FLOATING-POINT COMPARATOR WITH RELU OPERATOR FOR MACHINE LEARNING ENHANCEMENT

By

LANDON RAY BURLESON

Bachelor of Science in Computer Engineering
Oklahoma State University
Stillwater, Oklahoma
2019

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2021

FLOATING-POINT COMPARATOR WITH RELU OPERATOR
FOR MACHINE LEARNING ENHANCEMENT

Thesis Approved:

Dr. James E. Stine, Jr.

Thesis Adviser

Dr. Bingzhe Li

Dr. Gary Yen

Name: LANDON RAY BURLESON

Date of Degree: MAY, 2021

Title of Study: FLOATING-POINT COMPARATOR WITH RELU OPERATOR FOR MACHINE LEARNING ENHANCEMENT

Major Field: ELECTRICAL ENGINEERING

Abstract: This article provides various comparator designs that provide comparisons to double, single, half, and bfloat floating-point values as well as provide comparison modes for 32 and 64 bit two's compliment integer encoded numbers. The variety of different modes described are assessable via select signal to the proposed comparators. This comparator also houses a Rectified Linear Unit (ReLU) function to leverage performance in a machine learning environment. Many forms of machine learning architectures, such as Deep Neural Network (DNN) and Convolutional Neural Network (CNN), utilize the ReLU algorithm for weight updates to their respective computational layer networks. Providing a hardware level solution to these weight updates within these networks would produce faster results for the networks respective outputs due to the speed and reliability of hardware solutions over the traditional based software solutions found in the industry today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

Machine learning has become a staple in modern computing and processing. neural networks such as Deep Neural Network (DNN) or Convolution Neural Network (CNN) has become common among researchers specializing in image recognition and big data sciences. The performance of these networks are crucial as computation intensity and data grows in size and scope throughout a multitude of different industries. Both DNN and CNN networks operate using a network of layers that provide the means in which all calculations are carried out. These networks dynamically change the weights of these computational layers in order to provide an accurate prediction for the result of a respective input. The process in which layer weight updates are done by using a module called Rectified Linear Unit (ReLU). Currently, the ReLU operation is utilized through software implementation for neural network designs but this work proposes a hardware based approach to leverage performance in the weight update events of a neural network. This is made possible by utilizing a floating-point comparator with built in ReLU functionality as discussed within this paper. Past comparator implementations generally lack machine learning functionality and lacked details of the operation of the comparator design [4], [5], [6], [7]. In recent times however, comparators with machine learning functionality have begun to become more common to develop due to the demand for faster and more efficient neural network implementations [8]. However, the recent advancement of comparators with machine learning functionality generally lack details and versatility within their respective designs. The following quote from Hennessy depicts a large demand for

high performance neural network designs: "In addition to these large players, dozens of startups are pursuing their own proposals." [9, p. 60] "To meet growing demand, architects are interconnecting hundreds to thousands of such chips to form neural-network supercomputers." [9, p. 60] "This avalanche of DNN architectures makes for interesting times in computer architecture." [9, p. 60] "It is difficult to predict in 2019 which (or even if any) of these many directions will win, but the marketplace will surely settle the competition just as it settled the architectural debates of the past." [9, p. 60]. Taking this quote into consideration, these massive neural networks that are being utilized in industry are used strictly for computational performance. Any loss in performance over software based ReLU operations or poorly optimized floating-point comparator designs jeopardizes performance severely due to the sheer amount of weight updates found within these networks. This work provides a versatile floating-point comparator design with ReLU functionality to provide performance uplift and optimization to future and current neural network designs.

Comparison operations for both floating-point and integer encoded values were derived directly from the IEEE 754 standards [1]. The ReLU operation used with the machine learning variants of the proposed floating-point comparator designs is discussed in Section II along with the comparison functionality. The various approaches to the floating-point comparator designs and the inner workings of the proposed work are also described in detail within the aforementioned section. This work also provides various multi-function floating-point comparator designs with an emphasis on machine learning operations to leverage performance within the layer weight update events of a neural network [2]. In industry, it is common to use a subtractor as a comparator. Comparison outcomes using a subtractor are determined from the sign bit for both floating-point or integer operands. This work provides a way to directly compare between two floating-point or two's complement operands with the possible outcomes being greater than, less than, equal, and unordered respectively. Further-

more, a variety of floating-point encoding options are included in the various proposed designs such as double, single, half, and bfloat precision. This is to provide versatility inside an Floating-Point Unit (FPU) for any neural network encoding requirements. For this work, using a simple 2-bit comparison between Op1 and Op2 outputs a 0x1, 0x2, 0x3, or 0x4 to correspond to the floating-point condition codes (FCC) found in Table 2.0.2. This behavior is directly derived from comparison operations detailed in the IEEE 754 standard [1].

Section III discusses the implementation(s) of the proposed designs and the other work used to quantify performance differences. Details on the floating-point, two's complement, and machine learning functionality is discussed thoroughly for each independent design. The main focus of this work is the fpcomp_opt_ReLU design for floating-point and two's complement comparison operations in addition to the ReLU function for machine learning operations. A flowchart is provided for a visual demonstration of the operation and tie-ins to the various blocks that make up the aforementioned design(s). The design flow format used to iterate, test, and synthesis all designs is described as well.

All testing was conducted using the ModelSim simulation tool for all hardware descriptive language (HDL) implementations and the Synopsys DesignWare synthesis tool was used for all HDL synthesis trials. To ensure proper functionality, the Testfloat [10] floating-point test vector generation tool was used in tandem with a self-checking test bench for all ModelSim tests. Further details for testing is discussed in Section III.

Results were gathered based upon operation and synthesis results. All designed floating-point units are compared against the DesignWare standard floating-point comparator as well as a previous tree-based subtractor architecture [3]. The varying designs of this papers units is discussed further in Section IV.

CHAPTER II

BACKGROUND

Machine learning has become a mainstay in computer computation and arithmetic. Some workflows that are common within this industry include image recognition, Artificial Intelligence, and voice recognition. These networks are made up of input, computational, and output layers respectively. By placing a set of data or a specified input through these layers, the outputs produce an estimation of what the network expects is the correct result based on the weights of the computational layers and the pre-existing conditions determined from the learning phase of the initial neural network design. Using neural networks, work flows such as image recognition become relatively efficient and accurate for the desired results. However, in current implementations of these networks, a software based approach is used to update the weights of their layer map by using the ReLU operation found in Figure 2.0.3. As implementations of the various neural network types become more intensive, the performance of the overall network decreases drastically due to the abstraction a software based ReLU function imposes.

Using the IEEE 754 standards [1] [11] [12] [13] for the floating-point number encodings and the arithmetic, this paper provides a multi-function comparator that offers floating-point, and two's compliment comparison modes using a 2-bit magnitude compare approach. Further discussion of the implementation and inner workings of these modules are explained in later sections of this paper.

Combining the floating-point and two's complement comparison functionality with the ReLU operator used within various neural networks designs, provides an effective

hardware-based solution to the growing need for faster and efficient neural network designs. The following sections will cover the IEEE 754 comparison arithmetic, two's complement, and the ReLU operation.

2.0.1 Two's Complement and Comparison Arithmetic

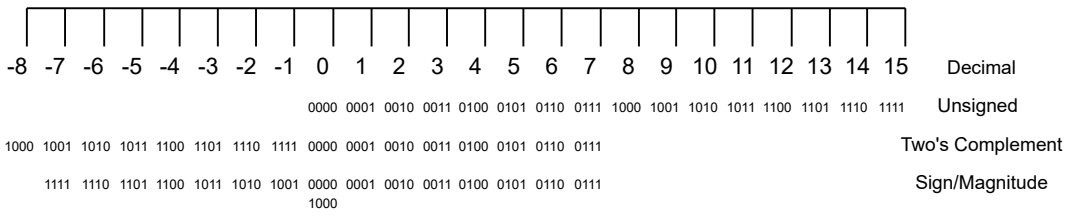

Figure 2.1: Two's Complement Number Encoding (4-bit)

Two's complement number encoding provides a format for use to represent both negative and positive integers. The bit on the furthest left of the encoding is considered the most-significant-bit (MSB) and the furthest right-most bit is the least-significant-bit (LSB). Similar to floating-point number encoding, the MSB bit is considered the sign bit of a given number. If set to high or '1', the number encoded in two's complement or floating-point is regarded as a negative value. If set to low or '0', the number encoded is considered a positive value. All two's complement operations conducted using any of the proposed designs yield an output of greater-than, less-than, or equal-to. These outcomes are directly correlated with the Floating-point Condition Codes (FCC) outputs found in later sections discussing floating-point comparison and the accompanying arithmetic. Unlike the floating-point operations discussed later, a two's complement number cannot be an invalid number. This is due to a lack of an exponent within the number encoding. To add to this, this work exclusively uses a fixed-point (constant radix) for all two's complement encoding. This means that all two's complement numbers are integers and therefore, have no fractional part of it's respective encoding. Since the two's complement operands are fixed-point, any com-

bination of a binary value will yield a valid value for comparison operations. There is no possibility of a not-a-number outcome in contrast to the floating-point encoding counterpart.

Possible combinations of operands (Op1 and Op2) for two's complement include: Op1 and Op2 are positive, Op1 is positive and Op2 is negative, Op1 is negative and Op2 is positive, or Op1 and Op2 are negative values. With this in mind, the methodology used to determine whether Op1 is less-than Op2 is determined from the magnitude of each individual number as follows: Op1 is negative and Op2 is positive, Op1 and Op2 are positive and the magnitude of Op1 is less-than the magnitude of Op2, Op1 and Op2 are negative and the magnitude of Op1 is greater than the magnitude of Op2. The equal-to flag is determined whether both operands are equivalent in value. The greater-than flag can be determined from the equal-to and the less-than flags respectively through boolean logic. The equations 2.1 and 2.2 acquire the greater-than and less-than flags respectively for a two's complement comparison.

Due to the way floating-point numbers are encoded, the magnitude comparison operation is valid for both floating-point numbers and two's complement numbers. The reason for this is due to the exponent being ahead of the fraction of a floating-point number. The 'exponent' of a two's complement number will always be positive in this circumstance and therefore will produce appropriate results when compared using the proposed floating-point comparator(s). For a two's complement comparison example, as shown in Figure 2.0.1, Op1 is set as negative -78 and Op2 is set as positive 42 yielding a 01 as the output which corresponds to the less-than flag in the FCCs described in the IEEE 754 floating-point section below. The process in which this outcome is produced is straightforward. The least-significant two bits for $Op1_{[1:0]} = 10$ and $Op2_{[1:0]} = 10$ are compared and yield an '00' output for equivalence. Moving to the compare module located to the left, $Op1_{[3:2]} = 00$ and $Op2_{[3:2]} = 10$ are compared against one another and yield a '01' since Op1 is less than Op2. Following the same

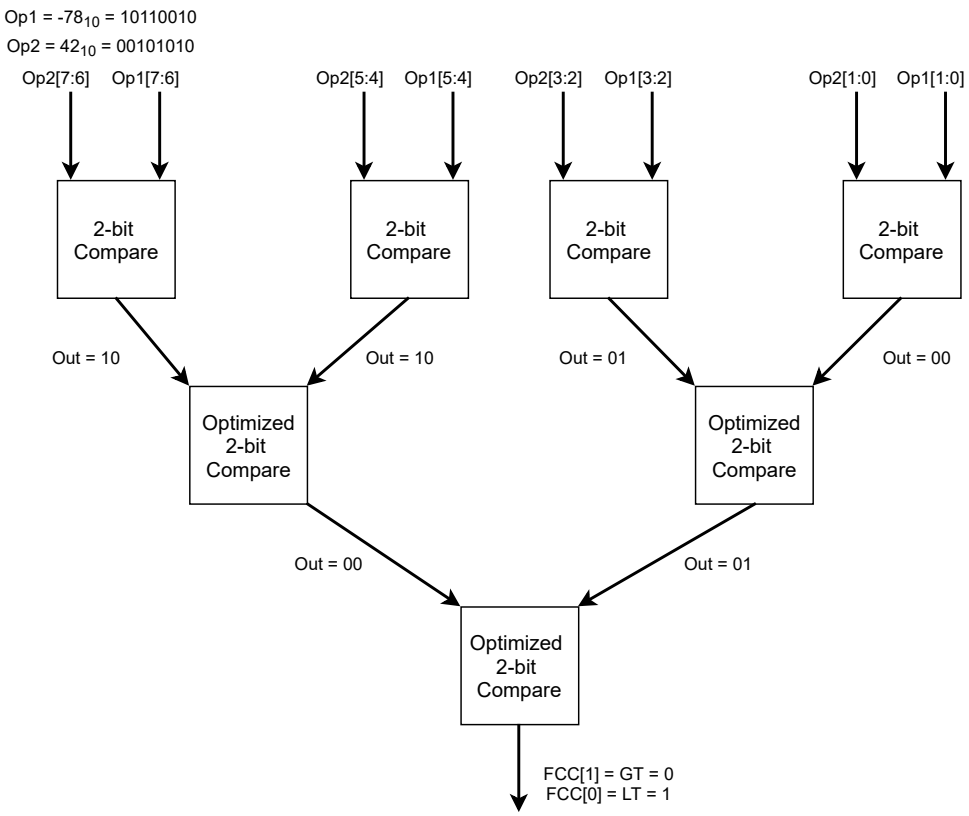

Figure 2.2: Two's Complement Comparison Example

pattern, $Op1_{[5:4]} = 00$ and $Op2_{[5:4]} = 10$ are compared and results in a FCC of '10' for greater-than outcome. The MSBs of each operand is compared $Op1_{[7:6]} = 10$ and $Op2_{[7:6]} = 00$ which gives the FCC '10' for a greater than outcome. The next stage of the comparator utilizes the FCC outputs from the previous stage for the basis of the next comparison operations. Starting with the least-significant bit side of the comparator, the FCC from the [1:0] bit comparison is compared against the FCC from the [3:2] comparison. This operation yields a '01' less-than outcome. The FCCs between bits [7:6] and [5:4] are compared and the output for this operation yields a '00' for equivalence. Finally, comparing the FCCs for bits [7:4] and [3:0] results in a final output of '01' for the comparison between Op1 and Op2 two's complement operations. The final stage of the comparison module uses the left-most comparison results as Op1 due to the MSBs being more significant in terms of finding the comparison results

for the inputted operands. The magnitude comparison modules shown in the figure utilize the aforementioned equations for the less-than and greater-than outcomes. The modules denoted as 'Optimized' use equations 2.3 and 2.4 respectively. These optimize the comparison functionality and is discussed further in the following section.

2.0.2 IEEE 754 Floating-Point and Comparison Arithmetic

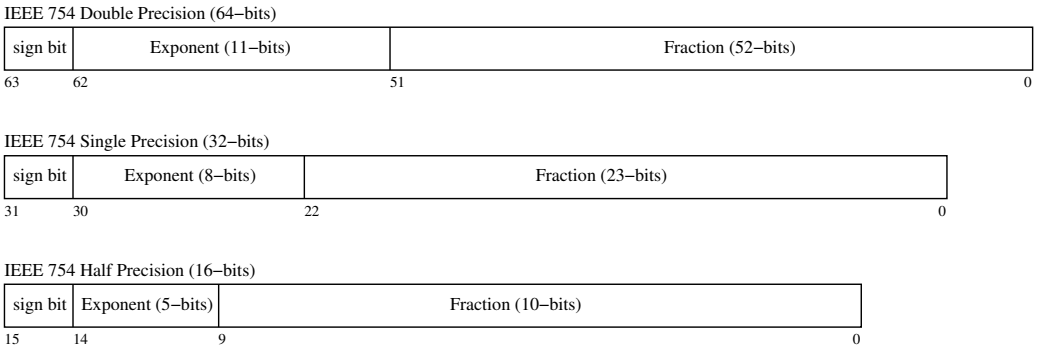

Figure 2.3: IEEE 754 Floating-Point Standards

The proposed work provides comparison operation for the following IEEE 754 floating-point types: double, single, and half precision floating-point numbers. Each of these encodings correspond to a 64, 32, and 16 bit floating point numbers respectively. These encodings all have various sizes for the exponent and fraction and can be viewed in Figure 2.0.2. A comparison operation presented in this work is detailed as a two bit magnitude comparison between Op1 and Op2. These two operands are compared using the boolean logic found in Equations 2.1 and 2.2. This operation will

FCC[1:0]	Description
00	A = B
01	A < B
10	A > B
11	Unordered

Table 2.1: Floating-Point Condition Codes and Descriptions

output the less-than (LT) and the greater-than (GT) flags for use in future commands of a processor. Once these flags are found, they are converted into the FCCs which is defined as the following: LT is mapped to FCC[0] and GT is mapped to FCC[1]. If FCC[1] = 0 and FCC[0] = 0, the comparison is a result of Op1 and Op2 being equivalent (EQ).

The final FCC combination is defined as FCC[1] = 1 and FCC[0] = 1 which corresponds to the unordered (UO) variant of the FCC codes. The unordered distinction of the FCC is determined from the exponent value found in Op1 or Op2. If the observed exponent of Op1 or Op2 is all 1's or high's, the number is considered NaN or infinite. As mentioned before, the FCC for this scenario is 0x3. Within this work, a set of select bits are used to specify the desired number encoding and accurately determine the validity of an input operand. The proposed design aspects of these modules along with these additional features are discussed further in Section III.

An optimized magnitude compare was also utilized to provide the most efficient performance possible with the proposed work within this paper [2]. The optimized GT and LT Equations are shown in Equations 2.3 and 2.4 respectively. The original equations 2.1 and 2.2 were optimized by introducing 'don't cares' into the equation and in turn optimize the 2-bit magnitude compare operation found within these designs. Some of floating-point comparator designs (denoted with '_opt_') proposed use the optimized variant of the comparison operation. The feature set differences and inner workings of each independently proposed design is discussed further in chapter III

$$GT = Op1[1] \cdot \overline{Op2[1]} + Op1[1] \cdot Op1[0] \cdot \overline{Op2[0]} + Op1[0] \cdot \overline{Op2[1]} \cdot \overline{Op2[0]} \quad (2.1)$$

$$LT = \overline{Op1[1]} \cdot Op2[1] + \overline{Op1[1]} \cdot \overline{Op1[0]} \cdot Op2[0] + \overline{Op1[0]} \cdot Op2[1] \cdot Op2[0] \quad (2.2)$$

$$GT = GT[1] + GT[0] \cdot \overline{LT[1]} \tag{2.3}$$

$$LT = LT[1] + \overline{GT[1]} \cdot LT[0] \tag{2.4}$$

In accordance to the IEEE 754 floating-point standard, there are two ways to accomplish a comparison operation result between two floating-point operands. The first is to return the floating-point condition codes with the possible outputs shown in Table 2.0.2. The second way to accomplish a comparison result is by specifying a desired outcome and designating the output of the comparator of either true or false for this event. For example, if the desired outcome is LT and if the comparison between two operands results in a LT outcome, the output from the comparator will yield a 'true' for this event. However, if the operation results in a outcome other than the expected result, the comparator will yield a 'false'. The other possible output is to yield an invalid flag if either operand is not a number or unordered. For this work, all comparator designs output the appropriate FCC codes for a given operation in accordance to the possible FCC code outputs found in Table 2.0.2. See Figure 2.0.2 for a visual representation of the two types of 754 comparators.

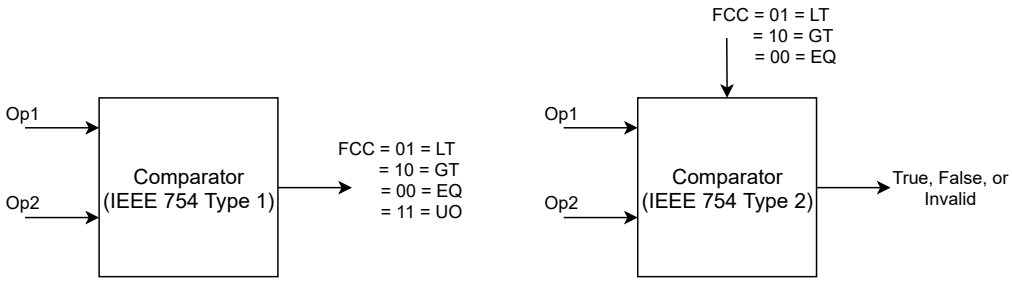

Figure 2.4: IEEE 754 Comparator Types [1]

As shown in Figure 2.0.2, the '2-bit Compare' modules use equations 2.1 and 2.2 for the GT and LT flags respectively. Similar to the two's complement example above, the 'Optimized 2-bit Compare' modules utilize the 2.3 and 2.4 respectively.

As comparison operations are conducted on each 2-bit operand, the GT and LT flags found within each comparison result is then used to find the next set of FCCs. This is done until all comparisons are exhausted and the final FCC is formed. As shown, Op1[7:0] is set to 01010111 and Op2[7:0] is set to 01011111 respectively. The final result of the comparison operation between the two binary values yield a '01' output for the FCC code. The following describes the process in which this finalized result is produced for the two operands. Following a similar process as the previous example, the least-significant two bits for $Op1_{[1:0]} = 11$ and $Op2_{[1:0]} = 11$ are compared and yield an '00' output for equivalence. Moving to the compare module located to the left, $Op1_{[3:2]} = 01$ and $Op2_{[3:2]} = 11$ are compared against one another and yield a '01' since Op1 is less than Op2 in magnitude. Continuing the pattern, $Op1_{[5:4]} = 01$ and $Op2_{[5:4]} = 01$ are compared and results in a FCC of '00' for equivalence. The MSBs of each operand is compared $Op1_{[7:6]} = 01$ and $Op2_{[7:6]} = 01$ which gives the FCC '00' for the equal-to outcome. The next stage of the comparator uses the previous outcomes to produce the FCC for the group of bits observed. Starting with the right-most side of the comparator, the FCC from the [1:0] bit comparison is compared against the FCC from the [3:2] comparison. This operation yields a '01' less-than outcome. The FCCs between bits [7:6] and [5:4] are compared and the output for this operation yields a '00' for equivalence. Finally, comparing the FCCs for the bit groups [7:4] and [3:0], this results in a final output of '01' for the comparison between Op1 and Op2. This code corresponds to the LT result and therefore, the comparison is valid for the example shown. Notice that the comparator does not make any distinctions for any specific section of an encoding. The edge cases for floating-point and the accompanying exponent and fraction sections associated with this encoding is handled within the 'exception handling' unit proposed within this work. These edge cases and further details of the inner workings of the magnitude comparator are discussed in chapter III.

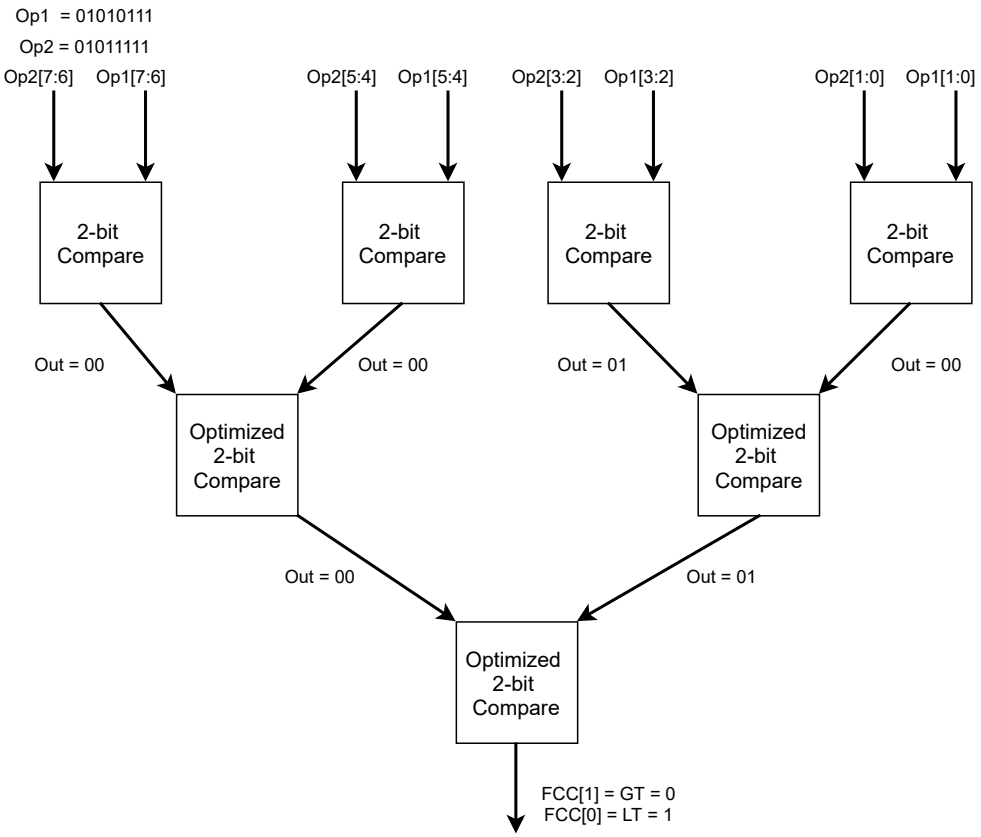

Figure 2.5: Example Comparison Operation in Accordance with IEEE 754[2]

2.0.3 ReLU

The Rectified Linear Unit (ReLU) operation is used heavily in machine learning environments to update the weights of the computational layers within a neural network. ReLU functionality can be added to any type of neural network. However, the most common neural networks that utilize the ReLU function are the Convolutional Neural Network (CNN) and multilayer perceptron architectures [14], [15], [16], [17], and [18]. The CNN variant consists of input, computational, convolution, and output layers. These architectures are often used for image processing and object recognition. The multilayer perceptron architecture uses nodes that are synonymous with human neurons to use input vectors as a source of computation. These input vectors are shuffled into neurons known as perceptrons which calculate the weights which correspond to

the likelihood that the given vector belongs in a desired class or set of data [19]. Once the weights are calculated using the perceptrons, the weights must be updated for a given computation layer. This need for a an operation is the reason for the ReLU operation being so common within the architectures described. The importance of this function within a respective neural network is immeasurably significant. The ReLU function dictates an appropriate moment in which to update the weight of a computational layer based on a previous result. This process ultimately maintains these weights to accurately predict a result of a respective input for a given neural network.

An example of a computation would be to use a CNN to predict if a input image has a desired object. Not all pictures are the same in terms of contrast, saturation, lighting, etc. However, the particular shape and characteristics of an object remain relatively constant in a given photo set. The layer weights found within the initial neural network design will be set to a default value to attempt a successful prediction of the object. Feeding this neural network pictures (with and without the object) will build up the layer weights to an appropriate value set to more accurately predict if the object is present in an input image. This process requires the ReLU function to update all layers within a timely manner to process data faster. As stated in the introduction, the neural networks of today are increasing in size and scope dramatically. As size increases, so does the number of computational layers and furthermore the number of weight updates required to maintain accurate predictions for a neural network.

Currently, the most common way to implement the ReLU operator is through software. A software approach is several layers abstracted from the hardware. This approach increases delay significantly by adding an unnecessary amount of instructions for a single operation. To visualize this, an example ReLU function code is shown below with the associated x86 assembly code (Listing II.1, II.2). The assembly code output from the C code results in approximately 10 instructions for a single

ReLU operation. Using the software variant of the ReLU operator introduces a execution delay over the hardware based approach proposed in this work. Using this works proposed hardware-based floating-point comparator with ReLU functionality, will exponentially increase the speed in which these calculations are conducted and allow a significant improvement of throughput for the supercomputer scale neural networks of today [9]. By implementing this function within hardware, this process is reduced to a single instruction for execution. As an example, assume a clock cycle time is 1 ps within a single-cycle architecture. A software based ReLU function would take 10 ps per ReLU operation. Using the same architecture and speed, the proposed work would execute the same operation in 1 ps. This speedup compounded with the shear amount of ReLU operations required in the supercomputer scale neural networks, would produce a significant performance boost over the conventional implementation.

Commonly, a subtractor is used to determine a comparison of Op1 and Op2 within a neural network. Once completed, the weights are updated in accordance accordance with Figure 2.0.3. In this work, Op1 is used to compare against '0' to determine the ReLU output for the layer weight in question. Once this operation is complete for every iteration of the learning process, the original weight value is cleared from the layer in question and is then OR'ed with the ReLU value to complete the update process. In summary, the ReLU operator provides a way to maintain a positive Op1 output for a given weight update event.

$$ReLU = \begin{cases} \text{Op1}, & \text{if Op1} > 0 \\ 0, & \text{if Op1} \leq 0 \end{cases} \tag{2.5}$$

Figure 2.6: ReLU Operation

```c
#include <stdio.h>
int main() {
  double a = 3.39030830803;
  double result;
  if (a > 0)
   result = a;
  else
   result = 0.0;
  printf("The_result_is_%lg\n", result);
}
```

Listing II.1: ReLU program in C

```
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        subq    $16, %rsp
        movsd   .LC0(%rip), %xmm0
        movsd   %xmm0, -8(%rbp)
        movsd   -8(%rbp), %xmm0
        pxor    %xmm1, %xmm1
        comisd  %xmm1, %xmm0
        jbe     .L7
        movsd   -8(%rbp), %xmm0
        movsd   %xmm0, -16(%rbp)
        jmp     .L4
.L7:
        pxor    %xmm0, %xmm0
        movsd   %xmm0, -16(%rbp)
```

Listing II.2: Assembly in x86 for the ReLU Function

CHAPTER III

IMPLEMENTATION AND TESTING

This section covers all implementations of the various proposed floating-point comparator (fpcomp) designs and the other works. This includes detailed inner-workings of all designs and the various differences incorporated into each unique unit. For better understanding, multiple diagrams are used to describe the various operations and feature sets.

3.0.1 Two's Complement Implementation

Similar to the fpcomp inner workings, the modules that have two's complement functionality (exception to the fpcomp_only and fpcomp_opt_only) use the 2-bit magnitude compare with or without the optimization to generate the LT and EQ flags respectively. These flags are then inserted into the exception handling block within all proposed designs to determine whether the GT flag is set or not. The UO verification is unnecessary for two's complement due to the respective encoding. It is not possible to have an infinite or an invalid number encoded with a two's complement format. Therefore, the only operation done on the two's complement comparison results within the exception handling block is the generation of the finalized FCC for a given comparison operation. The input Sel signal used to determine a floating-point encoding versus a two's complement number is Sel[2]. If Sel[2] is set to '1' for the input signal going into the comparator, the comparator treats both Op1 and Op2 as two's complement numbers. The remaining Sel[1] and Sel[0] signals determine the number of bits each number encompasses. A 16-bit value is represented as $Sel[2] \cdot \overline{Sel[1]} \cdot Sel[0]$,

a 32-bit value is represented as $Sel[2] \cdot Sel[1] \cdot Sel[0]$, and a 64-bit value is represented as $Sel[2] \cdot Sel[1] \cdot \overline{Sel[0]}$ respectively. These Sel[2:0] signal values are consistent across all proposed designs that include two's complement functionality.

3.0.2 Floating-Point Comparator and ReLU Implementation

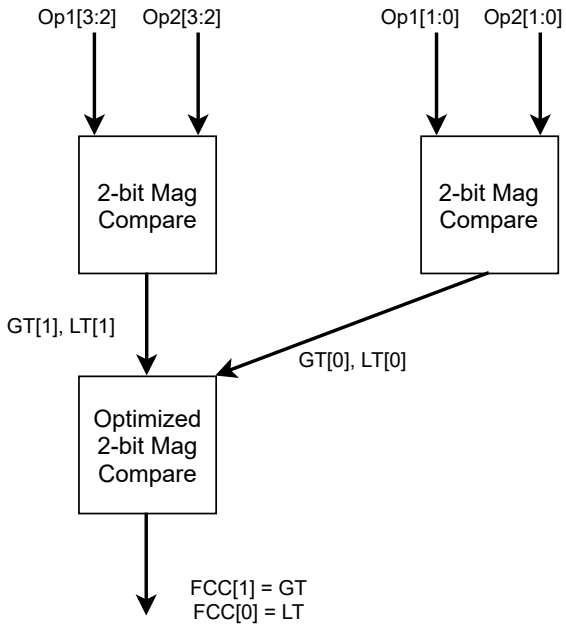

Figure 3.1: 4-bit Magnitude Comparator Utilizing the Optimized Modules

The implementation of the proposed design of this paper is shown in Figure 3.0.2. The operation of the fpcomp_opt_ReLU module goes as follows: sign extend operand 1 (Op1) and operand 2 (Op2) based on select signals, compare Op1 against Op2, generate FCC through the exception handling block, output the appropriate FCC, and generate the z0 signal from the ReLU operation.

For the sign extension module, both operands and the 3-bit Sel signal are inputted into this block. Based on the Sel signal, the sign extension block extends the sign of each operand inputted by 32 bits or 48 bits. This distinction is made by the size of each operand to result in a 64 bit output into the comparison block. The sign extension is done to ensure that the comparison operation is functional for all input

types into the proposed model. As an example operation, if both operands are 32-bit floating-point numbers, the $Sel[2] = 0$, $Sel[1] = 0$, and $Sel[0] = 1$. The operation of the sign extension function relies on $Sel[0] \cdot Sel[1] \cdot Sel[2]$ and $Sel[0] \cdot \overline{Sel[1]} \cdot Sel[2]$ to produce the Ext32 and Ext16 signals. If Ext32 is set to high, then the operands are sign extended by 32 additional bits. If Ext16 is set to high, the the operands are sign extended by 48 additional bits.

From the sign extension module, Op1[63:0] and Op2[63:0] are inputted into the comparison module for analysis. Within the comparison module, the 2-bit magnitude comparison sub-modules are used to calculate the appropriate less-than, greater-than, equal-to, or unordered result in correspondence with the IEEE 754 FCC values. The magnitude comparison operation is characterized in Equations 2.2 and 2.1. For the optimized variants of the comparator designs, these versions of the magnitude compare modules utilize a optimized form of the magnitude compare functions discussed in previous chapter. This enhancement allows the comparison operation to execute with fewer clock cycles than the standard comparison operation and use fewer logic gates for power and area savings [2].

Sel[2:0]	Description
000	Double Precision Numbers
001	Single Precision Numbers
010	Half Precision Numbers
101	16-bit Integers
110	64-bit Integers
111	32-bit Integers

Table 3.1: Comparator Options via 3-bit Select Signal

The magnitude comparison operation is visualized in figure 3.0.2 using a simplified 4-bit comparator. As mentioned before, the '2-bit Mag Compare' blocks are represen-

tative of the boolean equations 2.1 and 2.2 for the GT and LT flags respectively. Once the initial magnitude compare on the 2-bit input operands is complete, the generated GT and LT flags are used to determine the final FCC value inside of the optimized version of the aforementioned equations. The optimized variants are defined as 2.3 and 2.4 for the final GT and LT flags respectively. This portion of the operation substitutes the use of Op1 and Op2 in favor of GT[1:0] and LT[1:0] to obtain the expected and accurate results. If a design does not use the optimization, the optimization block shown in Figure 3.0.2 is replaced with a standard 2-bit magnitude comparator for all magnitude comparison operations. The MSB of the respective operands entering the comparison module are flipped to ensure magnitude comparison operation is conducted correctly for both two's complement and floating-point values. To validate this notion, suppose Op1 is negative and Op2 is positive. If a magnitude comparison is conducted on these operands without flipping the sign bit, the result would reveal that Op1 is greater than Op2 due to the magnitude comparator interpreting the signed value as a unsigned one and generating a GT flag. Flipping the sign bits in this case and other cases would ensure proper initial comparison results via FCC. This method produces appropriate FCC values for two's complement operands but doesn't cover all edge cases for floating-point encodings. The FCC generated from the magnitude comparator module is then inputted into the exception handling block to handle all edge cases as well as handle both floating-point and integer encoded numbers. The initial LT and EQ values are passed along with each operand to the exception handling unit of the design to finalize the results of the operation.

The exception handling block is used to generate the FCC per the IEEE 754 standard based upon the LT and EQ flags set from the comparison operation. This block checks for an unordered FCC output based on the floating-point operands before finalizing the FCC value. This distinction is done by utilizing the Sel signal coming into the exception handling block to determine whether the operands are floating-

point encoded or not. The Sel signal also determines the precision of the floating-point operands. These operations must be conducted to determine if the inputted floating point values should be checked for a not-a-number (unordered) encoding. This scenario is determined by observing the exponent values within the exponent segment of the floating-point encoding. An exponent shown to be all 1's is considered not-a-number for a floating-point encoding. A signalling unordered output is determined by the MSB of the fraction section of the floating-point number. If this bit is set to 0 and the exponent of the number is set to all 1's, the unordered output is signalling and the invalid flag is set to high. If not signalling, the UO flag is set and the final FCC is returned as '11' from the exception handling block. Once a non-unordered result is determined, the LT and EQ flags are found due to the UO flag being set to 0. If both input operands are found to be valid (not unordered), the finalized FCC flags are found with the following logic expressions. The EQ flag is set by using the following logic equation: $EQ = EQ_{mag} \mid (Op1zero \cdot Op2zero \cdot fp) \cdot \overline{UO}$. The EQ flag is set if the EQ_{mag} flag is set from the comparator module or if the floating-point operands are equal to zero and if the UO flag is not set. The LT flag is set using the following logic equation: $LT = ((LT_{mag} \cdot \overline{fp}) \mid (\overline{LT_{mag}} \cdot Op1[63] \cdot Op2[63] \cdot fp) \mid (LT_{mag} \cdot \overline{(Op1[63] \cdot Op2[63])} \cdot fp)) \cdot \overline{EQ} \cdot \overline{UO}$. The LT flag is set if LT_{mag} is set and the operation is not a floating-point comparison or if the LT_{mag} is not set and if both floating-point operands are negative and the magnitude of Op1 is greater than Op2 or if LT_{mag} is set and if both floating-point operands are positive and the magnitude of Op1 is less than Op2 and if EQ and UO are not set. The GT flag is determined by the following logic: $GT = \overline{(LT \mid EQ \mid UO)}$. The GT flag is directly calculated by the other possible flags because it cannot be set while the other flags are set. The final FCC bits are calculated using FCC[0] = LT + UO and FCC[1] = GT + UO respectively.

For the fpcomp_opt_ReLU module, upon generating the FCC values, the signals

are passed into the ReLU module along with Op1[63:0] to generate the z0 signal used as the output of the ReLU operation. This operation is done using a mux21 module and a signal defined by $\sim$ FCC[1] to select the Op1[63:0] operator or 64'h0 for the output z0. The FCC[1] is considered the GT bit in accordance to the FCC codes.

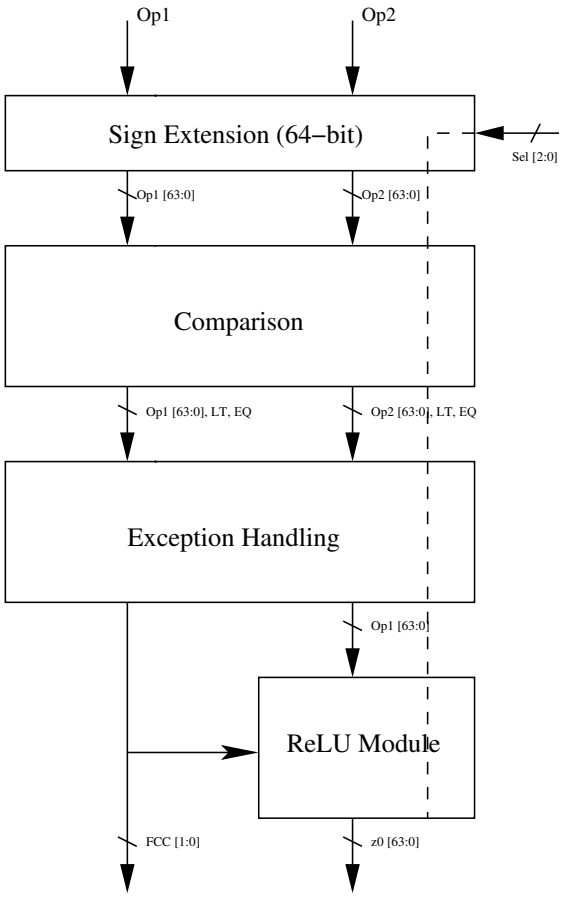

Figure 3.2: Flowchart for the Implementation of fpcomp_opt_ReLU Module

As shown in Table 3.0.2, the various fpcomp designs use a diverse feature set to accomplish similar goals to the fpcomp_opt_ReLU module. The various designs generally, all have basic floating-point comparison operations. However, these designs vary in terms of two's compliment functionality, the use of the magnitude comparison optimization, and machine learning functionality. The following designs have machine learning functionality: fpcomp_ReLU, fpcomp_ml, fpcomp_opt_ReLU, and fpcomp_maxmin. The differences

among these designs are directly related to the method used to implement the machine learning operations. The fpcomp_ReLU and fpcomp_opt_ReLU are functionally the same with exception to the ladder using the optimization for the floating-point comparison operations and the zctrl input that is used within the fpcomp_ReLU design. The zctrl signal inside of the fpcomp_ReLU module is used to switch between a comparison operation and a ReLU operation. This is done by setting Op2 to either 64'h0 or its original input value. The main advantage of the fpcomp_opt_ReLU design is that both the comparison output of the operands and the RelU output is always outputted without the need of an additional control signal. The fpcomp_maxmin and fpcomp_ml designs are based upon the DesignWare variation of the comparator used for machine learning operations [20]. Starting with the fpcomp_maxmin design, this design utilizes both 'max' and 'min' functions. These functions output the largest is smallest value in respect to the input operands for the maximum and minimum variables respectively. The fpcomp_ml design uses the 'max' function similarly to how the ReLU function works. It outputs either Op1 or 64'h0 depending on whether Op1 is greater than 0 or not.

The remaining fpcomp designs do not have the machine learning functionality. These designs include: fpcomp, fpcomp_comb, fpcomp_only, and fpcomp_opt_only. The standard fpcomp design is considered the baseline and it has both floating-point and two's compliment comparison operations. It however, doesn't utilize the optimization of the 2-bit magnitude compare function. The fpcomp_comb also doesn't utilize the optimization and it lacks the bfloat comparison functionality. Both the fpcomp_only and fpcomp_opt_only have only floating-point comparison operations. All designs designated with '_opt' supports the 2-bit magnitude optimization described in the previous chapter. See Table 3.0.2 for a visual of differing feature sets between all proposed designs.

Module	DP	SP	HP	Bfloat	16-bit 2's comp	32-bit 2's comp	64-bit 2's comp	max/min	ReLU	Optimization
fpcomp	X	X	X	X	X	X	X			
fpcomp_only	X	X	X	X						
fpcomp_opt_only	X	X	X	X						X
fpcomp_comb	X	X	X		X	X	X			
fpcomp_ml	X	X	X	X	X	X	X	X		
fpcomp_maxmin	X	X	X	X	X	X	X	X		
fpcomp_ReLU	X	X	X		X	X	X		X	
fpcomp_opt_ReLU	X	X	X	X	X	X	X		X	X
tree_subtractor_architecture [3]	X	X	X	X	X	X	X			
DW_fp_cmp [20]	X	X	X	X					X	

Table 3.2: Feature Sets for Proposed and Previous Designs

3.0.3 Tree-Based Subtractor Architecture and DesignWare Floating-Point Comparators

The proposed designs of this work described below are directly compared against the tree-based subtractor and the DW comparators respectively.

The tree-based subtractor architecture uses a magnitude compare similar to the proposed work with exception to the generate and propagate nodes that are used in place of the proposed 2-bit magnitude modules. These nodes are analogous to similarly designed tree-based adders. The subtractor based comparator is the most common comparator found in industry today and it uses the subtraction operation found within the (Floating-Point Unit) FPU or (Arithmetic Logic Unit) ALU to take the difference between two operands and ultimately check the sign bit of the output for the comparison result. If the sign bit is set, Op1 is less-than Op2, if the sign bit is not set, Op1 is greater-than Op2, and if the result of the operator is all zeros, Op1 is equal to Op2. For ease of use and proper operation, the tree-based subtractor design uses 1's complement for the comparison process [3]. Using a similar operation to what is found in the flowchart for fpcomp_opt_ReLU (3.0.2, the tree-based subtractor design was adapted to replace the 'Comparison' portion of the chart with the subtractor comparator. This was done to ensure similar operation between the subtractor comparator and the proposed designs 3.0.2 for a fair contrast between both architectures.

However, the ReLU module was left out of the subtractor architecture and was only tested against proposed designs that lacked this functionality as well. The tree-based subtractor architecture is visually represented in Figure 3.0.3. This figure shows both the black and the 'OR' nodes used to create the generate and propagate values to ultimately generate the GT and LT flags through a subtraction based process. The required boolean equations are also provided within the figure.

The DesignWare comparator provides a baseline floating-point comparator that is completely support all floating-point types targeted in this work (double precision, single precision, half precision, and bfloat). This particular design, however, does not include two's complement comparison operations. A direct comparison of feature sets between all proposed designs and other works are found in Table 3.0.2.

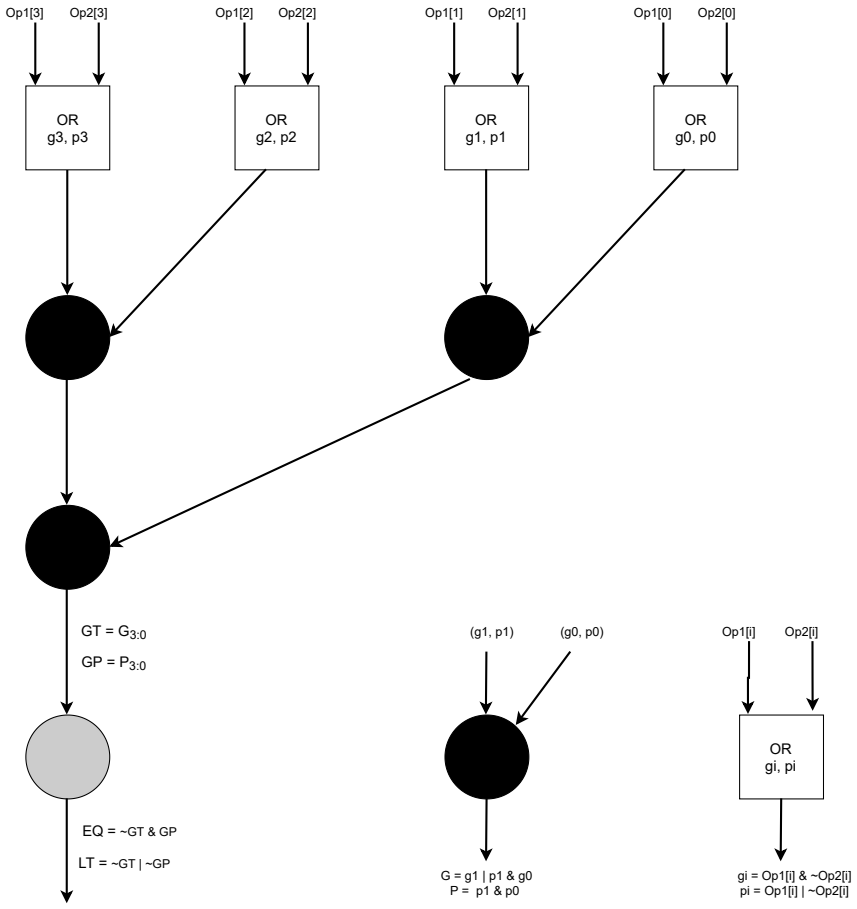

Figure 3.3: Tree-Based Subtractor Architecture [3]

3.0.4 Design Flow and SoC

Shown in Figure 3.0.4 is the general design flow used to implement the proposed designs into independent System on Chip (SoC) designs. These SoCs were designed in a way to directly implement any proposed design into a existing hardware design. Any proposed design could be directly added to a processors data-path for bolstered comparison and machine learning performance. All designs were written in SystemVerilog (HDL) and simulated using the ModelSim test suite. Once the designs were verified within the test suite, the HDL was taken through synthesis using all three of the technology nodes used within this work (SkyWater 130nm, SkyWater 90nm, and cmos32soi ARM SOI). Once the synthesis runs are complete, the results are analyzed and used to provide design feedback for the development of the HDL. This cycle is repeated until all optimization and design improvement options are extinguished. The final synthesis results are then recorded.

3.0.5 Testing

Testing was conducted using the ModelSim test suite for HDL simulation and DesignWare was used for all synthesis runs for all floating-point comparator designs. Floating-point test vectors were generated using the Testfloat tool for thorough analysis of floating-point and two's complement functionality within all designs. This tool generates both Op1 and Op2 along with the expected result FCC value in accordance to IEEE 754 standard. Using these values in tandem with self-checking test benches allowed for easy checking for operation error within the HDL.

The ModelSim test suite was initialized with a '.do' file with the appropriate test parameters for the various comparison operations. These included 64-bit, 32-bit, and 16-bit floating-point comparison values with emphasis on each FCC possibility. These possibilities include GT, LT, EQ, and UO to provide thorough analysis to the proposed designs. Using this testing methodology, the design in question is thoroughly

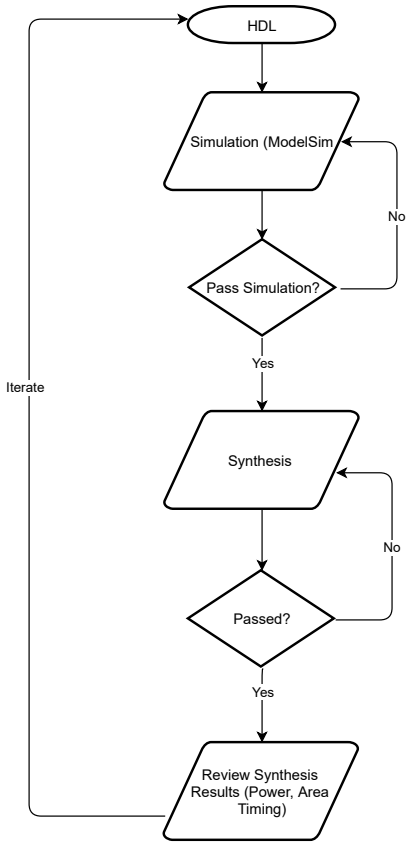

Figure 3.4: Design Flow

examined and analyzed for accuracy of the outputs. These tests used the Testfloat tool to generate all test vector values for these edge cases for the emphasized FCC possibilities described earlier. These simulations also produced a 'VCD' file that was used within the synthesis design flow for more accurate power measurements within each respective fpcomp design.

The Synopsys DC Shell synthesis tool was used once designs were verified within the ModelSim suite. The tool was configured for the SkyWater 130nm (Sky130) and SkyWater 90nm (Sky90) technology nodes using the typical-typical design corners for all standard cell models in the SkyWater technology. The cmos32soi ARM SOI (32nm) cells were however, set up to use the 'RVT' cells. These cells are used for the highest speeds possible for design. This design flow generated the power, area, and timing metrics used for the design comparison in the following section.

CHAPTER IV

RESULTS

This section describes the results of testing the individual designs using the methodology described in the previous section. As mentioned before, the Sky130nm, Sky90, and cmos32soi ARM SOI technology nodes was utilized to generate synthesis results for all fpcomp designs. All synthesis tests were conducted to find area, power, and timing. Further analysis of the power discrepancies between all designs is also included within this analysis.

4.0.1 Synthesis

The results of the synthesis runs are described in the following tables for the Sky130, Sky90, and cmos32soi ARM SOI respectively: Table 4.0.1, Table 4.0.1, and Table 4.0.1. The timing reported within the tables is measured using the timing results of the critical path of an individual design. Power is calculated using the summation of the internal, switching, and leakage power. Due to these designs lacking any sequential components, area is a calculation of the absolute area that encompasses purely the combinational circuits in each of the discussed fpcomp designs. The power-delay product was also provided to provide context of efficiency of power and timing. This figure of merit also excludes switching power to provide for a better performance comparison metric.

Modules	Area (μm^2)	Power (mW)	Period Met (ns)	Power-Delay Product (pJ)
fpcomp	8,892.2987	0.6402	1.8174	1.1630
fpcomp_only	5,471.0567	0.4119	0.8473	0.3491
fpcomp_opt_only	5,606.5877	0.4148	0.8818	0.3658
fpcomp_comb	7,673.2523	0.5685	1.2315	0.7002
fpcomp_ml	8,334.0575	0.6352	1.2413	0.7885
fpcomp_maxmin	7,581.6773	0.6679	1.2010	0.8021
fpcomp_ReLU	9,158.9651	0.6916	1.2808	0.8858
fpcomp_opt_ReLU	8,812.4453	0.6866	1.2754	0.8757
tree_subtractor_architecture [3]	7,314.7435	0.5142	1.2231	0.6289
DW_fp_cmp [20]	12,815.3718	3.7638	1.6917	6.3670

Table 4.1: Synthesis Results for the Various Floating-Point Comparator Designs in Sky130 Technology Node

Modules	Area (μm^2)	Power (mW)	Period Met (ns)	Power-Delay Product (pJ)
fpcomp	2,413.7400	0.2438	0.5832	0.1422
fpcomp_only	1,463.1400	0.0860	0.4003	0.0344
fpcomp_opt_only	1,748.3200	0.1496	0.3855	0.1600
fpcomp_comb	2,376.5000	0.1723	0.5644	0.0973
fpcomp_ml	2,287.3200	0.1282	0.6162	0.0790
fpcomp_maxmin	2,456.8600	0.1946	0.5777	0.1124
fpcomp_ReLU	2,655.8000	2.1693	0.1727	0.3746
fpcomp_opt_ReLU	2,424.5200	1.6007	0.1322	0.2116
tree_subtractor_architecture [3]	2,110.9200	0.1084	0.6195	0.0672
DW_fp_cmp [20]	3,082.1001	2.2527	0.5694	1.2830

Table 4.2: Synthesis Results for the Various Floating-Point Comparator Designs in Sky90 Technology Node

Modules	Area (μm^2)	Power (mW)	Period Met (ns)	Power-Delay Product (pJ)
fpcomp	1,738.3632	0.8473	0.1080	0.0985
fpcomp_only	1,056.2328	0.2795	0.0731	0.0204
fpcomp_opt_only	1,235.3040	0.3313	0.0770	0.0255
fpcomp_comb	1,617.0672	0.3918	0.1053	0.0413
fpcomp_ml	1,758.1536	0.4615	0.0991	0.0457
fpcomp_maxmin	1,498.6440	0.4014	0.1061	0.0429
fpcomp_ReLU	1,345.4280	0.3294	0.1089	0.0359
fpcomp_opt_ReLU	1,584.5088	0.4083	0.1135	0.0463
tree_subtractor_architecture [3]	1,469.5968	0.3848	0.1098	0.0422
DW_fp_cmp [20]	2,348.0352	14.3007	0.1200	1.7150

Table 4.3: Synthesis Results for the Various Floating-Point Comparator Designs in cmos32soi ARM SOI Technology Node

As shown in Table 4.0.1, the area varies drastically across all designed modules. The Synopsys DesignWare comparator fell behind every designed comparator within the table in terms of area and power in comparison to the proposed designs. The designs that yielded the best results across all metrics were the fpcomp_only and fpcomp_opt_only modules. This is expected due to the lack of machine learning and integer based operations. The tree_subtractor_architecture is most similar to the fpcomp_comb design in terms of feature set. The fpcomp_comb design provided marginally worse performance in all metrics tested. The percentage difference for area, power, and timing are defined as follows: 4.78%, 10.05%, and 0.68%. These values imply marginal differences between the two comparators in terms of both performance and feature set. The fpcomp_opt_ReLU provides the similar results in comparison to the fpcomp_ReLU design in terms of all metrics tested. The percentage difference corresponding to these results are: 3.86%, 0.72%, and 0.426%. As mentioned before, the fpcomp_opt_ReLU design provides results for both the ReLU function and the comparison operation without the need to choose between the two. The fpcomp_ReLU design on the otherhand, only provides a ReLU output or a comparison output for Op1

and Op2 at any given time. The results from this synthesis run shows little difference between the two designs. However, in practice the fpcomp_opt_ReLU would yield slightly better results over the fpcomp_ReLU counterpart in both area and timing. As shown in 4.0.1, the Sky90 technology synthesis runs resulted in similar trends within the margin of error for all modules.

As shown in Table 4.0.1, the area and timing found within the 32nm results are all improvements over the 90nm and 130nm technology runs. The Synopsys DesignWare comparator fell behind every designed comparator within the table in terms of area, power, and timing. The difference between the worst timing (DesignWare) and the best (fpcomp_only) was 143.6% difference. To solidify the performance of this papers proposed designs, the fpcomp_ReLU design provides the best power and area results out of the modules designed with the machine learning ReLU functionality in mind. As with the Sky130 and Sky90 synthesis results, the fpcomp_ReLU and fpcomp_opt_ReLU modules offer similar performance metrics with an slight advantage to the fpcomp_ReLU design. The strictly floating-point comparator designs show a similar conclusion in terms of their respective performance metrics. The fpcomp_only and fpcomp_opt_only designs showcase a 15.63% worse area, 16.96% worse power, and 5.24% worse timing for the fpcomp_opt_only design. The tree_subtractor_architecture shows promising results across the board in comparison to this papers proposed designs. However, the differences between the fpcomp_comb design and the tree-based subtractor is marginal but both power and timing prove to be better for the implementation of the fpcomp_comb design. The fpcomp_ReLU design manages to best the tree_subtractor_architecture in all three categories regardless of the additional machine learning functionality.

All technology nodes used for synthesis tests follow the same trends within their respective test suite. Whenever the technology node decreases in size, the area, power, and speed improve drastically. For the synthesis tests, all speed targets were varied

in accordance to the technology node being used. The clock frequency targets for the set 2,000MHz, 4,000MHz, and 20,000MHz for Sky130, Sky90, and cmos32soi ARM SOI respectively. This was done to force the timing to fail for their respective target periods to ensure the synthesis engine exhaustively tries to reach the timing goal. This allows for more accurate area and power results to be formulated with respect to the frequency target. The Power-Delay Product value shown in the above tables is used to normalize out the switching power metric to represent the efficiency of the design. The lower the number the better. Note that this metric is measured in a unit of energy (pJ).

Synthesis results are merely a prediction of a particular design's performance metrics. To fully realize the advantages of the designs, a Place-and-Route (PNR) run must be conducted due to the short-comings present within the synthesis results. Place-and-Route would reveal the performance benefits of the optimization (designs denoted '_opt') versus the unoptimized designs. It would also allow for realistic area, power, and timing results to better differentiate real performance between all proposed designs and other works.

4.0.2 Power

Shown in the following tables is a comprehensive look at the power results for each individual design. As mentioned before, the power is split into three groups: internal, switching, and leakage power. The internal power is given by the individual standard cell power delivery from the VDD and GND rails respectively. The switching power is determined from the capacitance at each node in a design. This capacitance includes gate, diffusion, and wire capacitance values. Leakage power is described by the amount of amperage that leaks through a gate of a transistor. This phenomenon becomes more common as transistor size decreases. It occurs due to a strong voltage potential on one side of the channel pulling electrons between the source and drain

inadvertently because of proximity of the terminals. The following tables showcase the power results for each technology node tested (Sky130, Sky90, and cmos32soi ARM SOI).

Modules	Internal (mW)	Switching (mW)	Leakage (nW)	Total (mW)
fpcomp	0.1745	0.4655	131.1926	0.6402
fpcomp_only	0.1215	0.2904	68.1806	0.4119
fpcomp_opt_only	0.1184	0.2963	67.2624	0.4148
fpcomp_comb	0.1608	0.4076	111.0391	0.5685
fpcomp_ml	0.1778	0.4573	128.0321	0.6352
fpcomp_maxmin	0.1632	0.5046	109.1492	0.6679
fpcomp_ReLU	0.1787	0.5127	137.1926	0.6916
fpcomp_opt_ReLU	0.1825	0.5040	127.8869	0.6866
tree_subtractor_architecture [3]	0.1499	0.3642	92.6337	0.5141
DW_fp_cmp [20]	0.8497	2.8931	221.1670	3.7430

Table 4.4: Power Results for the Various Floating-Point Comparator Designs in Sky130 Technology Node

Modules	Internal (mW)	Switching (mW)	Leakage (nW)	Total (mW)
fpcomp	0.0985	0.1438	1,408.2000	0.2438
fpcomp_only	0.0411	0.0444	619.7482	0.0860
fpcomp_opt_only	0.0683	0.0802	1,037.1000	0.1496
fpcomp_comb	0.0737	0.0972	1,423.5000	0.1723
fpcomp_ml	0.0576	0.0696	1,115.1000	0.1282
fpcomp_maxmin	0.0743	0.1188	1,538.3000	0.1946
fpcomp_ReLU	0.0769	0.0943	1,424.5000	0.1727
fpcomp_opt_ReLU	0.0551	0.0760	1,079.9700	0.1322
tree_subtractor_architecture [3]	0.0518	0.0557	861.8221	0.1084
DW_fp_cmp [20]	1.0556	1.1957	1,422.4300	2.2527

Table 4.5: Power Results for the Various Floating-Point Comparator Designs in Sky90 Technology Node

Modules	Internal (mW)	Switching (mW)	Leakage (uW)	Total (mW)
fpcomp	0.3172	0.3036	226.4267	0.8473
fpcomp_only	0.0741	0.0589	146.5311	0.2795
fpcomp_opt_only	0.0847	0.0707	175.9724	0.3313
fpcomp_comb	0.0913	0.0867	213.8396	0.3918
fpcomp_ml	0.1133	0.1025	245.6304	0.4614
fpcomp_maxmin	0.0903	0.1199	191.1570	0.4014
fpcomp_ReLU	0.0730	0.0855	170.9406	0.3294
fpcomp_opt_ReLU	0.0947	0.1028	210.8073	0.4083
tree_subtractor_architecture [3]	0.0855	0.0837	215.6767	0.3848
DW_fp_cmp [20]	8.1795	5.7783	342.8702	14.3007

Table 4.6: Power Results for the Various Floating-Point Comparator Designs in cmos32soi ARM SOI Technology Node

In order to obtain as accurate power figures as possible, 'saif' files were imported into the synthesis engine for analysis of the signals during a testbench conditions. These files are directly generated from the 'VCD' files mentioned in the earlier testing section. These files allows for accurate power predictions for the design by using the testvectors used from Testfloat inside of the ModelSim test suite to generate a power profile for the synthesis engine to utilize for power prediction.

As can be seen in the tables found above, as the technology node size decreased, the power for each respective design also generally decreased with exception to the 32nm synthesis flow. A noteworthy observation is that the leakage power for each design increased as the technology node size decreased. This is expected due to the channel length decreasing and allowing current to flow when a high voltage potential is placed on either the source or the drain terminals of a transistor. Switching and internal power generally decreased when the technology node decreased as well. This is due to the decreases in capacitance and threshold voltage as the transistor size decreased. The capacitance would decrease due to the size decrease of all gates,

diffusion, and wire traces throughout the design. The internal power would decrease since the design VDD associated within a cell design would decrease as the higher voltage potential is unnecessary for smaller transistors as it could possibly damage the designs. This reduction in voltage also implies a reduction in current and power. With these characteristics in mind, as a particular design falls below the 45nm technology node, the leakage current increases exponentially due to the channel length. These conclusions are easily seen in the 32nm table versus the 90nm table. However, these issues are resolved by using a 3D transistor designs such as FinFET [21] [22]. Further analysis of the power metrics will be conducted in the future using power analysis tools such as Voltus to verify these values further.

CHAPTER V

CONCLUSION

Future work for this proposal includes using the PNR flow to accurately assess all performance differences between all designs. These tests would yield closer-to-reality results in comparison to the synthesis runs. In addition to the PNR design flow, designing a CNN or DNN with each comparator utilizing machine learning functionality would be beneficial to compare the performance between a software and hardware based ReLU approach.

The results of these floating-point comparator modules are promising for both the comparison operation and the ReLU operation found within machine learning workflows. Seen in Section IV, the synthesis results of the fpcomp_opt_ReLU module shows promise in comparison to the other designs and works. This potentially sizable increase in performance provides a good argument for the proposed work be introduced into future neural network implementations that are stringent on execution time of each iteration of progress within a DNN or CNN. As progress is made within the field of machine learning, any and all possible performance enhancements should be sought to provide faster response times within these networks. The current ReLU software solution used in these networks is a performance hinderance in comparison to hardware-level solutions. Also, due to the frequency in which neural networks update their respective weights within the computational layers, the responsiveness of the ReLU operator becomes the most impactful operation within the network in terms of execution time and delay. This potential performance limitation can be resolved with the proposed hardware-based solution of this paper.

REFERENCES

[1] IEEE, "Ieee standard for interval arithmetic," standard, IEEE Computer Society, 2015.

[2] M. J. S. James E. Stine, "A combined two's complement and floating-point comparator," *IEEE*, 2005.

[3] F. Ntouskas, C. Efstathiou, and K. Pekmestzi, "Efficient design of magnitude and 2's complement comparators," *Integration, the VLSI Journal*, pp. 164–169, 2020.

[4] K. W. Glass, "Digital comparator circuit."

[5] D. Norris, "Comparator circuit."

[6] F. Murabayashi, T. Hotta, S. Tanaka, T. Yamauchi, H. Yamada, T. Nakano, Y. Kobayashi, and T. Bandoh, "3.3 v bicmos techniques for a 120-mhz risc microprocessor," 1994.

[7] E. S. Fetzer, M. Gibson, A. Klein, N. Calick, C. Zhu, E. Busta, and B. Mohammad, "A fully bypassed six-issue integer datapath and register file on the itanium-2 microprocessor," 2002.

[8] I. V. Zoev, A. P. Beresnev, E. A. Mytsko, and A. N. Malchukov, "Implementation of 14 bits floating point numbers of calculating units for neural network hardware development," in *Materials Science and Engineering* (I. Publishing, ed.), 2017.

[9] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, pp. 48–60, 2019.

[10] J. R. Hauser, "Berkeley testfloat," 2018.

[11] IEEE, "Ieee standard for binary floating-point arithmetic," standard, IEEE Computer Society, 1985.

[12] IEEE, "Ieee standard for floating-point arithmetic," standard, IEEE Computer Society, 2008.

[13] IEEE, "Ieee standard for floating-point arithmetic," standard, IEEE Computer Society, 2019.

[14] H. Yu, J. Cheng, X. Zhang, Y. Gao, and K. Mei, "Implementation of convolutational neural network with co-design of high-level synthesis and verilog hdl," *IEEE*, 2020.

[15] A. F. M. Agrap, "Deep learning using rectified linear units (relu)," 2019.

[16] F. Leofante, N. Narodytska, L. Pulina, and A. Tacchella, "Automated verification of neural networks: Advances, challenges and perspectives," 2018.

[17] J. A. A. Opschoor, P. C. Petersen, and C. Schwab, "Deep relu networks and high-order finite element methods," *Seminar for Applied Mathematics*, 2019.

[18] P. Hill, B. Zamirai, S. Lu, Y.-W. Chao, M. Laurenzano, M. Samadi, M. Papaefthymiou, S. Mahlke, T. Wenisch, J. Deng, L. Tang, and J. Mars, "Rethinking numerical representations for deep neural networks," 2018.

[19] Standford, "The perceptron,"

[20] DesignWare, "Dw_fp_cmp," tech. rep., Synopsys, 2020.

[21] Y. Tsividis and C. McAndrew, *Operation and Modeling of the MOS Transistor*. Oxford, 2011.

[22] N. H. E. Weste and D. M. Harris, *CMOS VLSI Design*. Addison-Wesley, 2011.

[23] J. L. Henneessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2019.

[24] S. L. Harris and D. M. Harris, *Digital Design and Computer Architecture: ARM Edition*. Morgan Kaufmann, 2016.

[25] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2004.

[26] J. Henry S. Warren, *Hacker's Delight*. Addison Wesley, 2013.

[27] DesignWare, "Dw_cmp_dx," tech. rep., Synopsys, 2019.

[28] DesignWare, "Dw_cmp_dx," tech. rep., Synopsys, 2019.

VITA

Landon Ray Burleson

Candidate for the Degree of

Master of Science

Thesis: FLOATING-POINT COMPARATOR WITH RELU OPERATOR FOR MACHINE LEARNING ENHANCEMENT

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2021.

Completed the requirements for the Bachelor of Science in Computer Engineering at Oklahoma State University, Stillwater, Oklahoma in December, 2019.

Experience:

Graduate Research Assistant - VLSI Computer Architecture Research Group OSU
January 2020 - May 2021