layers. These layers can be *nonlinearity layers*, which apply nonlinear activation functions as seen in section 2.3or *pooling layers*, that are used to help "blind" the network to small or insignificant changes by combining or "pooling" values. Examples of pooling operations are illustrated in Figure 2.11. *Normalization layers* apply a transformation of the outputs in order to adhere to some constraint. In most cases, this constraint is to have the output(s) within some range of each other.

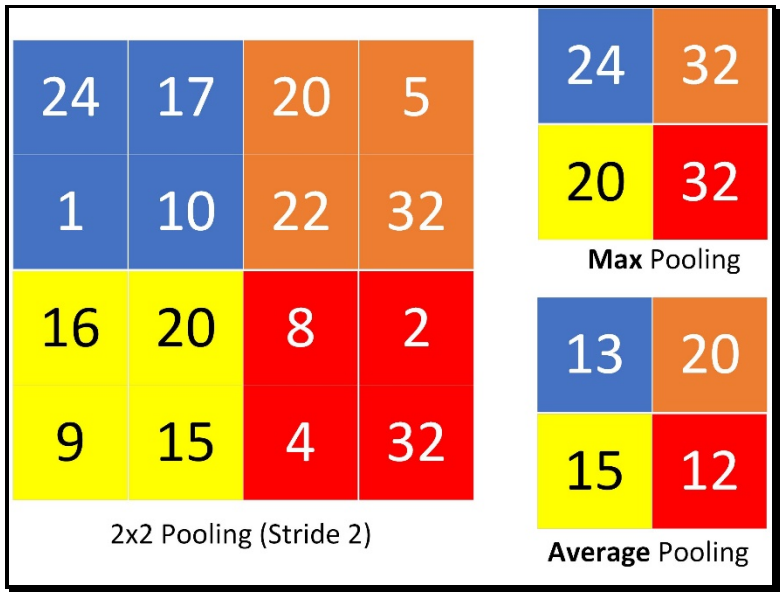

Figure 2.13: Pooling Variations

2.6.2 Recurrent Neural Networks (RNNs)

Recurrent Neural Networks (RNNs) are models designed to process and recognize relationships in sequential data, this means RNNs are typically more adept at recognizing time dependent relationships or sequential dependencies. These sequential dependencies typically lie in time series input commonly found in Speech and Natural Language Processing (SLP or NLP). For example, a well-designed RNN will recognize the similarities between the following two phrases: "I will be out of town on Friday" and "Friday I will not be in the city." The similarities, in this

instance, being the date and location being referenced. Structurally, RNNs are similar to the regular feedforward network except for the output layer. In an RNN, the output layer has some "internal memory" that allows for long-term dependencies to affect the output. The difference between these two networks is illustrated in Figure 2.12. A variation of this network is the Long Short-Term Memory Network (LSTMs). Other than Google's Tensor Processing Unit (TPU), developed in order to meet projected demands of speech recognition in voice search [Hen17], little focus or attention has been placed, to date, on hardware acceleration for most of these LSTMs.

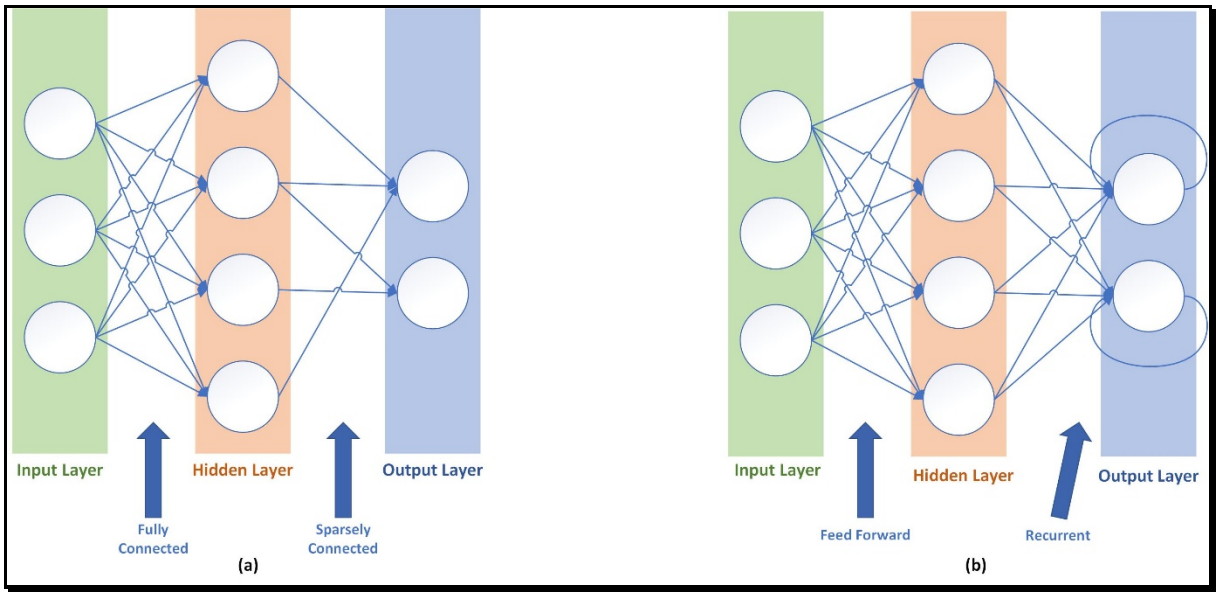

Figure 2.14: (a) Simple Feedforward Network and (b)Recurrent Neural Network

2.7 EMBEDDED VS. CLOUD MACHINE LEARNING

The profession has recently entered into a very interesting era in computing where there are two polar opposite computing schemes. The "traditional distributed computing" model is where heavy workloads are sent over the network to be processed at server farms. Juxtaposed to this is the "Internet of Things" (IoT) model, where every day items such as lightbulbs, speakers, TVs, and refrigerators, have some sort processor (typically to perform simple but specific computational tasks) and computing is pushed to the edge of the network. With the recent rise of the IoT paradigm, the reader might ask, "why bother speeding up computation at the edge of the network when all the computations can be performed on server farms?" This is especially relevant when training often requires huge amount of data sets and computational resources typically only available at these server farms. However, it is worth mentioning the advantages of having Domain Specific Architectures (DSAs) at the edge of the network, even though it is not the focus of this study.

For many DNN applications, it is favorable to have the ANN/DNN inference close to the sensor(s). In many cases, the information to be processed, such as images, video, and audio, have heavy bandwidth requirements which can come at a communication cost. Other applications are time sensitive such as autonomous vehicles, navigation, and robotics. So, the inherent latency of the network cannot be trusted with many of these applications.

2.7.1 Computer Vision

As video continues to be the primary source of traffic on the internet—accounting for over 70% of traffic [Cvn16]—it is becoming extremely desirable to have inferences happen close to the video source rather than transferring the information over a high latency network, waiting for it to

be processed at the clusters, and even still waiting for the result to be sent back. Other applications simply do not have the luxury of waiting: ranging from applications such as autonomous cars that rely on real-time, live video processing in order to identify hazards and obstacles, to security cameras that can benefit from being able to identify threats as they surge.

2.7.2 Speech Recognition

Speech recognition and natural language processing have dramatically improved our ability and experience when interacting with electronic devices. Although many of the virtual assistant services such as Google Assistant and Amazon Alexa are still cloud services, it is very attractive to have these services pushed onto a device. By having these computations on a local device, the dependencies of network connectivity are decreased or even eliminated, the latency between command and execution is reduced, and additionally, privacy is increased. This reduction in latency is important, as in many cases natural language processing is the first step before being able to perform any sort of task in AI.

2.7.3 Medical

It is undeniable that there is a need to move the medical industry from reactive care—treating the patient after the onset of a disease or condition—and preventative care—having the patient come in for weekly/monthly/yearly screenings—to predictive care: a medical system where a patient can wear a device that is capable of collecting long-term data that can help either detect or diagnose various diseases, or monitor treatment [Sze17a]. For example, blood sugar can be monitored, and based on the user's previous A1C levels and glucose levels, hypoglycemia and hyperglycemia can be predicted and prevented. However, due to the nature of these devices, wearable or implantable, the energy consumption must be kept to a minimum.

2.8 COMPUTER ARCHITECTURE

> "*Computers have led to a third revolution for civilization, with the information revolution taking its place alongside the agricultural and the industrial revolutions. The resulting multiplication of humankind's intellectual strength and reach naturally has affected our everyday lives profoundly and changed the ways in which the search for new knowledge is carried out… The computer revolution continues…*"
>
> - David A. Patterson & John L. Hennessey

Applications that were economically infeasible have suddenly become practical. In the recent past, the following applications were '*computer science fiction*'" [Hen12]: computers in automobiles, cell phones, the human genome project, the World Wide Web and search engines.

Now that ANNs have been covered extensively in the first half of this chapter, the history of computer architecture will be discussed next: Performance metrics when evaluating processors and proposed guidelines for designing DSA, according to popular literature.

2.8.1 Computer History

Though it may surprise the reader, the "modern" concept of the general-purpose computer is actually almost 200 years old! An English mathematician and inventor by the name of Charles Babbage is widely credited with conceptualizing the first general-purpose computer in the early 1800's. His Mechanical General-Purpose Computer included the concepts of *control flow*, including branch statements such as *if*, *then*, *else* and *looping*, as well as the first concepts of *memory* for storage of programs and instructions.

It took a little over 100 years to effectively use Babbage's ideas in practice: during the era after WWII when scientists and engineers from around the globe cooperated to further theory and produce electronic computers. One of the most famous is Alan Turing, whose algorithms were used to decrypt Nazi messages in WWII. Not coincidentally, he invented the notion of the "Turing

Machine," which is, at this point, a yet-unattainable goal. Another pair of less well-known scientists, John Eckert and John Mauchly, working at the University of Pennsylvania gave birth to what we know today as the von Neumann architecture. This architecture is composed of a processing unit, a control unit with instruction register and program counter, a main memory, an external mass storage and input/output mechanism.

Early computers like the Electronic Numerical Integrator and Calculator (ENIAC), Electronic Discrete Variable Automatic Calculator (EDVAC) and the Electronic Delay Storage Automatic Calculator (EDSAC), were massive in size and would typically fill up entire rooms with equipment. It would take roughly 20 years for the full development of transistors until we would get the first microprocessor or single computer on a chip, the Intel 4004, in 1971.

2.8.2 Computer Organization

2.8.2.1 Classes of Computers

For the purpose of this thesis, the literature was surveyed and resulted in identifying three major classes of computer systems: *Desktop, Server,* and *Embedded* computer systems. The *desktop* has a strong emphasis on good performance at a low cost, typically used to serve one or very few users. Most importantly however, is its Central Processing Unit (CPU), this CPU is designed with the guideline to "do everything *well enough.*" The *server* is designed, generally, with an emphasis on running larger programs for several hundred or thousand users at a time, and is typically accessed via the network. This class of computer's CPU is typically designed for very high performance, as the workload is usually very heavy. The last class of computing system is the *embedded* system, which is by far the most common type, as these are the computing systems that are inside other devices running very specialized application or software.

The proposed DSA in this article may be placed in any one of these previously mentioned computer systems, depending on whether the DSA is being used to train or infer. If an ANN architecture that has already been trained to infer is utilized, the DSA would be placed in either a *desktop* computer or *embedded* computer, ready to make these inferences at the edge of the network as quickly as possible (short execution time). If, on the other hand, the DSA is used to accelerate the learning process, it would be a DSA designed for training at a server farm.

2.8.2.2 Computing Metrics

How is it possible to quantify what is a "good" vs. "bad" or "fast" vs. "slow" computer? This is a not a trivial answer, and if asked of an engineer, the most common answer would be: "well, it depends." For example, if there are two *desktop* computers, it would be logical to test by running a sample program to see which one completes execution first. In this case, it could be said that the "fast" computer is the one with the shortest *execution time*. If, however, the setting is a datacenter, the "good" label would be more likely based on which of the two *servers* had the highest *throughput*, or which one of the two completed the most work in a given amount of time [Hen12].

In order to quantify the performance of CPUs, the focus will be primarily on two metrics: execution time and power consumption. The execution time of a program is affected by three factors: *Program Instruction Count (IC), Clock Cycles per Instruction (CPI)* and *Clock Rate (R)*, as shown in Equation 2.1, which yields Computing Execution Time, in seconds. Power consumption, in Watts, is computed as shown in Equation 2.2, and is dependent on three factors: *Capacitive Load (C), Transistor Supply Voltage (V)* and *Transistor Switching Frequency (F)*.

$$CPU_{time} = \frac{(IC \times CPI)}{R} \tag{2.3}$$

Equation 2.3: Processing time with respect to Instruction Time, Clock cycles per Instruction and Clock Rate

$$Power = \frac{(1)}{2} \times C \times V^2 \times F \tag{2.4}$$

Equation 2.4: Power consumption as a function of Capacitive load, Voltage and Frequency

In 1974, Robert Dennard made the observation that power density remained constant for a given area of silicon regardless of the increase in resistors because of the constantly decreasing dimensions of each transistor. This in turn meant that transistors could go faster and use less power. However, since 2004, this characteristic has started to plateau, as seen in Figure 2.13 [Hen12]. This phenomenon is referred to, by engineers, as hitting the *Power Wall*. Essentially what has happened is that we have reached a practical power limit for cooling the microprocessors in our systems!

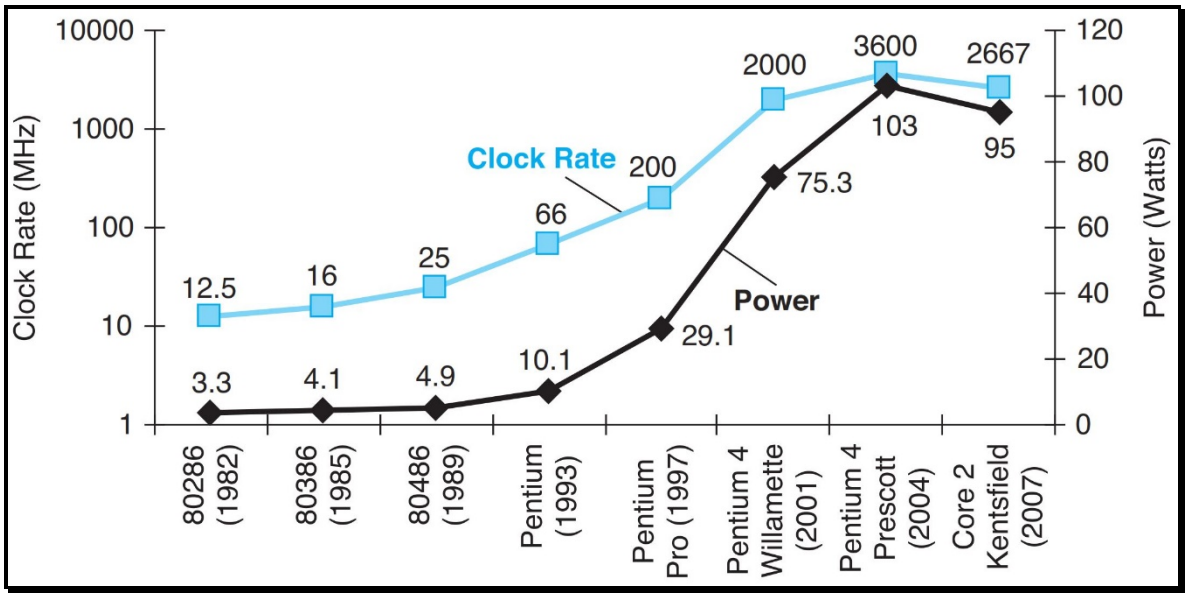

Figure 2.15: Clock rate and Power for Intel x86 microprocessors over eight generations and 25 years [Hen12]

The question is: how have we been able to achieve higher and higher clock rates while keeping power consumption at relatively minimal incline? As one might be able to observe from combining Equations 2.1 and 2.2, that by reducing the voltage within each generation, have higher clock speeds can be achieved, for approximately the same power consumption. The problem with this approach, however, is that by further lowering the voltage to the transistors, they start to become leaky and this, in turn, causes unpredictable transistor performance. This issue has called for newer but also substantially more expensive techniques to cool processors. However, another approach to solve this problem has been a call for complete redesign of some of these computer architectures (DSAs!).

2.8.3 Domain Specfic Architecture Design Guidelines

The following guidelines were certainly not the author's own original ideas, they were however meticulously researched through much reading and literature survey. Here, general guidelines for designing any Domain Specific Architecture are presented, but will emphasize the application of DNNs based on Hennessy and Patterson's book, *Computer Architecture, A Quantitative Approach*.

2.8.3.1 Memory

The first guideline suggested is to use dedicated memories to minimize distance over which data is moved. In the case of DNNs, we want to have the Processing Units (PUs) as close to the training/testing data as possible. An illustration of this type of architecture is shown in Figure 2.14. These types of architectures are referred to as *Spatial Architectures* [Sze17], in which each Arithmetic Logic Unit (ALU) can communicate with one another directly through dataflow processing. Additionally, these ALUs should have their own control logic and local memory, such

as a register file. If an ALU has its own local memory, it is also referred to as a Processing Engine (PE). Having this network of PEs, the time required for retrieval of data decreases by roughly two orders of magnitude because the traditional memory hierarchy (and overhead) is avoided. Additionally, because the DNNs have a lack of randomness in memory access patterns, these DSAs can be designed to support very specialized dataflows to each PE. In doing so, the size and shape of the DNN can be optimized to achieve the best energy efficiency.

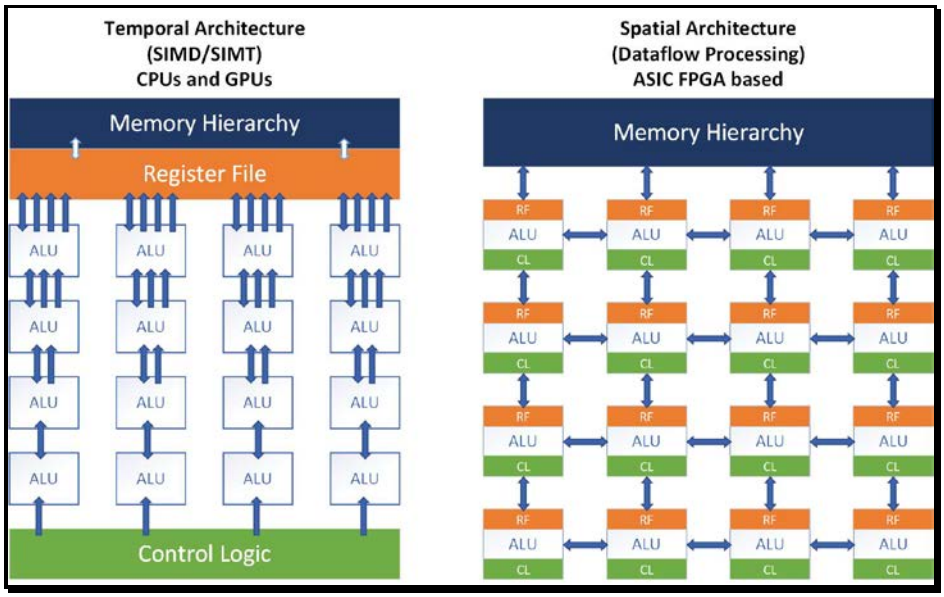

Figure 2.16: Architectures specialized for parallel computing

2.8.3.2 µ-Architectual Optimizations

The second suggested optimization is to invest the resources, saved from dropping advanced micro-architectural optimizations, into more arithmetic units or PEs and bigger memories. The advanced micro-architectural optimization to be removed in this type of DSA, currently found in most CPUs are: out of order execution, multithreading and prefetching. Similar to the previous guideline, as a DSA designer, there is a very clear idea of the memory access pattern in DNN processing. Therefore, in the case of DNNs, out of order execution, multithreading and

prefetching architecture would be a waste of silicon space that could be better spent on more PEs or bigger memories.

2.8.3.3 Parallelism

The third suggested guideline is to take advantage of the easiest form of parallelism for the domain. Due to the nature of DNNs, the most common type of computation is the MAC operation. This operation applies a single instruction (multiply an input vector and weight matrix) for all the data presented. This type of functionality is commonly referred to as Single Instruction Multiple Data (SIMD), according to Flynn's Taxonomy [Fly72] also illustrated in figure 2.15. This DSA could also be used to advantage, in the execution of Multiple Instruction Multiple Data (MIMD), where a trained network has a set of weight matrices, there could be multiple inputs going through the network at the same time in a "cascaded" fashion.

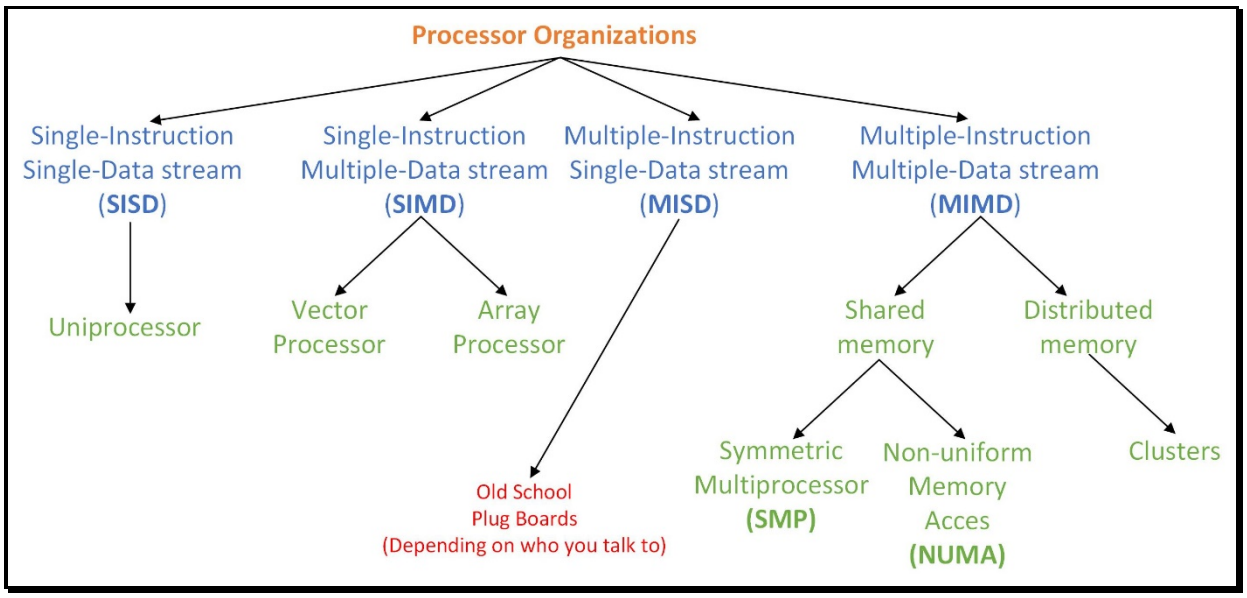

Figure 2.17: Flynn's Taxonomy of Parallel Processors in Computing

2.8.3.4 Data Size

The fourth guideline is commonly referred to in the literature as quantization, or quite simply, reduction of data size to the smallest needed for the domain. For example, by reducing the

numerical precision of the IEEE 754 standard from 16 bits to 8 bits, two benefits are achieved. A reduction in the power required to perform operations on 16-bit operands vs 8-bit operands and the reduction in roughly half of memory that would have been potentially wasted on over-precision in the 16-bit format.

2.8.3.5 Domain Specific Languages

The last guideline is to design for the most commonly used programming language in the domain. A common fallacy according to [Hen17] is assuming that the new computer is so attractive that programmers will rewrite their code just for the new hardware. Fortunately, domain-specific programming languages such as Python, Google's TensorFlow and Keras (written as a wrapper to TensorFlow) have made porting applications to DSAs much more feasible. Keras will be the tool used to develop test ANN's in this work.

3 DISCUSSION OF THE PROBLEM

3.1 PRIORITIZING WORKLOAD AND OPERATIONS

It is important to emphasize why the primary target of this work is to optimize the primary applications in Deep Neural Networks, the Multiply and Accumulate (MAC) operations. Additionally, it is important to discuss the datasets and models that are used in this analysis. These architectures range anywhere from 3 to 34 layers, some combine fully connected and recurrent layers while others combine fully connected layers with convolution layers, the target applications also vary widely from image processing to even an Atari-playing neural network [Rea17]! The Fathom Workloads [Rea17] are additionally described as;

> *"These are not toy problems—each of these models comes from a seminal work in the deep learning community and have either set new accuracy records on competitive datasets or pushed the limits of our understanding of deep learning."*
> - Deep Learning for Computer Architects Reagen et. all.

Table 3.1: The Fathom Workloads [Rea17]

Model Name	Application	Architecture	Layers	Learning Style
autoenc	Model used to learn feature encoding in data.	Full	3	Unsupervised
speech	Text to speech engine begin hand-tuned systems	Full/Recurrent	5	Supervised
memnet	Facebook's memory-oriented neural system	Memory Network	3	Supervised
seq2seq	Extracts meaning of sentence and produces it in a target language.	Recurrent	7	Supervised
residual	Image classifier developed by Microsoft's Research Division Asia. ILSVRC 15' winner.	Convolutional	34	Supervised
vgg	Image classifier leveraging power of small conv. Filters	Full Convolutional	19	Supervised
alexnet	Image classifier, beating hand-tuned systems at ILSVRC 2012	Full Convolutional	5	Supervised
deepq	Atari-playing neural network demonstrating near perfect gameplay by analyzing pixels from award winning players	Full Convolutional	5	Reinforced

As mentioned earlier, the reason for citing these works is to provide context and justification for primarily targeting the MAC operation. Figure 4.1 [Rea17], is a table that reports the percentage

of execution time spent on certain types of operations, for different Fathom Workload entities. Specifically, the Fathom Workload framework is listed on the vertical axis and the variety of operations is given as the horizontal axis. The intersection of these two axes is populated with a number that represents the percentage of execution time taken during computation for that particular operation. Now the reader might note that for some of the frameworks the summation of the entire row might not result in 100 (as would be expected to report 100% of execution time), but this is because the original study chose to leave out operations that contributed less than 1% of total computation time.

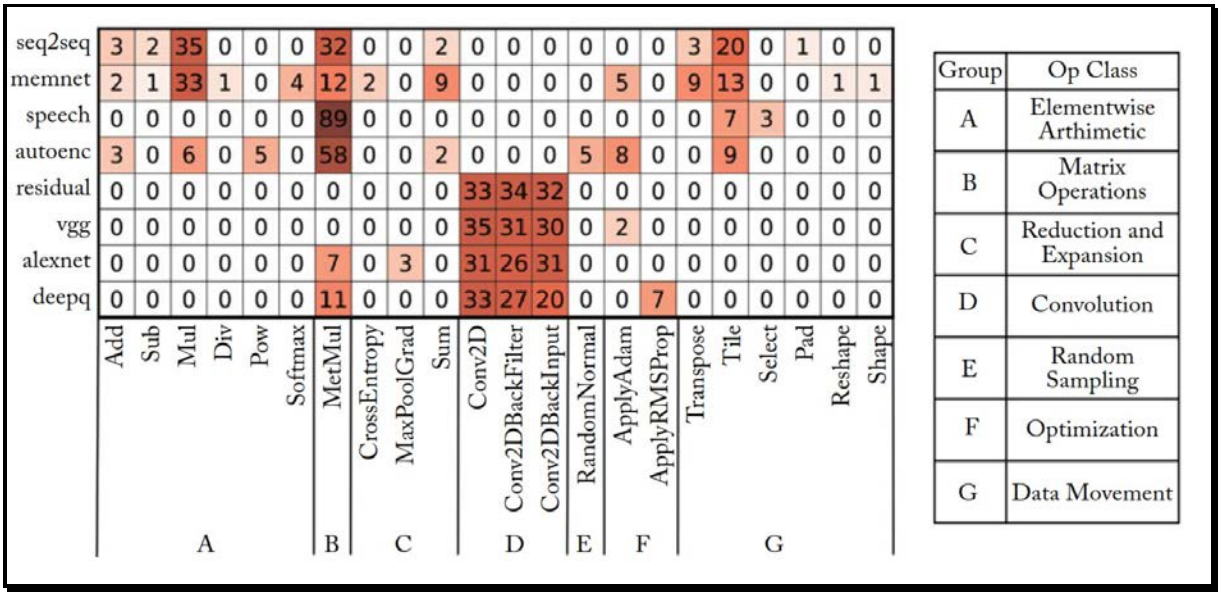

Figure 3.18: Principle component analysis of execution time per Fathom workload [Rea17]

This table demonstrates that for those networks that do not employ convolutional layers, the Matrix Multiply (MetMul) accounts for an average of 47.75% total computational time! Roughly another third of computational time is spent on Multiplication which is still one of the targeted operations in this work. Now, the reader might think that we are excluding those networks whose primary operation is the convolution but the reality is that the proposed architecture can also help convolutional neural networks. The *discrete convolution* operation is defined as: the ***summation***

from negative infinity to positive infinity of the ***product*** of two functions, one of which is described as itself flipped over the y-axis shifted *n* units, see Equation 4.1 [Phi14]. Which goes to show that the entirety of the discrete convolution is supported by the proposed architecture!

$$y[n] = \sum_{k=-\infty}^{\infty} x[k]h[n-k] \tag{3.1}$$

Equation3.1: Discrete Convolution Operation

3.2 BINARY MULTIPLICATION AND SUMMATION ALGORITHMS USING FLOATING-POINT OPERANDS

Now that the importance of the two primary operations in neural networks, multiplication and summation have been stressed, details of hardware multiplication will be covered. Binary floating-point representation of a number in hardware as per the Institute of Electrical and Electronics Engineers (IEEE) is discussed in the following section. Floating-point addition, multiplication and proposed algorithms intended to speed up multiplication operations while also attempting to reduce silicon surface area consumption are examined in subsequent sections.

3.2.1 IEEE FLOATING-POINT STANDARD 754

The 2008 IEEE 758 standard is a document that specifies the interchange and arithmetic formats and methods for binary and decimal floating-point arithmetic in computer programming environments. This document, however, focuses on the binary floating-point representation of numbers, and has a primary focus on the 16-bit (*half-precision)* and the proposed 12-bit representation of these numbers. The set of floating-point numbers representable within a format is determined by the following integer parameters [IEE08]:

- b = the radix (in our case 2 or 10)
- k, storage width in bits
- p = the number of digits in the significand (precision)
- e_{max} = the maximum exponent e, (also referred to as the *bias*)
- e_{min} = the minimum exponent e ($e_{min} = 1 - e_{max}$)
- e is any integer $e_{min} \leq e \leq e_{max}$

- *m* is a number represented by a digit string

Each floating-point number has one unique encoding in a binary interchange format. In order to make this encoding unique, the value of the significand *m* is maximized by decreasing the value *e* until either $e = e_{min}$ OR $m \geq 1$. After this is done, if $e = e_{min}$ and $0 < m < 1$, the floating-point number is considered to be *subnormal*. These *subnormal* numbers are encoded with a reserved biased exponent value. The representation of floating-point data in the binary interchange formats are encoded in *k* bits in the three fields illustrated in Figure 3.2. Specifically, the fields are: 1-bit sign, *S*.

- *w*-bit biased exponent, $E = e + bias$
- t-bit trailing significand, where $t = p - 1$

Figure 3.19: General Floating-Point Encoding

Table 3.2 presents common numerical formats by their binary interchange format parameters, for comparison purposes.

Table 3.2: Binary Interchange Format Parameters [IEE08]

Parameter	binary16 (half-precision)	binary32 (single-precision)	binary64 (double-precision)	binary128 (quadruple-precision
k	16	32	64	128

p	11	24	53	113
e_{max} **(*bias*)**	15	127	1023	16383
sign bit	1	1	1	1
w	5	8	11	15
t	16	32	64	128

For this work, the focus is on developing two architectures that make use of the *half-precision* format: 1 sign bit, 5 biased exponent bits and 11 significand bits; and a proposed *mid-precision* format: 1 sign bit, 5 biased exponent bits and only 6 significand bits. Now that the standard representations for floating-point numbers has been explained, the following section will elaborate on the two primary operations being performed in the DSA: the summation and multiplications operations.

3.2.2 BINARY FLOATING-POINT SUMMATION ALGORITHMS

The general algorithm for adding two floating-point numbers is as follows:

- **Step 1**: Align the decimal point of the number that has the smaller exponent (easy enough for a human)
- **Step 2**: Add the significands.
- **Step 3**: If the sum is not in scientific notation (have only 1 leading digit after the decimal point), normalize the summation.
- **Step 4**: Round the resulting number according to required storage specification.

In order to accurately highlight the issues and complexity with adding floating-point numbers, an example using two floating-point numbers both in base 10 is detailed next. If, for example, the sum of the following two numbers, $9.99_{10} \times 10^{1}$ and $1.32_{10} \times 10^{-1}$ is required, it would be necessary to first obtain a form of the smaller number that matches the larger exponent, $1.32_{10} \times 10^{-1} = 0.0132_{10} \times 10^{1}$. It can be observed that in order to accomplish this, it is necessary to shift the significant digits to the right until the exponent matches that of the larger number, in this case a shift right twice. Now that both exponents match, we can successfully add the significands as per step 2:

$$\begin{array}{r} 9.9900_{10} \\ +0.0132_{10} \\ \hline 10.0032_{10} \end{array}$$

Because the result is not normalized, it must be adjusted, resulting in the final summation equaling 1.00032×10^{1}. The final step is to round the resulting number according the storage specifications. Even though our example did not indicate any storage specifications, the reader might easily

observe that if we had storage limit of 3 significand digits, our end result would actually be 1.000×10^1. While this might not seem like a big issue, in the cases where the application is dealing with very high precision calculations, such as medical imaging or exploration of hydro-carbon resources, it can become evident how much more critical issue precision really is. Another issue that can arise from limited storage for precision comes from normalizing the result: how to deal with instances where the normalized result yields a value that the architecture can no longer represent? This issue is referred to as either an underflow or overflow error. It is common that a pattern of all 1's in the exponent is used to represent values outside the scope of the storage requirements. Figure 3.3 shows the block diagram of a digital circuit: an arithmetic unit dedicated to floating-point addition. The steps illustrated previously in the example correspond to each block from top to bottom in Figure 3.3. The exponents are initially subtracted in the small Arithmetic Logic Unit (ALU) to determine which number is larger and by how much. This result controls the three Multiplexers (MUX's) from left to right, which helps select the larger exponent, the significand of the smaller value and bigger value. The smaller significand is shifted right and added afterwards inside the big ALU. Normalization then happens by shifting the result either left or right and increments or decrements the exponent accordingly. Finally, rounding is performed according to storage specifications.

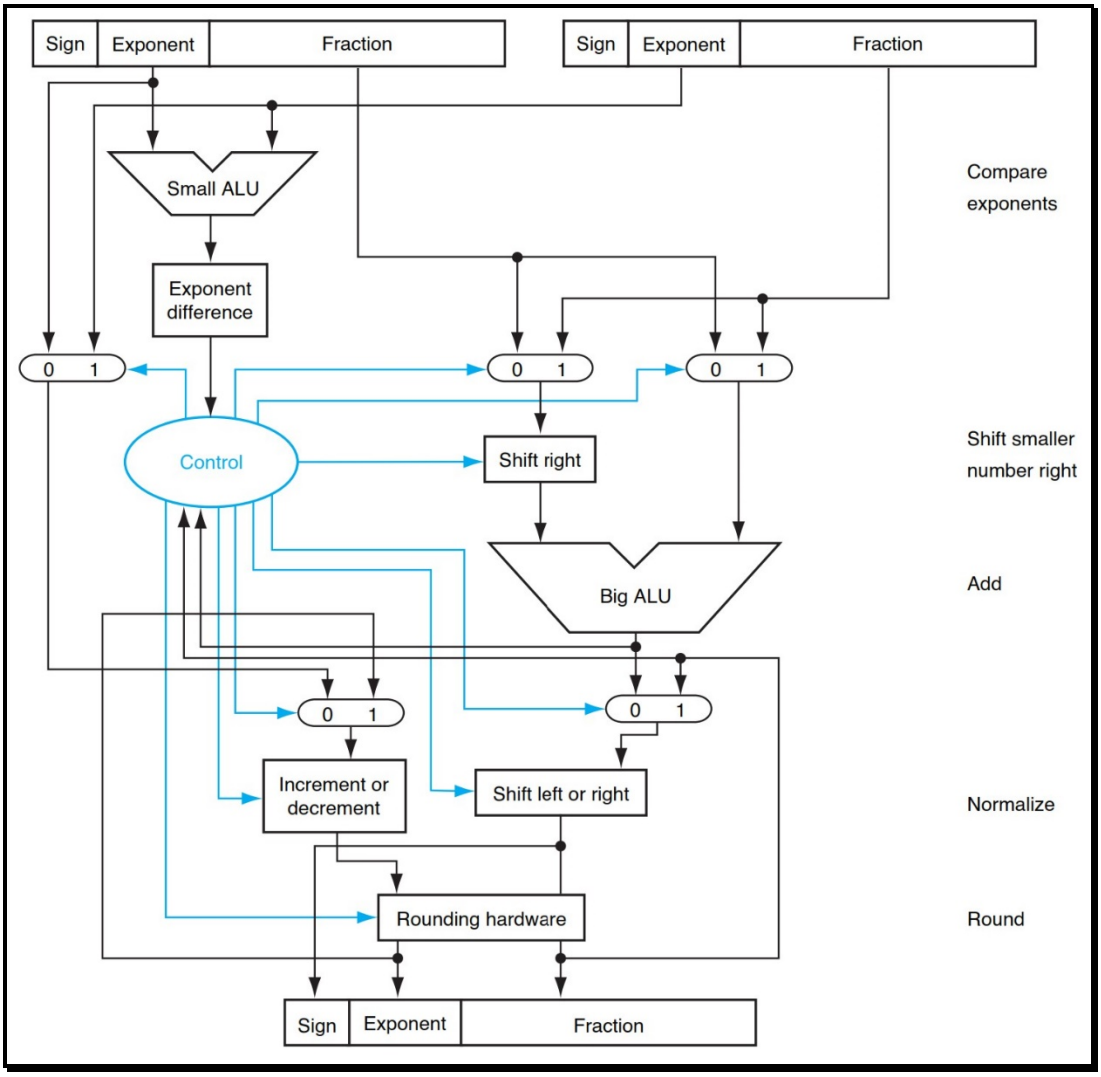

Figure 20.3: Block diagram of an arithmetic unit dedicated to floating-point addition [Hen12]

3.2.3 BINARY FLOATING-POINT MULTIPLICATION ALGORITHMS

Now that the floating-point addition algorithm has been detailed, the general floating-point multiplication algorithm will be illustrated. The general floating-point multiplication is as follows:

- **Step 1**: Calculate the exponents by adding the biased or un-biased exponents together, then removing the bias if necessary.

- **Step 2**: Multiply the significands (without forgetting to include the "hidden" leading 1) and place the decimal point in the correct place according the significands.

- **Step 3**: Normalize the product.
- **Step 5**: Round the product according the storage requirements.
- **Step 4**: Find the sign of the product by performing an XOR operation on the sign bits.

This algorithm may seem simpler than the addition algorithm because there is no need for comparisons operators (in order to store the smaller and larger operator), and also there is no requirement of constant shifting until operands match each other. There is one detrimental characteristic of this general algorithm, however, in an inefficient approach: *N* number of additions is required to find the product, where *N* is the size of the smallest significand, as illustrated in Figure 3.4. Referring back to the IEEE 754 specifications from section 3.2.1, we note that the number of additions can range anywhere from 16 to 128! In an inefficient implementation this is both slow and consumes a large amount of silicon space. The following sections address algorithms that aim to mitigate these effects and make the operation more efficient.

				X_3	X_2	X_1	X_0
			×	Y_3	Y_2	Y_1	Y_0
				X_3Y_0	X_2Y_0	X_1Y_0	X_0Y_0
			X_3Y_1	X_2Y_1	X_2Y_1	X_0Y_1	
		X_3Y_2	X_2Y_2	X_1Y_2	X_0Y_2		
	X_3Y_3	X_2Y_3	X_1Y_3	X_0Y_3			
P_7	P_6	P_5	P_4	P_3	P_2	P_1	P_0

Figure 3.21: Multiplication size 4 operands through addition

3.2.3.1 WALLACE MULTIPLIER

The Wallace multiplier has generally three stages under which it operates [Abr17]. The first stage is the "Initialization," illustrated in Figure 3.5. During this stage, the focus is on forming a matrix of partial products and grouping these partial products into groups of three rows. The following stage focuses on "Compressing" the matrix to a height of only two rows. In the third

and final stage, the sum of the final two rows is computed in order to obtain the final result. The following is a detailed explanation of the Wallace algorithm using numbers 109_{10} = 0110 1101_2 and 78_{10} = 0100 1110_2, illustrated in Figure 3.5.

First, we must initialize the matrix of partial products and group the resulting matrix into groups of three rows, those NOT in a complete group will be left alone as shown in the "Initialization". The objective of the following stage, or the "Compressing" phase of the Wallace algorithm, is to compress the matrix to rows of 6, 4, 3 until it has reached only 2 rows as shown in steps 1, 2, 3 and 4 respectively. This is accomplished using Full Adders (FA's) or a (3,2) counter in column groupings with 3 bits in combination with Half Adders (HA's) or a (2,2) counter in column groupings with only 2 bits [Abr17]. Once a column in a group is added, the sum bit is stored within the same column in the first row of that grouping and the carry bit is carried to the following column and stored in the second row, this is repeated until we have reached the end of the row starting from the least significant bit (LSB) and ending with the most significant bit (MSB). The result of a FA is illustrated with orange highlighting and the result of a HA is illustrated with yellow highlighting. Those bits that are simply carried down or left alone are highlighted in green. This iterative process is repeated until we have a matrix with only 2 rows. We then finally add the remaining two rows together to obtain our final result of 0010 0001 0011 $0110_2 = 8502_{10}$.

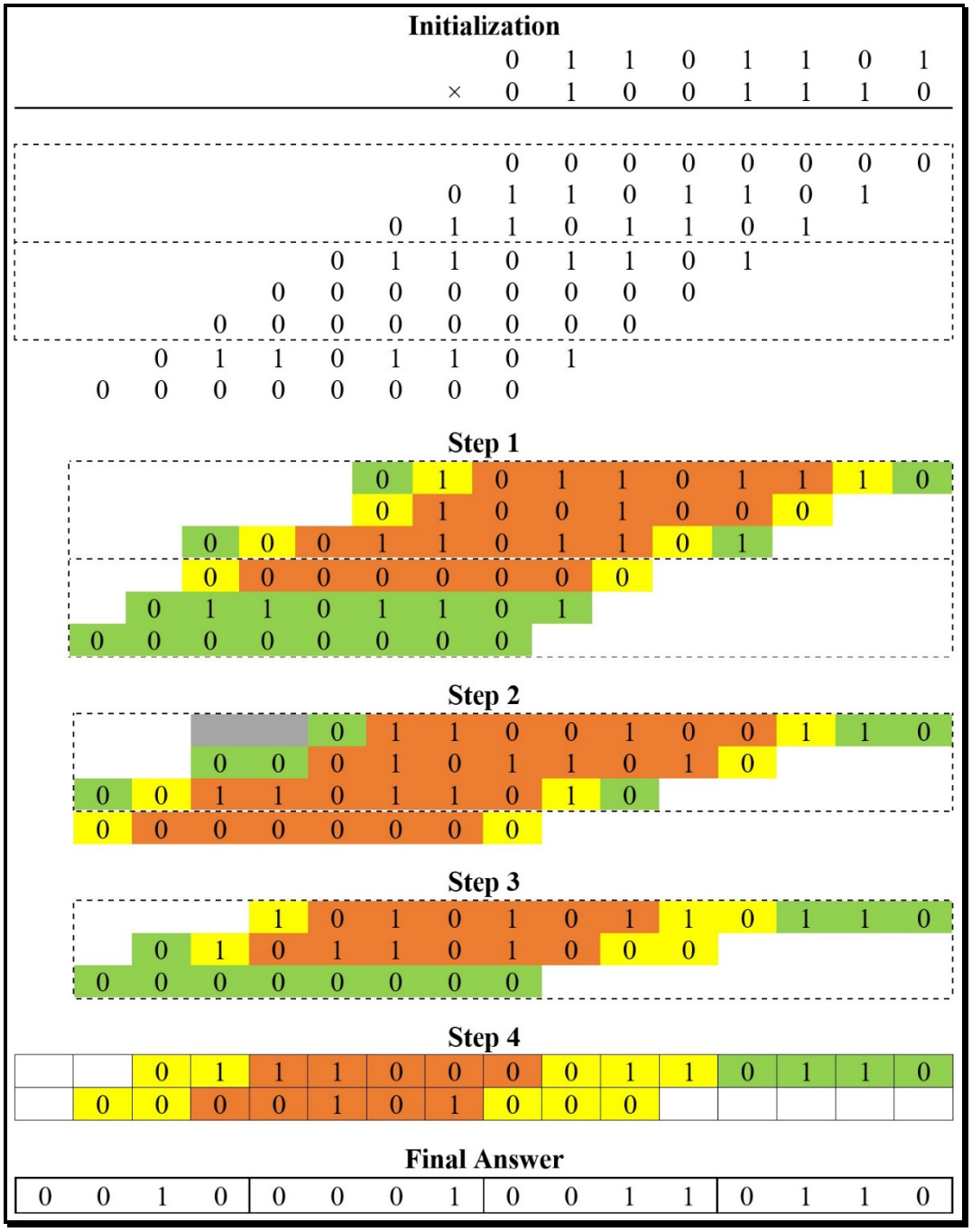

Figure 3.22: Step by step illustration of Wallace Tree Multiplier, multiplying 1092 and 78

3.2.3.2 DADDA MULTIPLIER

The Dadda Multiplier works in three stages, similar to the Wallace Multiplier. In the "*Initialization*" stage, it also forms a partial product matrix similar to the Wallace Multiplier, however herein lies one of the key differences. The Dadda Multiplier shifts those bits enclosed

inside the triangle illustrated in Figure 3.6 and shifts them upward in order to have a now "inverted triangle" shape [Abr17]. As in the Wallace Multiplier, the next stage focuses on "*Compressing*" the matrix to a height of two rows. The third stage is used to find the summation of the last two rows in order to obtain the final result. Studies have shown that as the number of bits being multiplied increases, the Dadda Multiplier operates at the same speed as the Wallace Multiplier however using fewer components, ultimately saving silicon space [Hab70]. The following is a detailed illustration with an example of the Dadda algorithm using numbers 109_{10} = 0110 1101_2 and 78_{10} = 0100 $1110_{2,}$ illustrated in figure 3.6, the same numbers are used as before in order to easily verify the accuracy of this algorithm.

First, the matrix of partial products is initialized, but before the "*Compressing*" stage is started, one must first shift up those bits encapsulated within the green triangle illustrated in figure 3.6. Once this is complete, the resulting partial product matrix should look similar to an inverted triangle. The following stage is the "*Compressing*" phase of the Dadda algorithm. This phase focuses on compressing the matrix to rows of 6, 3, 2 until it has also reached only 3 rows as show in steps 1, 2, 3 and 4 respectively in figure 3.6 [Abr17]. This is achieved by using FA's in column groupings with 3 bits in combination with HA's in column groupings with only 2 bits [Abr17]. Because of this scheme, the Dadda Multiplier has been found to require fewer FA's and HA's in comparison to the Wallace Multiplier [Tow03]. Once the columns in a group are added, the sum bit is stored within the same column in the first row of that grouping and the carry bit is carried to the following column and stored in the second row. As in figure 3.5, the result of a FA is illustrated with orange highlighting and the result of HA is illustrated with yellow highlighting. Those bits that are either simply carried over or left alone are shown in green. This recursive process is

repeated until we have a matrix with only 2 rows. The final phase is to find the sum of the last two rows and verify that our result is indeed, $0010\ 0001\ 0011\ 0110_2 = 8502_{10}$.

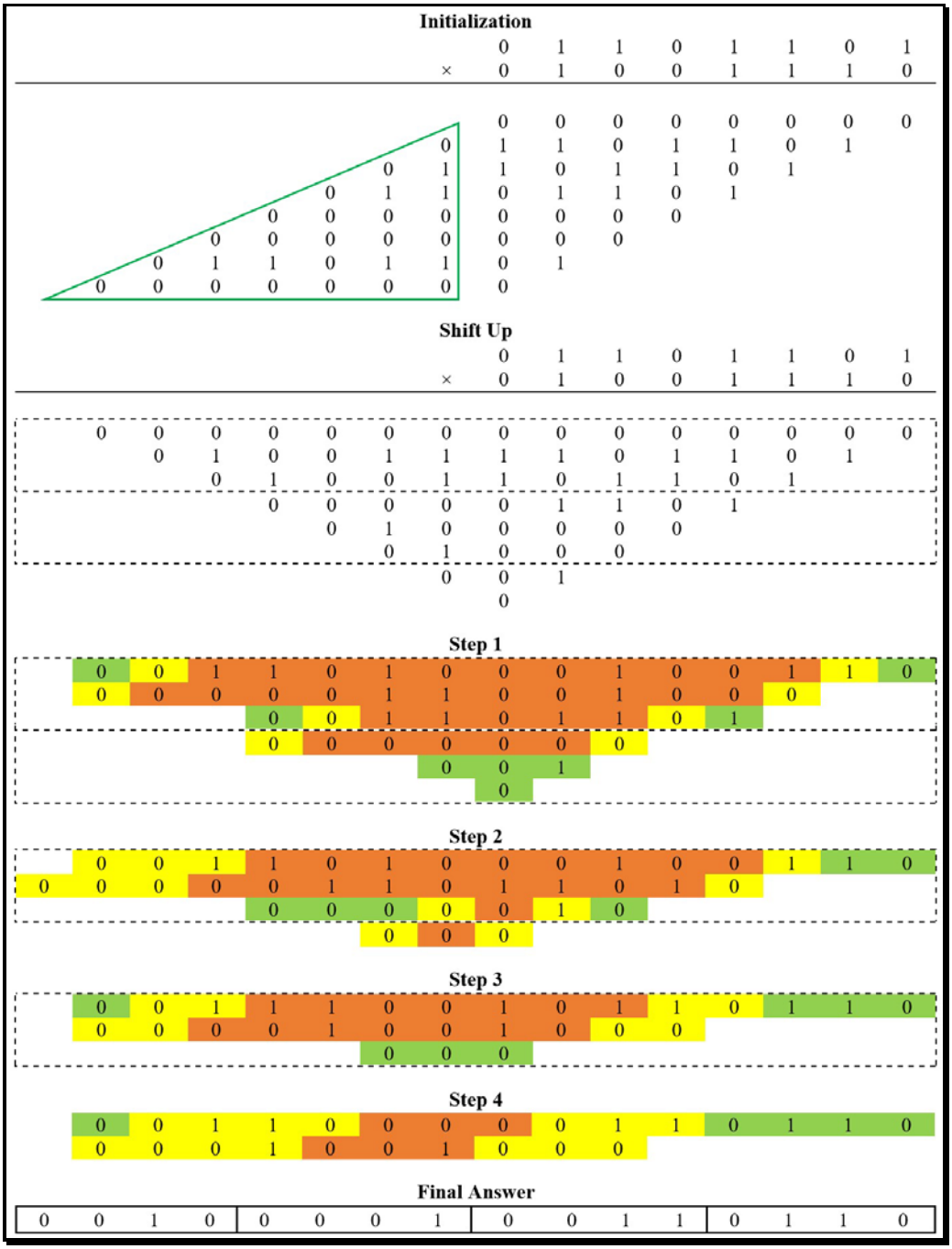

Figure 3.23: Step by step illustration of Dadda Multiplier, multiplying 1092 and 78

3.2.3.3 BOOTH'S ALGORITHM

The third multiplication algorithm explored in this work is Booth's Algorithm. In this algorithm we have an **Accumulator** (**A**) register, initially set to zero, a **Q** register used to store the multiplier, **M** register used to store the multiplicand and an **-M** register used to store the two's compliment of **M**. The size of these registers is determined by the size of the operands, in the case of the example at hand, the register size for **A**, **Q**, **M** and **-M** is 8 bits. Note: assigning the multiplier and multiplicand to either **Q** or **M** is not important, assigning values either way will result in the final correct with possible different intermediate results. In addition to these registers we have a bit referenced as $\mathbf{Q_0}$.

Booth's Algorithm works by analyzing the LSB in **Q** and the $\mathbf{Q_0}$ bit, in that corresponding order. Based on the resulting bit pattern we perform 1 of 3 different actions:

- **00** or **11**: Right shift once **A** and **Q**, shifting the both the LSB's from **A** into **Q** and **Q** into $\mathbf{Q_0}$.
    - This action will be referred to as **Shift Right** (**SR**).
- **01**: Add the value of **M** to the value of **A** and store the result back into **A**, then perform **SR**.
    - This action will be referred to as **Add M Shift Right** (**AMSR**).
- **10**: Subtract the value of **M** (add the value of **-M**) to the value of **A** and store the result back into **A**, the perform **SR.**
    - This action will be referred to **Subtract M Shift Right** (**SMSR**).
- Note: in all the above actions the shift in **A** is performed with sign extension.

The process of analyzing $\{\mathbf{Q},\mathbf{Q_0}\}$ and performing the corresponding action is repeated *n* number of times where *n* is the size of the operands, illustrated as **S**. Figure 3.7 is used to illustrate Booth's Algorithm again using numbers 109_{10} = 0110 1101_2 and 78_{10} = 0100 1110_2 in order to easily demonstrate and verify the accuracy of the algorithm.

M = 0100 1110 -M = 1011 0010

A								Q								Q_0	S	A
0	0	0	0	0	0	0	0	0	1	1	0	1	1	0	1	**0**		
1	0	1	1	0	0	1	0										8	**SMSR**
1	1	0	1	1	0	0	1	0	0	1	1	0	1	1	0	**1**		
0	1	0	0	1	1	1	0										7	**AMSR**
0	0	0	1	0	0	1	1	1	0	0	1	1	0	1	1	**0**		
1	0	1	1	0	0	1	0										6	**SMSR**
1	1	1	0	0	0	1	0	1	1	0	0	1	1	0	1	**1**	5	**SR**
1	1	1	1	0	0	0	1	0	1	1	0	0	1	1	0	**1**		
0	1	0	0	1	1	1	0										4	**AMSR**
0	0	0	1	1	1	1	1	1	0	1	1	0	0	1	1	**0**		
1	0	1	1	0	0	1	0										3	**SMSR**
1	1	1	0	1	0	0	0	1	1	0	1	1	0	0	1	**1**	2	**SR**
1	1	1	1	0	1	0	0	0	1	1	0	1	1	0	0	1		
0	1	0	0	1	1	1	0										1	**AMSR**
0	0	1	0	0	0	0	1	0	0	1	1	0	1	1	0	**0**	0	**FA**

Final Answer

0	0	1	0	0	0	0	1	0	0	1	1	0	1	1	0

Figure 3.24: Step by step illustration of Booth's Algorithm, multiplying 1092 and 78

3.3 USE OF DOMAIN SPECIFC ARCHITECTURE GUIDELINES

3.3.1 USE OF DEDICATED MEMORIES TO MINIMIZE DATA MOVEMENT

As previously mentioned in section 2.8.3.1, the first guideline towards designing any DSA [Hen17] is to have dedicated memories in order to minimize the distance over which data is moved. The benefits of this design strategy are quicker and more energy efficient computations, due to the removal of the traditional memory hierarchy seen in general CPU's. This style of architecture is referred to as a *Spatial Architecture* [Sze17], in this type of architecture each ALU is allowed to communicate with one another, and additionally have local control logic and local memory, typically as a register file. The combination of ALU, local register file and local control logic will be referred to as a Functional Unit (FU). In this work we will also attempt to leverage the topology of the *Spatial Architecture* and these FU's in combination with the *systolic array*. The *systolic array* is a two-dimensional collection of arithmetic units, in the case of this project FU's, that can independently compute a partial result as a function of inputs from other arithmetic units that are considered upstream to each unit. It relies on data from different directions arriving at cells in an array at specific time intervals where they are to be combined. Because the data flows through the array as a staggered advancing wave front, it is similar to blood being pumped through the human circulatory system by the heart, which is the origin of the systolic name [Hen17].

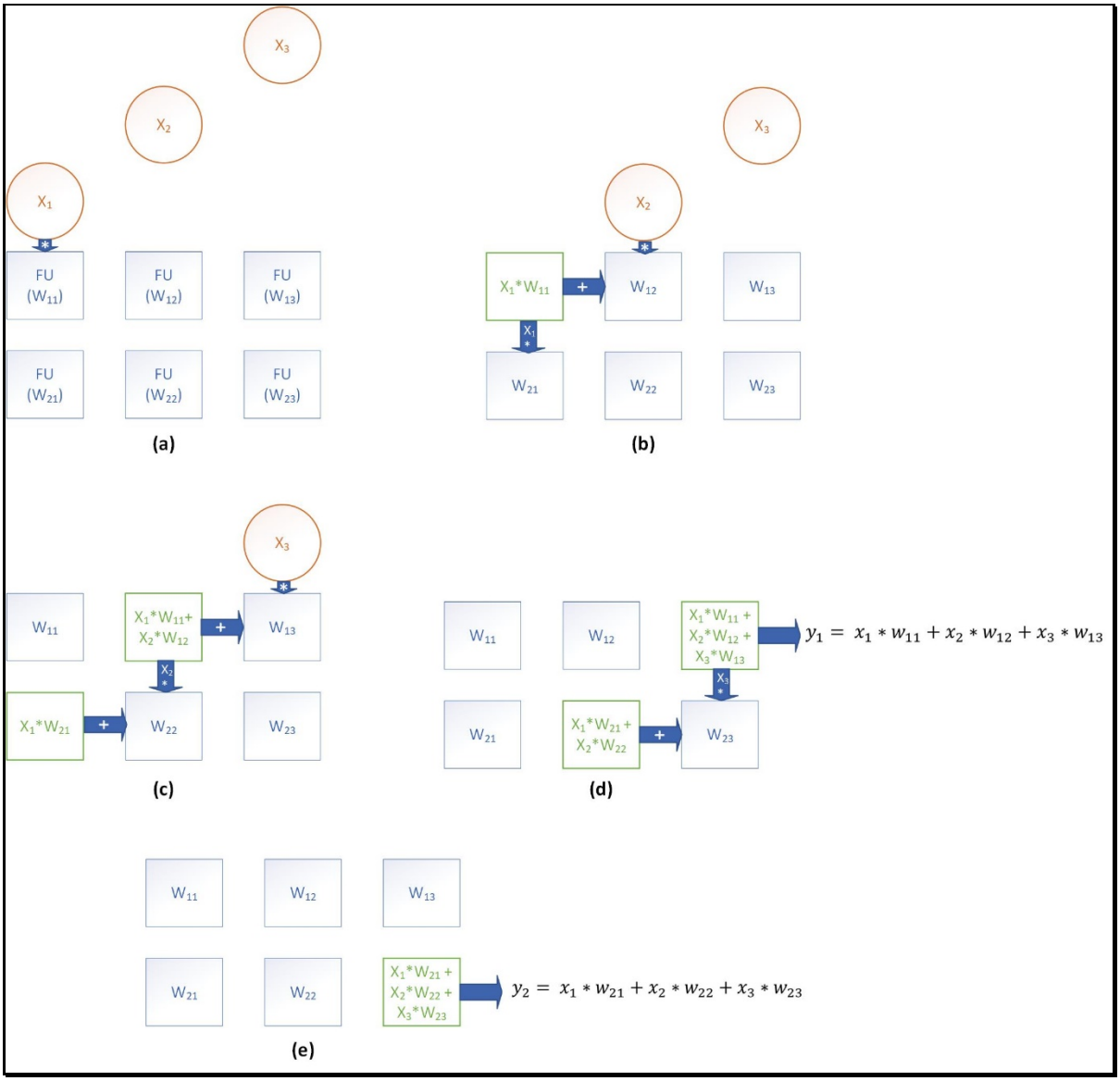

Figure 3.25: Example of systolic array operation starting from top-down, left-right

An example of the systolic array is given in figure 3.8. The orange circles represent an input vector being introduced to the Spatial Architecture in a staggered fashion. In figure 3.8 the input vector travels through the array downward, the input vector is being multiplied with the respective weights of each neuron stored locally in each FU as it travels down to the following set of FU's. The resulting product at each FU is added (the accumulate portion of the MAC operation)

by communicating the result to the FU to its right side which is then added to the result of the multiplication of that FU.

The proposed architecture results in several advantages, as mentioned before, eliminating the memory hierarchy seen in typical CPUs and "recycling" the input vector, the distance over which information has to travel is therefore minimized, also reducing the overall power consumption. This is many times a critical constrain in many embedded system design applications. Another advantage of this type of architecture is that it would give the illusion of x_n inputs being read, processed and inferred on simultaneously after only a single feed delay cycle.

3.3.2 SCRATCH MICRO-ARCHITECTURAL TECHNIQUES TO INVEST IN MORE PROCESSING UNITS

There is, at this point, a very deep understanding of the execution and algorithm in this specific domain–that is, multiply and accumulate the inputs and weights, pass this result to an activation function, pass this result to the next layer, and repeat this process until the final output layer is reached, and produce the final result or inference. Now, instead of wasting precious space and resources on advanced micro-architectural techniques such as out-of-order execution, multithreading, prefetching, address coalescing, it is now possible to focus on spending these saved resources on more processing units and more on-chip memory. More on-chip memory would be certainly be much more beneficial to recurrent neural networks (RNN's), where, if the reader references back to section 2.6.2, these networks require some sort of "memory of previous inputs/outputs. The processing unit that will serve as the primary computational power house for this DSA is described in the following section.

3.3.2.1 THE FUNCTIONAL UNIT ARCHITECTURE

The proposed functional unit illustrated in figure 3.9 for this work is composed of 4 primary parts: the floating-point multiplication hardware illustrated as the circle with the hollow X symbol inside, the floating-point addition hardware illustrated as the circle with the hollow + symbol inside, the register file containing the stored weights of a given neuron and finally some sort of control logic. In the case of an RNN, the register file will also hold some sort of "memory" of previous inputs.

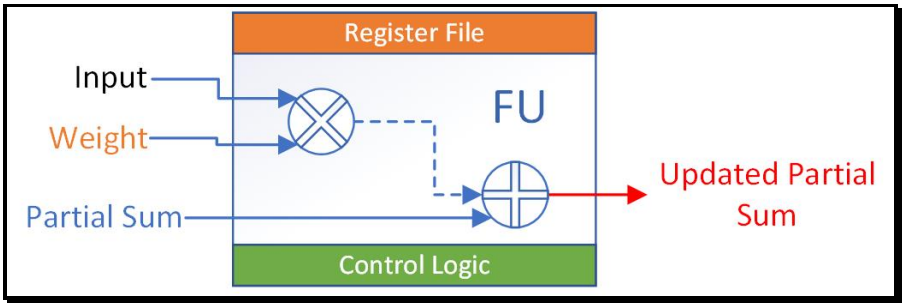

Figure 3.26: Proposed Functional Unit Architecture

3.3.3 LEVERAGE EASIEST FORM OF PARALLELISM THAT MATCHES THE DOMAIN

Flynn's Taxonomy [Fly72] classifies computer into one of four categories:

- **Single-Instruction Single-Data stream (SISD) –** where there is a single-instruction performed on a single-data stream, this type of computer is also sometimes referred to as the uniprocessor.

- **Single-Instruction Multiple-Data stream (SIMD) –** there is still a single-instruction but now this instruction is manipulating multiple-data streams, in practice each processor has its own data memory but there is only one instruction being performed typically dispatched by a control processor.

- **Multiple-Instruction Single-Data stream (MISD) –** this class of computer performs multiple-instructions simultaneously on a single-data stream, no commercial multiprocessor of this type has been built to date [Hen17]. Though some may argue though that old-school plug-board machines are a derivative form of MISD, since the instruction streams were single instructions and derived data passed from program step *i* to step *i*+1 [Fly72].

- **Multiple-Instructions Multiple-Data streams (MIMD)** – the final classification is the multiple-instructions with multiple-data streams computer, the "multiprocessor." In this architecture, each processor has the ability to fetch its own instructions and can operate on its own set of data, these types of computers are usually those found in clusters or data-centers.

Attempting to follow the third suggested guideline, "take advantage of the easiest form of parallelism for the domain" [Hen17], we shall leverage MIMD. The reason for leveraging this form of parallelism is due to the nature of ANNs. An ANN has a set of weight matrices stored, hence one part of the MD. There would also be multiple inputs going through the networks at the same time in a "cascaded" fashion. This idea of "cascaded" inputs is the same as that one presented in figure 3.8 with the systolic array. The MI comes from simultaneous operations being performed inside the ANN, multiplication of weights and inputs, summation of results in order to determine whether a neuron will "fire" and the inference of the network.

3.3.4 REDUCE DATA OPERATION SIZE

The third suggested guideline for designing a DSA is to reduce the data size and data type to the simplest needed for the domain. Utilizing smaller and simpler data types has three advantages: it allows us to pack more FU's onto the same limited chip area, to increase the effective

memory bandwidth and on-chip utilization, and the time and energy associated with moving data around is diminished because of the smaller data representations.

If the reader references back to section 2.2.1, one is able to see that the IEEE 754 standard for representing single-precision and half-precision floating-point numbers is 32 and 16 bits, respectively. For this project, the focus is on using only one of these two data representations, the IEEE 754 standard half-precision representation illustrated in figure 3.10(a), since minimization of the data size is important. We have also designed a proposed 12-bit representation referenced as *mid-precision,* illustrated in figure 3.10(b). In this proposed *mid-precision* format, the sign bit *S* is still kept to a single bit (we couldn't figure out how to split a bit and we haven't successfully achieved quantum computing). Bits 6 through 10 are still used to express the biased exponent, *E*. What makes this a *mid-precision* representation is that the significant field, *t*, is limited to only 5 bits, so we have essentially "cut" the significand field down the middle. The reason for this approach is that during initial testing of the DSA implementation of the *mid-precision* variation, it was found that there was not a significant reduction in performance when inferring compared to its half-precision counterpart. Results are further discussed in the following chapter.

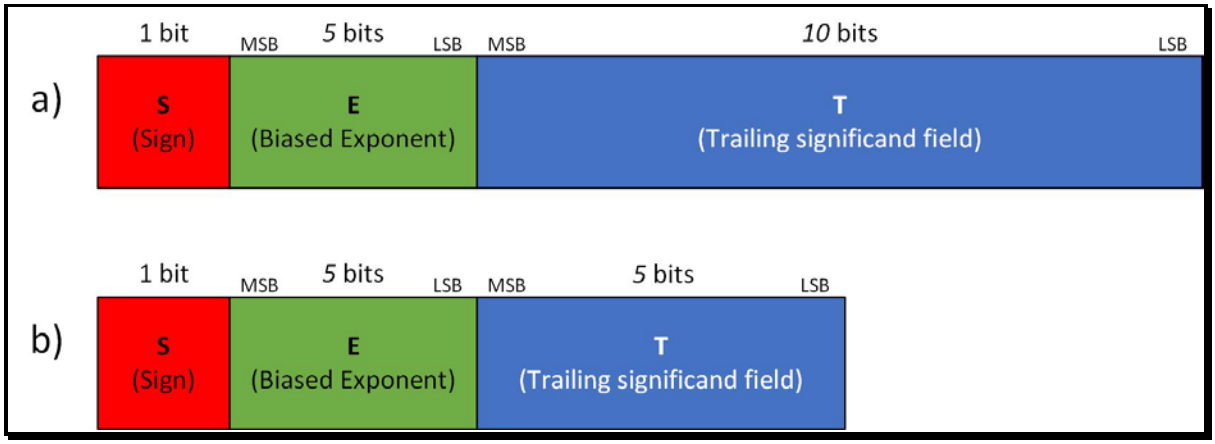

Figure 3.27: Comparison of (a) IEEE 754 half-precision; and (b) proposed mid-precision

3.3.5 DESIGN DSA ACCORDING TO DOMAIN-SPECIFIC LANGUAGE

As languages like JavaScript and Python that come with just-in-time compilers and frameworks that are gaining increasing popularity such as AngularJS and Django, it is has become increasing clear that programmers have started to value productivity over performance. As computer engineers, we cannot fall prey to the misconception that our new computing platform will be so attractive or so revolutionary, that scientist and programmers will rewrite entire programs and thousands of lines of code, just to accommodate our new hardware. Thus, we come to the last suggested guideline, we must design according to a domain-specific programming language to port code to the DSA [Hen17]. In designing these DSA's we should be very careful and consider the flexibility of the architecture if we want it to be useful for as long as possible in order to mitigate some of the inherent non-reusable engineering (NRE) cost that inherently comes from developing such a specialized piece of hardware.

As we continue this shift from performance to productivity an adoption of high-level programming frameworks comes hand-in-hand. These frameworks provide primarily two benefits: they are capable of easily abstracting the underlying hardware interface from the programmer, the programmer can now focus on just "driving" instead of what is "under the hood." The second benefit comes from the capability of developing code and programs leveraging a variety of libraries, regardless of whether these libraries are developed open-source or closed-source. These libraries and kernels allow for rapid prototyping and development even in a fast-paced environment such as the tech industry. This project will emphasize working with Keras, which is a high-level interface completely utilizing Python as its framework. Keras then uses TensorFlow as its backend service, the TensorFlow service is built using C++ for its fundamental or primitive

operations [Ian17]. This project particularly uses Keras to develop working Artificial Neural Networks that are then modeled using the proposed DSA.

3.4 FINAL PROPOSED DOMAIN SPECIFIC ARCHITECTURE

We have now covered which primary operations should be targeted; workloads; guidelines set by the IEEE in order to standardize floating-point storage and arithmetic operations such as multiplication and summation (MAC); and efficient schemes and algorithms for multiplying floating-point numbers. We have also covered the recommended guidelines to follow when designing a Domain Specific Architecture, which are use of faster dedicated memories; scratching advanced micro-architectural techniques; leveraging the easiest form of parallelism according to the domain; and reduction of the data operation size and design of DSA according to domain-specific language. We will now illustrate through an example how all of these ideas and concepts will be brought together.

Figure 3.11a) is an illustration of a theoretical ANN designed to solve the XOR problem. It can be observed that in this ANN, there is a total of three layers: an input layer with neurons x_1 and x_2, a hidden layer with neurons h_1, h_2 and a bias b, and an output layer with an output neuron o_1 and also a bias b. The output of this last output layer is the result y. What y actually means depends on whether the model is still in training or has already been trained, which means the model is inferring already. For this project, we have chosen to focus solely on the inferring task of ANN's.

Figure 3.11(b) is an illustration of the hardware implementation of this ANN following the five DSA guidelines. Still illustrated as circles are the inputs x_1, x_2 and b. These inputs are passed into the first hidden layer neuron, h_1. Each "neuron" is highlighted with the dotted orange rectangles. Each Functional Unit inside the dotted rectangle corresponds to a weight from an input. Once the input and the stored weight in the Functional Unit have been multiplied, a partial sum is

computed and passed on to the unit to its right. This action is illustrated with the red arrows in Figure 3.11(b). The input is "recycled" by passing it down to the next hidden neuron, h_2. This action is denoted with the blue arrows in Figure 3.11(b). The last Functional Unit inside each neuron is a little different than the rest as it has to additionally compute the activation function. For the purposes of this work, we will use the Rectified Linear Unit (ReLU) illustrated earlier in Figure 2.7. Once the activation function has been computed on the fully computed MAC, the output of that neuron is sent to the next layer; in this case it is just o_1. this action is represented using the green arrows in Figure 3.11(b). Similar to the previous layer of neurons, the partial MAC is computed in each Functional Unit and passed on to the next until reaching the final FU, where the activation function is computed and a final output y is computed.

Utilizing this proposed architecture, we have directly been able to successfully follow four out of the five guidelines for designing a DSA [Hen17]. First, we have used dedicated memories within each FU to store the weights of each neuron in order to reduce the inherent latency that comes from a traditional memory hierarchy. Secondly, we have scratched any advanced micro-architectural techniques such as out-of-order execution, multithreading, prefetching address coalescing in order to leverage more FUs on the limited silicon space we may have. We have also leveraged the easiest form of parallelism–in our case MIMD, since we are performing multiplication of weights and inputs–summation of both partial and full MACs, and computation of activation functions, all concurrently! The fourth guideline was to reduce the data size to the simplest need for the domain. We accomplished this by designing and implementing the proposed *mid-precision* floating-point representation.

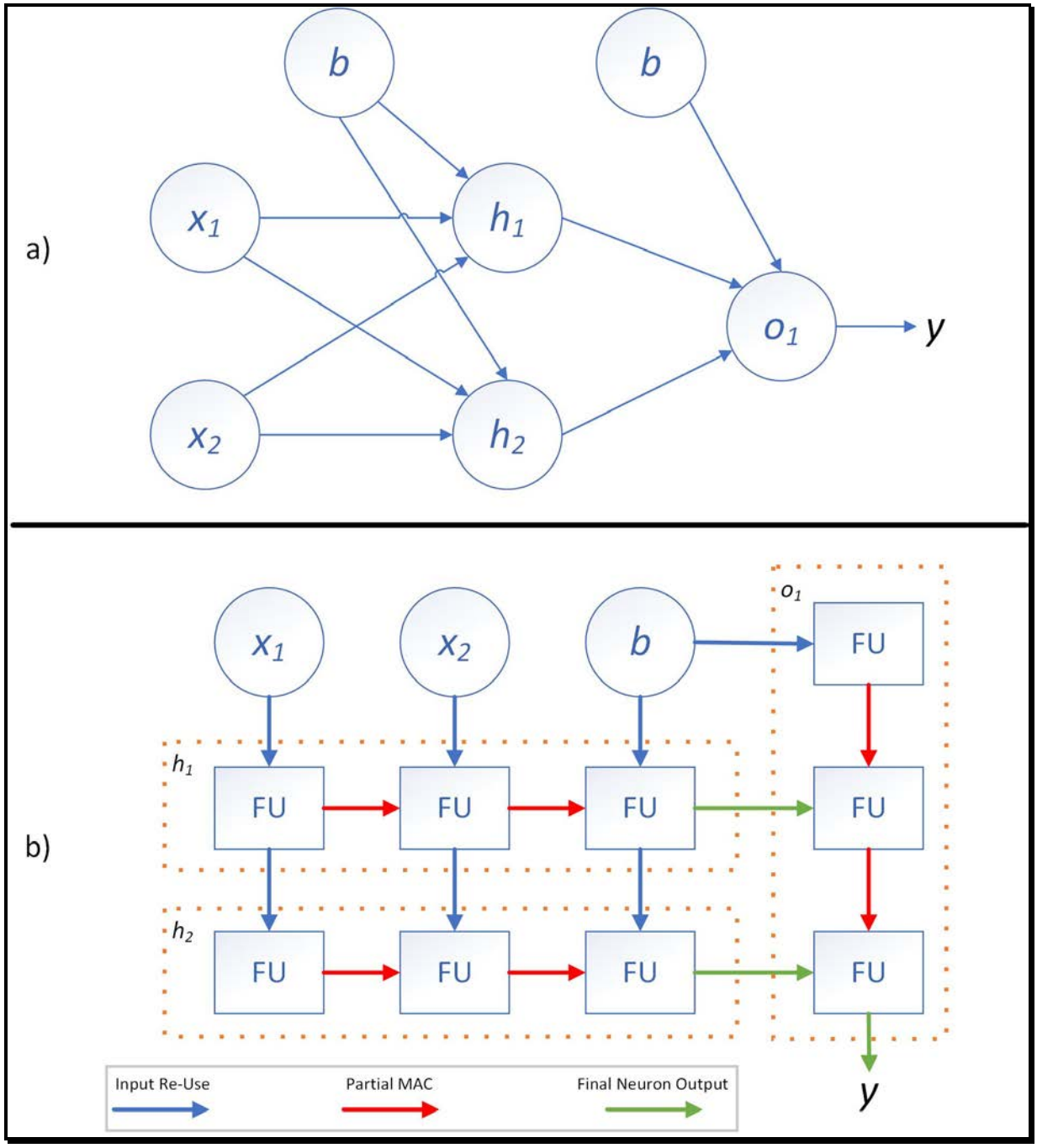

Figure 3.28: Domain Specific Architecture for an Artificial Neural Network solving the XOR problem

4 TEST-BED DESIGN & RESULTS

4.1 PRECISION REDUCTION ANALYSIS

At this point the reader may notice that the error due to the reduction in precision, from the standardized "half-precision" floating-point to the proposed "mid-precision" representation, has not been discussed yet. In order to make an appropriate assessment of the error produced in the reduction in precision, the Relative Error Percentage (REP) calculation as per Equation 5.1 is used. This formula takes the value measured, $v_{measured}$ and subtracts the target value, v_{target} and divides it by v_{target}. The result is then divided by v_{target} and multiplied by 100 to get the percentage. The reader may notice that this formula has the ability to produce a negative result which is not common with percentages, however the result of this calculation has the ability to tell us whether we "overshoot" or "undershoot" the mark given the sign of the result.

$$Relative\ Error\ \% \ = \ \delta \ = \ \frac{\left(v_{measured} - v_{target}\right)}{v_{target}} \times 100 \tag{4.1}$$

Equation 4.5: Relative Error Percentage formula

To get an estimate of the loss of precision, the difference between the numbers highlighted in green and the numbers highlighted in yellow, are used. The value of the difference is divided by the 16-bit value. This results in an average REP of -0.395%, meaning that when truncating the floating-point representation from 16-bits to 12-bits, on average the numbers can be expected to be only 0.395% larger than the original representation. This is also a very positive result because it can also be expected to see very little difference in the end result, when the network is trained and ready to infer.

```
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++Hidden Layer++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
########################################16-bit##############################################
(2, 2)
[[ 0.7864033 -0.5324216]  [[0.7866  -0.532]
 [-0.9421562  0.5320883]]  [-0.9424  0.532]]

[[0011101001001011 1011100001000010]
 [1011101110001010 0011100001000010]]

Bias
(2,)
[-2.1904702e-03 -3.6245037e-05] [-2.19E-3 -3.624E-5]

[1001100001111100 1000001001100000]

########################################12-bit##############################################
(2, 2)
[[ 0.7864033 -0.5324216]  [[0.7813  -0.5313]
 [-0.9421562  0.5320883]]  [-0.9375  0.5313]]

[[001110100100 101110000100]
 [101110111000 001110000100]]

Bias
(2,)
[-2.1904702e-03 -3.6245037e-05] [-2.167E-3 -3.624E-5]

[100110000111 100000100110]

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++Output Layer++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
########################################16-bit##############################################
(2, 1)
[[1.2867637]      [[1.287]
 [1.8850328]]      [1.885]]

[[0011110100100110]
 [0011111110001010]]

Bias
(1,)
[-0.00084873]     [-8.49E-4]

[1001001011110100]

########################################12-bit##############################################
(2, 1)
[[1.2867637]      [[1.281]
 [1.8850328]]      [1.875]]

[[001111010010]
 [001111111000]]

Bias
(1,)
[-0.00084873]     [-8.47E-4]
```

[100100101111]

4.2 COMPARISON OF 16 BIT FLOATING-POINT MULTIPLIERS

First, verification of the accuracy of the proposed floating-point multiplier, implementing Booth's algorithm using 16-bit operands, is made. In order to produce the operands and verify the final result an online tool developed by Dr. E. Weitz at the Hamburg University of Applied Sciences [Wei16] is used. This tool allows the user to provide two operands in decimal floating-point format, then it translates these operands to binary floating-point format. The user is then allowed to select an operation (+, -, ×, ÷) which, in turn, produces the final result in both binary and decimal format. The testbench was manually created to include situations in which the operands where either both are 0, one is 0, both are positive, one is positive and the other negative, or both are negative. Figure 4.1 is an illustration of the final results, and close examination shows that the multiplier does, in fact, provide accurate results, and with very low latency.

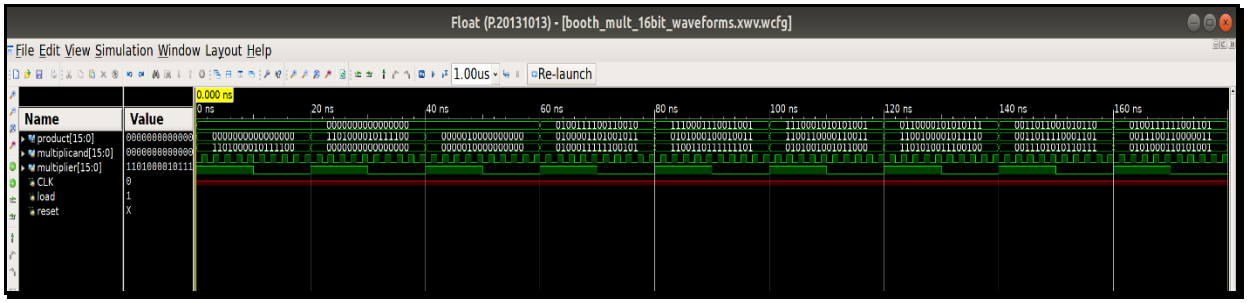

Figure 4.29: Testbench waveform of designed Booth's Algorithm multiplier using 16bit operands

Once the accuracy of the Booth's Alg. multiplier is verified, it was decided to test the latency of performance of this design vs. the "out-of-the-box" multiplier supplied by the Xilinx LogiCORE IP Floating-Point Operator v6.1. Figure 4.2 is a schematic representing the latency testbench setup. The overall module consists of 4 inputs: the multiplicand (multA), the multiplier (multB), the clock (CLK), the "operands ready/valid" signal (LOAD) and a reset signal. The outputs used for comparison, to test latency, are the final output of the Booth's Algorithm module (booth_product) and the final output of the Xilinx module (Xilinx_product). It was also decided to incorporate a final accuracy test of the Booth's Algorithm module by connecting both multipliers to a 16-bit comparator and verifying that the final products of both modules were equal.

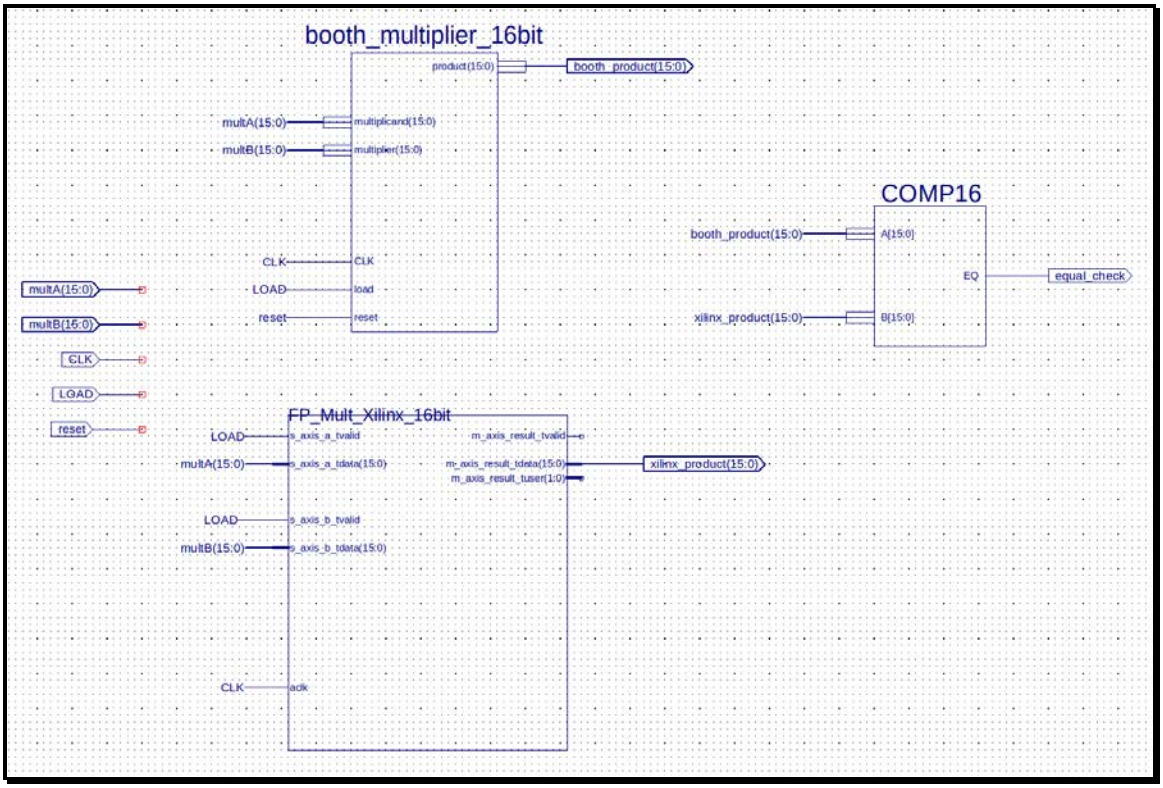

Figure 4.30: 16bit Floating-Point multiplier comparison schematic used to compare latency

In order to make this a fair and valid comparison, when creating the floating-point multiplier using the LogiCORE toolkit, we opted for a non-blocking implementation of the module. According to the documentation [Xil12], the latency of a floating-point multiplication

with a mantissa of 11 bits can range anywhere from 0 to 6 clock cycles. Figure 4.3 represents the final waveforms of the testbench designed to test the head-to-head latency of both multipliers. The equality check signal is highlighted as the blue signal in Figure 4.3, and can be seen to go active as soon as both multiplier outputs are final. The yellow signal is the output from the Xilinx multiplier and the top most bright green signal is the output from the Booth multiplier. It is satisfying to report that the Booth module outperformed the Xilinx module in every test case, by providing nearly instant results (Xilinx does not have a way to model propagation delay of a circuit unless a target FPGA board is selected). Though Booth's algorithm seems as it should be sequential, once synthesized, it was actually implemented as a purely combinational implementation hence outperforming the Xilinx module. Because the Xilinx documentation does not provide an average value for latency, we can only estimate and predict what speed up this new multiplier would provide. Assuming that an average latency for the Xilinx multiplier is 3 clock cycles [(0+6)/2 = 3], we could actually expect an average speed up of ×3!

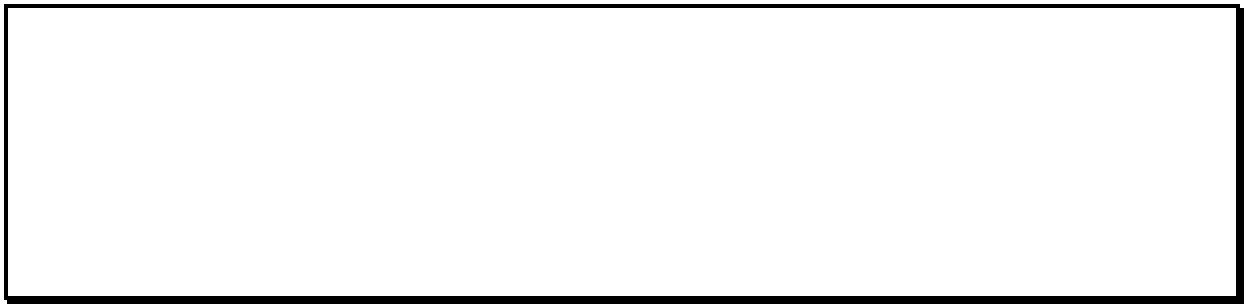

Figure 4.31: Waveform results from 16bit floating-point multiplier comparison

4.3 COMPARISON OF 12 BIT FLOATING-POINT MULTIPLIERS

Simlar to the initial accuracy testing in section 4.1, the operands were used to produce and verify the final result using a tool [Wei16] similar to that in section 4.1. However, this time since we are dealing with 12 bit operators, the last 4 bits were reset in order to measure the overall loss in accuracy. The test bench was also created to include multiple situations, in which the operands are either both 0, one is 0, both are positive, one is positive and the other negative, and both negative. Figure 4.4 is an illustration of the final results and, again, the multiplier does, in fact, provide close to accurate results (with respect to the 16 bit representation), and does so with very low latency.

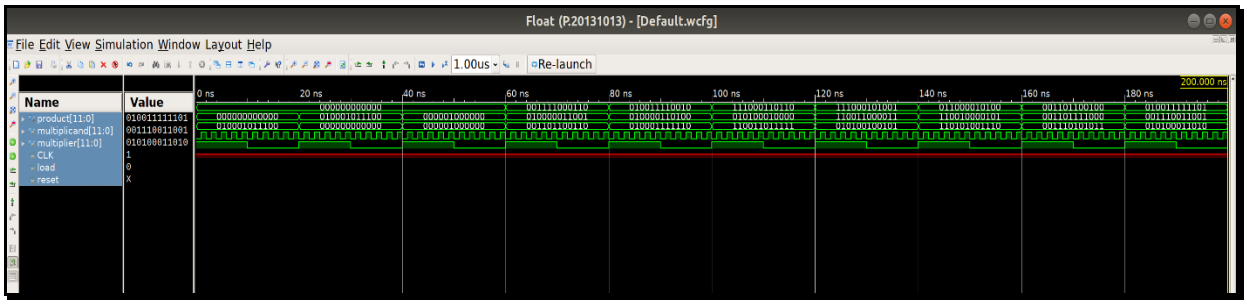

Figure 4.32: Testbench waveform of designed Booth's Algorithm multiplier using 12bit operands

Once the accuracy of the results of the Booth multiplier were verified, it was decided to also test its latency performance. Thus the newly designed multiplier is compared to the multiplier supplied by the Xilinx LogiCORE IP Floating-Point Operator v6.1, where it was customized to have a mantissa of only 6 bits. Figure 4.2 is a schematic representing the latency testbench setup. The overall module consists of 4 inputs, the multiplicand (multA), the multiplier (multB), the clock (CLK), the “operands ready/valid” signal (LOAD) and again a reset signal. The reader may notice a difference between this schematic and that one shown in Figure 4.2, due to the way modules are generated in the LogiCORE toolkit, inputs and outputs can only be produced in mutliples of 8. Hence the need to ground bits and store intermediate results in buffers waiting to feed both

multiplier modules. The outputs used to test the latency are the final output of the Booth module (booth_product) and the final output of the Xilinx module (Xilinx_product). It was also decided to incorporate a final accuracy test of the Booth module by connecting both multipliers to a 4 bit comparator and an 8 bit comparator verifying that the final products did, in fact, match.

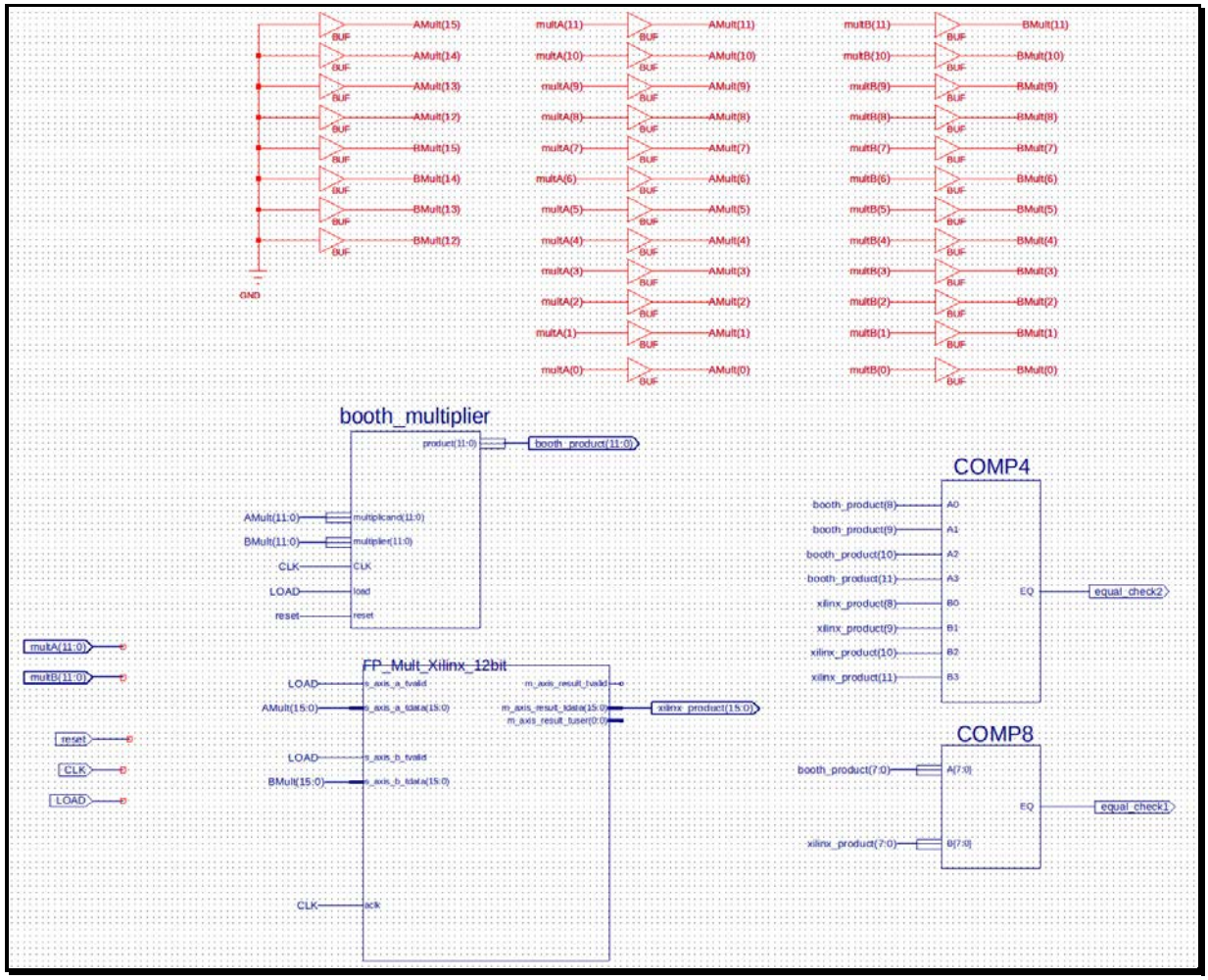

Figure 4.33: 12bit Floating-Point multiplier comparison schematic used to compare latency

In order to make this a fair and valid comparison, when creating the floating-point multiplier using the LogiCORE toolkit, a non-blocking implementation of the final module was chosen. According to the Xilinx documentation [Xil12], the latency of a floating-point multiplication with a mantissa of 6 bits can range anywhere from 0 to 6 clock cycles, as reported previously. Figure 4.6 represents the final waveforms of the testbench that was designed to test the

head-to-head latency of both multipliers. The equality check signals are highlighted as before, as indicated by the blue signals in Figure 4.3, and can be seen to go active as soon as both multiplier outputs are final. The yellow signal is the output from the Xilinx mutliplier and the top-most bright green signal is the output from the Booth multiplier. It is very satisfying to report that the Booth module again outperformed the Xilinx module in every test case. By similar analysis as in section 4.1, one can come to the conclusion that the expected average speed can be up to ×3!

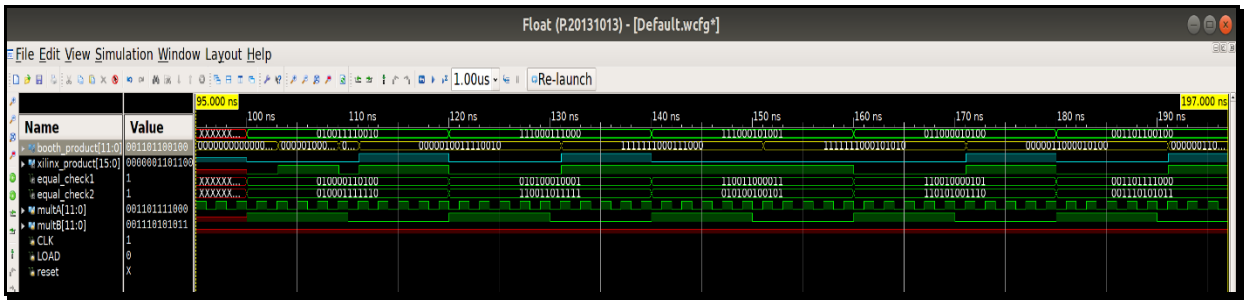

Figure 4.34: Waveform results from 12bit floating-point multiplier comparison

4.4 TESTING DSA ANN WITH XOR PROBLEM

Historically, the XOR problem has proven to be a challenging area for early ANN models. The manner in which early perceptron models failed had initially cast much doubt on the practicality of ANNs for most of the 1960's and 70's. As mentioned in Chapter 1, the reason that these early models struggled so much with this problem was due to the non-linear separability of the problem. In other words, this problem generates two categories of outputs that cannot be divided by a single line. Accurate categorization can only be achieved by using two methods: by suing the backpropagation training method, discussed in section 2.4 [Day90] and by employing the use of MLPs or Deep Neural Networks [Rea17]. Due to the historically challenging aspect of this problem, in addition to its naturally simplistic interpretation, it was decided to use the XOR problem as the Proof of Concept (POC). That is, the proof that the proposed DSA does, in fact, perform as expected.

4.4.1 SOFTWARE BASED DESIGN ANN USING KERAS

The code used to develop a working simulated ANN is shown in part A of Appendix A. Here, the final product produced with this simulation is illustrated. The network consists of a total of three layers, an input layer, a hidden, and a final hidden layer. The input layer consists of two fully connected neurons, and a bias neuron. The hidden layer consists of two fully interconnected neurons, plus a bias neuron, and the final layer consist of a single neuron with a bias input. The output of this final neuron is a value determing whether the final output will be a 1 or 0. Figure 4.7 is the final output produced from a fully trained ANN using Keras as the modeling tool. The optimizer used was Adam with a learning rate of 0.0035 and the activation function used was the Rectified Linear Unit (ReLU), as discussed in Chapter 2. The reasoning for using the ReLU is due

to its simple, yet modern, implementation. Figure 4.8 is a theoretical model illustrating the neurons with the associated weights of each input and output.

```
PREDICTION WITH MODEL
[[0.       ]
 [1.0020868]
 [1.0082479]
 [0.       ]]

#############
MODEL WEIGHTS
#############
HIDDEN LAYER
(2, 2)
[[ 0.7864033 -0.5324216]
 [-0.9421562  0.5320883]]
(2,)
[-2.1904702e-03 -3.6245037e-05]

OUTPUT LAYER
(2, 1)
[[1.2867637]
 [1.8850328]]
(1,)
[-0.00084873]

MODEL SUMMARY
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 2)                 6
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 3
=================================================================
Total params: 9
Trainable params: 9
Non-trainable params: 0
_________________________________________________________________
```

Figure 4.35: ANN used to solve XOR problem modeled in Keras

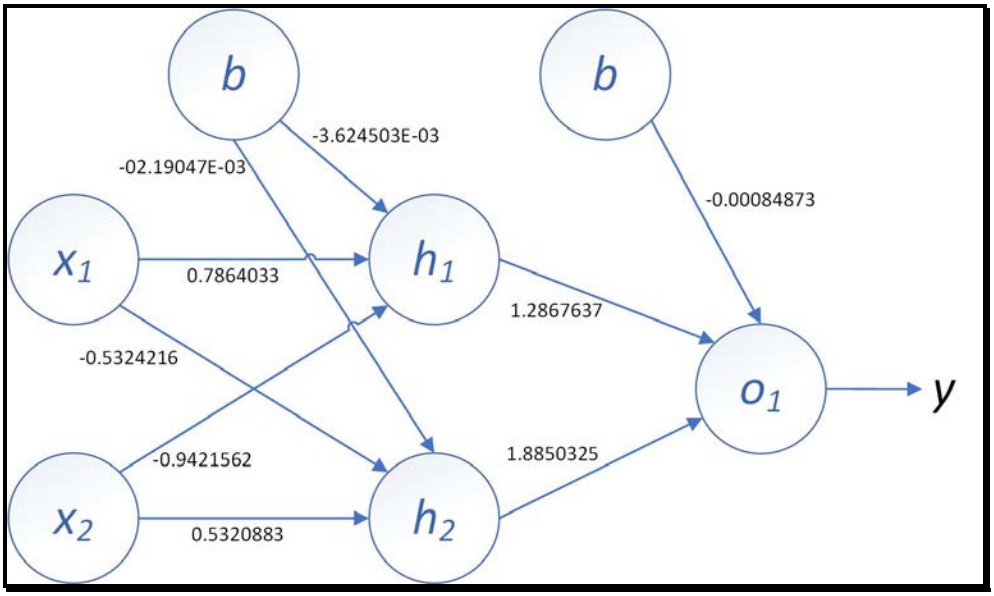

Figure 4.36: Theoretical illustration of ANN designed to solve the XOR problem

4.4.2 DSA-BASED DESIGN ANN USING 16 BIT OPERANDS

First, the schematic for the Functional Unit (FU) utilizing the 16-bit half-precision operands is presented. Illustrated in Figure 4.9 is the designed FU, starting from left to right in the schematic, the top-left shows the inputs to the unit. These consist of the *input vector* which, in most cases, is the data used to make an inference, the *weight* to be stored and associated with the FU, the bit flag indicating whether the unit is ready to store a new weight, the operands ready flag, a clock, and a global reset enabling an override of the current weight stored in the FU. The outputs of the FU are: the newly produced updated partial sum, denoted as UPDATED_P_SUM(15:0), a result ready flag, the "recycled" input vector enabling the re-use of input vectors enabling the suggested Temporal Architecture and flags indicating if there was an overflow or underflow in the FU.

The processing portion of the FU consists of four major components: two registers, a floating-point multiplier and a floating-point adder. The register on the far left in Figure 4.9 is used to store the weight of the FU and serves to provide one of the operands for the floating-point

multiplier. The register illustrated in the top of the diagram serves as a temporary storage for the input vector in order to enable the re-use and communication among these FUs. The designed floating-point multiplier, implementing Booth's algorithm, is used in order to obtain the project ×3 speed up. The "bottleneck" in this design comes from the Xilinx LogiCORE IP Floating-Point Adder. The adder produced from the LogiCORE toolkit can take anywhere from 0-8 clock cycles to produce the final result. Thus, taking a conservative average of a 4 clock cycle delay, one can see that this is where most of the delay comes from, as all other modules in this unit only take a clock cycle to complete operation.

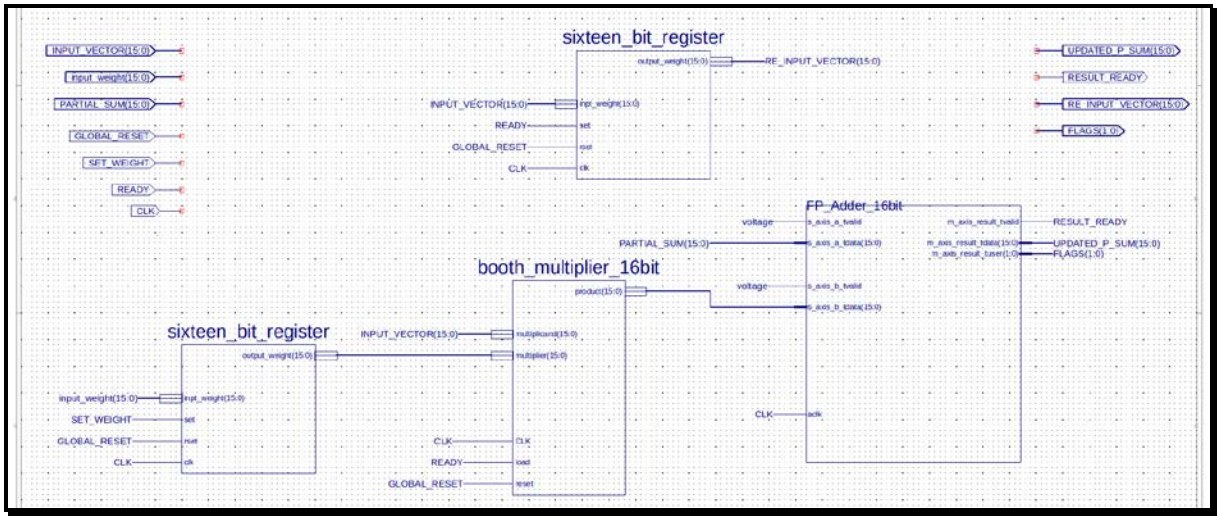

Figure 4.37:Functional Unit Schematic utilizing 16-bit (half-precision) operands

Figure 4.10 is a waveform verification that the FU operates as expected. The test-bench for this portion of the verification is designed to produce an output given three 16-bit floating point operands, 1.0 as the input vector, 0.786 as the FU weight and 0.0 as the partial product, these input values were chosen because they are real inputs the final MLP XOR DSA will see during final testing. The resulting output (0.786) was then re-introduced into the FU as the new partial product with an input vector equal to 0.0, another test case of true inputs the tested DSA will see in final testing. The final result of the updated partial sum was 1.572, as expected.

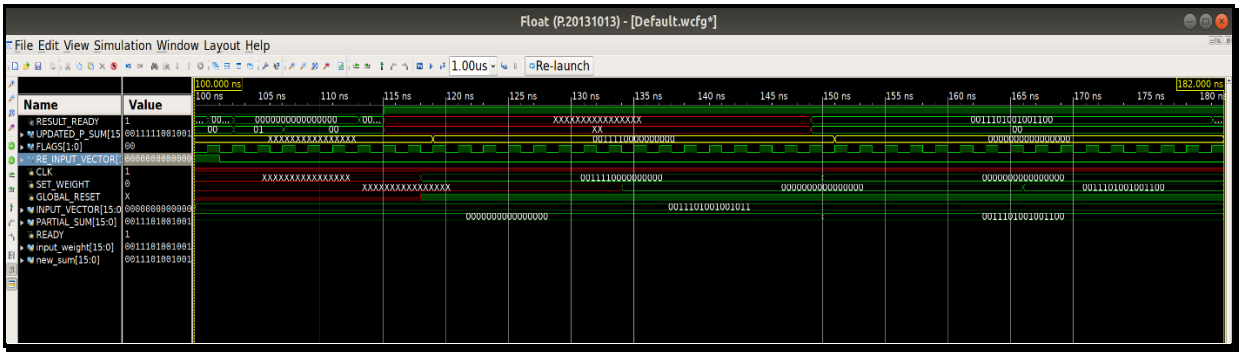

Figure 4.38: Waveform test verification of 16-bit Functional Unit

Figure 4.11 is the finalized MLP DSA designed to solve the XOR problem. The first row of FU's (or Processing Engines, PEs) corresponds to the three weights beloging to h_1 in Figure 4.8. The first FU is associated with the weight coming in from x_1, the second is the weight coming in from x_2 and the final FU is the weight associated with the bias (b). The seccond row of FUs is set up the same way as the previous row. It can be observed from the schematic that there are connections running from the first set of FUs to the second set of FUs. This is where the re-use of the input vectors is leveraged, in order to implement the combination of the systolic array and the temporal architecture. The two components below the second row of FUs serve as the activation functions of the artificial neurons. Because of the simple, yet modern, function of ReLU, as illustrated in section 2.7, it is possible to implement the ReLU activation function using a 23 to 16 MUX with one select line. The MUX is connected to the final MAC (denoted as NeuronX_excitment(15:0) in the schematic) of its corresponding neuron, the select line is then connected to the sign bit of the neuron excitement. If this bit is a 1, it means the excitement is negative, therefore "firing" all 0's. A bit 0 will "fire" the neuron excitement as per the ReLU activation function. The last row of FUs implement the weights of the final neuron o_1 as per Figure 4.8. It should be noted that the input vector re-use output is not connected to any aditional FUs as this layer consists of only one neuron. Similar to the previous layer, this layer is connected to a MUX implementing the ReLU activation function. The very bottom of the schematic consists of

all the inputs and outpus the module contains. More explicitly, these of course include the input vectors, the bias, a zero vector feeding that initial partial summation, ready flags, weight initializer vectors, overflow/underflow flags and the final output which is the inferrence of the network.

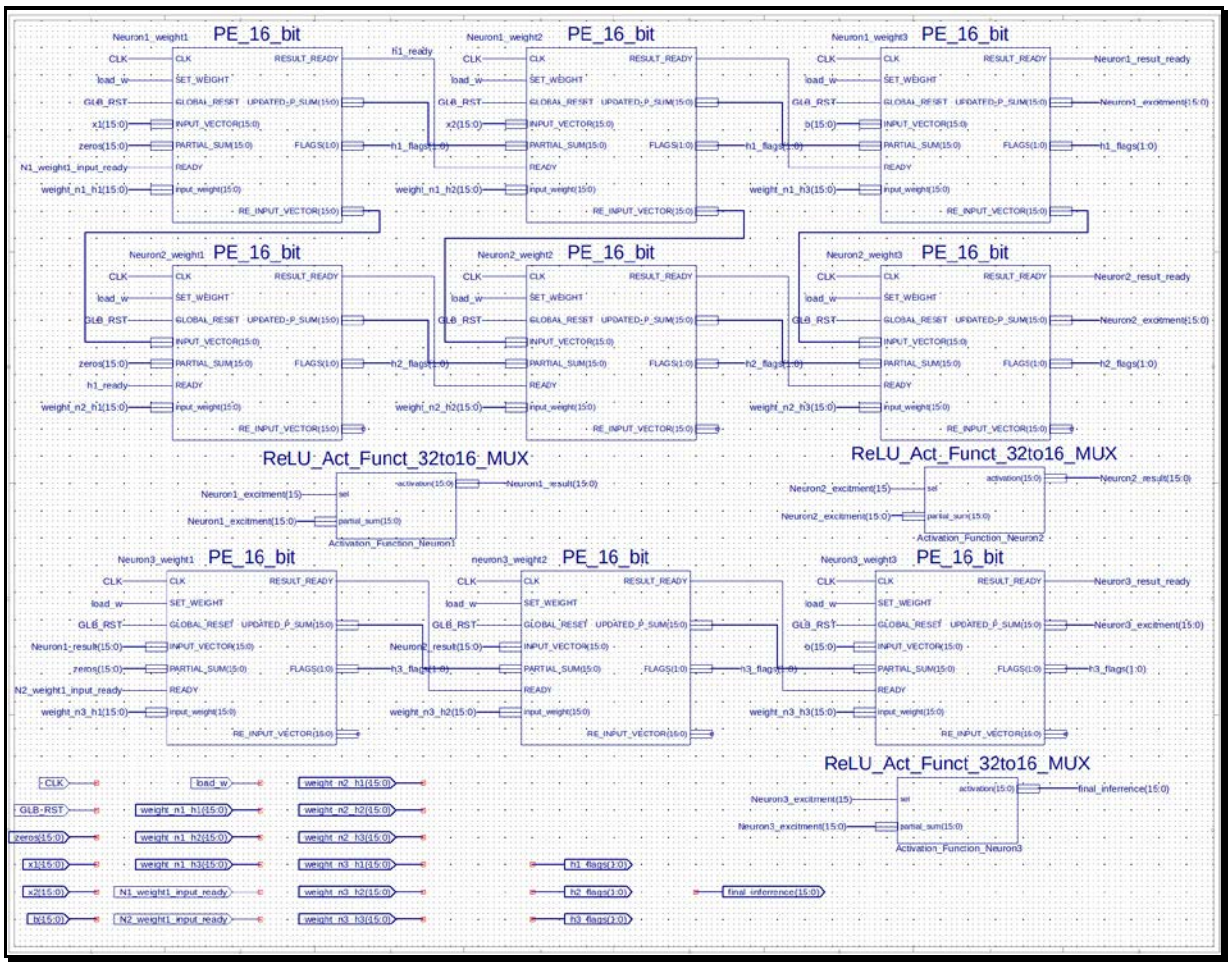

Figure 4.39: MLP DSA designed to solve XOR problem using 16-bit (half-precision) operands

Figure 4.12 is the final waveform output showing that the architecture does in fact work as designed. It can be seen that the final_inference, marked as the blue signal in Figure 4.12, is set after the ninth cycle of inputs, which is consistent with that "delay" of one feed cycle that was projected, due to the systolic array nature of this architecture. The clock pulses also illustrate the overall operation cycle delay of 8 clock cycles, due to the floating-point adder provided from the Xilinx LogiCORE toolkit.

In order to verify the accuracy of our final inferrences, the output signal is translated to a decimal format. Equation 5.1 is used to obtain the REP per set of inputs, and a final average REP is obtained. These results are illustrated in Table 5.1, where it can be observed that the average REP turns out to be only 0.188%, meaning that on average the true result is “overshot” only by a miniscule margin!

Table 4.3: Relative Error Percentage Calculation, MLP DSA 16-bit operands

X_1	X_2	$Y_{EXPECTED}$	$Y_{MEASURED}$	REP
0.0	0.0	0.0	0.0	0.0%
1.0	0.0	1.0	1.009	0.9%
0.0	1.0	1.0	0.9985	-0.15%
1.0	1.0	0.0	0.0	0.0%
			Avg. REP	**0.188%**

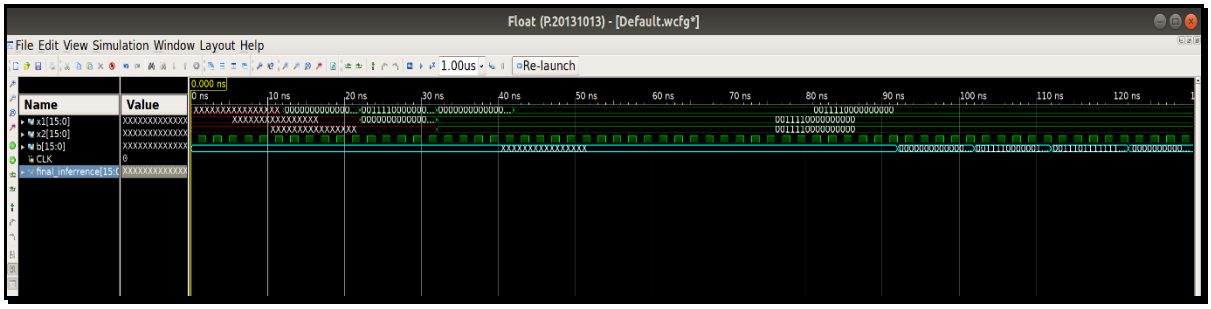

Figure 4.40: MLP DSA designed to solve XOR waveform verification, 16-bit operands

4.4.3 DSA-BASED DESIGN ANN USING 12 BIT OPERANDS

Now the schematic design for the Functional Unit utilizing the proposed 12-bit mid-precision operands, illustrated in Figure 4.12, is presented. Similar to the schematic from Figure 4. 11, the top left of the schematic shows the same inputs as in the 16-bit FU. There are 12-bit vectors for the weight associated with the FU and the input vector. There are a set weight, input ready, global reset bits and clock inputs. The most notable detail the reader may note is that the

partial sum vector is actually composed of 16-bits instead of 12. This nuisance is due to how the floating-point modules are created and synthesized in the Xilinx LogiCORE toolkit. Regardless of how the user specifies the mantissa and exponent size, these module can only be created in multiples of 8. If there are bits left over, as is the case in this instance where we have four unused bits, they are the four MSBs, and treated as Don't Cares. This, however, proves to be an issue when trying to connect a 12-bit bus to a 16-bit bus. In the lower half of the schematic it can be observed how the issue has been solved: the 12-bit bus is passed through an intermediate set of buffers in order for this to be connected to the wider bus. The outputs of this FU are similar to the ones in the 16-bit FU: a newly produced updated partial sum, denoted also as UPDATED_P_SUM(15:0), a result ready flag, the "recycled" input vector also enabling the re-use of input vectors implementing the suggested Temporal Architecture and systolic array.

The processing portion of the FU comes from the same four major components; two registers, a floating-point multiplier, and a floating-point adder. The only thing differentiating this FU from the previous one in section 4.2.2 are the operand sizes. Here, the same performance "bottleneck" with the Xilinx LogiCORE toolkit exists as before, as it also takes anywhere from 0-8 clock cycles to produce a final result, where all other modules in this unit only take one clock cycle to complete operation.

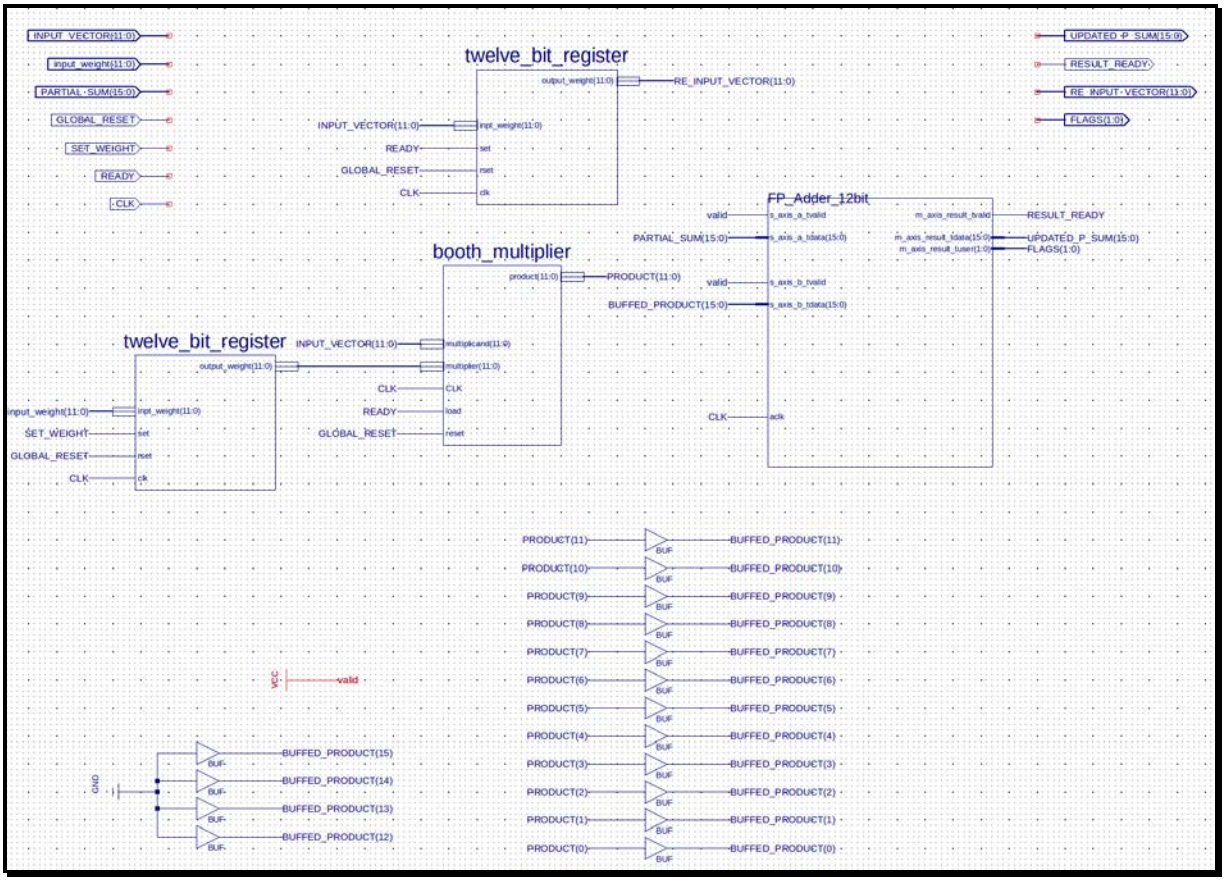

Figure 4.41: Functional Unit Schematic utilizing proposed 12-bit (mid-precision) operands

Figure 4.13 is a waveform verification showing that the FU operates as expected. The test-bench here is similar to that in section 4.2.2: we have 1.0 as the input vector, 0.7813 as the FU weight and 0.0 as the partial product. The resulting output (0.7813) was re-introduced into the FU as the new partial product with an input vector equal to 0.0. Another test case of true inputs to the tested DSA will be seen in final testing. The final result of the updated partial sum was 1.5625. This is not exactly 1.5626, due to the negligible loss of precision, but extremely close, and within tolerance parameters.

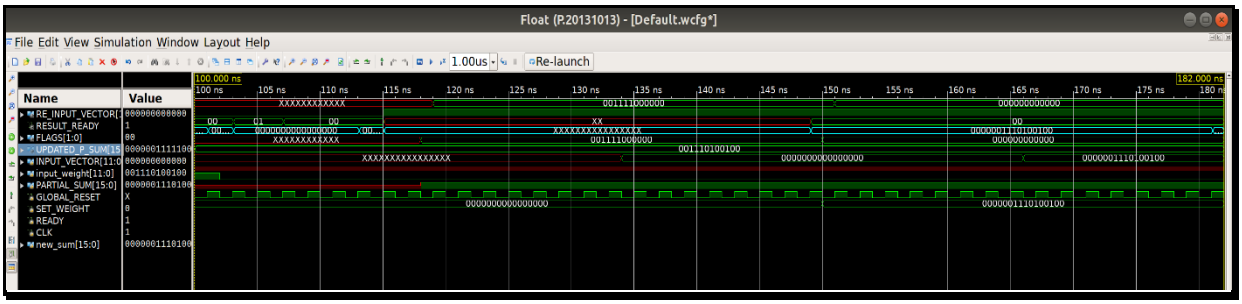

Figure 4.42: Waveform test verification of 12-bit Functional Unit

Figure 4.15 is the final MLP DSA designed to solve the XOR problem. The most notable difference between this architecture and the one shown in Figure 4.11 is the fact that this architecture uses 12-bit operands, therefore also affecting the MUXs used to implement the ReLU activation function. This schematic also contains the same inputs and outputs as its 16-bit counterpart: the input vector's x_1, x_2, the bias, a zero-vector feeding the initial partial summation, ready flags, weight initializing vectors, overflow/underflow flags and the final output which is the inference of the network.

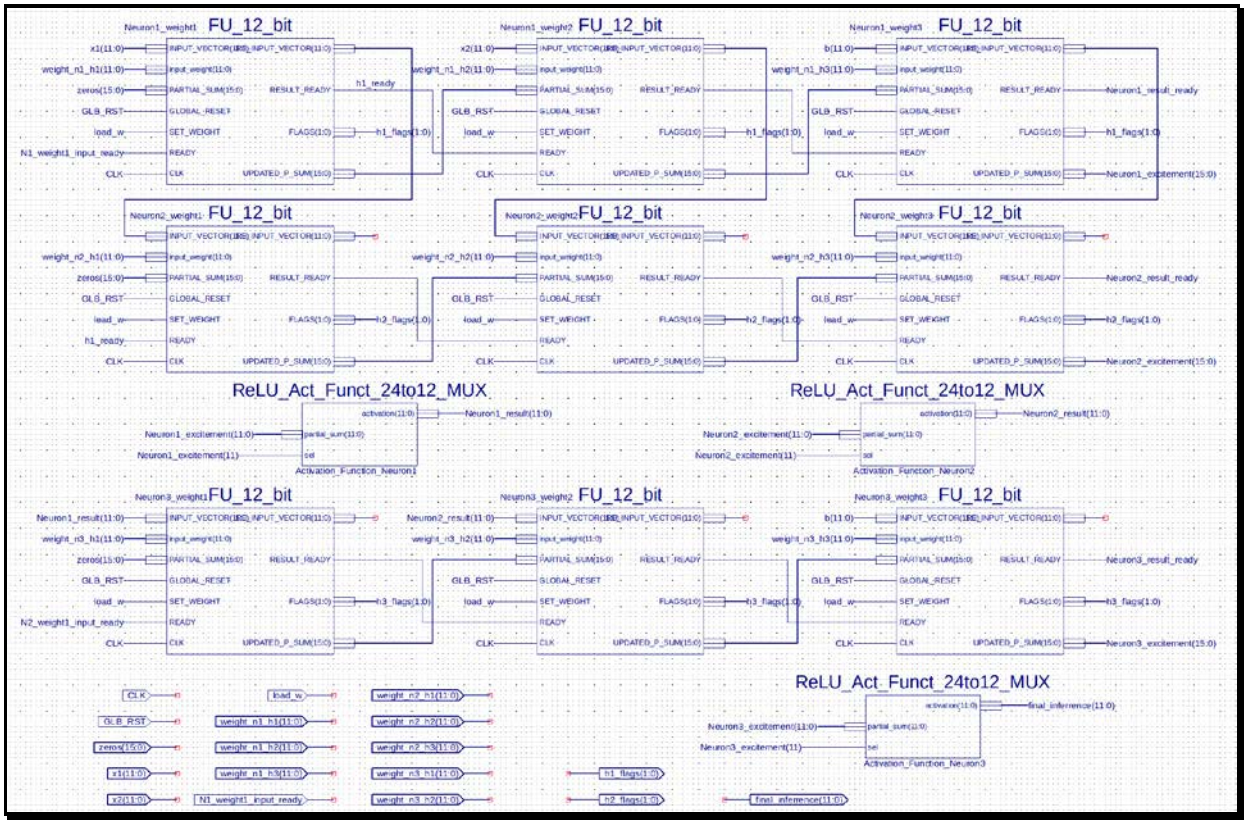

Figure 4.43: MLP DSA designed to solve XOR problem using the proposed 12-bit (mid-precision) operands

Figure 4.16 is the final waveform output showing that this architecture also does in fact perform as designed. It can be seen that the final_inference, again illustrated as the blue signal in Figure 4.16, is set after the ninth cycle of inputs, which is consistent with the feed cycle delay originally projected. Here it can also be seen, as per the clock pulses, how each overall operation has a cycle delay of 8 clock cycles, due to the floating-point adder provided from the Xilinx LogiCORE toolkit.

The accuracy of our final inferences is verified, with this architecture and operand size, by translating the final output into decimal format. The REP formula is used, again, to obtain the error per set of inputs, as well as a final average REP. These results are illustrated in Table 5.2, and it can be observed that the decrease in accuracy has caused the system to “undershoot” or underestimate the result by a margin of 0.958%, which is still a very miniscule margin.

Table 4.4: Relative Error Percentage Calculation, MLP DSA 12-bit operands

X_1	X_2	$Y_{EXPECTED}$	$Y_{MEASURED}$	REP
0.0	0.0	0.0	0.0	0.0%
1.0	0.0	1.0	0.9766	-2.34%
0.0	1.0	1.0	0.9844	-1.56%
1.0	1.0	0.0	0.0	0.0%
			Avg. REP	**-0.968%**

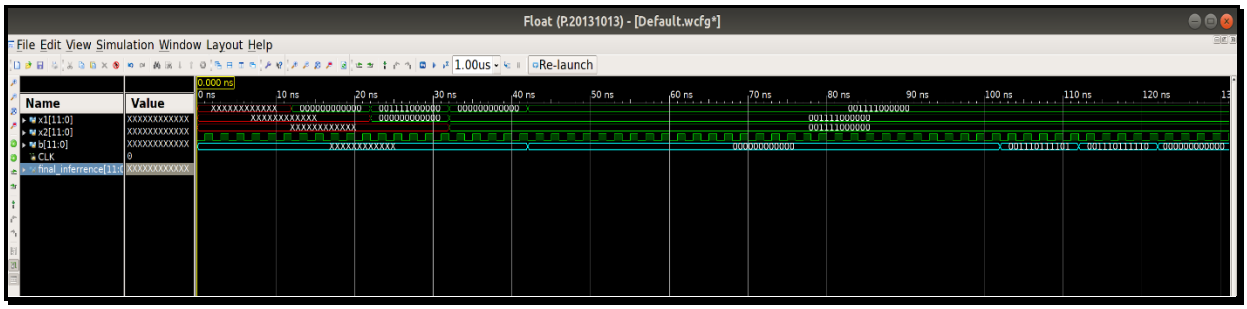

Figure 4.44: MLP DSA designed to solve XOR waveform verification, 12-bit operands

5 CONCLUSION & FUTURE WORK*CONCLUSION*

Artificial Intelligence, Machine Learning and Artificial Neural Networks have taken huge strides the last 20 years and it's in part greatly due to the Era of Big Data and the Internet of Things (IoT). This saturation of data has enabled scientists and engineers to create ANNs that are capable of doing things we thought was science fiction only a few years ago. However, as these networks become more complicated and more computationally intensive, a demand has risen for computer engineers to develop hardware that is capable of handling these workloads. The aim of this study is to explore the design of a Domain Specific Architecture for Artificial Neural Networks, as per suggested guidelines from experts in the field. These guidelines aim to provide several benefits that could not be found in traditional CPUs and GPUs: they aim to lower power consumption and to provide quicker computations. These guidelines are as follows:

- The use of dedicated memories to minimize the distance over which data is moved;
- Investing resources saved from dropping advanced µ-architectural optimization into more FUs or bigger memories;
- Leveraging the easiest form of parallelism per the domain;
- Reducing the data size to the absolute minimum as per the domain and application; and
- Use of domain-specific programming language to port the code to the DSA.

In this study, we have managed to leverage 4 of these 5 suggested guidelines. We realized the use of dedicated memories by designing the FUs with a local register's capable of holding the associated weights of the neuron. We completely eliminated µ-architectural optimizations and designed a floating-point multiplier, following Booth's algorithm, that was able to obtain the output of the operands in one clock cycle. Furthermore, the easiest form of parallelism was leveraged by applying a combination of a Temporal Architecture and Systolic array, allowing for multiple instructions' execution, with multiple sets of data, simultaneously (MIMD). The final guideline leveraged was to reduce the data size to the absolute minimum. This was accomplished by producing two architectures, one with a *half-precision* 16-bit operand and proposing a new *mid-precision* 12-bit operand by truncating some of the mantissa bits. This approach, on average, only reduced precision by -0.39%, meaning we "undershot" the target output, on average, by a 0.39% margin.

5.2 FUTURE WORK

For future work, there are three primary areas that would be important to target:

- performance assessment with CNN and Recurrent Neural Networks (RNN);
- the development of a "tool" with the capability of mapping and allocating the appropriate hardware resources based on a software implementation of an Artificial Neural Network; and
- the exploration of fixed-point operands for inferring.

Due to the nature of many of today's applications of Artificial Neural Networks, image processing, object identification, natural language processing and real time translation, all of which employ some form of CNN or RNN, we believe that it would be of great benefit to test and the performance of this DSA and its FU's. If we can recall back from section 3.1, the discrete convolution is in short defined as, the ***summation*** of the ***product*** of two functions. Because of this fact we believe CNN's will greatly benefit primarily from the Booth multiplier developed in this study, and also the proposed Spatial Architecture. We can expect to see at the minimum ×3 speed with help from the Booth multiplier module and a much more efficient use and re-use of updated partial summations, therefore positively affecting power consumption of this DSA. We believe RNN's will see the most improvement from the FU architecture, because the RNN architecture relies on some sort of "memory" of previous inputs we believe that if by slightly expanding that internal register file or the internal memory we will also get a more efficient use of data movement again having a positive effect on the power consumption of the DSA.

We would also like to develop a "tool" that has the capability of identifying the implemented ANN in software and from this identification maps the software ANN architecture

to a hardware implementation with the proposed DSA in this study. An illustration of this tool is shown in Figure 5.1, this tool is very similar to the steps taken by a compiler/assembler in traditional software engineering. In traditional software engineering a compiler is in charge of taking the source code or program and translates it to assembly code (in some cases optimized too) that is specific to a machines µ-architecture. It is then the assembler's job to take this newly produced assembly code and assembles it to machine code which the computer can now finally understand. The tool proposed works in a similar manner, the tool would be able to map and optimize an ANN modeled in software to the proposed DSA using FU's and the systolic array. The final step would be the ability to check if it is possible to load the newly compiled DSA to a targeted FPGA board.

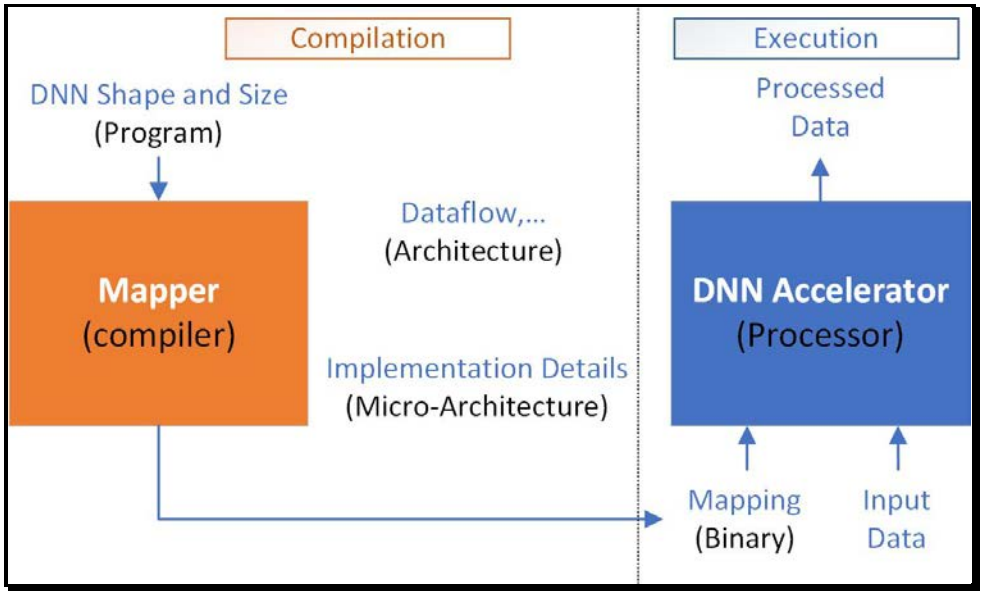

Figure 5.45: Analogy between resource "mapping" tool and traditional software compiler

The last suggestion for an area of future work is to explore the use of fixed-point operands as opposed the floating-point operands used in this study. Google's *TPU* [Jou17] makes use of the 8-bit fixed-point operands by default, this is done to improve the overall throughput and energy efficiency of the architecture. This 8-bit implementation has shown to have less energy and less

area consumption than the IEE 754 16-bit floating-point operation by a factor of 6. Of course, this is from an inferring standpoint only, due to the aggressive reduction in precision of the operands it becomes extremely difficult to train these networks using these types of operands.

REFERENCES

[Abr17] Prasad, B. M., Singh, K. K., Ruhil, N., Singh, K., & OKennedy, R. (2017). Study of Various High-Speed Multipliers. Boca Raton: CRC Press.

[Che16] Chen, Y., Krishna, T., Emer, J., & Sze, V. (2016). 14.5 Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. 2016 IEEE International Solid-State Circuits Conference (ISSCC). doi:10.1109/isscc.2016.7418007

[Cvn16] "Complete Visual Networking Index (VNI) Forecast," Cisco, June 2016

[Day90] Dayhoff, J. E. (1990). Neural Network Architectures: An Introduction. New York, NY: Van Nostrand Reinhold.

[Fis36] Fisher, R. A. (1936). The Use Of Multiple Measurements In Taxonomic Problems. Annals of Eugenics, 7(2), 179-188. doi:10.1111/j.1469-1809.1936.tb02137.x

[Fly72] Flynn, M. J. (1972). Some Computer Organizations and Their Effectiveness. IEEE Transactions on Computers, C-21(9), 948-960. doi:10.1109/tc.1972.5009071

[Hab70] A. Habibi and P. A. Wintz, "Fast Multipliers," in IEEE Transactions on Computers, vol. C-19, no. 2, pp. 153-157, Feb. 1970. doi: 10.1109/T-C.1970.222881

[Hen12] Alexander, P., Ashenden, P. J., Bruguera, J., Patterson, D. A., & Hennessy, J. L. (2012). Computer Organization and Design: The Hardware/Software Interface (4th ed., Revised Printing). Amsterdam: Elsevier.

[Hen12a] Hennessy, J. L., Arpaci-Dusseau, A. C., & Patterson, D. A. (2012). Computer Architecture: A Quantitative Approach Fifth Edition. San Francisco: Morgan Kaufmann.

[Hen17] Hennessy, J.L., Arpaci-Desseau, A. C., & Patterson, D. A. (2017) Computer Architecture: A Quantitative Approach Sixth Edition. San Francisco: Morgan Kaufmann.

[Hml18] A history of machine learning. (n.d.). Retrieved April 17, 2018, from https://cloud.withgoogle.com/build/data-analytics/explore-history-machine-learning

[Ian17] Goodfellow, I., Bengio, Y., & Courville, A. (2017). Deep Learning. Cambridge, MA: MIT Press.

[IEE08] 754-2008 IEEE Standard for Floating-Point Arithmetic. (2008). IEEE / Institute of Electrical and Electronics Engineers Incorporated.

[Jou17] Jouppi, N. P. et al. In-datacenter performance analysis of a Tensor Processing Unit. Proc. 44th Annu. Int. Symp. Comp. Architecture Vol. 17 1–12 (2017)

[Len90] Douglas B. Lenat, R. V. Guha, Karen Pittman, Dexter Pratt, and Mary Shepherd. 1990. Cyc: toward programs with common sense. Commun. ACM 33, 8 (August 1990),30-49. DOI=10.1145/79173.79176 http://doi.acm.org/10.1145/79173.79176

[Mor04] Morris, M., Charles R. K. (2004) Logic and Computer Design Fundamentals, Third Edition. Essex: Pearson.

[Niu12] Niu, X. (n.d.). System-on-a-Chip (SoC) based Hardware Acceleration in Register Transfer Level (RTL) Nov. 2012 Design. doi:10.25148/etd.fi13042902

[Phi14] Phillips, C. L., Parr, J. M., Riskin, E. A., & Prabhakar, T. (2014). Signals, Systems, and Transforms Fifth Edition. Harlow, Essex: Pearson.

[Ram12] B. Ramkumar, V. Sreedeep, H. M. Kittur, “A Design Technique for Faster Dadda Multiplier”

[Rea17] Reagen, Brandon, et al. Deep Learning for Computer Architects. Morgan & Claypool Publishers, 2017.

[Rui17] R. Zhao, T. Todman, W. Luk and X. Niu, "DeepPump: Multi-pumping deep Neural Networks," 2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP), Seattle, WA, 2017, pp. 206-206. doi: 10.1109/ASAP.2017.7995281

[Sal18] Salvo, B. D. (2018). Brain-Inspired technologies: Towards chips that think? 2018 IEEE International Solid - State Circuits Conference - (ISSCC). doi:10.1109/isscc.2018.8310165

[Sid01] Siddique, M., & Tokhi, M. (n.d.). Training neural networks: Backpropagation vs. genetic algorithms. IJCNN01. International Joint Conference on Neural Networks. Proceedings (Cat. No.01CH37222). doi:10.1109/ijcnn.2001.938792

[Sze17] V. Sze, Y. H. Chen, T. J. Yang and J. S. Emer, "Efficient Processing of Deep Neural Networks: A Tutorial and Survey," in Proceedings of the IEEE, vol. 105, no. 12, pp. 2295-2329, Dec. 2017. doi: 10.1109/JPROC.2017.2761740

[Sze17a] Sze, V., Chen, Y., Emer, J., Suleiman, A., & Zhang, Z. (2017). Hardware for machine learning: Challenges and opportunities. 2017 IEEE Custom Integrated Circuits Conference (CICC). doi:10.1109/cicc.2017.7993626

[Tak17] Takamaeda-Yamazaki, S., Ueyoshi, K., Ando, K., Uematsu, R., Hirose, K., Ikebe, M., . . . Motomura, M. (2017). Accelerating deep learning by binarized hardware. 2017 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC). doi:10.1109/apsipa.2017.8282183

[Tow03] J. Townsend, Whitney & Jr, Earl & Abraham, J.A.. (2003). A comparison of Dadda and Wallace multiplier delays. Proceedings of SPIE - The International Society for Optical Engineering. 5205. 10.1117/12.507012.

[Wei16] Weitz, E., Dr. (2016). Retrieved from http://www.weitz.de/ieee/

[Xil12] LogiCORE IP Floating-Point Operator v6.1, 6th ed. Xilinx., San Jose., CA, USA, Jul. 2012. Accessed on: Feb, 08, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/floating_point/v6_1/pg060-floating-point.pdf

[Zha17] R. Zhao, W. Luk, X. Niu, H. Shi and H. Wang, "Hardware Acceleration for Machine Learning," 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), Bochum, 2017, pp. 645-650. doi: 10.1109/ISVLSI.2017.127

APPENDIX A: XOR ARTIFICIAL NEURAL NETWORK CODE

```
# Author:                 Angel Izael Solis
# Section:                EE 5399 (Thesis 2)
# Date Created:           02/21/2019
# Description:
#
# Artificial Neural Network Applications
# Neural Network that classifies the XOR problem
# Developed using Keras

# Import necessary modules
import keras
from keras.layers import Dense
from keras.models import Sequential, load_model
from keras.callbacks import EarlyStopping
from keras.optimizers import Adam
import pandas as pd

# Train Model
def trainModel():
       #read in CSV file
       data = pd.read_csv('XOR-data.csv')
       predictors = data.drop(['out'], axis=1).as_matrix()
       n_cols = predictors.shape[1]
       target = data.out

       # Set up the model
       model = Sequential()

       # Add the first layer
       model.add(Dense(2,activation='relu',input_shape=(n_cols,)))

       # Add the output layer
       model.add(Dense(1,activation='relu'))

       # Compile the model
       monitor = EarlyStopping(patience=2)
       adam = Adam(lr=0.0035)
       model.compile(optimizer=adam, loss='binary_crossentropy',)

       # Fit the model
       print("FITTING THE MODEL")
       model.fit(predictors,target,batch_size=1,verbose=0,nb_epoch=10000,callbacks=[m
onitor])


       # Save model architecture/weights
       model.save('XOR-arch.h5')

# Test Model
def inferring(net_arch):
```

```
        # load testing data
        data = pd.read_csv('XOR-data.csv')
        predictors = data.drop(['out'], axis=1).as_matrix()
        n_cols = predictors.shape[1]

        # Load saved model
        model = load_model(net_arch)

        # Predict with the model
        print("PREDICTION WITH MODEL")
        print(model.predict_proba(predictors))

        for weights in model.get_weights():
                print(weights.shape)
                print(weights)
        model.summary()

# "Main"
#trainModel()
inferring('XOR-arch.h5')
```

APPENDIX B: IRIS CLASSIFICATION ARTIFICIAL NEURAL NETWORK CODE

```
# Author:                   Angel Izael Solis
# Section:                  EE 5399 (Thesis 2)
# Date Created:             02/21/2019
# Description:
#
# Artificial Neural Network Applications
# Neural Network that classifies the
# IRIS dataset flower classification problem
# Developed using Keras

# Import necessary modules
import keras
from keras.layers import Dense
from keras.models import Sequential, load_model
from keras.optimizers import Adam
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import pandas as pd

def preprocess():

        dataset = pd.read_csv('iris.csv')
        X = dataset.iloc[:,1:5].values
        y = dataset.iloc[:,5].values

        encoder =  LabelEncoder()
        y1 = encoder.fit_transform(y)
        Y = pd.get_dummies(y1).values

        x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.3, random_state
= 0)

        return x_train, x_test, y_train, y_test


def training(x_train, y_train):

        # Set up model
        model = Sequential()

        # Add first hidden layer
        model.add(Dense(4, input_shape = (4,), activation ='relu'))

        # Add third hidden layer
        model.add(Dense(2, activation = 'relu'))

        # Add output layer
        model.add(Dense(3, activation = 'softmax'))

        # Compile model and set up learning rate
        model.compile(Adam(lr=0.04), loss = 'categorical_crossentropy', metrics=['accuracy'])

        # Fit model according to training sample
        print("FITTING THE MODEL")
        model.fit(x_train, y_train, verbose = 1, epochs = 1000)

        # Save model Architecture/Weights
```

```
        model.save('IRIS-arch.h5')


def inferring(arch, x_test, y_test):

        # Load saved model
        model = load_model(arch)

        # Infer with the model
        print("MODEL EVALUATION")
        performance = model.evaluate(x_test, y_test, verbose = 1)

        print("Test loss", performance[0])
        print("Test accuracy", performance[1], "\n")


def printModel(arch):

        model = load_model(arch)

        print("PRINTING MODEL ARCHITECTURE\n")
        print("PRINTING MODEL WEIGHTS")
        for weights in model.get_weights():
                print(weights.shape)
                print(weights)

        print("PRINTING MODEL PARAMETERS\n")
        model.summary()

###### MAIN ######

# Preprocess data...
(x_train, x_test, y_train, y_test) = preprocess()

# Create and train model
training(x_train, y_train)

# Load and use model
inferring('IRIS-arch.h5', x_test, y_test)

printModel('IRIS-arch.h5')
```

APPENDIX C: BOOTH'S ALG. 12 BIT FLOATING-POINT MULT. CODE/TB

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: The University of Texas at El Paso
// Engineer: Angel Izael Solis
//
// Create Date:       12:27:15 02/16/2019
// Design Name:       Booth's Algorithm 12 bit Multiplier
// Module Name:       booth_mult
// Project Name:      Domain Specific Architecture, Deep Neural Networks
// Target Devices:
// Tool versions:
// Description:       This floating-point multiplier aims to accelerate the
//                                  floating-point multiplication process by implementing
//                                  booth's algorithm, which consist of only comparing
//                                  two bits and adding a binary number based on the result
//                                  of that comparison.
//
//                                  In this module we also implement a "mid-precision"
//                                  form representation of floating-point numbers
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
// Proposed FP format: 12 bits
//
// First bit:                sign bit
// Following 5 bits:  exp. bits
// Last 6 bits:              Mantissa bits
//
//////////////////////////////////////////////////////////////////////////////////
module booth_multiplier(multiplicand, multiplier, CLK, load, reset, product);


       input [11:0] multiplicand, multiplier;
       input CLK, load, reset;

       output reg [11:0] product;

       reg signed [15:0] Q;
       reg [7:0] M;
       reg [7:0] Mn;
       reg [4:0] exp;
       reg Qn;

       task compare_and_shift;
              begin
                     case ({Q[0], Qn})

                            2'b00,2'b11: begin
                                   Qn = Q[0];
                                   Q = Q>>>1;
                            end

                            2'b01: begin
                                   Q = Q + {M, 8'b0};
```

```
                                Qn = Q[0];
                                Q = Q>>>1;
                        end

                        2'b10: begin
                                Q = Q + {Mn, 8'b0};
                                Qn = Q[0];
                                Q = Q>>>1;
                        end
                endcase
        end
endtask

always @ (posedge reset) begin

        Q = 16'b0;
        M = 8'b0;
        Mn= 8'b0;
        exp =5'b0;
        Qn = 1'b0;
        product = 12'b0;

end

always @ (posedge load) begin

        Q = {9'b0, 1'b1, multiplicand[5:0]};
        M = {1'b1, multiplier[5:0]};
        // 2's compliment of M
        Mn = (~({1'b1, multiplier[5:0]}) + 1);
        Qn = 1'b0;

        // Case where either input is 0, so product is 0
        if( (multiplicand == 12'b0) || (multiplier == 12'b0) ) begin
                product = 12'b0;
        end

        //Case where there is an underflow, so product is 0
        else if ( ((multiplicand[10:6]) + (multiplier[10:6])) < 6'b10000) begin
                product = 12'b0;
        end

        else begin

                repeat(8) begin
                        compare_and_shift;
                end

                case(Q[13])
                        1'b1: begin
                                exp = (((multiplicand[10:6] - 5'b10000) +
(multiplier[10:6]) - 5'b10000) + 5'b10010);
                                product = {(multiplicand[11] ^ multiplier[11]), exp,
Q[12:7]};
                        end

                        1'b0: begin
                                exp = (((multiplicand[10:6] - 5'b10000) +
(multiplier[10:6]) - 5'b10000) + 5'b10001);
```

```
                                    product = {(multiplicand[11] ^ multiplier[11]), exp,
Q[11:6]};// + 1'b1;
                                end
                        endcase

                end //else
        end //always

endmodule // booth_multiplier
```

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: The University of Texas at El Paso
// Engineer: Angel Izael Solis
//
// Create Date:    17:17:53 02/18/2019
// Design Name:
// Module Name:    booth_multiplier_tb
// Project Name:      Domain Specific Architectures, Deep Neural Networks
// Target Devices:
// Tool versions:
// Description:       This test-bench aims to verify the correctness of the
//                                 multiplier implementing the "mid-precision" proposed
floating
//                                 point representation.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module booth_multiplier_tb();

	// Inputs
	reg [11:0] multiplicand, multiplier;
	reg CLK;
	reg load;
	reg reset;

	// Output
	wire [11:0] product;

	booth_multiplier DUT(
		.multiplicand(multiplicand),
		.multiplier(multiplier),
		.CLK(CLK),
		.load(load),
		.reset(reset),
		.product(product)
	);

	initial forever #1 CLK = ~CLK;
		initial begin

			CLK = 0;

			// Testing first IF statement, first condition
			multiplicand <= 12'b000000000000;
			multiplier <= 12'b010001011100;
			load <= 1'b1;#10;
			load = ~load;#10;

			// Testing first IF statement, second condition
			multiplicand <= 12'b010001011100;
			multiplier <= 12'b000000000000;
			load <= 1'b1;#10;
			load = ~load;#10;
```

```
// Testing second IF statement, underflow condition
multiplicand <= 12'b000001000000;
multiplier <= 12'b000001000000;
load <= 1'b1;  #10;
load = ~load;#10;

// Testing actual multiplication
// multiplicand = 2.781 ~= 2.8
// multiplier = 0.3984 ~= 0.4
// expected_product = 0 01111 000110 = 1.108
multiplicand <= 12'b010000011001;
multiplier <= 12'b001101100110;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = 3.625 110100
// multiplier = 7.875 111110
// product = 28.55
// produced_product = 28.5 = 0 10011 110010
multiplicand <= 12'b010000110100;
multiplier <= 12'b010001111110;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = 40
// multiplier = -23.75
// product = -950
// produced_product = 944 = 1 11000 110110
multiplicand <= 12'b010100010000;
multiplier <= 12'b110011011111;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = -16.75
// multiplier = 50.5
// product = -846
// produced_product = -840 = 1 11000 101001
multiplicand <= 12'b110011000011;
multiplier <= 12'b010100100101;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = -8.625
// multiplier = -78.0
// product = 673
// produced_product = 672 = 0 11000 010100
multiplicand <= 12'b110010000101;
multiplier <= 12'b110101001110;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = 0.4688
// multiplier = 0.836
// product = 0.3918
// produced_product = 0.3906 = 0 01101 100100
```

```
                multiplicand <= 12'b001101111000;
                multiplier <= 12'b001110101011;
                load <= 1'b1;#10;
                load = ~load;#10;

                // Testing performance of half precision with booth multiplier
                // multiplicand = 0.6953
                // multiplier = 45
                // product = 31.28
                // produced_product = 31.25 = 0 10011 111101
                multiplicand <= 12'b001110011001;
                multiplier <= 12'b010100011010;
                load <= 1'b1;#10;
                load = ~load;#10;

                $finish;
            end

endmodule
```

APPENDIX D: BOOTH'S ALG. 16 BIT FLOATING-POINT MULT. CODE/TB

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: The University of Texas at El Paso
// Engineer: Angel Izael Solis
//
// Create Date:    12:27:15 02/16/2019
// Design Name:        Booth's Algorithm 16 bit Multiplier
// Module Name:    booth_mult
// Project Name:   Domain Specific Architecture, Deep Neural Networks
// Target Devices:
// Tool versions:
// Description:       This floating-point multiplier aims to accelerate the
//                                 floating-point multiplication process by implementing
//                                 booth's algorithm, which consist of only comparing
//                                 two bits and adding a binary number based on the result
//                                 of that comparison.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//     Implementing algorithm with half-precision representation of operands
//
//     First bit:                          Sign bit
//     Following 5 bits:           Exp. bits
//     Last 11 bits:               Mantissa bits
//
//////////////////////////////////////////////////////////////////////////////////
module booth_multiplier_16bit(multiplicand, multiplier, CLK, load, reset, product);

       input [15:0] multiplicand, multiplier;
       input CLK, load, reset;

       output reg [15:0] product;

       reg signed [23:0] Q;
       reg [11:0] M;
       reg [11:0] Mn;
       reg [4:0] exp;
       reg Qn;

       task compare_and_shift;
              begin
                     case ({Q[0], Qn})

                            2'b00,2'b11: begin
                                   Qn = Q[0];
                                   Q = Q>>>1;
                            end

                            2'b01: begin
                                   Q = Q + {M, 12'b0};
                                   Qn = Q[0];
                                   Q = Q>>>1;
```

```
                        end

                        2'b10: begin
                                Q = Q + {Mn, 12'b0};
                                Qn = Q[0];
                                Q = Q>>>1;
                        end
                endcase
        end
endtask

always @ (posedge reset) begin

        Q  = 24'b0;
        M  = 12'b0;
        Mn = 12'b0;
        exp = 5'b0;
        Qn  = 1'b0;
        product = 16'b0;

end

always @ (posedge load) begin

        Q = {13'b0, 1'b1, multiplicand[9:0]};
        M = {1'b1,  multiplier[9:0]};
        // 2's compliment of M
        Mn = (~({1'b1, multiplier[9:0]}) + 1'b1);
        Qn = 1'b0;

        // Case where either input is 0, so product is 0
        if( (multiplicand == 16'b0) || (multiplier == 16'b0) ) begin
                product = 16'b0;
        end

        //Case where there is an underflow, so product is 0
        else if ( ((multiplicand[14:10]) + (multiplier[14:10])) < 6'b10000) begin
                product = 16'b0;
        end

        else begin

                repeat(12) begin
                        compare_and_shift;
                end

                case(Q[21])
                        1'b1: begin
                                exp = (((multiplicand[14:10] - 5'b10000) +
(multiplier[14:10]) - 5'b10000) + 5'b10010);
                                product = {(multiplicand[15]) ^ (multiplier[15]), exp,
Q[20:11]};
                        end

                        1'b0: begin
                                exp = (((multiplicand[14:10] - 5'b10000) +
(multiplier[14:10]) - 5'b10000) + 5'b10001);
                                product = {(multiplicand[15]) ^ (multiplier[15]), exp,
Q[19:10]} + 1'b1;
                        end
```

```
                        endcase

                end //else
        end //always

endmodule // booth_multiplier
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company: The University of Texas at El Paso
// Engineer: Angel Izael Solis
//
// Create Date:    17:17:53 02/18/2019
// Design Name:
// Module Name:    booth_multiplier_tb
// Project Name:      Domain Specific Architectures, Deep Neural Networks
// Target Devices:
// Tool versions:
// Description:       This test-bench aims to verify the correctness of the
//                                 multiplier implementing the "half-precision"  floating
//                                 point representation.
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////

module booth_multiplier_16bit_tb;

        // Inputs
        reg [15:0] multiplicand, multiplier;
        reg CLK;
        reg load;
        reg reset;

        // Output
        wire [15:0] product;

        // Instantiate the Unit Under Test (UUT)
        booth_multiplier_16bit uut (
                .multiplicand(multiplicand),
                .multiplier(multiplier),
                .CLK(CLK),
                .load(load),
                .reset(reset),
                .product(product)
        );

        initial forever #1 CLK = ~CLK;
                initial begin

                        CLK = 0;

                        // Testing first IF statement, first condition
                        multiplicand <= 16'b0;
                        multiplier <= 16'b1101000010111100;
                        load <= 1'b1;#10;
                        load = ~load;#10;
```

```
// Testing first IF statement, second condition
multiplicand <= 16'b1101000010111100;
multiplier <= 16'b0;
load <= 1'b1; #10;
load = ~load;#10;

// Testing second IF statement, underflow condition
multiplicand <= 12'b0000010000000000;
multiplier <= 12'b0000010000000000;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = 3.646 1101001011
// multiplier = 7.895 1111100101
// expected_product = 28.78 = 0 10011 1100110010
multiplicand <= 16'b0100001101001011;
multiplier <= 16'b0100011111100101;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = 40.6
// multiplier = -23.95
// expected_product = -972.5 = 1 11000 1110011001
multiplicand <= 16'b0101000100010011;
multiplier <= 16'b1100110111111101;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = -16.8
// multiplier = 50.75
// expected_product = -852.5 = 1 11000 1010101001
multiplicand <= 16'b1100110000110011;
multiplier <= 16'b0101001001011000;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = -8.734
// multiplier = -78.25
// expected_product = 28.78 = 0 11000 0101010111
multiplicand <= 16'b1100100001011110;
multiplier <= 16'b1101010011100100;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = 0.472 01101 1110001101
// multiplier = 0.8394 01110 1010110111
// expected_product = 0.396 01101 1001010110
multiplicand <= 16'b0011011110001101;
multiplier <= 16'b0011101010110111;
load <= 1'b1;#10;
load = ~load;#10;

// Testing performance of half precision with booth multiplier
// multiplicand = 0.698 01110 0110000011
```

```
                    // multiplier = 45.28 10100 0110101001
                    // exptected_product = 31.2 10011 1111001101
                    multiplicand <= 16'b0011100110000011;
                    multiplier <= 16'b0101000110101001;
                    load <= 1'b1;#10;
                    load = ~load;#10;

                    $finish;
                end

endmodule
```

CURRICULUM VITA

Angel Izael Solis was born on July 17, 1995 in El Paso, Texas. The first child of Juan Carlos Solis and the fourth child of Magdalena Ochoa, he graduated high school from El Paso High School in El Paso, Texas, June 2013. Always a proud Tiger! He entered The University of Texas at El Paso (UTEP) the following Fall of 2013.

While pursuing his bachelor's degree in Electrical and Computer Engineering with a minor in Computer Science and Mathematics, he held several leadership positions within the university. He first held a position as an Engineering Ambassador under the supervision of Gabby Gandara. It was then that he became increasingly active within organizations and activities at UTEP such as the Institute of Electrical and Electronics Engineers (IEEE), Latinos in Science and Engineering (MAES, formerly known as the Mexican American Engineering Society), Society of Hispanic Professional Engineers (SHPE), and the Texas Society of Professional Engineers (TSPE). In addition to these professional organizations, Mr. Solis became a member of the *Tau Beta Pi* Engineering Honor Society and the *Alpha Chi* Honor Society. He has also held positions as Technology Support Center Technician and University 1301 Peer Leader.

During Mr. Solis's undergraduate career at UTEP, he managed to obtain three internship positions: one with Entergy Corp. in New Orleans, LA and two with Lockheed Martin Space Systems Company (SSC) in Sunnyvale, CA and Denver, CO, consecutively. During his time at Entergy Corp. he held the position of Power Distribution Intern. There he assisted in designing multiple residential and commercial power distribution projects using proprietary CAD software requiring detailed understanding of construction plans and customer relations skills. His first internship with Lockheed Martin SSC was a Software Engineering position, which required him to reconstruct an existing kernel design to follow standard Object-Oriented Programming guidelines as opposed to the then Function Oriented program. This was done in order to increase the program's agility and robustness. His second internship with Lockheed Martin SSC was also a Software Engineering role that allowed him to develop, test, and deploy a RESTful Interface that enabled two sets of databases in order to speed up the manufacturing and production process This, in turn, allowed Mr. Solis to also learn about Systems Engineering. Mr. Solis received his Bachelor of Science in Electrical and Computer Engineering on May 2018. Forever a Miner! During the summer of 2018 he worked as a Telecommunications Engineer with ExxonMobil in Houston, TX. During his time there, he worked on an $11M digital radio infrastructure upgrade project.

Mr. Solis leveraged the Fast-Track program offered at UTEP which allows undergraduate students to take upper-division courses at the graduate level. Therefore, he started his graduate career early on but was officially accepted into the Masters' of Science in Computer Engineering graduate program for the Fall of 2018. Whilst in this program, he has worked as Graduate Teaching Assistant under the supervision of both Dr. Patricia Nava and Dr. Michael McGarry. In this position he has assisted faculty by teaching and performing teaching-related duties, such as teaching lower and upper level material, developing teaching materials, preparing and giving examinations, and grading examinations and papers. He managed to pay for both his undergraduate and graduate degrees through generous grants, scholarships, fellowships, and endowments from the National Science Foundation, STARS foundation, Texas Space Grant Consortium, Microsoft Corporation, Texas Instruments, Lockheed Martin, and thesis advisor's support.