EXPLOITING AREA-SPEED-POWER TRADEOFF OF FPGA DESIGNS ON MULTILAYER PERCEPTRON (MLP) NEURAL NETWORK

by

April Lora Reed, MS

THESIS

Presented to the Faculty of

The University of Houston-Clear Lake

In Partial Fulfillment

Of the Requirements

For the Degree

MASTER OF SCIENCE

in Computer Engineering

THE UNIVERSITY OF HOUSTON-CLEAR LAKE

DECEMBER, 2020

EXPLOITING AREA-SPEED-POWER TRADEOFF OF FPGA DESIGNS ON MULTILAYER PERCEPTRON (MLP) NEURAL NETWORK

by

April Lora Reed

APPROVED BY

_____

Xiaokun Yang, PhD, Chair

_____

Hakduran Koc, PhD, Committee Member

_____

Ishaq Unwala, PhD, Committee Member

RECEIVED/APPROVED BY THE COLLEGE OF SCIENCE AND ENGINEERING:

_____

David Garrison, PhD, Interim Associate Dean

_____

Miguel A Gonzalez, PhD, Dean

Dedication

I would like to dedicate my thesis to my husband, my Mom, my Dad, my late Sister, my Derkinator, and my two best friends. Without their love, support, motivation, and help I would be at a very different place in life.

Acknowledgements

I would like to acknowledge the faculty and staff in the Computer Engineering department at the University of Houston – Clear Lake. I would like to acknowledge my thesis committee members for inspiring and helping me. I would like to thank Dr. Xiaokun Yang for his support and knowledge. Without his insight my thesis work would have been a very difficult journey. I also would like to thank Dr. Hakduran Koc for mentoring me throughout my computer engineering academic career and by helping me prioritize my thesis work. I feel very lucky that I am at a school that has outstanding professors who have helped me grow academically and professionally.

I would like to acknowledge my computer engineering friends who have helped me with my thesis related and academic work.

Finally, I would like to acknowledge my husband and parents for their love and support.

ABSTRACT

EXPLOITING AREA-SPEED-POWER TRADEOFF OF FPGA DESIGNS ON MULTILAYER PERCEPTRON (MLP) NEURAL NETWORK

April Lora Reed
University of Houston-Clear Lake, 2020

Thesis Chair: Xiaojun Yang, PhD

This dissertation presents four Field Programmable Gate Array (FPGA) design architectures for handwritten digit recognition, in order to improve hardware efficiency in terms of resources, power, and speed of the neuromorphic processor. Multipliers are used as the basis for each of the processor designs. Additional methods are explored to compare hardware efficiency with implementing floating point adders and multipliers in register-transfer level (RTL) designs. These implementations are then instantiated into each of the processor designs and the results are compared to the IPs-based demonstrations using Xilinx Vivado. Experimental results show that the 196-MUL design architecture achieves the highest speed but consumes a large amount of power and FPGA resource including look-up Tables (LUTs), flip-flops (FFs), and digital signal processing elements (DSPs). In contrast, the 28-MUL design architecture spends the minimum LUTs, FFs, DSPs, and power dissipation, however, the latency is greater than 3× compared with the hardware cost of the 196-muliplier structure. The conclusion of the

dissertation is that the proposed works offer different levels of hardware efficiency corresponding to different design specifications. The proposed designs allows for the user to choose which aspect is important to them in regards to area, power, and speed and then specialize the system to their needs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I:

INTRODUCTION

1.1 Background

Handwritten digit recognition is one of the emerging problems in the applications of image processing and computer vision [1]. Many prior works have shown high accuracy by demonstrating various digit classification platforms [2–4]. As results showed in [3], up to 99.21% recognition accuracy can be achieved by the designs on convolutional neural networks (CNN), through obtaining more diverse features from each handwritten digit image. Similarly, in [2] authors proposed a combination of learning parameters to the design on CNN, including number of layers, stride size, receptive field, kernel size, padding and dilution for CNN-based handwritten digit classification, which achieved a very high recognition accuracy of 99.87%.

From the hardware design perspective, the inherent tolerance to the imprecise digit classification has a potential to improve hardware computation efficiency in terms of gate/slice count, speed, and power dissipation [5–7]. Under this context, an FPGA-based neuromorphic processor and several parallel architectures were proposed in [6]. By using the approximate multipliers in the baseline processor design, up to 20% reduction in energy consumption was achieved while slightly reducing the recognition performance for handwritten digit recognition. Another example proposed by [7] demonstrated that using residue number system (RNS) arithmetic in the convolutional layer of CNN can reduce hardware costs by 32% compared with the traditional approach based on the binary number system. Likewise, computation speed can be improved by FPGA acceleration [8] and application-specific designs such as employing a novel digit extraction method to reach up to 2.47× speedup in comparison to software solutions [9].

To find the optimal balance of area-latency-power consumption on the design with a FPGA, this dissertation proposes a dynamic design architecture on multi-layer perceptron (MLP) neural networks. By configuring different design specifications to the slice cost, speed, and power dissipation, my proposed work is able to minimize the latency while maintaining a digit recognition accuracy up to 95.82%. Specifically, the contributions are below.

This dissertation first presents a single-hidden-layer MLP neural network to classify handwritten digits with an accuracy up to 95.82%. Further, many different design architectures on FPGA containing floating-point multiplication, addition, subtraction, accumulation, exponential, and reciprocal are proposed to provide hardware acceleration corresponding to different FPGA resource costs.

Second, the floating-point adders and multipliers are implemented by register-transfer level (RTL) designs, in order to compare area-latency-power cost on the MLP network, as well as evaluate the differences compared to the Xilinx Vivado IP-based implementations.

Finally, the area-latency-power consumption on a FPGA is evaluated and compared with the performance of the prior works.

The organization of this dissertation is as follows: we first review the relevant related papers. Our approach is discussed in more detail in Section 3. In Section 4, the hardware implementations using floating-point adders and multipliers are discussed. In Section 5 the implementation of those floating-point adders and multipliers are combined with a previous architecture design and the RTL designs are compared to the Xilinx IP designs. In Section 6 the corresponding results in terms of slice count, speed, and power consumption are estimated and compared. Finally, in the last section we present our conclusion.

1.2 Related Works

Research in the FPGA design of handwritten digit classification has focused on two subjects: the implementation and evaluation, mostly with regard to the FPGA acceleration, and the performance comparison between different designs and platforms.

Previous works to the hardware acceleration have experienced significant technological breakthroughs in recent years [10–16]. For instance, in [13] authors presented a 5-layer accelerator for MNIST digit recognition enabling it to spend 25.4 microseconds to process one frame of image. In [14] a CNN network using a reconfigurable IP core was implemented, showing a time consumption of 17.6 us to recognize a handwritten digital picture. Additionally as a result of [15] the implementation using Xilinx XC7Z045 can reach 14.2 us per image recognition. To the best of our knowledge, the latest work proposed in [16] achieved the highest speed of 1.55 microseconds per image to the application of digit classification.

Other researches focused on comparing and evaluating performance of designs using different platforms, for example, to speed up the system computation by using a programmable artificial neural network coprocessor [17], to employ high-level synthesis (HLS) tools and software-hardware co-design platforms [18, 19], and to implement the network in IP-level and RT-level [20, 21] in order to reduce the design cost.

In this dissertation, a dynamic design architecture of MLP neural network is proposed in order to minimize FPGA area-latency-power cost by slightly decreasing the classification rate. The database of Modified National Institute of Standards and Technology (MNIST) is used to build the MLP network and evaluate the accuracy of the neural network models [22].

CHAPTER II:

PROPOSED DESIGN

2.1 Neural Network

Recently neural networks (NNs) are becoming one of the most populer machine learning algorithms widely used in multiple applications such as speech/music recognition [23, 24], language recognition [25-27], image classification and video segmentation [28-30]. In general NNs attempt to emulate how a human brain works. NNs learn, identify, and decipher complex tasks in science and engineering [31]. In a human brain, each neuron has dendrites entering and axons exiting. The axons then connect to other dendrites by synapses [32]. In NNs the neurons are referred to as nodes, the dendrites are inputs into the nodes, the axons are outputs from the nodes, and the synapses are the interconnections between the nodes. Each interconnection has a weight associated with it. Each neuron then sums all those adjusted weights and it checks if a certain threshold is reached. If the threshold is reached then an output is generated [32]. NNs are made up of layers of nodes. There are usually multiple layers: input layer, output layer, and at least one hidden layer [33]. The NNs learn by adjusting those weights and biases until they have the same output that is expected during training. For example, we can have hundreds of labeled images of ice cream cones and hot dogs. The weights and biases in the NN will be adjusted until it is correctly predicting the labeled images of ice cream cones and hot dogs.

FPGAs, as well as application-specific integrated circuits (ASICs), have a great advantage over traditional software based designs and implemenations [34-36]. The hardware acceleration will be most noted in the resource costs and latency. For FPGAs the advantages lie in its highly parallel architecture, flexible design architectures, real-time data processing, and low power consumption [37-40]. Further, the approximate design on FPGAs can improve the energy-efficiency with the corresponding quality constraints [41-43].

2.1.1 Multilayer Perceptron

The most utilized class of NNs is the MLP version [31]. Briefly, I want to cover single layer perceptron NN. Both single layer perceptron neural networks (SLPNN) and the multilayer perceptron neural network (MLPNN) are feedforward networks. A SLPNN will take in associated weights and inputs. The input is a vector of numbers. The weights are adjusted in training to help produce the correct output. Then the weighted sum of all inputs is adjusted by a bias and the final result is computed. A general single layer NN architecture will have one input and one output layer of processing units. Single layer perceptrons can only solve linearly separable problems [44]. Below you can find the formula for a SLPNN.

$$SLPNN = \sum_{i}^{m} (w_i\, x_i) + b$$

In the SLPNN formula, w equals weights, x equals inputs, and b equals bias. MLPNN consists of at least three layers: one input layer, at least one hidden layer, and one output layer [45]. Every layer except for the input layer will have a node that uses a nonlinear activation function. An activation function then takes that weighted sum of all inputs with the adjusted bias and conforms those results to some desired range to help

produce the final output. A general MLPNN architecture will have one input and one output layer of processing units as well as one or more hidden layers of processing units [46]. MLPNN can solve nonlinearly separable problems.

MLPNN uses a variety of learning techniques, but one of the most popular supervised learning techniques is back-propagation. The outputs are compared to provided answers for the problem that is trying to be solve. An error rate is then calculated and sent back through the network. Weights and biases are adjusted and fine-tuned with each iteration during training.

2.1.2 Sigmoid Neurons

Activation functions, sometimes referred to as threshold functions, are very important to MLPNNs [47]. The difference between linear and nonlinear activation functions, is that a linear activation functions produces an output that is proportional to the input, while non linear activation functions produces complex outputs that are not linearly related to the input. Most NNs will have non linear activation function in order to solve complex problems. There are many activation functions, some of the most referenced are sigmoid, relu, tanh, and activation. The most popular nonlinear activation function is the sigmoid function. Below is the sigmoid function.

$$Sigmoid\ function = \frac{1}{1 + e^{-x}}$$

Sigmoid functions produce outputs between 0 and 1. They have a smooth gradient, since they are able to make small changes instead of changes in large steps, therefore helping to fine tune the weights and biases needed for the NN [16]. An example of a MLPNN is shown in Figure 2.1.2.

A sigmoid neuron employees the information we know about the SLPNN formula and the sigmoid function. Below is the formula for a sigmoid neuron. An example of a MLPNN fully connected three layer design then follows.

$$SLPNN = x = \sum_{i}^{m} (w_i\, x_i) + b$$

$$Sigmoid\ function = \frac{1}{1 + e^{-x}}$$

$$Sigmoid\ neuron\ formula = \frac{1}{1 + e^{-\sum_i^m (w_i x_i) + bias}}$$

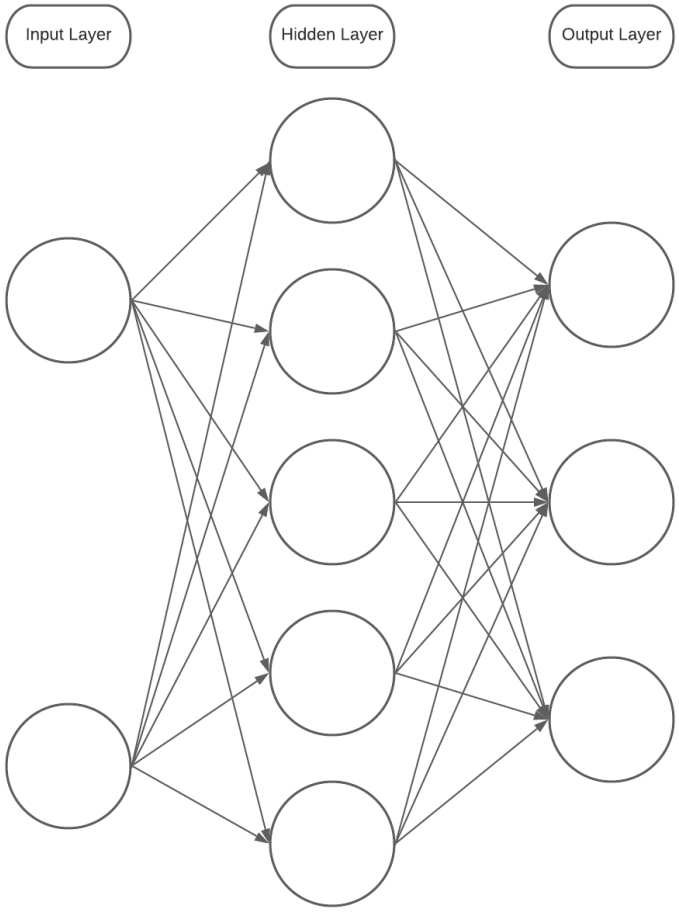

Figure 2.1.2
A general MLPNN fully connected three layer design.

2.4 Floating Point Single Precision

In an attempt to further reduce the area-latency-power for each of the MLPNN architecture designs, the floating point components were examined. The Institute of Electrical and Electronic Engineers (IEEE) 754 Standard was established in 1985 and it standardized floating-point arithmetic [48]. The current version is a revision of the IEEE Standard 754-2008, the IEEE Standard 754-2019 [49]. Floating point numbers can represent a larger range and higher precision of real numbers versus fixed point numbers [50]. There is a tradeoff between range and precision in floating point numbers [51]. The range is determined by the size of the exponent and the precision is determined by the size of the fraction.

The IEEE Standard 754 standardized the formats for floating point numbers. The format is specified by a radix, precision, and an exponent range [*49*]. The format comprises of three parts: the sign, the exponent, and the significand. The IEEE binary basic format describes three types of precision: 32-bit single precision, 64-bit double precision, and 128-bit quadruple precision. For this project, single precision was used and will be referenced as the precision used in the rest of this paper.

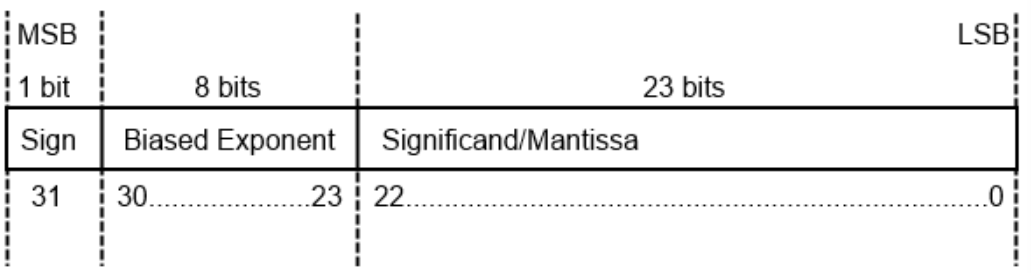

Figure 2.4.1
IEEE 754 Standard's Single Precision Format

In Figure 2.4.1, the distribution of the 32-bits can be seen. The most significant bit (MSB) represents the sign bit. The sign bit denotes whether the number is positive or negative, with '0' representing a positive number and '1' representing a negative number [52]. The next 8-bits represent the biased exponential. The bias is calculated as follows:

$$B = 2^{n-1} - 1$$

Where B represents the bias and n represents the number of bits that represent the biased exponent. For single precision, n = 8, therefore B = 127. The exponent is biased by a constant in order for the exponent to represent positive and negative numbers [*52*].

The last 23-bits represent the significand. The significand represents the precision. It is comprised of the significant digits of the number, led by an implicit non-zero digit. To normalize a number in scientific notation the leading bit is required to be a non-zero digit. In binary digits normalizing the number helps maximize the amount of representable numbers that can be stored and the leading bit will always be '1'. The significand is represented as follows:

$$S = 1.s_{22}s_{21}s_{20}s_{19}s_{18}s_{17}s_{16}s_{15}s_{14}s_{13}s_{12}s_{11}s_{10}s_{9}s_{8}s_{7}s_{6}s_{5}s_{4}s_{3}s_{2}s_{1}s_{0}$$

$s_{22}s_{21}s_{20}s_{19}s_{18}s_{17}s_{16}s_{15}s_{14}s_{13}s_{12}s_{11}s_{10}s_{9}s_{8}s_{7}s_{6}s_{5}s_{4}s_{3}s_{2}s_{1}s_{0}$ represents the significant digits for single precision, the digit 1 is the implied 1, and S represents the significand.

The IEEE 754 Standard also covers exceptions that may occur. Some of the exceptions that can occur are an an invalid operation, a divid by zero, and an inexact. The

output for floating points arithmetic numbers can be normal, subnormal, infinity, zero, and Not a Number (NaN) [53].

There are five rounding rules in the IEEE 754 Standard. They are as follows: Round towards zero, round towards positive infinity, round towards negative infinity, round to nearest, ties to even, and round to nearest, ties away from zero. Round to nearest, ties to even is the default for rounding for binary point. Round to nearest, tie to even is the rounding rule that will be implemented in the adder and multiplier components used in the designs.

2.5 Adder

The IEEE 754 Standard also specifies addition. In arithmetic logic units (ALUs), adders are the main processing component [54]. There are three main stages for performing floating point addition: Pre-normalizing before addition, addition, and post-normalizing after addition [55]. When traversing through these three main stages there are several components that are necessary to properly complete the order of operations and they help to ensure that the floating point addition holds up to the IEEE 754 Standard. These main components are exponential comparison, significand addition, normalization, rounding, and exceptions [51, 56].

Exponential Comparison

This component will compare two inputs' exponential bias to see which is larger. The smaller exponential bias is replaced with the larger exponential bias. Before this occurs, the difference between the two exponential biases is found and stored.

Significand Addition

Once the exponent biases match and the smaller number has been shifted accordingly, addition occurs.

Normalization

This component needs to count the leading zeroes and shift accordingly so that the first implicit digit is a one [57]. The amount that is shifted will also be used to either increment or decrement the exponent. We may need to use this component again after rounding has occurred.

Rounding

After normalization we are left with the outputs' significand plus additional bits. These bits help with rounding. As mentioned above, there are five rounding rules in the IEEE 754 Standard, though by default rounding to nearest, ties to even is used.

Exceptions

Exceptions are checked to shorten the process if one of the inputs is zero or to produce specific outputs for special cases.

2.6 Multiplier

The IEEE 754 Standard specifies multiplication. Multipliers are some of the most frequently used components in digital signal processing (DSP), graphics processing, image processing, and robotics [58, 59]. Multipliers are used because with those applications computational taxing matrix multiplication is involved [60]. There are three main stages for performing floating point multiplication: exponential addition, significand multiplication, and normalization. Although those are the three main stages there are several components that are necessary to properly complete the order of operations and they help to ensure that the floating point multiplication holds up to the IEEE 754 Standard. These main components are exponential addition, significand multiplication, normalization, rounding, and exceptions [61,62]. Normalization,

rounding, and exceptions components were described in the Adder subsection, therefore I will only go over the exponential addition and significand multiplication below.

Exponential Addition

The component will add the exponential bias from each input. This will introduce an extra bias, since not only were we adding the exponents, but also the biases. Subtracting a bias from the added exponential biases together will result in the correct exponential bias addition.

Significand Multiplication

This component will multiply the significands from each input. With decimal point multiplication we need to keep track of each input's digit placement. Since the significands are normalized, the sum of the significands length will give us the decimal placement for the result. With single precision there are 23 bits allocated for the significand. From this we can deduce that the decimal will fall right before the 46th bit in the result.

CHAPTER III:
DESIGN ARCHITECTURES

This dissertation is an extended work to reference [63] presented by Isaac Westby. Therefore, in this chapter the prior work – a design on 98-MUL (98MUL) architecture is firstly introduced. Then what follows are three different architectures, 28 multiplier architecure design (28-MUL), 49 multiplier architecure design (49-MUL), and 196 multiplier architecure design (196-MUL). These designs are mainly proposed to compare and evaluate the design performance in terms of latency, slice count, and power consumption.

3.1 Previous Design

Earlier work [64] presented the algorithm of a typical MLPNN, and further in [63] the hardware design on the MLPNN was implemented and estimated to classify handwritten digits. In general, it consists of 784 input nodes, 12 neurons in the hidden layer, and 10 output neurons. The goal of reference [63] was to create a low-latency and highly-accurate digit recognition NN. They achieved this with a significant speedup for digit recognition compared to other related works. Below is their final network design.

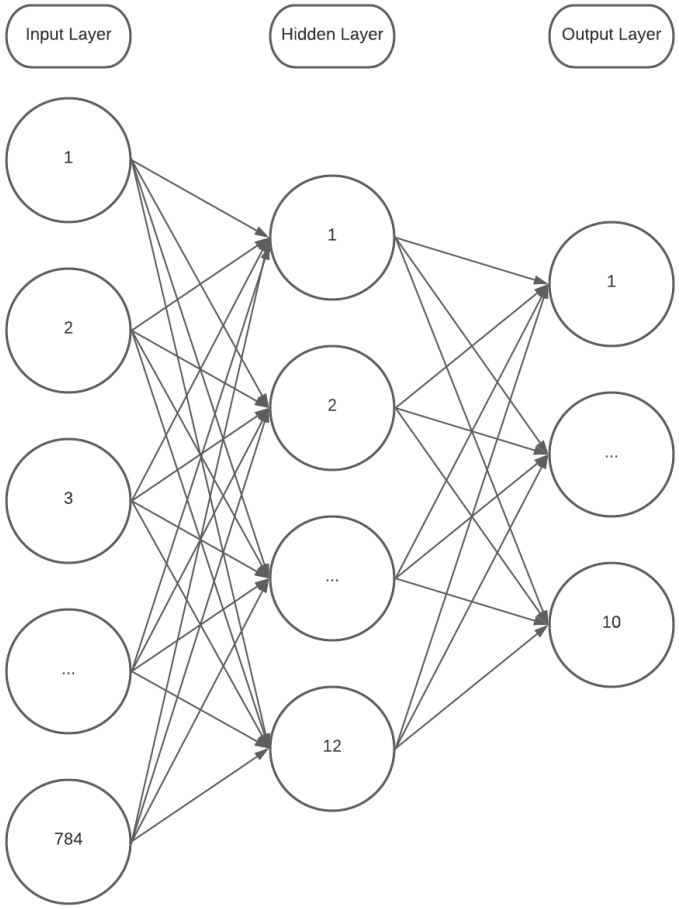

Figure 3.1.1
Final network design for a digit recognition MLPNN.

The final network design is also the basis for my work. I will now explain the different parts for this specific MLPNN. The image that is going to be processed will be resized to a 28 pixel by 28 pixel image, therefore there needs to be 784 input nodes for each pixel in the image. It was determined in Westby's work that only one hidden layer would suffice, since there was not a difference in accuracy between one hidden layer and two hidden layers [16]. Westby also examined the different accuracies between the amount of neurons in one and two hidden layers. He found in his research that 12 neurons would give an accuracy of 92.96%, given the accuracy versus resource cost, it was determined that 12 neurons in one hidden layer would be used in the network design [16].

The outputs represent the digits 0 through 9, therefore there are 10 neurons in the final layer.

When Westby trained the network, he was able to produce the weights and biases for the hidden layer and for the outer layer. For the hidden layer, there are 9,408 single precision weights. The 9,408 single precision weights are distributed in 12 rows and 784 columns [16]. There are 12 biases' values in vector form for the hidden layer, one for each neuron [16]. For the outer layer, there are 120 single precision weights. The 120 single precision weights are distributed in 10 rows and 12 columns. There are 10 biases' values in vector form for the outer layer, again, one for each neuron [16]. For the hidden layer neuron, the input contains 784 pixels that are distributed in 28 rows and 28 columns. Also to note, the outer layer input will be received as the output from the hidden layer.

Now that we have the information for the inputs, weights (hidden), bias (hidden), weights (hidden), and bias (outer), we can see how this information fits into the hidden neuron output equation and outer neuron output equation.

$$hidden\ neuron\ output(x) = Z = \frac{1}{1 + \exp(-\sum_{i=1}^{m=784}(w(x)_i * p_i) - b(x))}$$

For the hidden neuron output equation, w(x) stands for the weights associated with the hidden layer, p stands for the input, and b(x) stands for the biases associated with the hidden layer.

$$outer\ neuron\ output(x) = \frac{1}{1 + \exp(-\sum_{i=1}^{m=12}(w(x)_i * Z_i) - b(x))}$$

For the outer neuron output equation, w(x) stands for the weights associated with the outer layer, Z stands for the input, which is also the output for the hidden layer, and b(x) stands for the biases associated with the outer layer.

Using this information, Westby compared a non-pipelined design, a pipelined 784-multiplier design, and a multiple pipelined 98-MUL design [16]. When comparing latency and resources, it was noted that even though the 98-MUL design had the highest lantency between the three, it would be a fast enough execution time for real-world applications, while using the least amount of resources in comparison with the other two designs [16]. Therefore, Westby designed and created a 98-MUL architecture design [16]. Below are the figures for the 98-MUL power summary and utilization summary.

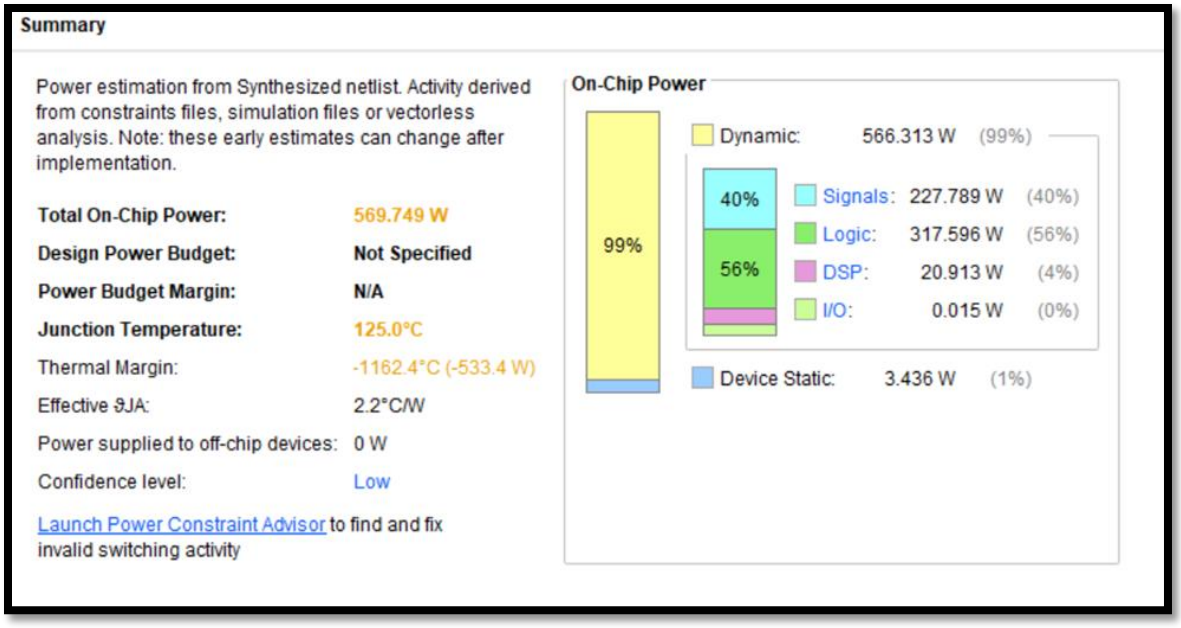

Figure 3.1.2
Power summary for 98-MUL

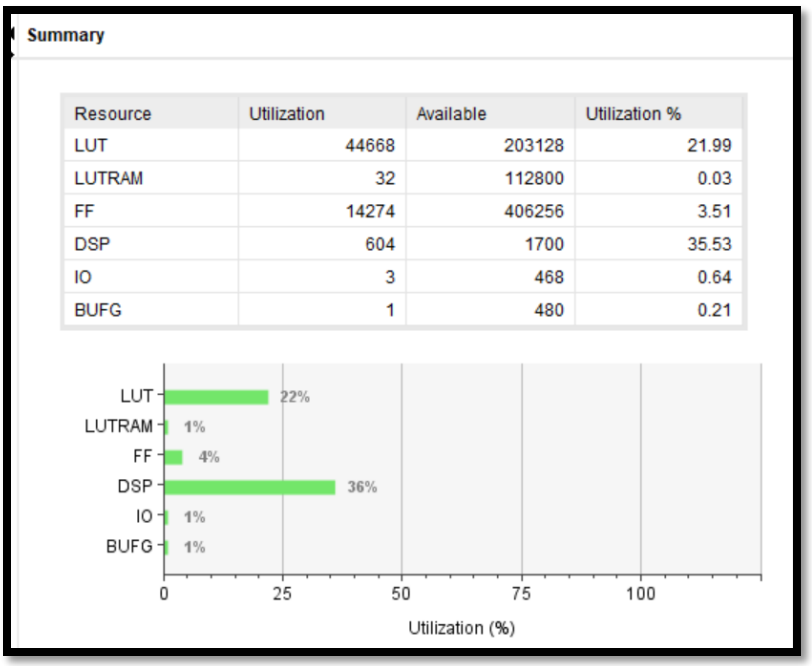

Figure 3.1.3
Utilization summary for 98-MUL

3.2 28-MUL

One part of my contributions for this project was to create three different architecture designs with the multiplers as the baseline. Therefore, there is a 28 multiplier architecture design, a 49 multiplier architecture design, and a 196 multiplier architecture design. There was one more architecture design that was created, a 392 multiplier architecture design, but there were not enough resources on the FPGA, specificly the DSP resources, therefore the 392 multiplier architecture design was dropped.

Since the outer neuron output equation, has its own associated weights and biases, and its input always depends on the hidden neuron output, the outer neuron output design will be the same for each multiplier based architecture design. Therefore, we only need to consider the hidden neuron output equation for each of these designs. Below is the outer neuron output's design that is used for all multiplier based architecture designs.

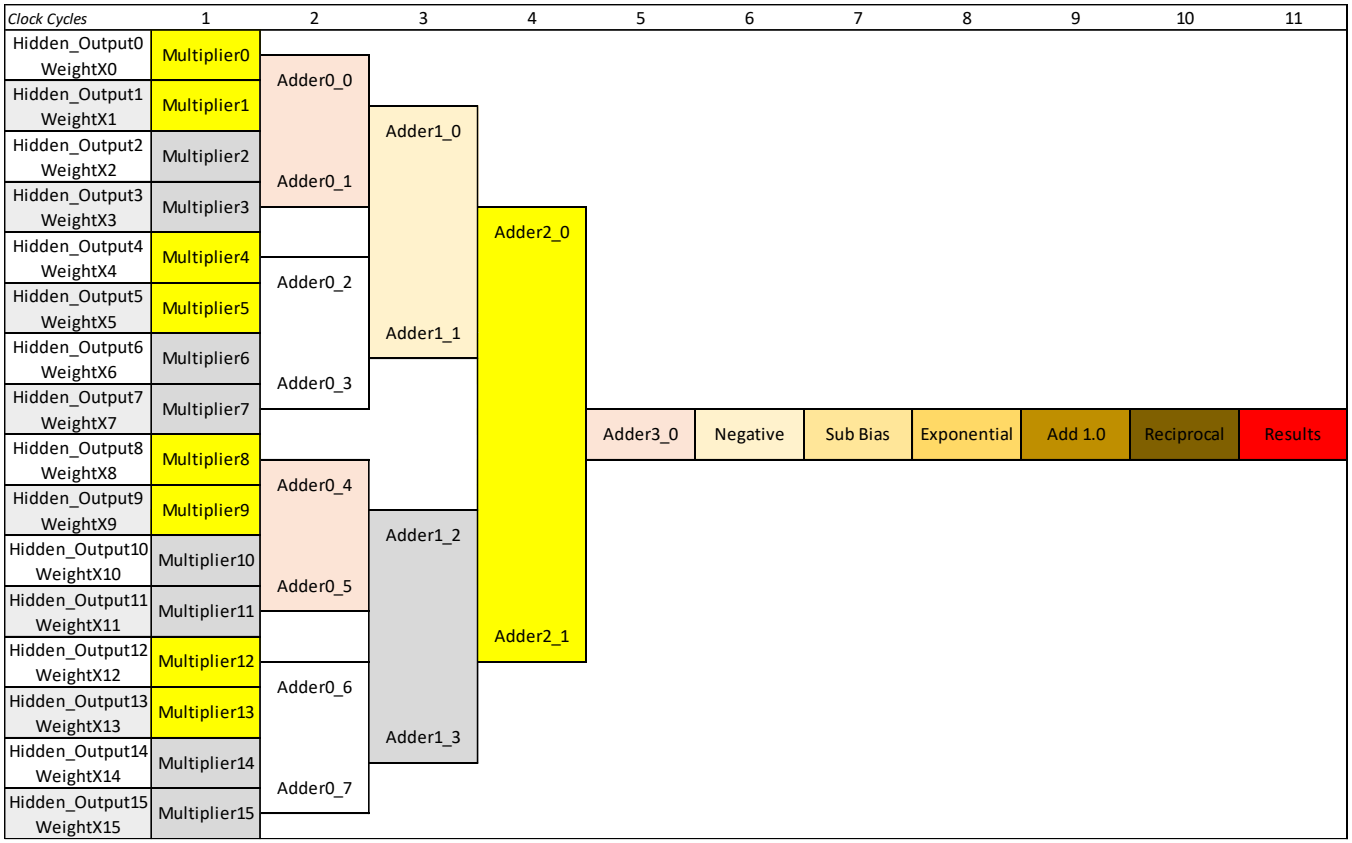

Figure 3.2.1
Outer layer output design used for all multiplier based designs.

When deciding the multiplier base architecture designs, the information that was known was that there are 784 inputs and that the previous work contributed a 98-MUL design. The goal was to compare the differences in area, power, and speed between different designs. Therefore, the first step was to look at the divisors of 784. Those divisors are listed as follows: 1, 2, 4, 7, 8, 14, 16, 28, 49, 56, 98, 112, 196, 392, and 784. My goal from here was to choose a design that could still be used in real-world application as well as be varied enough from the other designs, so that area-power-speed comparisons would be notable. I decided that the lowest design that I was going to create was with 49-MUL architecture design. Therefore that leaves: 56, 112, 196, and 392. Since, 112-multipliers are close enough to the 98-MUL design and similarly with the 56-multipliers being too close to a 49-MUL design, I was worried there would not be a notable difference. Therefore, I removed those possibilities and I was left with a 196-MUL design, and a 392-multiplier design. I then exceuted the 49, 196, and 392 designs. As noted before I was not able to execute the 392-multiplier architecture design. I then went back to the list of divisors and choose the highest, most varied number, which was 28.

The 28-MUL architecture design, is unique in that the number of inputs into the input layer also equals the number of times the design needs to be iterated. Below, in Figure 3.2.2 and Figure 3.2.3 you can see the design for the 28-MUL hidden layer output design and Figure 3.2.4 is the timing diagram for the same design. The timing diagram is very intricate therefore, I included in Figure 3.2.5 the hidden layer timing and in Figure 3.2.6 I included the very first iteration for the outer layer timing. Figure 3.2.7 is the power dissipation for the 28-MUL design and Figure 3.2.8 indicated the overall resources used.

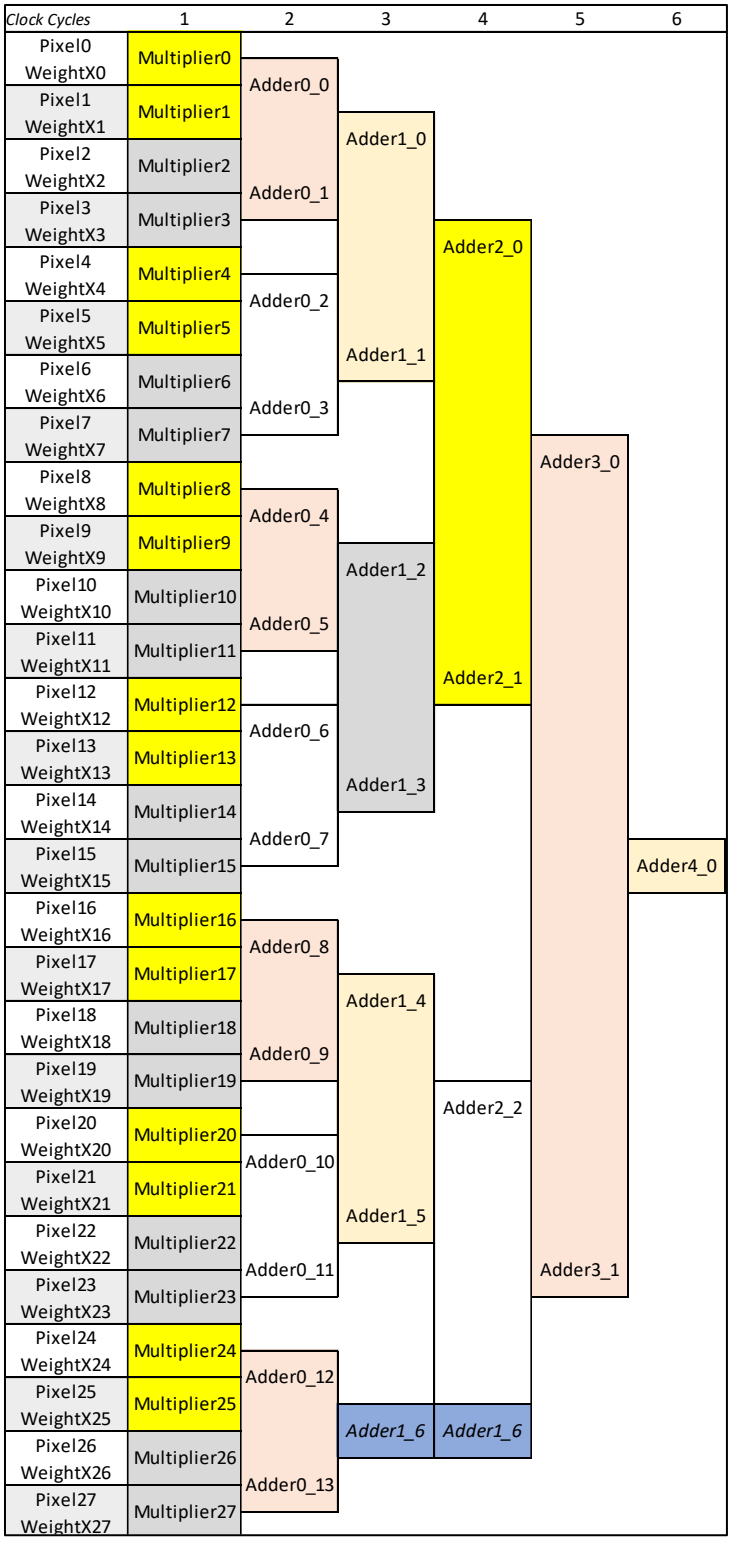

Figure 3.2.2

First half of the hidden layer output design used in the 28-MUL architecture design.

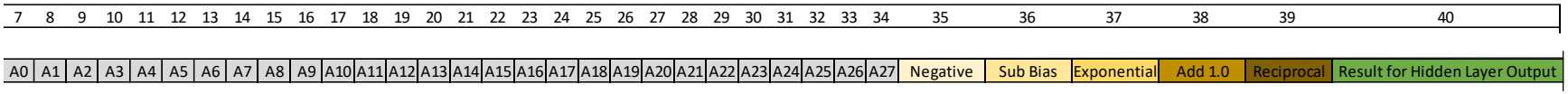

Figure 3.2.3

Second half of the hidden layer output design used in the 28-MUL architecture design.

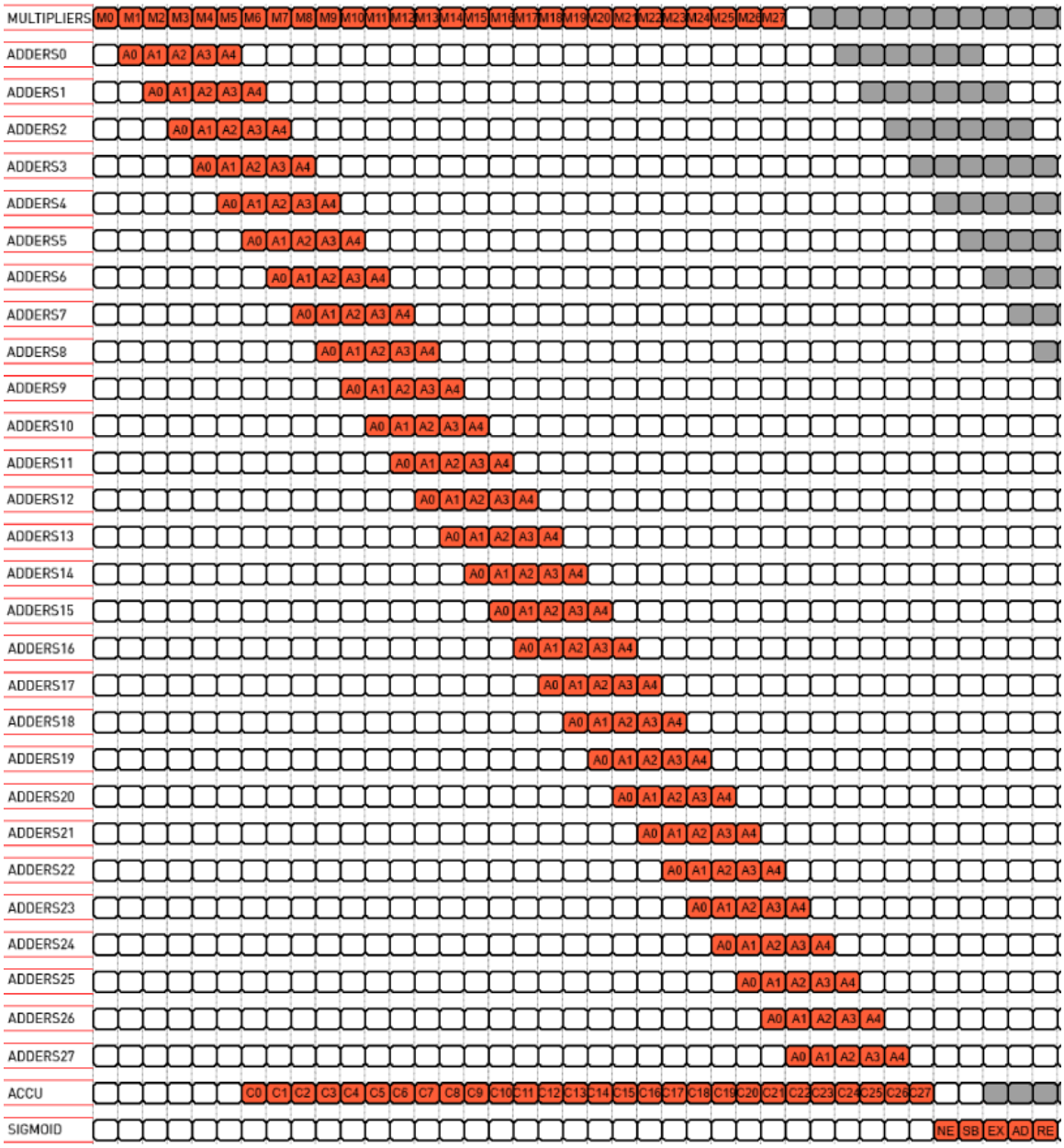

Figure 3.2.4

Partial timing diagram for the 28-MUL architecture design, emphasizing the first iteration of the hidden layer timing.

Figure 3.2.5

Partial timing diagram for the 28-MUL architecture design, emphasizing the outer layer timing.

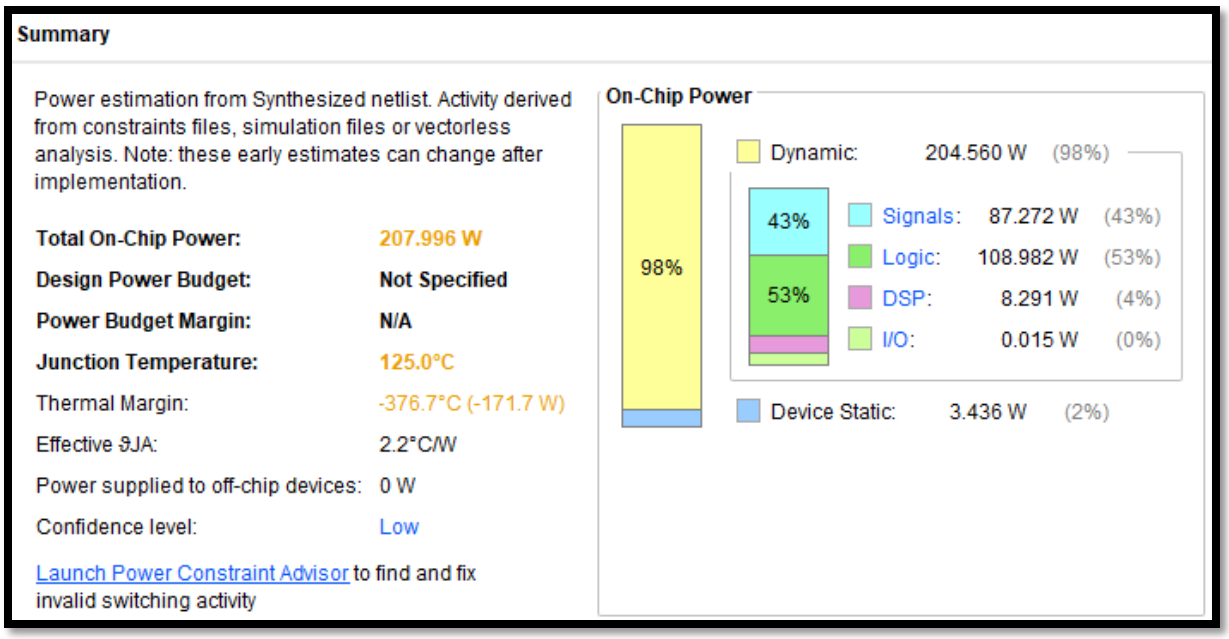

Figure 3.2.6
Power Summary for 28-MUL

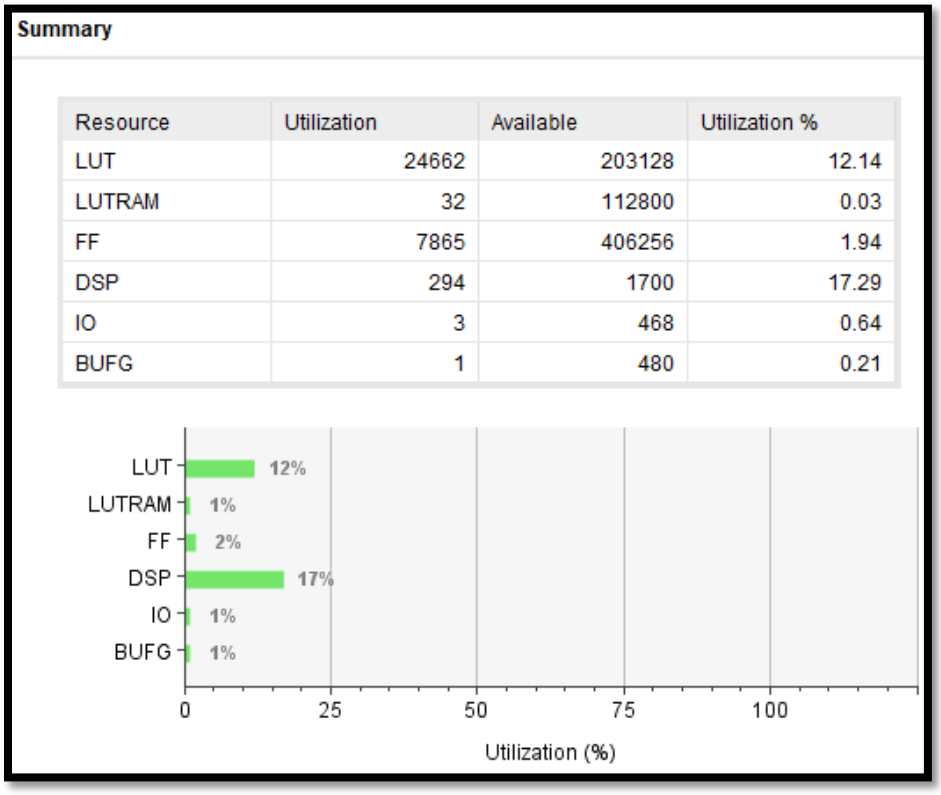

Figure 3.2.7
Utilization Summary for 28-MUL

3.3 Architecture Design 49

The 49-MUL architecture design, has 49 multipliers and needs to be iterated 16 times to process all 784 inputs. Similarly done with the 28-MUL architecture design I will include the figures below. The first figures in this section are Figure 3.3.1 and Figure 3.3.2. Those figures contains the design for the 49-MUL hidden layer output design. Figure 3.3.3 is the timing diagram for the same design. The design is also very intricate therefore, Figure 3.3.4 is included to show in greater detail the hidden layer timing and Figure 3.3.5 is included to show the outpur layer timing. Figure 3.3.6 is the power dissipation for the 49-MUL design and Figure 3.3.7 indicated the overall resources used.

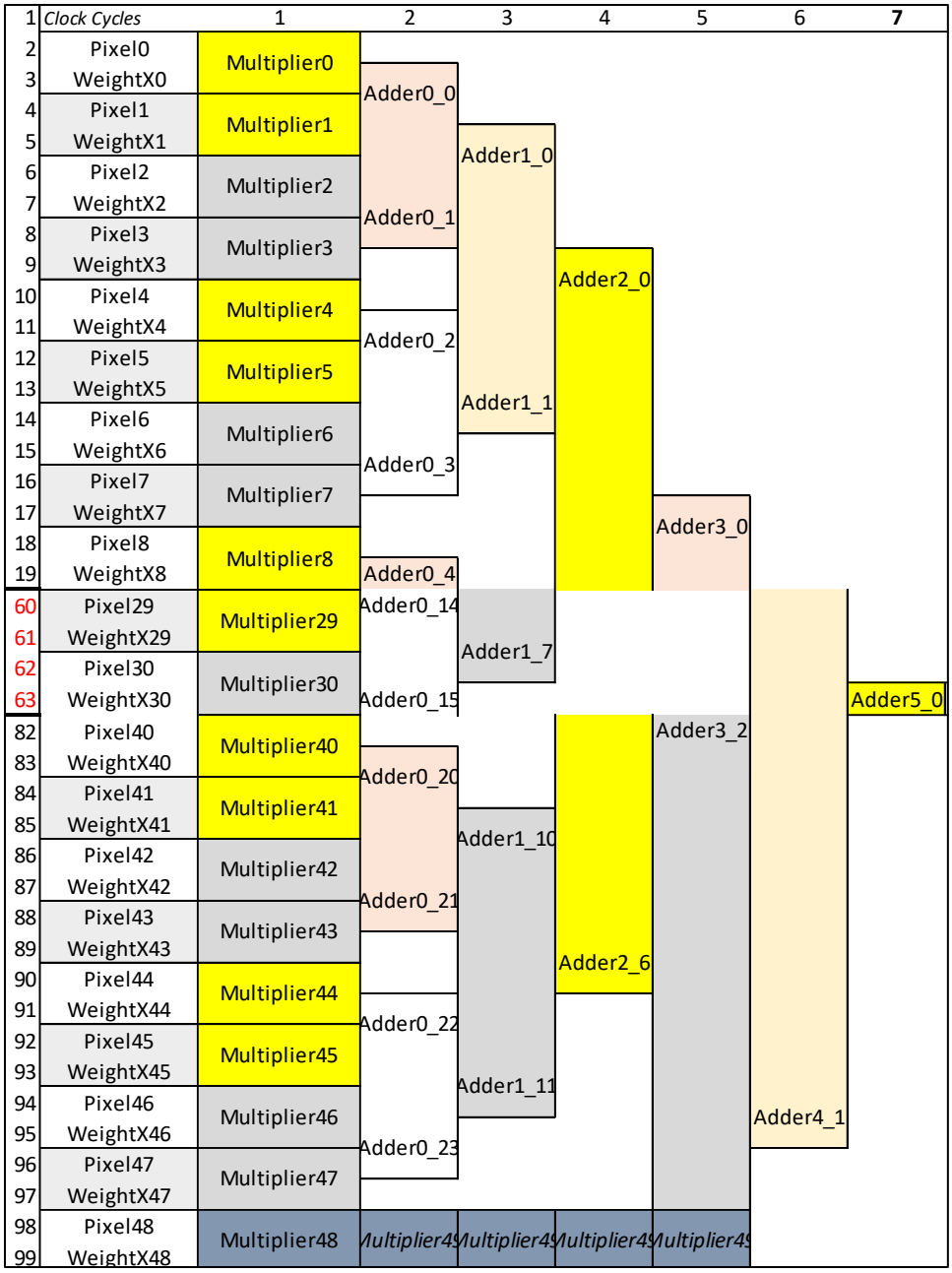

Figure 3.3.1

First half of the hidden layer output design used the 49-MUL architecture design. Make note that there are hidden rows in order to show overall design.

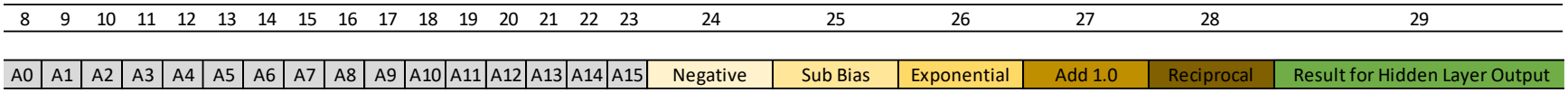

Figure 3.3.2
Second half of the hidden layer output design used the 28-MUL architecture design.

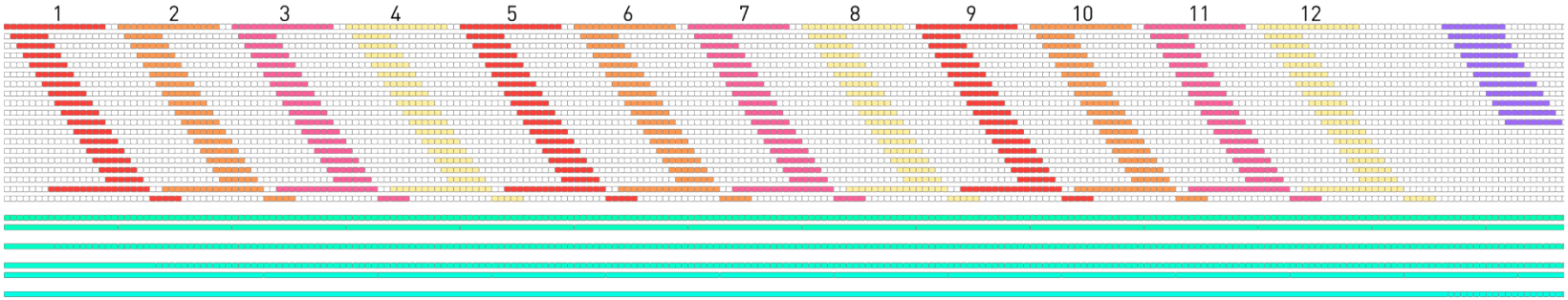

Figure 3.3.3
Timing diagram to show overall structure for the 49-MUL architecture design.

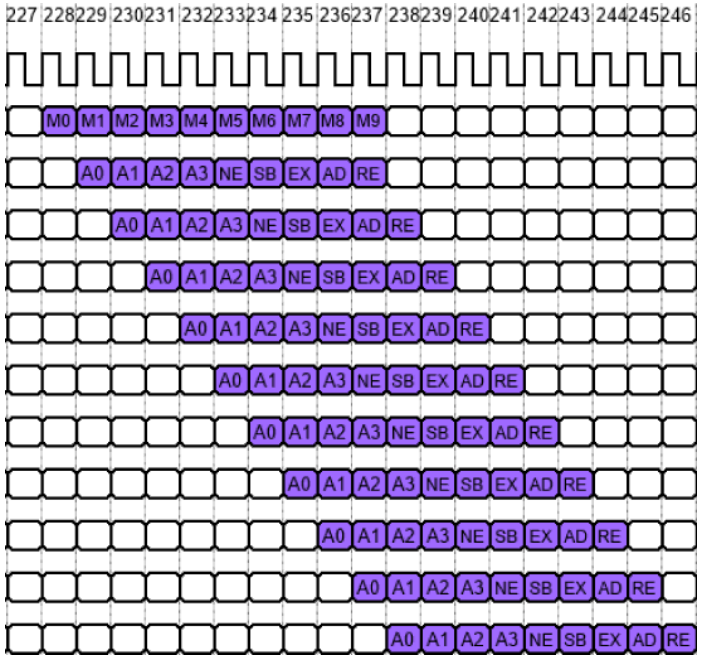

Figure 3.3.4

Partial timing diagram for the 49-MUL architecture design, emphasizing the outer layer timing.

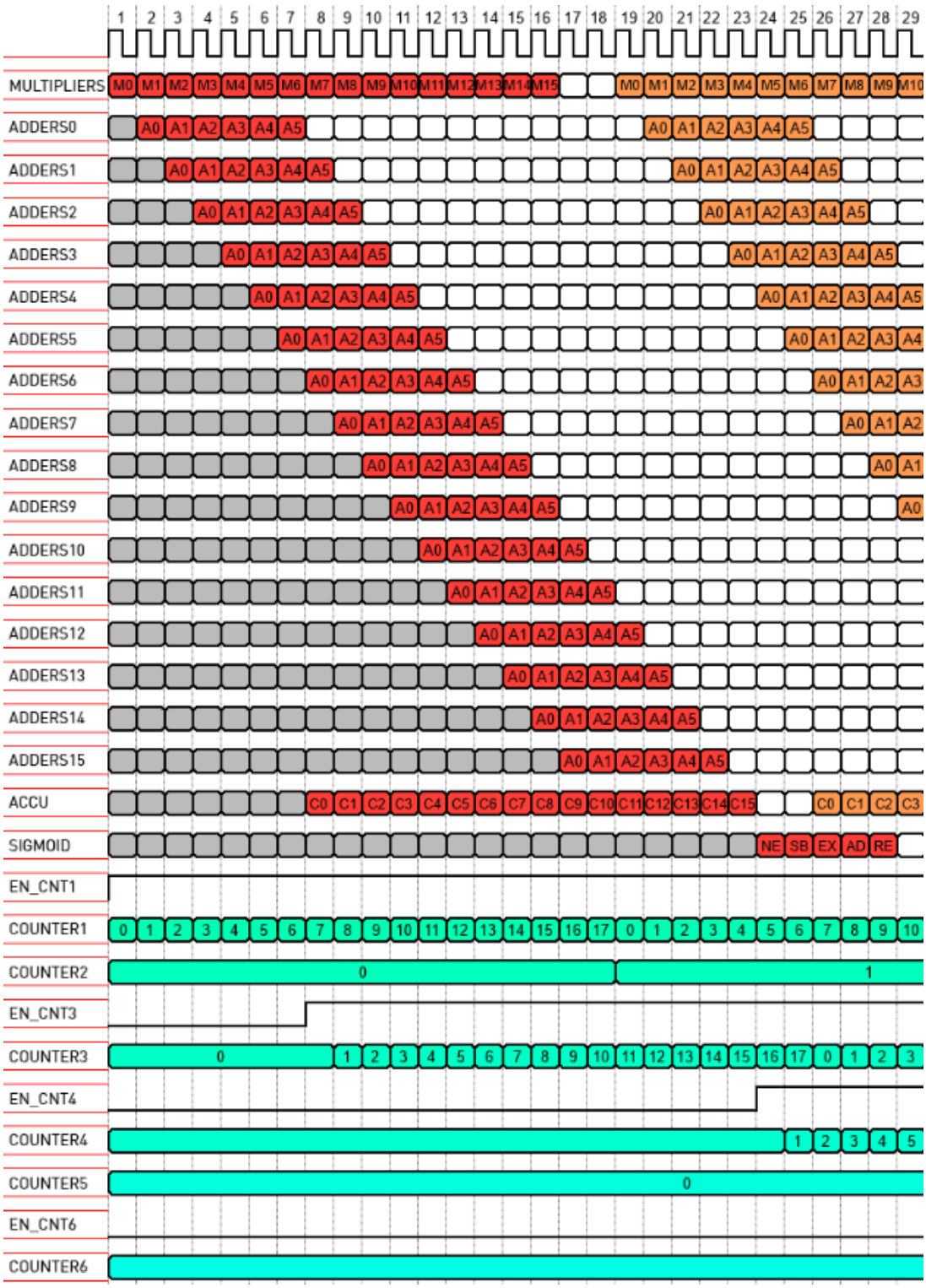

Figure 3.3.5
Partial timing diagram for the 49-MUL architecture design, emphasizing the first iteration of the hidden layer timing. Counters can also be viewed.

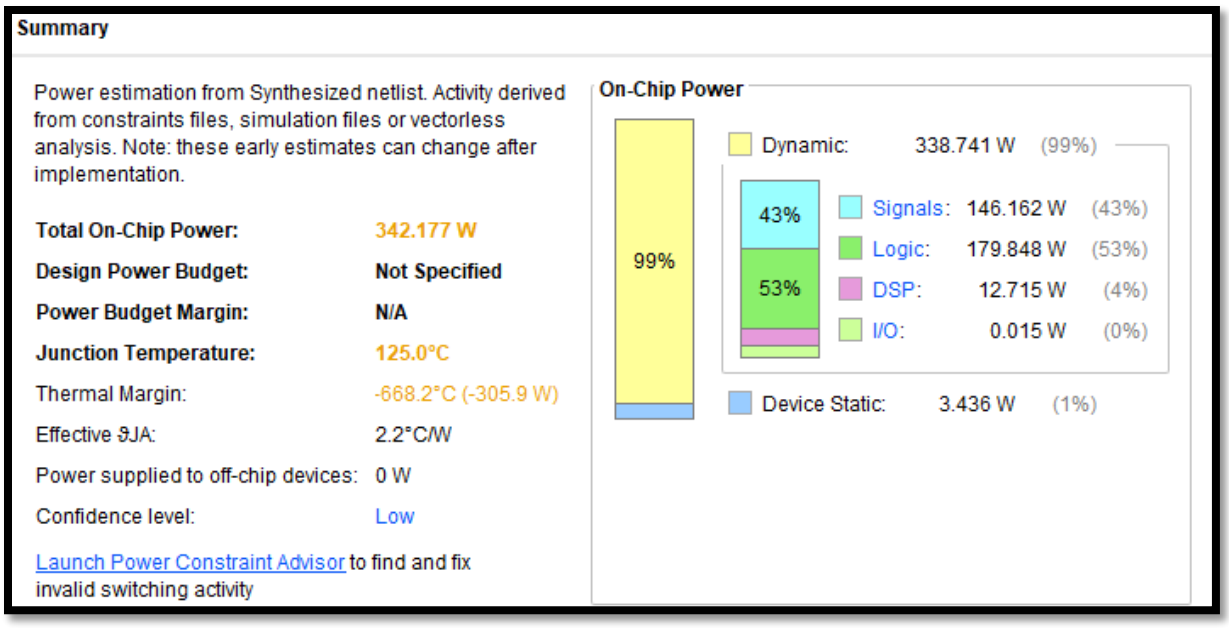

Figure 3.3.6
Power Summary for 49-MUL

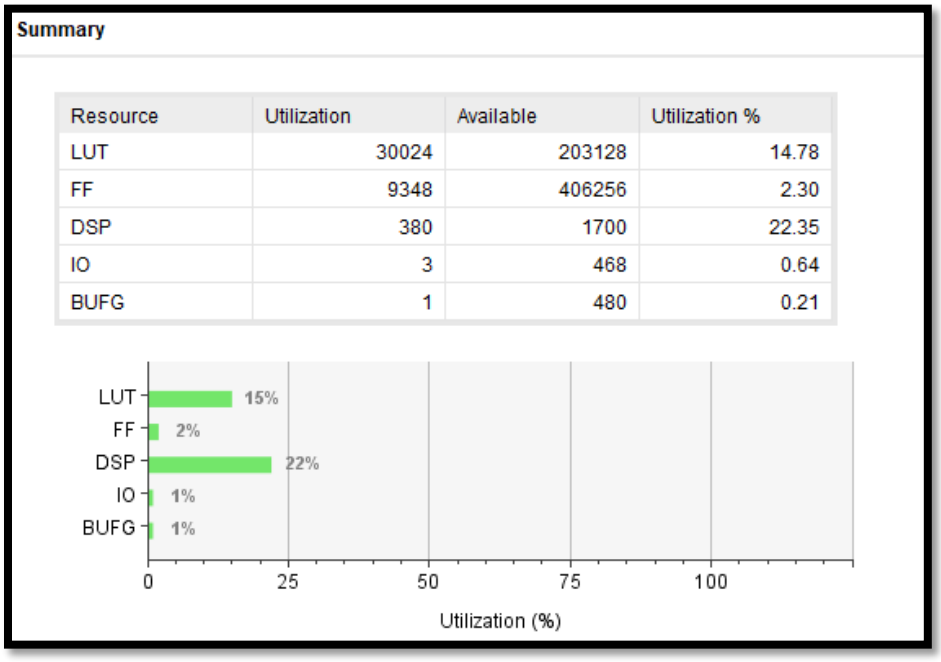

Figure 3.3.7
Utilization Summary for 49-MUL

3.4 Architecture Design 196

The 196-MUL architecture design, has 196 multipliers and needs to be iterated 4 times to process all 784 inputs. Similarly done with the 28 and 49 multiplier architecture designs I will include the figures below. The first figure in this section is Figure 3.4.1. It contains the first part of the design for the 196-MUL hidden layer output design. Figure 3.4.2 shows the second half of the hidden later output design. Figure 3.4.3 is the timing diagram for the same design. The design is also very intricate, therefore Figure 3.4.4 includes the very first iteration of the hidden layer timing and Figure 3.4.5 includes the output layer timing. Although the timing diagram is intricate, the 196-MUL timing diagram does not have as many iterations, therefore it may be easier to view than the previous designs. Lastly, included is Figure 3.4.6 that discloses the power dissipation for the 196-MUL design and Figure 3.4.7 which indicates the overall resources used.

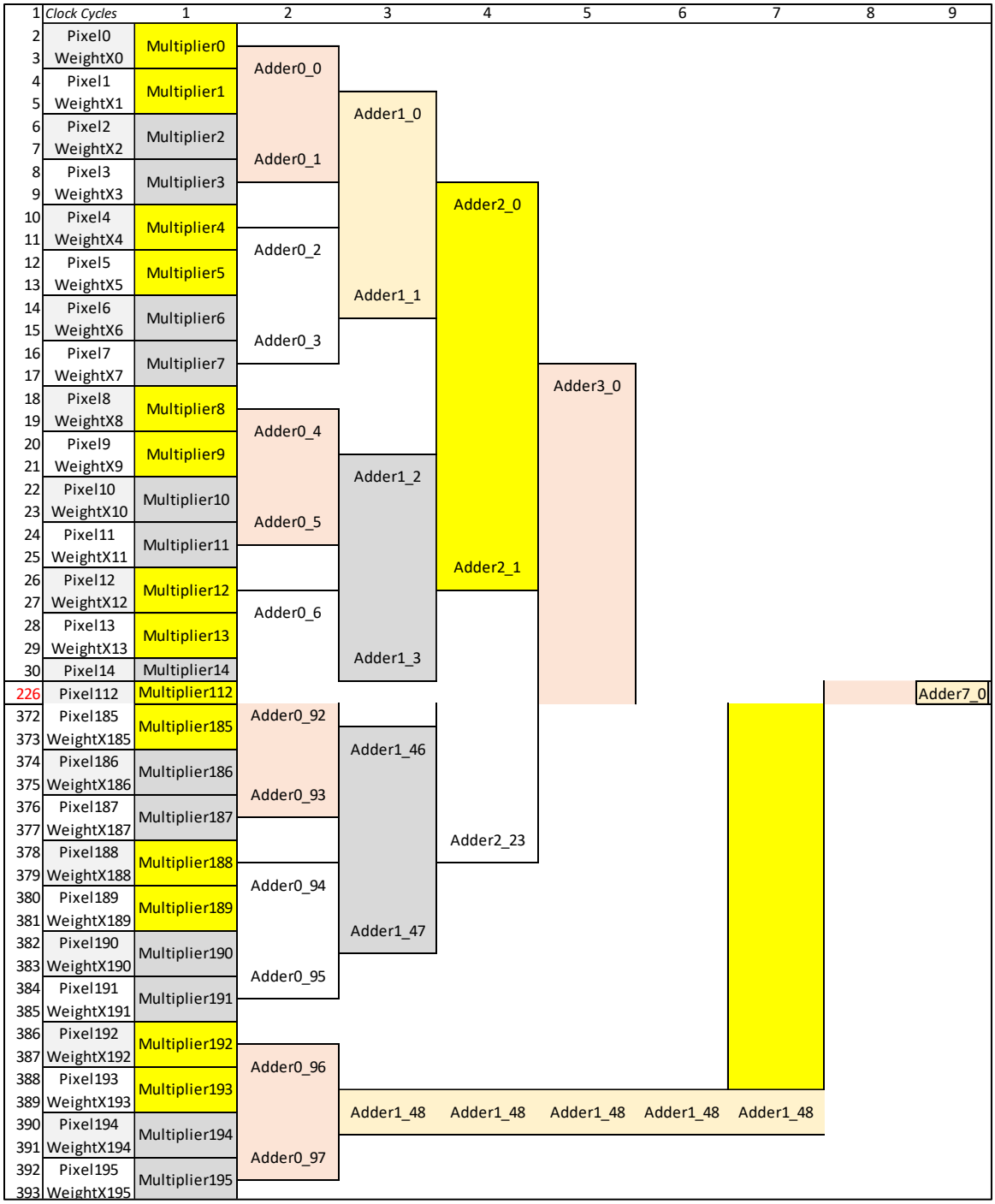

Figure 3.4.1

First half of the hidden layer output design used in the 196-MUL architecture design. Make note that there are hidden rows.

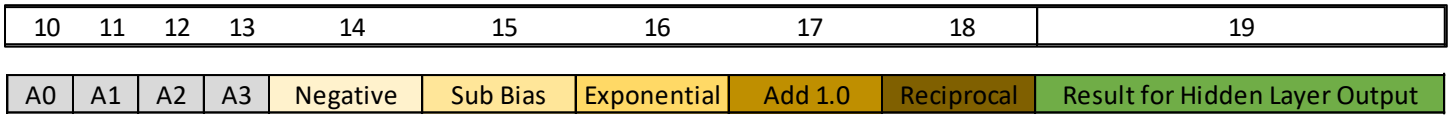

Figure 3.4.2

Second half of the hidden layer output design used in the 196-MUL architecture design.

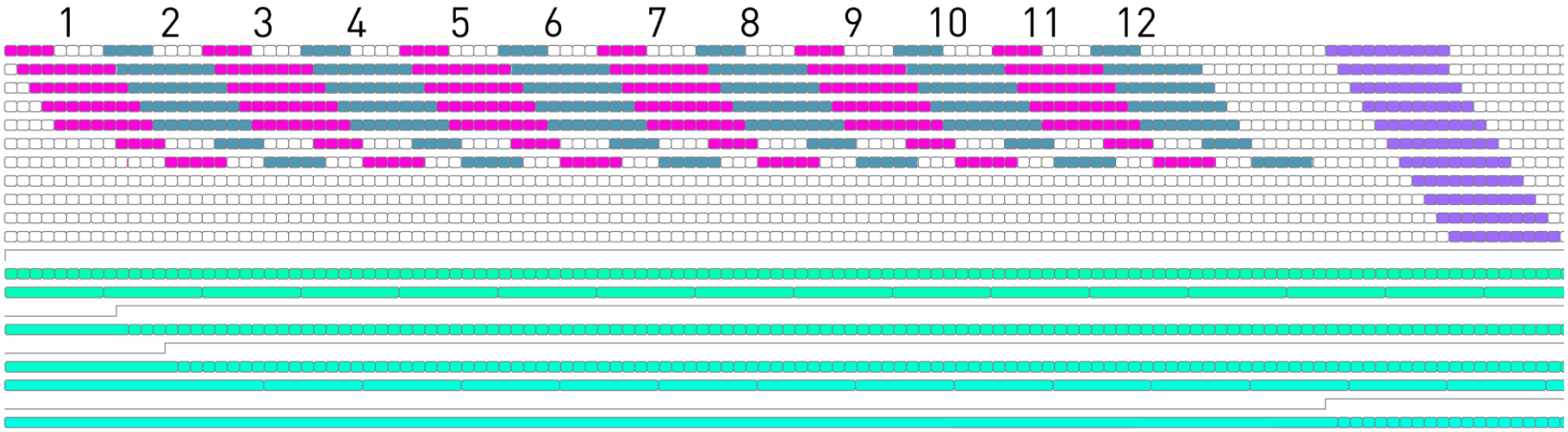

Figure 3.4.3

Timing diagram to show structure for the 196-MUL architecture design.

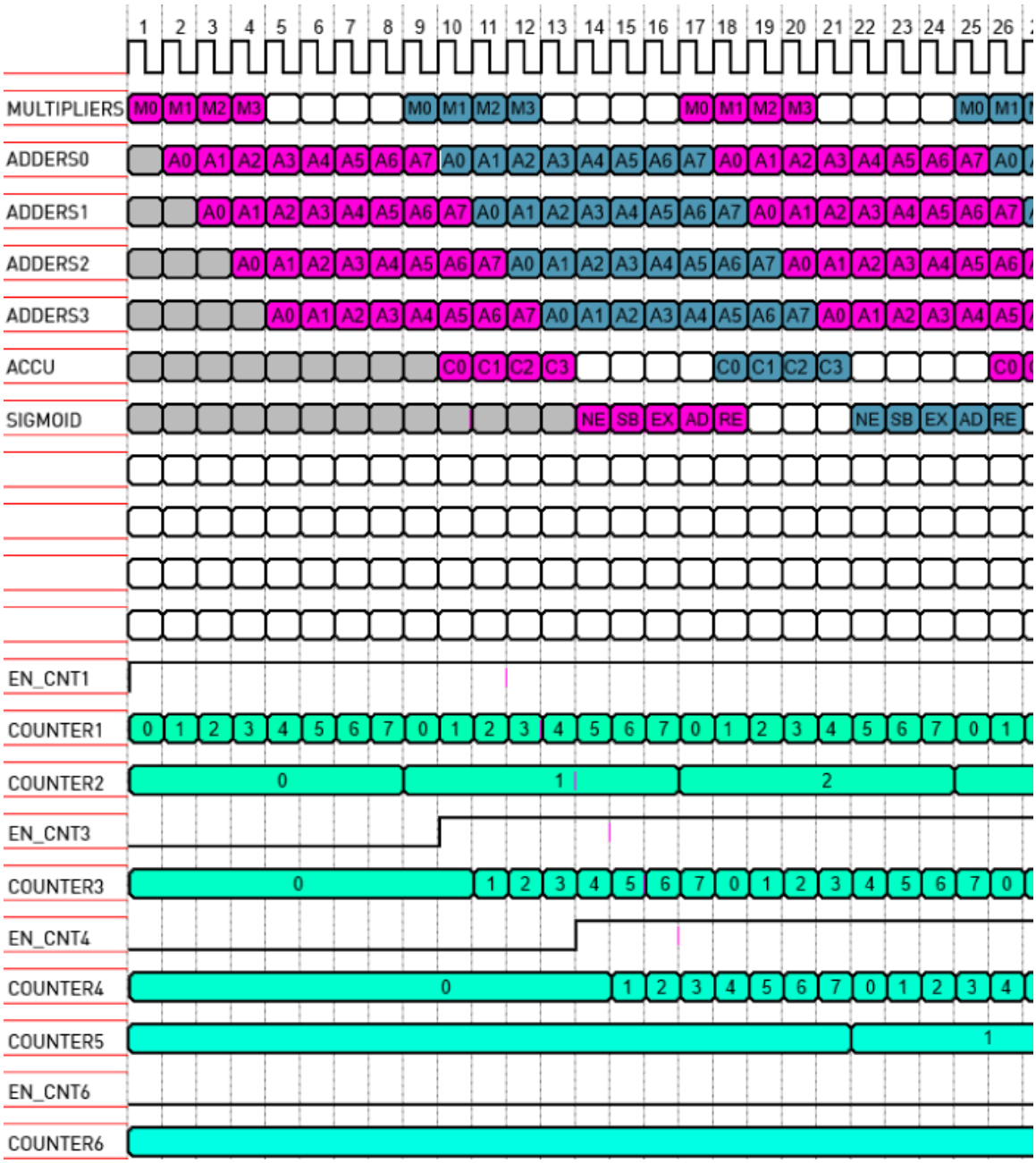

Figure 3.4.4

Partial timing diagram for the 196-MUL architecture design, emphasizing the first iteration of the hidden layer timing. Counters can also be partially viewed.

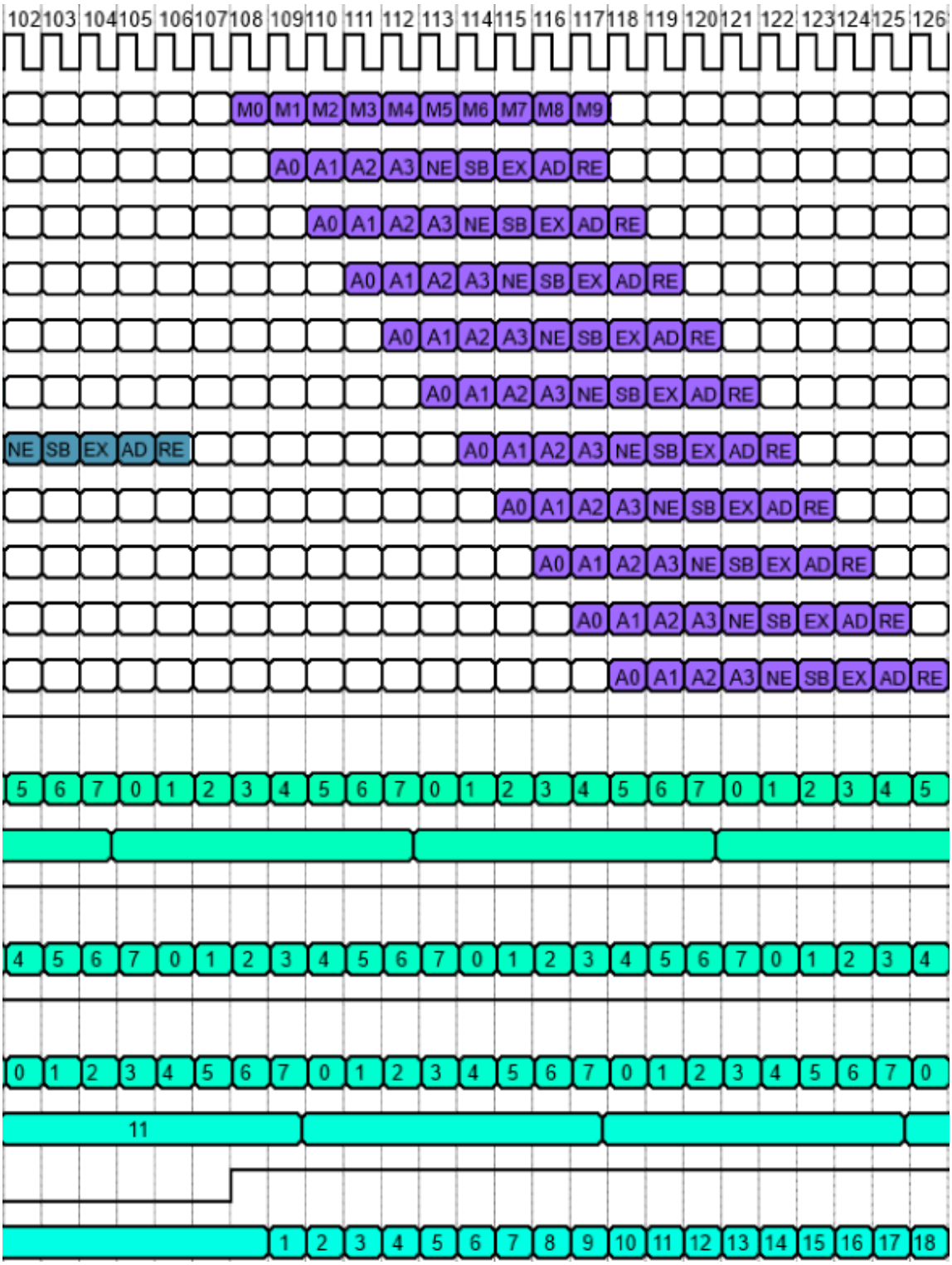

Figure 3.4.5

Partial timing diagram for the 196-MUL architecture design, emphasizing the outer layer timing. The counters are partially visible in the green-blue gradient colors.

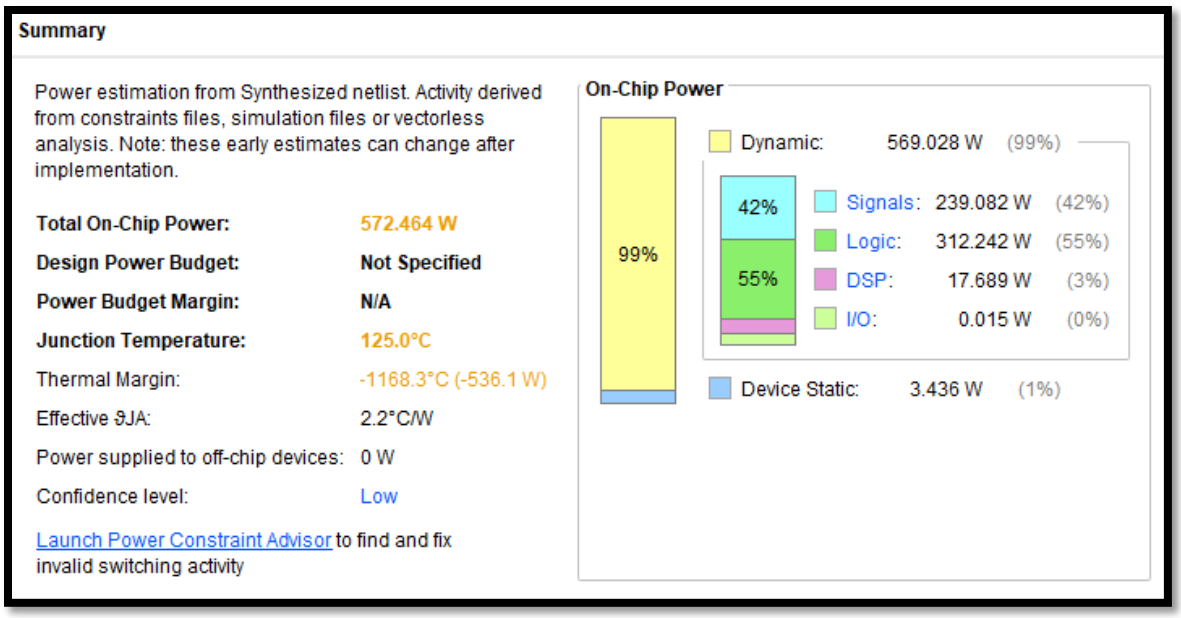

Figure 3.4.6
Power Summary for 196-MUL

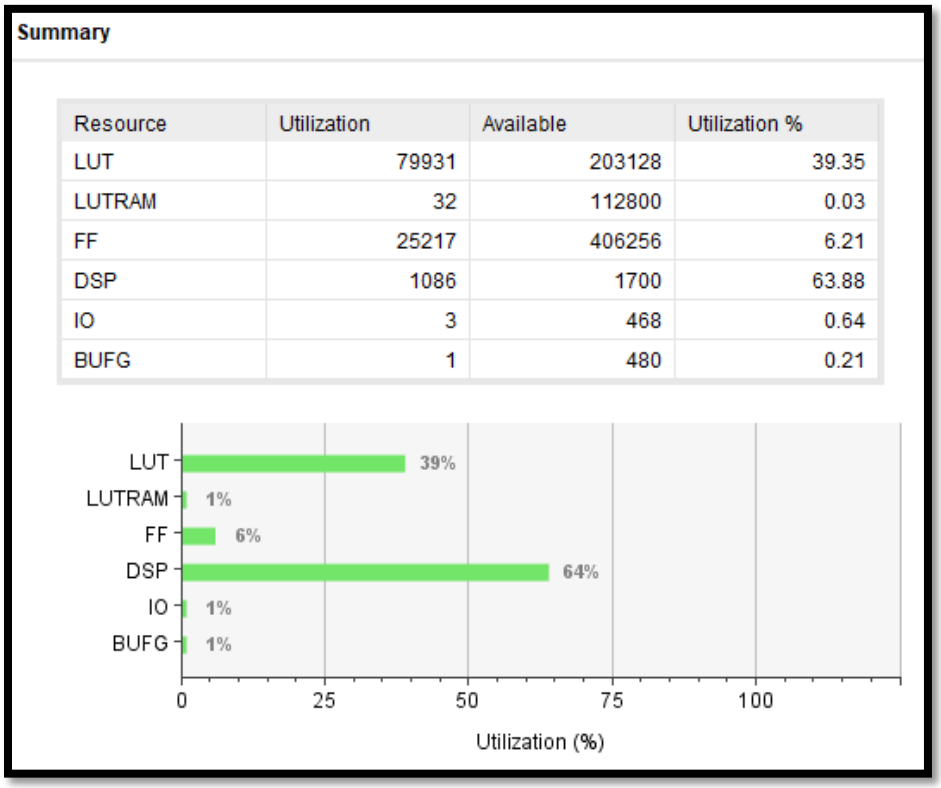

Figure 3.4.7
Utilization Summary for 196-MUL

CHAPTER IV:

FLOATING POINT COMPONENTS

In this chapter, the designs on floating point adders and multipliers are mainly discussed. Further, the simulation and synthesis results are shown by using Xilinx Vivado and the FPGA design flow [65, 66].

4.1 Floating Point Adder Design

First, data is extracted from the two inputs and the data is then distributed to the three main parts (sign, exponential bias, and significand) for each input. Next, it is determined if any exceptions are present. For example, if we have a Not a number (NaN) added to a normal number the result would be NaN (32'hFFFFFFFF). Another example would be with a positive infinity added to a normal number. That would result in the answer to be positive infinity (32'h7F800000). If an exception is found, then the predetermined results are sent to the output. Otherwise, concatenation to each of the inputs' significands occur followed by the exponential comparison. During the exponential comparison the signs will be determined. If the inputs' exponential biases are equal, then move onto the next step which is significand addition. If the inputs' exponential biases are not equal, then the next step is finding the smaller exponential bias. The smaller exponential bias is subtracted from the larger and that difference determines how much the small exponential bias's input's significand needs to be shifted. The smaller exponential bias then takes on the same exponential bias of the larger exponential bias. Now significand addition can occur. A leading zero counter is implemented to see if the results from the significand addition need to be normalized. If yes, then normalization occurs and move onto the step of special cases. If no, then move onto the next step, special cases. If a special case is valid then the results are sent to the

output, otherwise rounding will occur. The result of the normalized significand addition needs to be rounded. For rounding, we will have four bits that will help determine whether rounding up or rounding down should occur. The check bit which is the least significant bit. The next three bits are to the right of the check bit, in the subsequent order: guard bit, round bit, and sticky bit. The sticky bit is a reduction or of all the rest of the bits that are to the right of the round bit. The guard bit, round bit, and sticky bit are concatinated together and they form grs. If grs = 100, then the check bit will be checked if it contains a one or a zero. If a zero is present then we round down, by not doing anything to the significand. If a one is present then we round up. Rounding up calls for the significand to be incremented by one. Rounding up also occurs when the gbs equals 101, 110, or 111. Rounding down occurs when the guard bit equals 0 or when the gbs equals 0XX. Once rounded, the result is then checked to see if it is still normalized. If normalization needs to occur then the normalization step will be repeated. If the result is still normalized then send the significand and exponent to the final output.

These were the steps that I followed in order to build a floating point adder adhereing to the IEEE 754 Standard. Below you can find a flow chart for this adder.

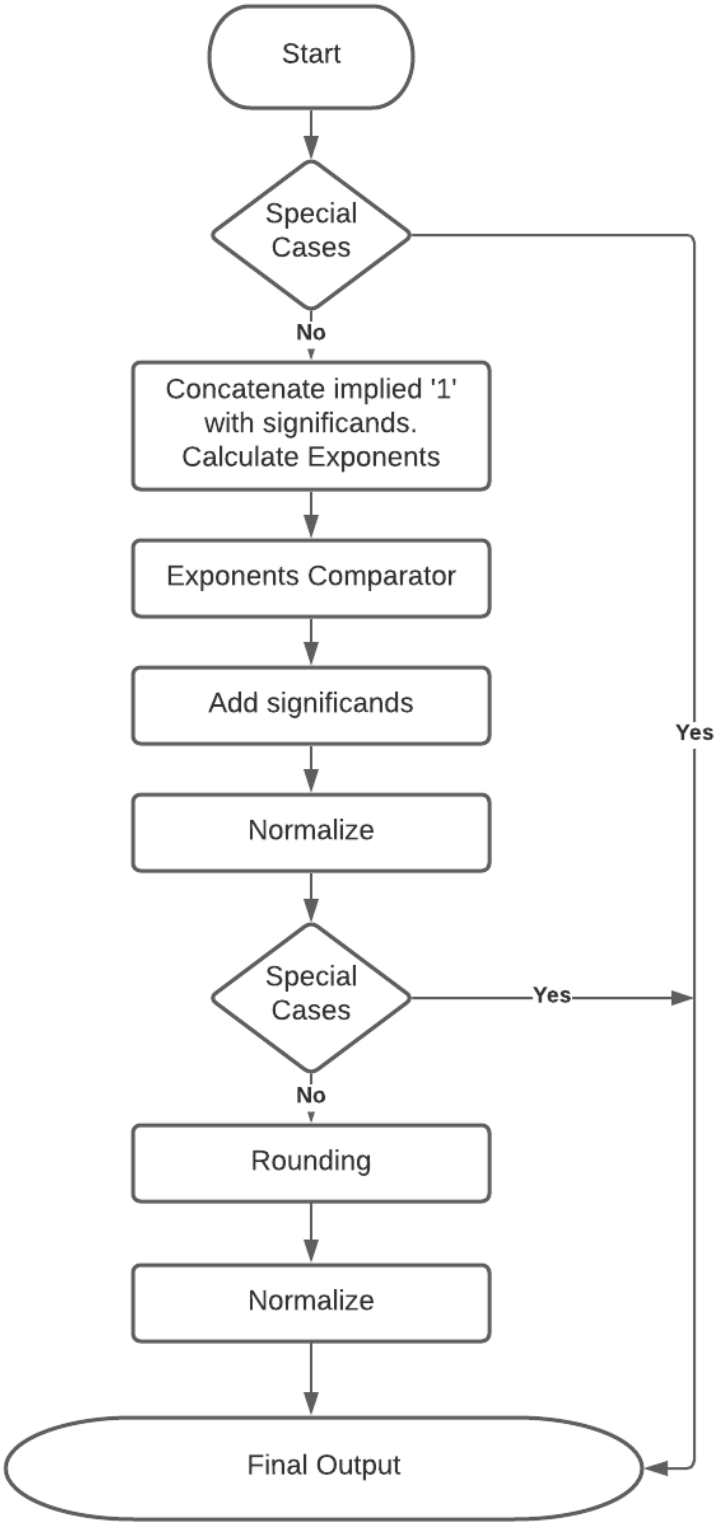

Figure 4.1.1
Adder flow chart

4.1.2 Adder Simulation Results

The adder was constructed using the Hardware Description Language, Verilog HDL. The design was created, simulated, and synthesized in Xilinx Vivado. The results were verified with handwritten computation and an online IEEE 754 Standard floating point calculator, IEEE 754 Calculator. At the bottom of the webpage I selected binary32, inputted the two inputs in hexadecimal format, and selected the addition symbol. The online calculator can be found at: http://weitz.de/ieee/. For the test bench, I tested with 20 different test cases. The simulation results matched my expected results for all test cases.

Below, you can see in Figure 4.1.2 nine test cases are displayed in the waveform. For the sixth test case my input for A was 0xDFF236E1 and my input for B was 0x5FCFC469. My output for F was 0xDE89C9E0. For the eighth test case my input for A was 0x510D36B7my input for B was 0x510D58B5. My output for F was 0x518D47B6. The other signals that are displayed represent some of the component stages that were described in my adder design and flow chart. For example, “ExAB” the larger exponent that will be sent forward and evaluated when normalization occurs. .

In Figure 4.1.3, we can analyze the power implementation estimate. The total on-chip power estimation is 10.471 W. The on-chip power distribution for static power is 6% while the dynamic power is the remaining 94%.

In Figure 4.1.4, we can analyze the adder’s resource utilization. The Look up tables (LUT) utilizes 880 out of the 203,128 available. The flip flops (FF) utilizes 333 out of 406,256 available. The inputs/output ports utilizes 97 out of 368 available. The LUTs and FFs utilize less than 1% of the board’s corresponding resources while the IOs utilize a little over one-fifth of the board’s IO resources.

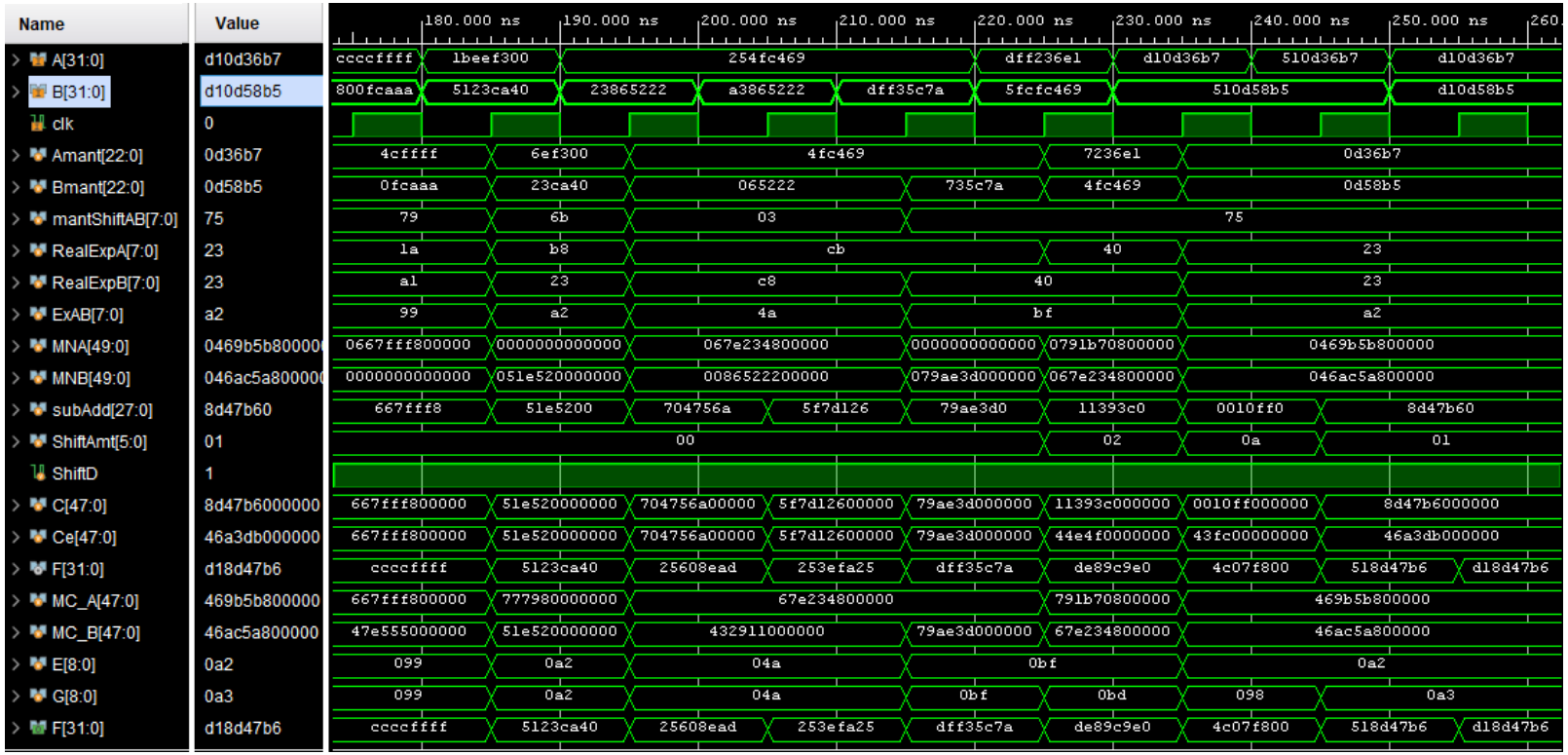

Figure 4.1.2

Adder component showing the test bench simulation results. A and B are inputs and F is the output.

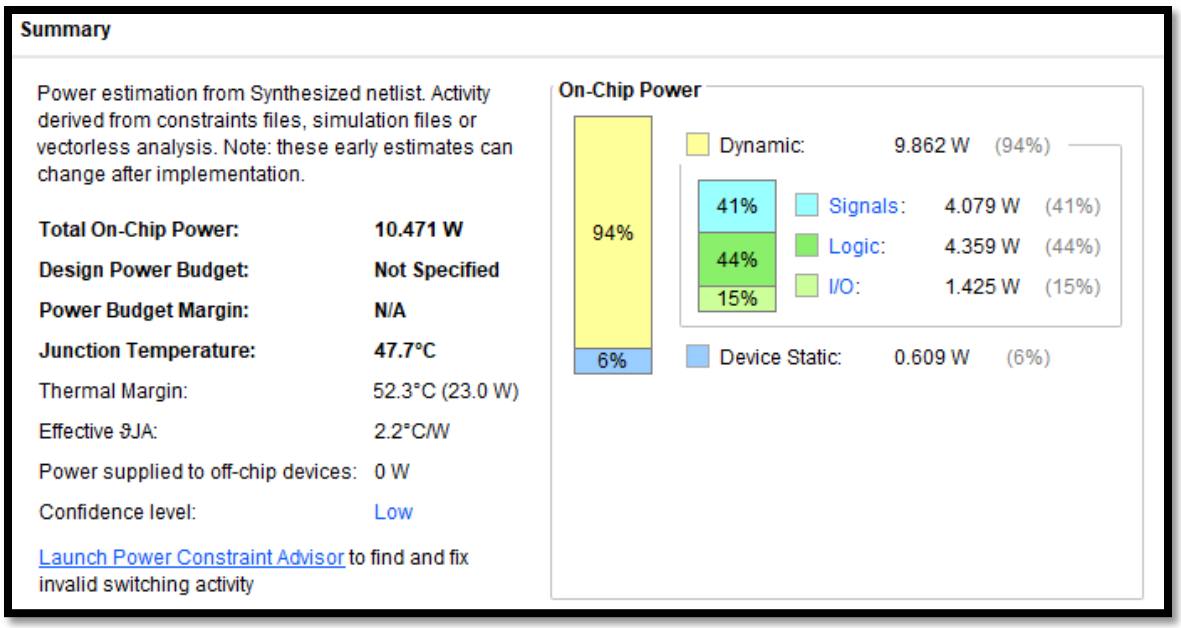

Figure 4.1.3
Adder summarization for estimate power implementation.

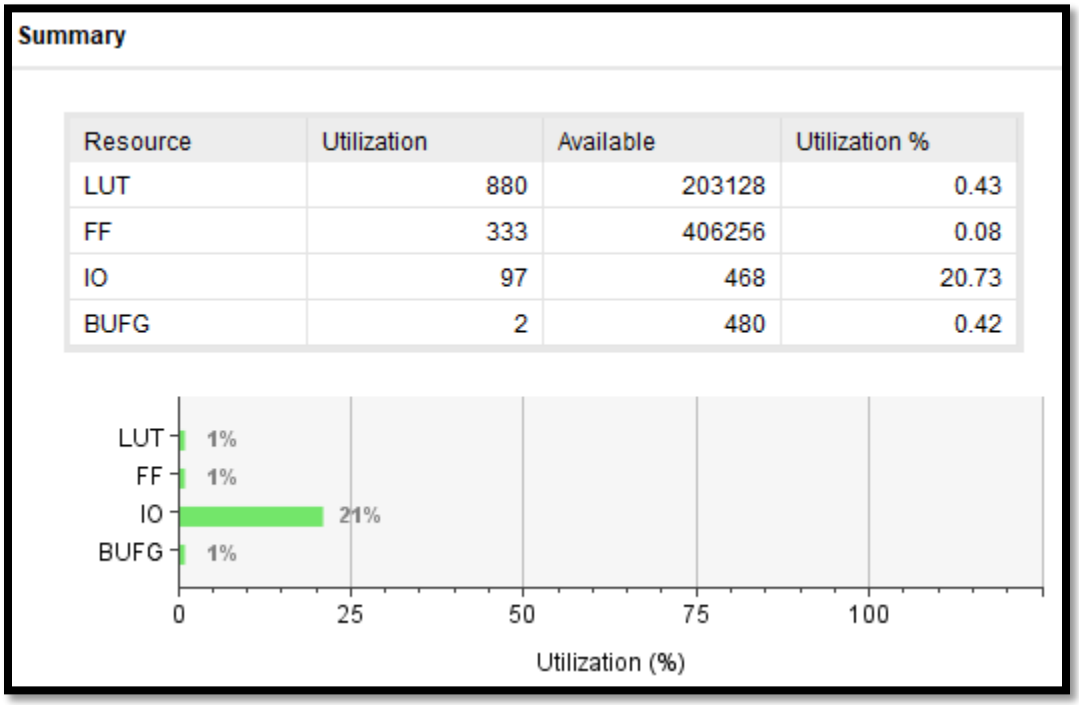

Figure 4.1.4
Adder summarization for resource utilization.

4.2 Floating Point Multiplier Design

Some of the steps that are needed for the multiplier will replicate some of the steps we used for the adder. First, data is extracted from the two inputs and the data is then distributed to the three main parts (sign, exponential bias, and significand) for each input. Next, it is determined if any exceptions are present. If this is the case, then the exception is determined and the results are sent to the output. If no exceptions are present then we XOR the signs from each input and send that result to the sign bit in the final output.

The next step involves calculating the exponential bias. First, we add the exponential biases from each of the inputs. Then we subtract out the bias associated each precision. Since we are using single precision in our project, we will subtract 127 from the summed exponential biases.

Multiplication of the significands follows exponential addition. As mentioned before, we can find the placement of the multiplication result by summing the significand's length for each input. This will let us know where our decimal placement will be in our result. Therefore in our result significand, if our leftmost bit is 0 then our decimal lies between the 45th and 46th bit.

Normalization step occurs next. We need to find the leading one and count the zeroes before that leading one. This is done to know how much to shift the resultant significand. This information is also used to adjust the exponent bias value. Once normalization is complete or if normalization is not needed then move to the step of special cases. If the special case is valid then the results are sent to the output, otherwise rounding will occur.

Once normalization is complete, the result of the normalized significand multiplication needs to be rounded. For rounding, we followed the round to nearest even.

If the grs was 0XX or 100 with a checkbit equal to zero then we would leave the result. Otherwise we would increment the signficand by one. After the result is rounded, the rounded result is then checked to see if it is still normalized. If it is not normalized, normalization needs to occur again. Once the result is normalized, send the significand and exponent information to the final output. Multiplication of the significands is now complete.

This concludes the steps that I followed in order to build a floating point multiplier. On the following page is the flow chart for this multiplier.

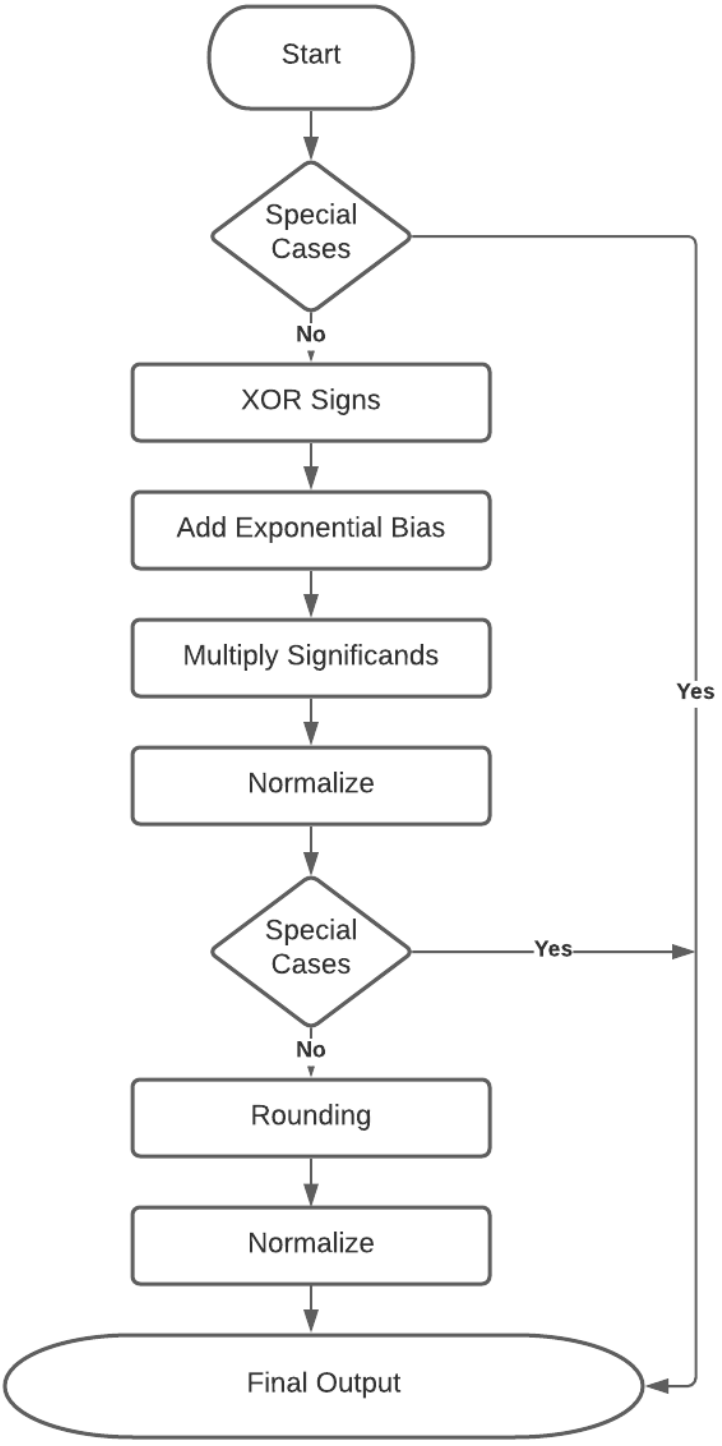

Figure 4.2.1
Multiplier flowchart

4.2.2 Multiplier Simulation Results

The multiplier was constructed using the Hardware Description Language, Verilog HDL. The design was created, simulated, and synthesized in Xilinx Vivado. The results were verified the same way that the adder results were verified, with handwritten computation and an online IEEE 754 Standard floating point calculator, IEEE 754 Calculator that was mentioned before during the Addition Simulation Results. For the test bench, I tested with 20 different test cases. The simulation results matched my expected results for all test cases.

Below, you can see in Figure 4.2.2 two test cases are displayed in the waveform. For the first test case my input for A was 0xf0000001 and my input for B was 0xb000000f. My output for F was 0x60800010. For the second test case my input for A was 0x254fc469 and my input for B was 0xc7f236e1. My output for F was 0xadc49435. The other signals that are displayed represent some of the component stages that were described in my multiplier design and flow chart. For example, “E” represents the step in which two exponential biases are summed together, then the bias is subtracted from the sum.

In Figure 4.2.3, we can analyze the power implementation estimate. The total on-chip power estimation is 32.451 W. The on-chip power distribution for static power is 6% while the dynamic power is the remaining 94%.

In Figure 4.2.4, we can analyze the multiplier’s resource utilization. The Look up tables (LUT) utilizes 452 out of the 203,128 available. The flip flops (FF) utilizes 32 out of 406,256 available. The inputs/output ports utilizes 97 out of 468 available. The LUTs and FFs utilize less than 1% of the board’s corresponding resources while the IOs utilize almost one-third of the board’s IO resources.

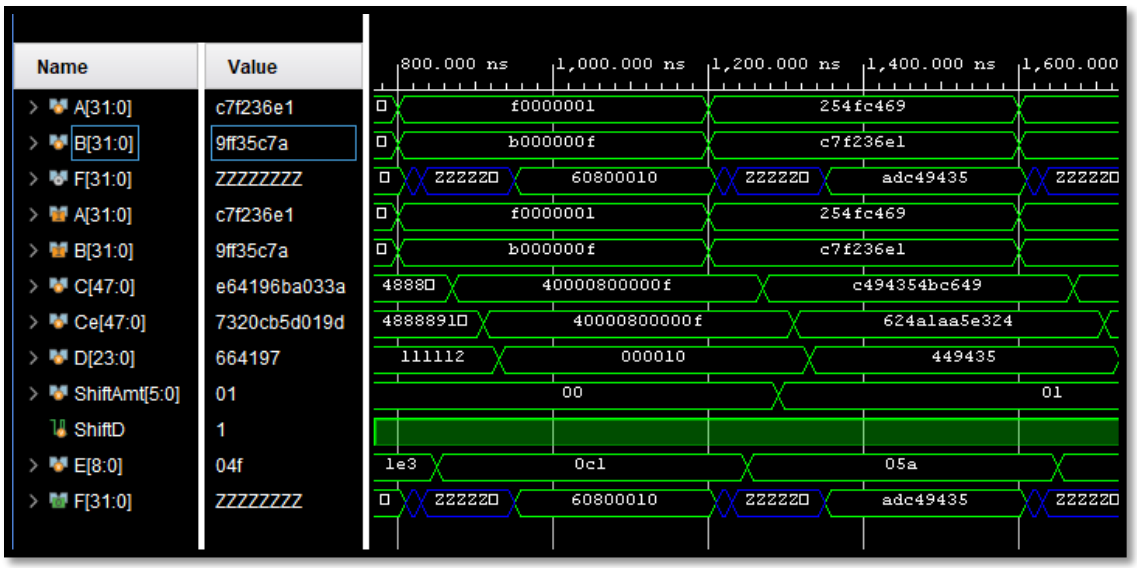

Figure 4.1.2
Multiplier component showing the test bench simulation results

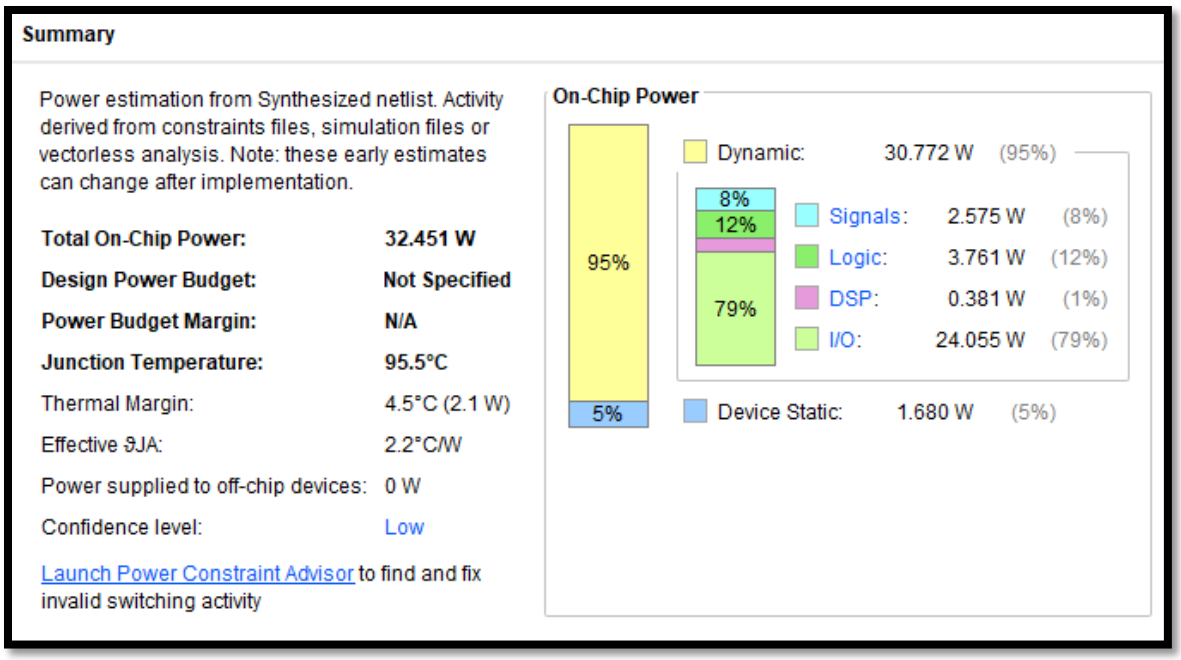

Figure 4.2.3
Multiplier summarization for estimate power implementation.

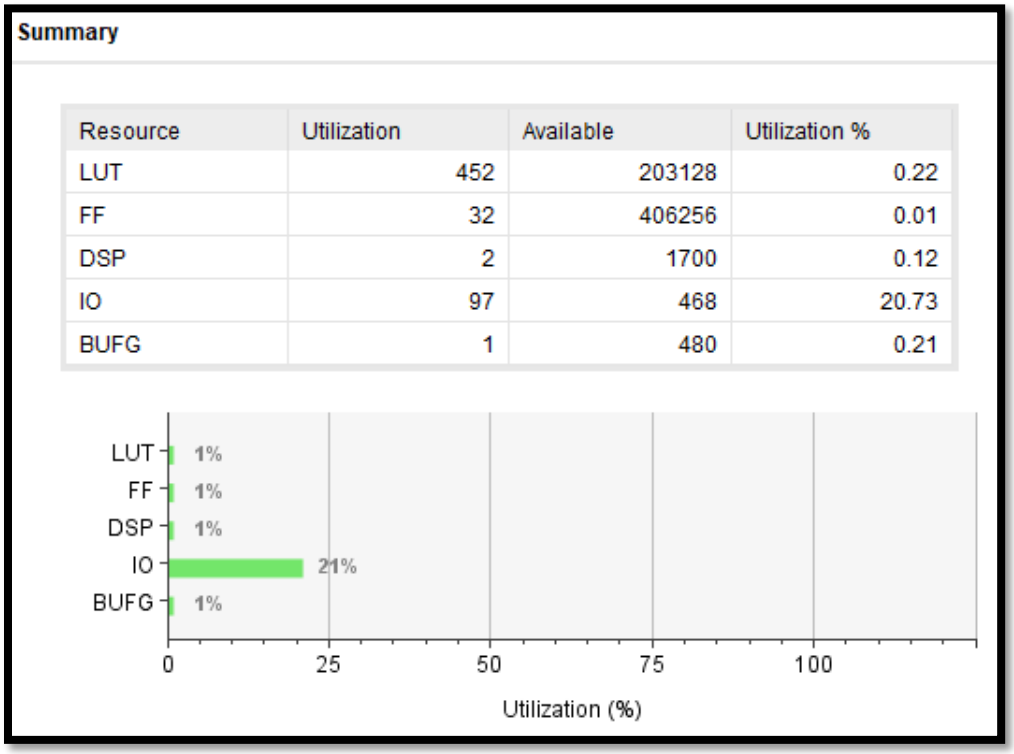

Figure 4.2.4
Multiplier summarization for resource utilization.

CHAPTER V:

ADDER AND MULTIPLIER RTL VERSUS IP

The goal of creating the RTL adder and multiplier was to be able to instantiate them into our 28-MUL, 49-MUL, 98-MUL, and 196-MUL designs, instead of using the Xilinx IP adder and multiplier. Instantiating these designs creates our 28-MUL-AM, 49-MUL-AM, 98-MUL-AM, and 196-MUL-AM designs. The latency should be the same between the same numbered MUL and MUL-AM designs, but there should be a significant difference between the designs in regards to power and area. After running the 28-MUL-AM design and comparing the results with the 28-MUL design, it became apparent that the Xilinx IP adders and multipliers were superior in resource and power consumption. Therefore, instead of detailing the differences between all of the MUL and MUL-AM designs, in this section, I will only go over the 28-MUL and 28-MUL-AM design, as well as compare the RTL and Xilinx IP adder and multiplier.

Now that the floating point multiplier and adder have been made we can instantiate those files into our 28-MUL design and not use the Xilinx IP floating point multiplier and adder. This creates our 28-MUL-AM design. Although the RTL adder and multiplier did not have lower resources and power costs than the IP versions, this still allows for user flexibility in the future, in regards to accuracy in digit detection. Custom design implementations can be made with the main goal of power costs and/or utilization costs for the user. The custom floating point designs was slightly more accurately in testing for the digit one, but the cost in resources and power have increased tremendously in comparison to the designs using the Xilinx IPs. The 28-MUL-AM strongly detected the digit one with a decimal result of 0.999954, while the 28-MUL strongly predicted the digit one with a decimal result of 0.9997094 precision. Even though the prediction was slightly higher on the 28-MUL-AM design for the digit one, there was a digit that the 28-

MUL-AM also tried to predict; a prediction of the digit six with a decimal result of 0.4561733.

Figure 5.1.1 displays the power cost for the 28-MUL-AM design. Figure 5.1.2 displays the utilization cost for the 28-MUL-AM design. Figure 5.2.1 shows the 28-MUL simulation results for digit one. Figure 5.2.1 shows the 28-MUL simulation results for digit one. Table 5.2.1 examines the previous two simulation results for digit recognition between the 28-MUL and 28-MUL-AM designs and shows the digit predictions. Lastly, we will examine the power and utilization figures for the RTL and Xilinx IP adder and multiplier.

5.1 Design 28-MUL-AM

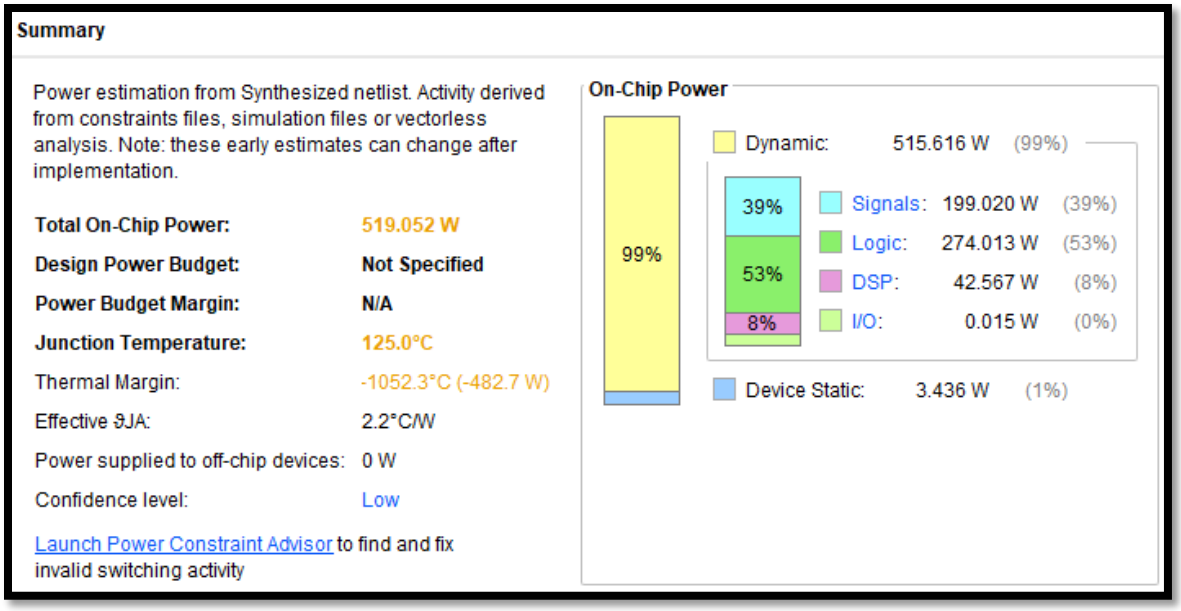

Figure 5.1.1
Power Utilization Summary for 28-MUL-AM

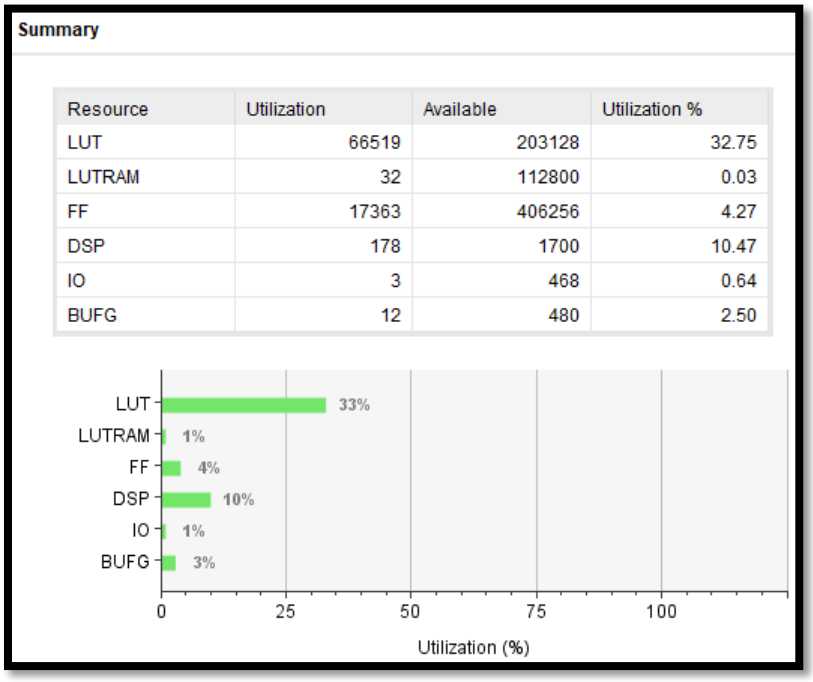

Figure 5.1.2
Utilization Summary for 28-MUL-AM

5.2 Simulation Results Comparison

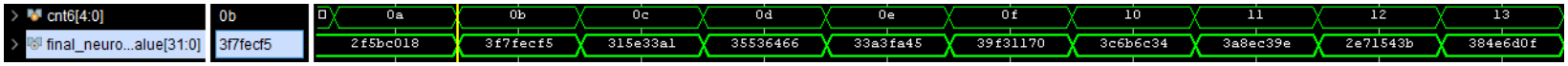

Figure 5.2.1
28-MUL Simulation Results for digit #1

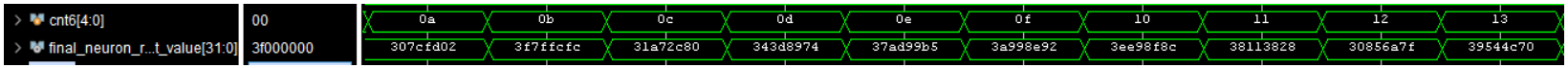

Figure 5.2.2
28-MUL-AM Simulation Results for digit #1

Comparing Simulation Results for Digit Recognition of Digit #1					
Possible Digit	**28_MUX**		**28_MUX_AM**		**Percent Difference**
	Probability of Digit		**Probability of Digit**		
0	0x2f5bc018	2.00E-10	0x307cfd02	9.20E-10	0.00000007%
1	0x3f7fecf5	0.9997094	0x3f7ffcfc	0.999954	0.02446000%
2	0x315e33a1	3.23E-09	0x31a72c80	4.87E-09	0.00000016%
3	0x35536466	7.87E-07	0x343d8974	1.77E-07	-0.00006110%
4	0x33a3fa45	7.64E-08	0x37ad99b5	2.07E-05	0.00206184%
5	0x39f31170	4.64E-04	0x3a998e92	1.17E-03	0.07079307%
6	0x3c6b6c34	1.44E-02	0x3ee98f8c	4.56E-01	44.18042410%
7	0x3a8ec39e	1.09E-03	0x38113828	3.46E-05	-0.10545809%
8	0x2e71543b	5.49E-11	0x30856a7f	9.71E-10	0.00000009%
9	0x384e6d0f	4.92E-05	0x39544c70	2.02E-04	0.01532479%

Table 5.2.1
A comparison of the simulation results for digit recognition between 28-MUL and 28-MUL-AM for digit #1

5.3 Simulation Results Comparison

Figure 5.3.1 is the Xilinx IP adder Power Summary. Figure 5.3.2 is the Xilinx IP adder Utilization Summary. Figure 5.3.3 is the Xilinx IP multiplier Power Summary. Figure 5.3.4 is the Xilinx IP multiplier Utilization Summary. Lastly, Table 5.3.5 is the table that compares the previous four figure's results with the RTL figures that were mentioned earlier.

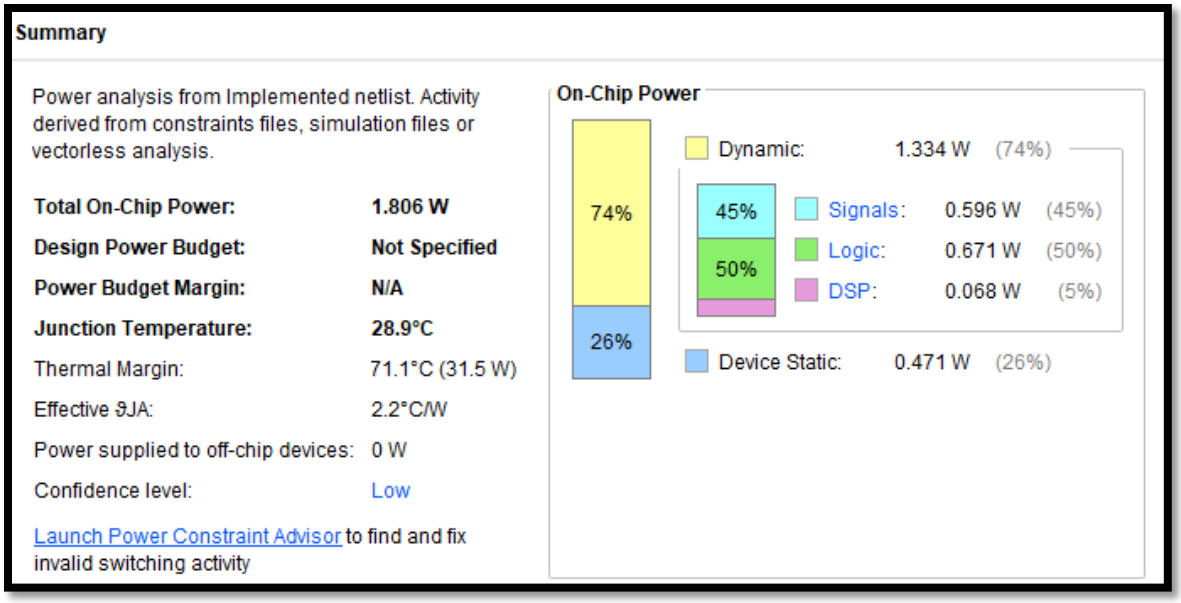

Figure 5.3.1
Xilinx IP Adder summarization for estimate power cost.

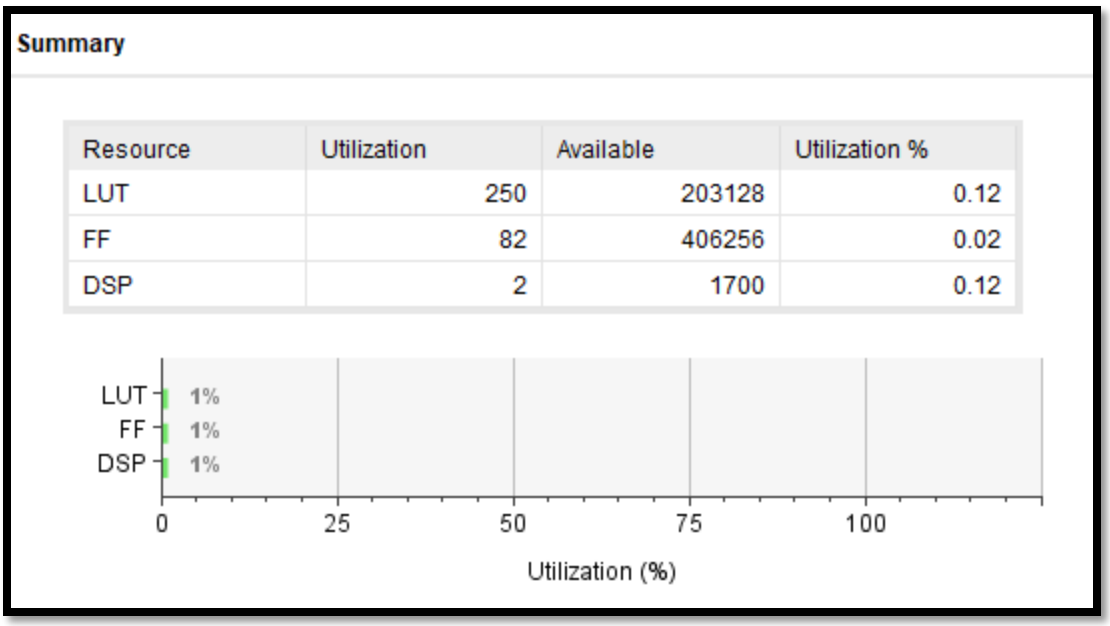

Figure 5.3.2
Xilinx IP Adder summarization for estimate utilization cost.

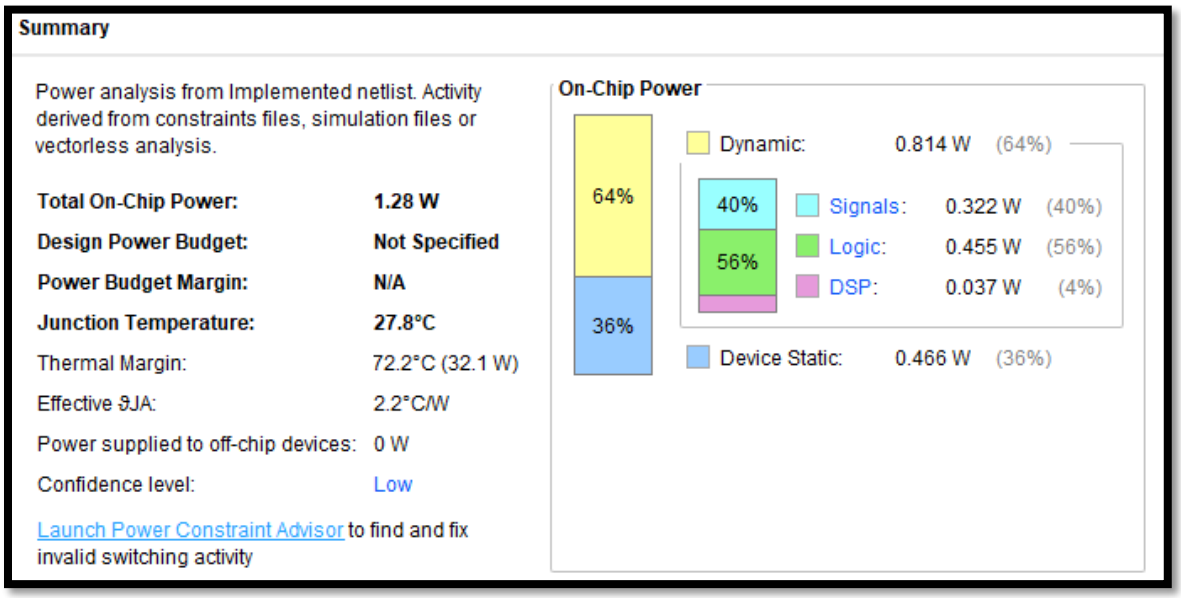

Figure 5.3.3
Xilinx IP Multiplier summarization for estimate power cost.

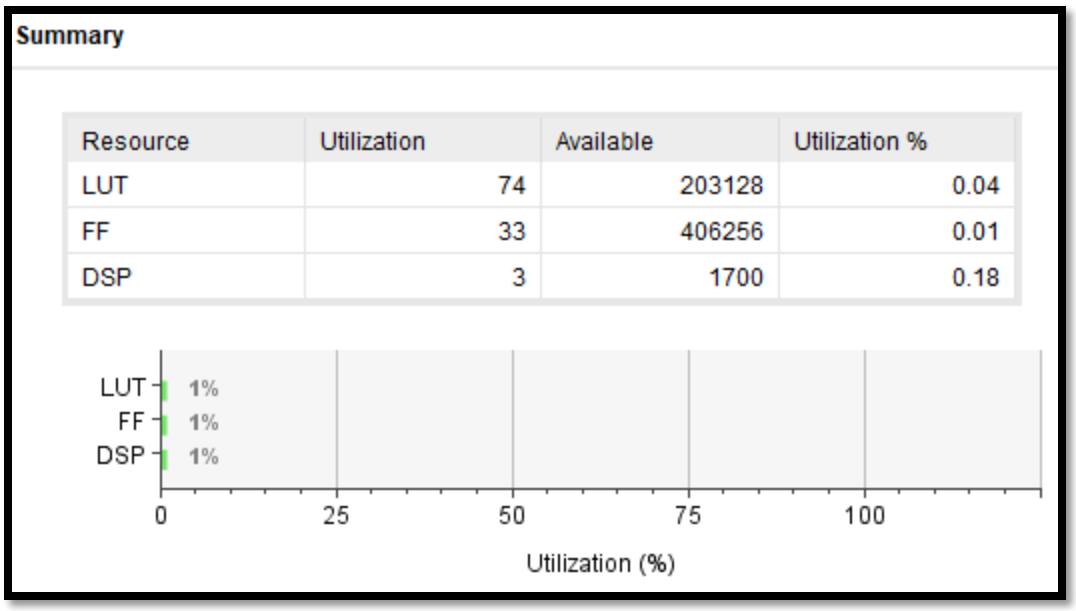

Figure 5.3.4
Xilinx IP Multiplier summarization for estimate utilization cost.

IP and RTL Adder and Multiplier Comparison				
	Adder		Multiplier	
	Xilinx IP	**RTL design**	**Xilinx IP**	**RTL design**
LUT	250	880	74	452
FF	82	333	33	32
DSP	2	N/A	3	2
Power	1.806 W	10.471 W	1.28 W	32.451 W

Figure 5.3.5
Comparison between the Xilinx IP and RTL designs for the adders and Multipliers

CHAPTER VI:

EXPERIMENTAL RESULTS

After the implementation of the MLPNN, this chapter discusses the hardware performance in terms of execution time, slice count, and power dissipation. Finally the comparison between the existing works and my proposed design is discussed.

6.1 Execution Time

The designs that had the lowest latency between the multiplier architecture designs was the 196 multiplier architecture design. The 196 multiplier architecture design executed in 126 clock cycles. The 98 multiplier architecture design executed in 129 clock cycles. The 49 multiplier architecture design executed in 246 clock cycles. The 28 multiplier architecture design executed in 390 clock cycles. In the tables below, we can see the different designs with their latency followed by another table that emphasizes the percentage change in latency compared to the 196 multiplier design.

Latency Comparison				
Designs	28-MUL	49-MUL	98-MUL	196-MUL
Clock Cycles	390	246	129	126

Table 6.1.1
Latency comparison displayed in clock cycles for all designs

Latency Percent Difference				
Designs	28-MUL	49-MUL	98-MUL	196-MUL
Clock Cycles	390	246	129	126
% difference in comparison to the 196-MUL design	210%	95%	2%	0%

Table 6.1.2
Percent difference calculations for latency compared to the 196-MUL design.

6.2 Resource Cost

The design that had the lowest resource cost between the multiplier architecture designs was the 28-MUL architecture design. The 28-MUL architecture design utilized 294 DSPs, 7865 FFs, and 24,662 LUTs. The 49-MUL architecture design utilized 380 DSPs, 9,348 FFs, and 30,024 LUTs. The 98-MUL architecture design utilized 604 DSPs, 14,274 FFs, and 44,668 LUTs. The 196-MUL architecture design utilized 1,086 DSPs, 25,217 FFs, and 79,931 LUTs. The 28-MUL-AM architecture design utilized 178 DSPs, 17,363 FFs, and 66,519 LUTs. The LUTs and FFs were overall much lower in the IP designs (MUL designs), while the DSP count was overall lower in the MUL-AM designs. In the table below, we can see the resources utilization for the LUTs, FFs, and DSPs for our designs. In the following table, the percentage difference is displayed for resource costs compared to the 28-MUL design.

Resources Comparison				
Design	**28-MUL**	**49-MUL**	**98-MUL**	**196-MUL**
LUT	24,662	30,024	44,668	79,931
FF	7,865	9,348	14,274	25,217
DSP	294	380	604	1,086

Table 6.2.1
Resource cost comparison for LUT, FF, and DSP for all designs

Resources Percent Difference				
Design	**28-MUL**	**49-MUL**	**98-MUL**	**196-MUL**
LUT	0%	22%	81%	224%
FF	0%	19%	81%	2%
DSP	0%	29%	105%	96%

Table 6.2.2
Percent difference calculations for resource cost compared to the 28-MUL design.

6.3 Power Cost

The design that had the lowest power cost between the multiplier architecture designs while instantiating the adder and multiplier components was the 28-MUL architecture design. The 28-MUL design power implementation cost is 207.996 W, versus 519.052 W for the 28-MUL-AM design. The 49-MUL design power implementation cost is 342.177 W. The 98-MUL design power implementation cost is 569.749 W. The 196-MUL design power implementation cost is 572.464 W. In the table below, we can see the power cost comparison between all MUL designs. The next table displays the percentage change in power cost between the MUL designs, using the 28-MUL design as the initial design since it had the lowest power cost.

Power Comparison				
Designs	**28-MUL**	**49-MUL**	**98-MUL**	**196-MUL**
Watts	207.996	342.177	569.749	572.464

Table 6.3.1
Power cost comparison in Watts between all designs

Power Percent Difference				
Designs	**28-MUL**	**49-MUL**	**98-MUL**	**196-MUL**
Watts	0%	65%	174%	175%

Table 6.3.2
Percent difference calculations for power cost compared to the 28-MUL design.

6.4 Comparison of Related Work

As mentioned in the beginning, there has been many breakthroughs in hardware acceleration in recent years. Some of the work that has been published focuses on image recognition latency. A 5-layer accelerator for MNIST digit recognition executed processing one image correctly in 25.4 microseconds [13]. Another author was able to processing an image in 17.6 microseconds [14]. Another related work was able to use the same hardware as our design and they were able to processing an image in 14.2 microseconds. The work that this thesis is built upon was able to process an image in 1.55 microseconds [16]. The design with the 196-MUL was able to process an image in 1.52 microseconds. There is room for improvement and there are ways to improve the latency for our system.

With latency, our designs can be competitive, but much work needs to be done to lower our utilization of resources. According to [19] they were able to execute image recognition in 3.2 milliseconds with only 18,426 LUTs and 8,264 FF. [15] has excellent latency with 14.2 microseconds. Their [15] numbers for their resource cost are 213,593 LUTs and 136,677 FFs. The 196-MUL-AM design had the lowest latency and therefore had the highest utilization of resources. The 196-MUL-AM resource cost was 297,302

LUTs and 76,715 FFs. The 28-MUL design has the highest latency and therefore its resource cost is lower with 24,662 LUTs and 7,865 FFs.

Lastly, the power consumption on all my designs were high. The 28-MUL design had the lowest power cost of 208 W. The highest power cost design was the 196-MUL-AM design. Below you can find a table that summarizes the comparison of related work with our 28-MUL design.

Comparison with Existing Work									
Designs	**[12]**	**[10]**	**[19]**	**[11]**	**[13]**	**[14]**	**[15]**	**[16]**	**Proposed Work**
Accuracy	0.9862	0.9864	0.96	0.9467	0.968	0.9757	-	0.9325	0.9582
Latency (us)	26000	3580	3200	637	25.4	17.6	14.2	1.55	3.9
LUTs	14832	32589	18426	38899	80175	12588	213593	44668	24662
FFs	54075	33585	8264	40534	40140	48765	136677	14274	7865
Energy (mJ)	-	-	4.83	-	-	-	-	0.88	0.81

CHAPTER VII:

CONCLUSION

7.1 Conclusion

In conclusion, this dissertation is summarized. Firstly, the paper started with the goals of what should be accomplished, as well as mention the related work to this project. The next section explained a neural network, a single layer perceptron neutral network, a multilayer perceptron neural network, the sigmoid function, and the sigmoid neutron. The next part covered the explanation of the IEEE 754 Standard.

The following section, described the previous work that was done regarding this project and explained how the network was determined. I then went over the three designs that were completed: 28-MUL, 49-MUL, and 196-MUL architecture designs. The next section covered the design aspects of the adder and multiplier. Then all the results were covered.

Overall, this dissertation presented several parallel architectures for handwritten digit recognition in order to improve hardware efficiency in terms of resources, power, and speed in a Field Programmable Gate Array (FPGA) neuromorphic processor. The conclusion of the project is that the proposed designs offered different levels of hardware efficiency depending on what is needed. The proposed designs allow for the user to choose what they deem important, whether they value area, speed, or power. There are many improvements and customization that can be made.

7.2 Future Work

Implementation of custom modules should bring the resource costs lower. With custom modules for adders and multipliers, there is much room to improve. Depending on the desired output regarding area, power, and speed, there are many algorithms that

are geared towards just lowering latency, resources, and power for adders and multipliers. If the goal is low power or low utilization of resources then the entire system can be custom tailored to reach that desired outcome.

In order to demonstrate the FPGA application of classifying handwritten digits, a data path design on image or video processing should be instantiated [67]. By capturing frames of image through a camera the dissertation enables a platform to recognize handwritten digits in real time.

REFERENCES

[1] F. Chen, N. Chen, H. Mao, and H. Hu, "Assessing Four Neural Networks on Handwritten Digit Recognition Dataset (MNIST)," *Computer Vision and Pattern Recognition,* Nov. 2018.

[2] S. Ahlawat, A. Choudhary, A. Nayyar, S. Singh, and B. Yoon, "Improved Handwritten Digit Recognition Using Convolutional Neural Networks (CNN)," *Sensors,* Vol. 2020, No. 20, June 2020.

[3] S. Ali, Z. Shaukat, M. Azeem, et al., "An efficient and improved scheme for handwritten digit recognition based on convolutional neural network," *SN Applied Sciences* Vol. 1, No.9, 2019.

[4] Y. Wang, R. Wang, D. Li, D. Adu-Gyamfi, et al., "Improved Handwritten Digit Recognition using Quantum K-Nearest Neighbor Algorithm," *International Journal of Theoretical Physics,* Vol. 58, PP. 2331-2340, 2019.

[5] S. Hadjis and K. Olukotun, "TensorFlow to Cloud FPGAs: Tradeoffs for Accelerating Deep Neural Networks," *2019 29th International Conference on Field Programmable Logic and Applications (FPL),* 2019.

[6] Q. Wang, Y. Li, B. Shao, S. Dey, and P. Li, "Energy efficient parallel neuromorphic architectures with approximate arithmetic on FPGA," *Neurocomputing,* Vol. 221, PP. 146–158, 2017.

[7] N. I. Chervyakov, P. A. Lyakhov, M. V. Valueva, et al., "Area-Efficient FPGA Implementation of Minimalistic Convolutional Neural Network Using Residue Number System," *2018 23rd Conference of Open Innovations Association (FRUCT),* 2018.

[8] A. Shawahna, S. M. Sait and A. El-Maleh, "FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review," in *IEEE Access*, Vol. 7, PP. 7823-7859, 2019, doi: 10.1109/ACCESS.2018.2890150.

[9] M. Pantho, F. Hategekimana, and C. Bobda, “A System on FPGA for Fast Handwritten Digit Recognition in Embedded Smart Cameras,” *Proceedings of the 11th International Conference on Distributed Smart Cameras September (ICDSC 2017)*, PP. 35–40, 2017.

[10] M. Cho and Y. Kim, "Implementation of Data-optimized FPGA-based Accelerator for Convolutional Neural Network," *2020 International Conference on Electronics, Information, and Communication (ICEIC)*, Barcelona, Spain, 2020, pp. 1-2, doi: 10.1109/ICEIC49074.2020.9050993.

[11] T. Tsai, Y. Ho and M. Sheu, "Implementation of FPGA-based Accelerator for Deep Neural Networks," *2019 IEEE 22nd International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, Cluj-Napoca, Romania, 2019, pp. 1-4, doi: 10.1109/DDECS.2019.8724665.

[12] S. Ghaffari and S. Sharifian, "FPGA-based convolutional neural network accelerator design using high level synthesize," *2016 2nd International Conference of Signal Processing and Intelligent Systems (ICSPIS)*, Tehran, 2016, pp. 1-6, doi: 10.1109/ICSPIS.2016.7869873.

[13] Y. Zhou and J. Jiang, "An FPGA-based accelerator implementation for deep convolutional neural networks," *2015 4th International Conference on Computer Science and Network Technology (ICCSNT)*, Harbin, 2015, pp. 829-832, doi: 10.1109/ICCSNT.2015.7490869.

[14] R. Xiao, J. Shi and C. Zhang, "FPGA Implementation of CNN for Handwritten Digit Recognition," *2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, Chongqing, China, 2020, pp. 1128-1133, doi: 10.1109/ITNEC48623.2020.9085002.

[15] J. Park and W. Sung, "FPGA based implementation of deep neural networks using on-chip memory only," *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Shanghai, 2016, pp. 1011-1015, doi: 10.1109/ICASSP.2016.7471828.

[16] I. Westby and X. Yang, “Exploring FPGA Acceleration on a Multi-Layer Perceptron Neural Network for Digit Recognition,” *The Journal of Supercomputing (J SUPERCOMPUT)*, Under Review, 2020.

[17] S. Wisayataksin and G. Boonyuu, "A Programmable Artificial Neural Network Coprocessor for Handwritten Digit Recognition," *2019 International Conference on Information and Communications Technology (ICOIACT)*, Yogyakarta, Indonesia, 2019, pp. 139-142, doi: 10.1109/ICOIACT46704.2019.8938541.

[18] H. Sharma, et al., "From high-level deep neural models to FPGAs," *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, 2016, pp. 1-12, doi: 10.1109/MICRO.2016.7783720.

[19] H. Madadum and Y. Becerikli, "FPGA-Based Optimized Convolutional Neural Network Framework for Handwritten Digit Recognition," *2019 1st International Informatics and Software Engineering Conference (UBMYK)*, Ankara, Turkey, 2019, pp. 1-6, doi: 10.1109/UBMYK48245.2019.8965628.

[20] J. Si, E. Yfantis and S. L. Harris, "A SS-CNN on an FPGA for Handwritten Digit Recognition," *2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, New York City, NY, USA, 2019, pp. 0088-0093, doi: 10.1109/UEMCON47517.2019.8992928.

[21] J. Si and S. L. Harris, "Handwritten digit recognition system on an FPGA," *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, Las Vegas, NV, 2018, pp. 402-407, doi: 10.1109/CCWC.2018.8301757.

[22] Y. LeCun, C. Cortes, and C. Burges, “MNIST handwritten digit database,” 2010.

[23] K. Vaca, A. Gajjar, and X. Yang, “Real-Time Automatic Music Transcription (AMT) with Zync FPGA,” *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, PP. 378-384, Miami, FL, US, Jan. 13, 2020.

[24] K. Vaca, M. Jefferies, and X. Yang, “An Open Real-Time Audio Processing Platform on Zync FPGA,” *International Symposium on Measurement and Control in Robotics (ISMCR)*, PP. D1-2-1-D1-2-6, Houston, TX, USA, 2019.

[25] H. He, et al., “Synthesize Corpus for Chinese Word Segmentation,” *The 21st International Conference on Artificial Intelligence (ICAI)*, Las Vegas, USA, PP. 129-134, July 29 - August 1, 2019.

[26] H. He, et al., “Dual Long Short-Term Memory Networks for Sub-Character Representation Learning,” *The 15th International Conference on Information Technology - New Generations (ITNG)*, Las Vegas, NV, USA, 2018.

[27] H. He, et al., “Iterated Dilated Convolutional Neural Networks for Word Segmentation,” *Neural Network World (NNW)*, In Press, 2020.

[28] A. Gajjar, et al., “An FPGA Synthesis of Face Detection Algorithm using HAAR Classifiers,” International. *Conference on Algorithms, Computing and Systems (ICACS 2018)*, PP.133-137, July 27-29, Beijing China, 2018.

[29] X. Yang, et al., "An Edge Detection IP of Low-cost System-on-Chip for Autonomous Vehicles," *The 22nd International Conference on Artificial Intelligence (ICAI 2020)*, In Press, March 2020.

[30] A. Gajjar, et al., "An IoT-Edge-Server System with BLE Mesh Network, LBPH, and Deep Metric Learning," *The 22nd International Conference on Artificial Intelligence (ICAI 2020)*, In Press, March 2020.

[31] A. A. Heidari, H. Faris, I. Aljarah, and S. Mirjalili, “An efficient hybrid multilayer perceptron neural network with grasshopper optimization,” *Soft Computing*, Vol. 23, No. 17, PP. 7941–7958, 2019.

[32] V. Sze, Y. Chen, T. 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.

[33] K. Gurney, *An introduction to neural networks*. Boca Raton, Florida: CRC Press, 1997.

[34] X. Yang, et al., “A Vision of Fog Systems with Integrating FPGAs and BLE Mesh Network,” *Journal of Communications (JoC)*, Vol. 14, No. 3, PP. 210-215, March 2019.

[35] X. Yang and X. He, “Establishing a BLE Mesh Network using Fabricated CSR mesh Devices,” *The 2nd ACM/IEEE Symposium on Edge Computing (SEC 2017)*, No. 34, San Jose/Fremont, CA, US, 2017.

[36] A. Gajjar, Y. Zhang, and X. Yang, “A Smart Building System Integrated with An Edge Computing Algorithm and IoT Mesh Networks,” *The Second ACM/IEEE Symposium on Edge Computing (SEC 2017)*, Article No. 35, San Jose/Fremont, CA, US, 2017.

[37] Y. Hao, "A General Neural Network Hardware Architecture on FPGA", PP. 1-6, 2017. [Accessed 17 May 2020].

[38] X. Yang and W. Wen, “Design of A Pre-Scheduled Data Bus (DBUS) for Advanced Encryption Standard (AES) Encrypted System-on-Chips (SoCs),” *The 22nd Asia and South Pacific Design Automation Conference, (ASP-DAC 2017)*, PP. 506-511, Chiba, Japan, Jan. 2017.

[39] X. Yang, W. Wen, and M. Fan, “Improving AES Core Performance via An Advanced IBUS Protocol,” *ACM Journal on Emerging Technologies in Computing (JETC)*, Vol. 14, No. 1, PP. 61-63, Jan. 2018.

[40] X. Yang, S. Sha, I. Unwala, and J. Lu, “Towards Third-Part IP Integration: A Case Study of High-Throughput and Low-Cost Wrapper Design on A Novel IBUS Protocol,” *IET Computers & Digital Techniques (IET-CDT)*, Vol. 14, No. 6, PP. 353-362, Nov., 2020.

[41] Y. Zhang and X. Yang, "A Case Study on Approximate FPGA Design With an Open-Source Image Processing Platform," *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, Student Forum, PP.372-377, Miami, FL, US, 2019.

[42] X. Yang and S. Shi, “Exploiting Energy-Quality (E-Q) Tradeoffs on Approximate FPGA Designs of Scalable Sequential Circuits,” *Journal of Circuits, Systems and Computers (JCSC)*, Aug. 27, 2020.

[43] Y. Zhang, et al., “Exploring Slice-Energy Saving on A Video Processing FPGA Platform with Approximate Computing,” *International Conference on Algorithms, Computing and Systems (ICACS)*, PP.138-143, July 27-29, Beijing China, 2018.

[44] N. K. Manaswi, *Deep learning with applications using Python: chatbots and face, object, and speech recognition with TensorFlow and Keras*. Berkeley, California: Apress., 2018.

[45] Z. Ali, I. Hussain, M. Faisal, H. M. Nazir, T. Hussain, M. Y. Shad, A. M. Shoukry, S. H. Gani, "Forecasting Drought Using Multilayer Perceptron Artificial Neural Network Model", *Advances in Meteorology*, Vol. 2017, doi: 10.1155/2017/5681308.

[46] P. S. Geidarov, “Clearly defined architectures of neural networks and multilayer perceptron,” *Optical Memory and Neural Networks,* Vol. 26, No. 1, PP. 62–76, 2017.

[47] S. G. Ramasubramanian, R. Venkatesan, M. Sharad, K. Roy and A. Raghunathan, "SPINDLE: SPINtronic Deep Learning Engine for large-scale neuromorphic computing," *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, La Jolla, CA, 2014, pp. 15-20, doi: 10.1145/2627369.2627625.

[48] "IEEE Standard for Floating-Point Arithmetic," in *IEEE Std 754-2008,* Vol., No., PP.1-70, 29 Aug. 2008, doi: 10.1109/IEEESTD.2008.4610935.

[49] "IEEE Standard for Floating-Point Arithmetic," in *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, Vol., No., PP.1-84, 22 July 2019, doi: 10.1109/IEEESTD.2019.8766229.

[50] Kumar, B.V.V., and Basha, S.M. (2016). Design and Simulation of Single-Precision Inexact Floating-Point Adder/Subtractor. *I-manager's Journal on Electronics Engineering*, 6(4), 7-12, doi: 10.26634/jele.6.4.8087.

[51] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design MIPS Edition: The Hardware/Software Interface*, 5th ed. Oxford, England: Morgan Kaufmann Publisher, 2013.

[52] R. R. Taksande, M. N. Thakare, G. D. Korde, "Design of Floating Point Adder/Subtractor and Floating Point Multiplier for FFT Architecture Using VHDL", *International Journal of Advanced Research in Electrical, Electronics, and Instrumentation Engineering*, Vol. 6, No. 1., 2017.

[53] N. Grover and M. K. Soni, "Design of FPGA based 32-bit Floating Point Arithmetic Unit and verification of its VHDL code using MATLAB," *International Journal of Information Engineering and Electronic Business*, Vol. 6, No. 1, PP. 1–14, 2014.

[54] R. Omidi and S. Sharifzadeh, "Design of low power approximate floating-point adders," *International Journal of Circuit Theory Applications*, 2020.

[55] *2015 International Conference on Soft Computing Techniques and Implementations (ICSCTI)*. Faridabad, India: IEEE, 2015.

[56] H. Zhang, D. Chen, and S.-B. Ko, "High performance and energy efficient single-precision and double-precision merged floating-point adder on FPGA," *IET Computers and Digital Techniques*, Vol. 12, No. 1, PP. 20–29, 2018.

[57] R. Dhobale and S. Chaturvedi, "Implementation of 32 Bit Binary Floating Point Adder Using IEEE 754 Single Precision Format", *IOSR Journal of VLSI and Signal Processing (IOSR-JVSP)*, Vol. 5, No. 1., February 2015.

[58] A. Sharma and T. K. Rawat, "Truncated Wallace Based Single Precision Floating Point Multiplier," *2018 7th International Conference on Reliability, Infocom*

Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, India, 2018, PP. 407-411, doi: 10.1109/ICRITO.2018.8748843.

[59] V. R. Krishnan, A. S. Rajiv, and N. R. Deborah, "A comparative study on the performance of FPGA implementations of high-speed single-precision binary floating-point multipliers," *2019 International Conference on Smart Systems and Inventive Technology (ICSSIT)*, Tirunelveli, India, 2019, pp. 1041-1045, doi: 10.1109/ICSSIT46314.2019.8987800.

[60] S. Arish and R. K. Sharma, “Run-time-reconfigurable multi-precision floating-point matrix multiplier intellectual property core on FPGA,” *Circuits, Systems, and Signal Processing*, Vol. 36, No. 3, PP. 998–1026, 201.

[61] M. Al-Ashrafy, A. Salem and W. Anis, "An efficient implementation of floating point multiplier," *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*, Riyadh, 2011, pp. 1-5, doi: 10.1109/SIECPC.2011.5876905.

[62] *2018 International Conference on electrical, electronics, communication, computer, and optimization techniques (ICEECCOT).* Karnataka, India: IEEE, 2018.

[63] I. Westby, "FPGA Acceleration on Multilayer Perceptron (MLP) Neural Network for Handwritten Digit Recognition", Master Thesis, The University of Houston - Clear Lake, May 2020.

[64] I. Westby, X. Yang, H. Koc, and J. Lu, "Accelerating Digit Recognition with Neural Network," *The 22nd International Conference on Artificial Intelligence (ICAI 2020),* Under Review, March 2020.

[65] X. Yang and J. H. Andrian, "A High-Performance On-Chip Bus (MSBUS) Design and Verification," in *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 7, pp. 1350-1354, July 2015, doi: 10.1109/TVLSI.2014.2334351.

[66] X. Yang, N. Wu, and J. Andrian, “A Novel Bus Transfer Mode: Block Transfer and A Performance Evaluation Methodology,” *Elsevier, Integration, the VLSI Journal*, Vol. 52, Issue: C, PP. 23-33, January 2016.

[67] X. Yang, Y. Zhang, and L. Wu, "A Scalable Image/Video Processing Platform with Open Source Design and Verification Environment," *20th International Symposium on Quality Electronic Design (ISQED 2019)*, PP. 110-116, Santa Clara, CA, USA, 2019.

VITA:

APRIL REED

2020	M.S., Computer Engineering University of Houston Clear Lake (UHCL), Houston, Texas
2019	B.S., Computer Engineering University of Houston Clear Lake (UHCL), Houston, Texas
2014	M.S., Counseling University of Houston Clear Lake (UHCL), Houston, Texas
2010	B.S., Psychology University of Houston, Houston, Texas

WORK-IN-PROGESS:

April L. Reed and Xiaokun Yang. 2018. Area-Speed-Power Tradeoff: FPGA Design and Implementation on Handwritten Digit Recognition," Submitted, The 58th Design Automation Conference (DAC), 2021.