UC Davis
UC Davis Electronic Theses and Dissertations

Title
Computing Numerical Functions on Many-Core Processor Arrays

Permalink
https://escholarship.org/uc/item/7tq3r619

Author
Huo, Yuxuan

Publication Date
2024

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Computing Numerical Functions on Many-Core Processor Arrays

By

YUXUAN HUO

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Electrical and Computer Engineering

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Chair, Bevan M. Baas

Hussain Al-Asaad

Venkatesh Akella

Committee in Charge

2024

© Copyright by Yuxuan Huo 2024
All Rights Reserved

Abstract

Numerical algorithm is a fundamental part of a chip and it plays a crucial role in a chip. The efficient manipulation of numerical data is essential for achieving optimal performance and desired functionality of a chip. The algorithms are designed on the chip to solve complex mathematical problems in different fields. Therefore, an efficient and accurate numerical algorithm can improve the practicality of a chip.

This paper presents some basic numerical algorithms that can apply to the target chip, and the target platform is Asynchronous Array of Simple Processors 3(AsAP3)[1]. The paper uses shift division as the basic dividing function throughout the algorithms to replace the traditional divisions. This paper implements Trigonometric functions, Exponential function, Natural Logarithm function, and LRN function on the AsAP3 platform. This paper applies Taylor series, CORDIC, and binary search algorithms to the implemented functions. Furthermore, this paper records the numerical results of these functions generated by AsAP3 and compares them with the reference values calculated by the MATLAB program. It analyzes the difference, SNR value, and throughput of simulated results to examine the accuracy of the calculation. The paper also displays difference and ratio graphs to visually present the magnitude of the difference. The results and comparisons show that the numerical algorithms offer a satisfactory performance in the target platform.

The applications are programmed with C in Visual Studio and transferred to the AsAP3 platform. The comparison between the generated value and reference value is completed on MATLAB.

Acknowledgments

I would not have accomplished this thesis without the guidance and help of several individuals. I would like to extend my sincere thanks to all of them.

First and foremost, I would like to express my deepest appreciation to my major professor, Dr. Bevan M. Baas, for his guidance and continuous support throughout the entire research process. His insightful advice and inspiring feedback are instrumental in shaping the direction of this work.

I am also grateful to the members of my thesis committee, Professor Hussain Al-Asaad, and Professor Venkatesh Akella, for their valuable feedback on refining my thesis.

It has been an honor for me to be a member of the VLSI Computation Laboratory (VCL). I would like to give special thanks to my talented peers who provided numerous intuitive ideas which helped to complete my thesis.

And finally, I am extremely thankful to my parents and my uncle, who provided me with financial and emotional help throughout the thesis. Their unwavering support and unlimited love have been my motivation to get through this challenging journey.

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Thesis Organization 2

2 Background **3**
2.1 Shift 3
2.2 Taylor series 4
2.3 CORDIC 5
2.4 Binary Search 6
2.5 LRN 6
2.6 AsAP3 9

3 The Implemented Functions **10**
3.1 Shift Division 10
3.2 Trigonometric functions 12
3.2.1 Taylor Series Algorithm 12
3.2.2 CORDIC Algorithm 13
3.3 Exponential 13
3.4 Natural Logarithm 14
3.5 Square Root 14
3.6 LRN 14

4 Algorithms **16**
4.1 Taylor series 16
4.2 CORDIC 17
4.2.1 Derivation of CORDIC 17
4.2.2 CORDIC Algorithm 19
4.3 Binary Search 23

5 Implement Functions on AsAP3 **24**
5.1 Shift Division 24
5.2 Trigonometric Functions 26
5.2.1 Taylor Series 26
5.2.2 CORDIC 29
5.3 Exponential Function 33
5.4 Natural Logarithm Function 33
5.5 Square Root 34
5.6 LRN 35

6 Analysis of Implemented Functions **37**
6.1 Trigonometric Functions 38
6.1.1 Analysis for Trigonometric functions in Taylor series 39
6.1.2 Analysis for Trigonometric functions in CORDIC 42
6.1.3 Data Analysis for Trigonometric Functions 43
6.2 Exponential Function 45
6.3 Natural Logarithm Function 47
6.4 Square Root 50
6.5 LRN 52

7 Thesis Summary and Future Work **54**
7.1 Thesis Summary 54
7.2 Future works 55

List of Figures

2.1 Figure for displaying logical shift, rotation, and arithmetic shift 4
2.2 Various Normalizations 7
2.3 Diagrams for AlexNet 7

3.1 The flowchart of shift division 11

4.1 Rotating the input vector by θ 18
4.2 The flowchart of CORDIC algorithm 20

6.1 Graph for Taylor series, Sine and Cosine 39
6.2 Graph for Taylor series, Arctangent 40
6.3 Graph for Taylor series, Arctangent 41
6.4 Graph for Taylor series, Arctangent 41
6.5 Graph for CORDIC, Sine 42
6.6 Graph for CORDIC, Arctangent 43
6.7 Graph for Taylor series, Exponential 45
6.8 Graph for Taylor series, Exponential 46
6.9 Graph for Taylor series, Natural Logarithm 48
6.10 Graph for Taylor series, Natural Logarithm with input from $X = -0.5$ to $X = 0.5$ 49
6.11 Graph for Taylor series, Natural Logarithm with 4096 inputs from $X = -0.5$ to $X = 0.5$ 49
6.12 Graph for Square Root function 51
6.13 Graph for LRN 52

List of Tables

4.1 $arctan(2^{-i})\, value\, for\, i^{th}\, iteration$ . . . 21
4.2 $2^{-i}\, value\, for\, i^{th}\, iteration$ . . . 22

6.1 Results of Trigonometric functions in Taylor series . . . 38
6.2 Results of Trigonometric functions in CORDIC . . . 39
6.3 Comparison for Sine . . . 44
6.4 Comparison for Arctangent . . . 44
6.5 Results of Exponential function in Taylor series . . . 45
6.6 Results of Natural Logarithm function in Taylor series . . . 47
6.7 Throughput Results of Natural Logarithm function in Taylor series . . . 47
6.8 SNR results for Natural logarithm function with different number of terms 50
6.9 Results of Square Root function . . . 50
6.10 Results for LRN function . . . 52

Chapter 1

Introduction

1.1 Motivation

The numerical algorithm is a fundamental operation in arithmetic calculation supported by every computer system. The numerical algorithm serves as the backbone for solving complex mathematical problems such as scientific simulations and engineering optimizations. In the chip design, the pursuit of efficiency and accuracy is a paramount part of the numerical algorithm. Therefore, this paper seeks to explore and analyze the numerical calculations applied on the AsAP3[2] platform, meanwhile aims to enhance the applicability of the calculations.

The fixed-point calculation is an important part of the AsAP3 platform. The platform currently has applications including Fast Fourier Transform, low-density parity check, and sorting algorithm. Adding the calculating application of the arithmetic function to the AsAP3 increases the functionality of the platform.

Therefore, this thesis proposes and implements two basic arithmetic operations, division, and square root calculation, and applies them to several numerical calculating functions. These functions include Trigonometric functions, exponential function, natural logarithm function and LRN function. For Trigonometric functions, Sine, Cosine and Arctangent functions use the Taylor series and CORDIC algorithm to finish the computation. To examine the performance of each function, this paper analyzes the difference

and SNR value between simulation and reference values. This thesis uses AsAP3 as the platform to test the proposed algorithms.

1.2 Thesis Organization

The following chapter descriptions are the organization of this thesis. Chapter 2 presents the background for the shift first. Then it introduces the basic information about the Taylor series, CORDIC, and binary search algorithm. It also studies the LRN function. Lastly, it introduces the AsAP3 platform.

Chapter 3 introduces the implemented functions in this paper. It introduces shift division first as it is a basic computational method in the paper. Then, it presents Trigonometric functions in the Taylor series and CORDIC, Exponential function, and Natural Logarithm function. Lastly, it discusses the square root and LRN functions.

Chapter 4 explains the algorithms shown in the paper. It first describes the procedure of the Taylor series. Then, it presents a derivation of the CORDIC with an explanation of the algorithm. It also illustrates the binary search algorithm.

Chapter 5 demonstrates the implementation of functions on the AsAP3 platform. It describes the implementing procedure for Trigonometric functions, Exponential function, Natural Logarithm function, Square Root function, and LRN function. It also provides pseudo-codes for each function.

Chapter 6 explains the testing environment for the thesis. Then, it displays the testing results with tables and graphs for each function. It lists difference, SNR value, and throughput as parameters to determine the performance of each function. This paper also conducts more tests with the Taylor series functions that have abnormal results and analyzes these test results.

Chapter 7 summarizes the thesis and proposes some thoughts on future work.

Chapter 2

Background

This chapter introduces the background for all the functions used in the thesis. First, it introduces the shifting method with the basic idea of shift operation. Then, it researches the background of the Taylor series in this paper. It also presents the CORDIC algorithm with the invention background and useful scenarios. In addition, this paper reviews the background of binary search and square root. Lastly, it introduces and explains the LRN function as well as the Alexnet, the platform that uses LRN in its calculations.

2.1 Shift

The bit-wise shift is an operation that moves every digit in a number's binary representation left or right. Shift operation includes three categories, logical shift, rotate, and arithmetic shift. Figure 2.1 below displays the three shifts[3].

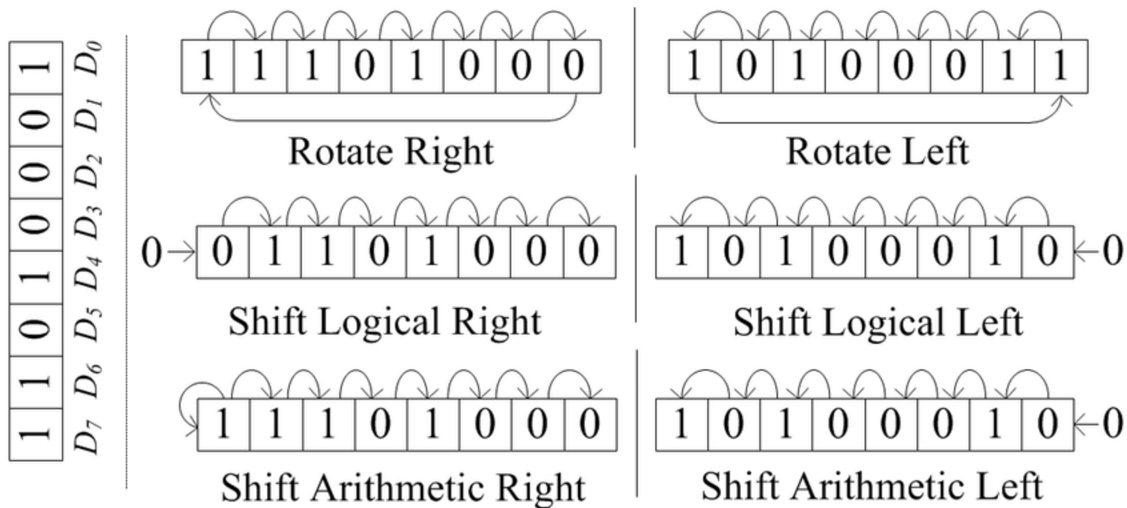

Figure 2.1: Figure for displaying logical shift, rotation, and arithmetic shift

A logical shift moves bits to the left or right. The left or right shift removes the leftmost or rightmost bits and adds a 0 to fill the space from the opposite side. Therefore, a logical left shift is a multiplication by 2, and a logical right shift is a division by 2. The rotation operation shifts the bits circularly. The rotation moves the leftmost or rightmost bit to the opposite end of the binary string. The arithmetic right shift is similar to a logical right shift, but it fills the leftmost bits with the sign bit of the original number.

This paper uses shift in the calculations of Taylor series, CORDIC, and LRN.

2.2 Taylor series

The Taylor series is a mathematical tool that represents a function as an infinite sum of terms. It is named after the English mathematician Brook Taylor in 1715. The fundamental concept of Taylor series is Isaac Newton's work on interpolation, where Newton approximates the functions using finite polynomial series at given points. James Gregory explores the infinite series representation of trigonometric functions. Later, Taylor formalizes and generalizes the ideas from both Newton and Gregory, and creates the Taylor series. It is useful for approximating functions in terms of polynomials. The series expansion is based on derivatives of the function at a specific point. The series enables the approximation of complex functions using polynomials and makes mathematical cal-

culation easier. The following equation shows the general form of the Taylor series,

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 ... \quad (2.1)$$

The Taylor series can obtain accurate approximate $f(x)$ by adding more terms to the series. In the above equation, $f(a)$ approximates the value of $f(x)$ near point a, and adding more terms can refine the approximation and increase a larger approximating range for x values.

This paper utilizes the Taylor series in calculating the approximate value of the functions including Sine, Cosine, Arctangent, Exponential, and Natural Logarithms.

2.3 CORDIC

Jack Volder introduced the coordinate rotation digital computer (CORDIC) in 1956 to replace the analog resolver with a more accurate and faster real-time calculator. Flight control and radar computer systems use the CORDIC algorithm. CORDIC uses trigonometric identities as the basic principle to achieve the calculation of an arbitrary angle for trigonometric functions. Since CORDIC only uses addition, subtraction, and shifting, it is a hardware-efficient algorithm and is commonly used when there is no multiplier or divider available, such as FPGA and micro-controllers[4].

The CORDIC algorithm has two modes: rotation and vector modes. The representative functions of rotation mode are Sine and Cosine, and the representative function for vector mode is Arctangent. The calculation includes bit conversion in each iteration. Each iteration will decide the rotation direction of the next iteration. Therefore, additional iterations can increase accuracy and reliability.

This paper uses the CORDIC algorithm in calculating the value of trigonometric functions including Sine, Cosine, and Arctangent.

2.4 Binary Search

In 1946, John Mauchly first mentioned the binary search algorithm in a report. In the mid-20th century, binary search became a popular and formal algorithm as mathematicians and computer scientists started to focus on the efficiency of sorting algorithms. The binary search algorithm has played an important part in data structures and algorithm designs ever since.

The binary search method was developed in computer science and mathematical calculation due to the efficiency of finding a specific element in a sorted collection of data. Binary search reduces the number of comparisons by dividing the search interval in half repeatedly. The algorithm has a time complexity of $O(logn)$ for finding the specific element in an array size of size n. The efficiency makes it a fundamental concept and algorithm in computer science.

The paper uses binary search in the calculation of the square root function.

2.5 LRN

Normalization is a crucial procedure in a convolutional neural network. It uses certain algorithms to restrict the growth of the unbounded activation function. Local Response Normalization (LRN) is one of the normalizations that is commonly used[5]. Figure 2.2 below shows the normalizations currently used[6].

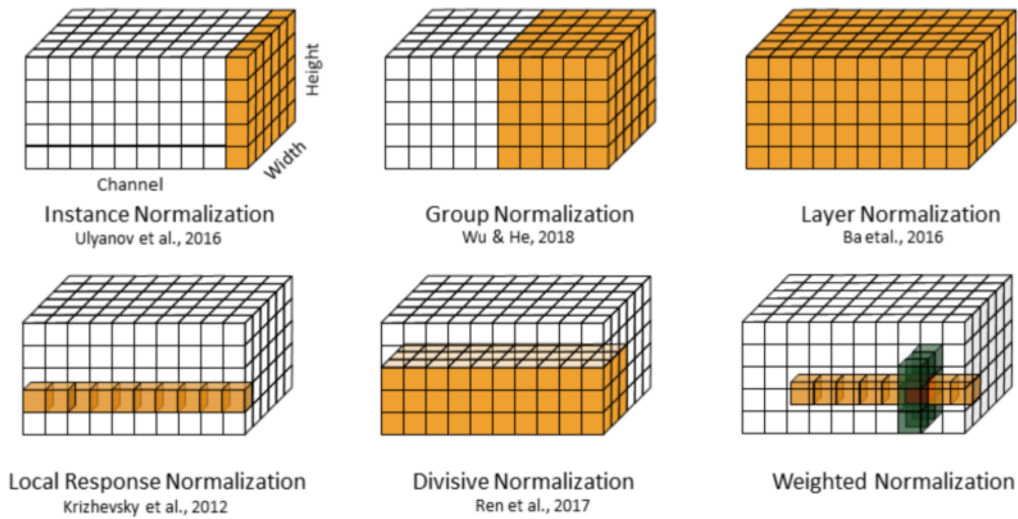

Figure 2.2: Various Normalizations

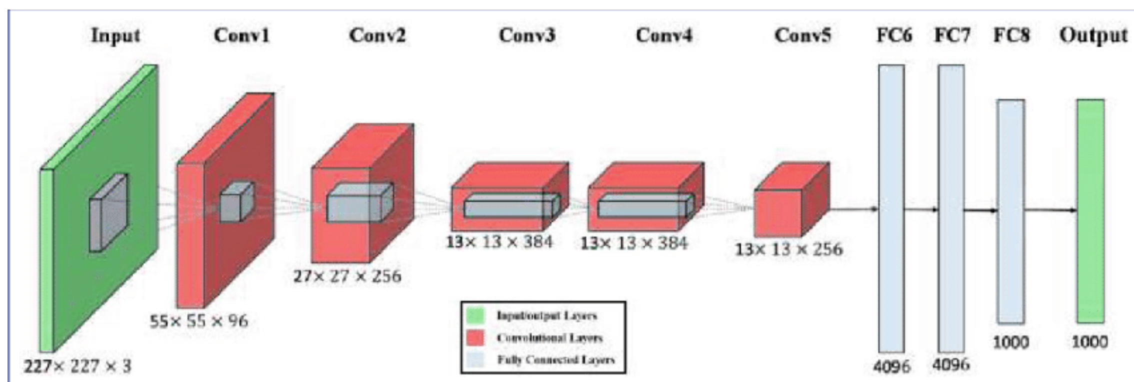

Figure 2.3: Diagrams for AlexNet

AlexNet is the name of a convolutional neural network architecture designed by Alex Krizhevsky, and it introduces the LRN. Figure 2.3 shows the diagram for AlexNet[7]. The AlexNet is eight layers deep and uses more than one million images on the ImageNet database as training objects. The input size for AlexNet is $227*227*3$. In AlexNet, LRN implements lateral inhibition. Lateral inhibition is the capacity of the excited neuron, and it can restrain the neighboring neurons. Lateral inhibition can create a clear contrast between the excited neuron and neighboring neurons. AlexNet uses Rectified Linear Unit(ReLU) as the activation function. However, ReLU does not have a normalization method to limit the growth of output value. ReLU uses LRN to help normalize the function. Therefore, it's getting easier to detect high-frequency features, and excited

neurons can be separated from their neighboring neurons[8].

The equation to calculate LRN includes inter-Channel LRN and Intra-Channel LRN. The equation for inter-Channel LRN shows as below,

$$b^i_{x,y} = a^i_{x,y}/(k + \alpha \sum_{j=max(0,i-\frac{n}{2})}^{min(N-1,i+\frac{n}{2})} (a^j_{x,y})^2)^\beta \tag{2.2}$$

In the equation, the constants (k, α, β, n) are the parameters that shape the normalization. In the AlexNet paper, the parameters are set to $(2, 10^{-4}, 0.75, 5)$ respectively. $a^i_{x,y}$ is the pixel value before the normalization, while $b^i_{x,y}$ is the result after the normalization. $a^j_{x,y}$ are the surrounding neurons of $a^i_{x,y}$. The range of surrounding neurons bases on the constants n and the total number of channels N. The parameter constant α normalizes the sum of surrounding neurons, and the parameter constant β sets a boundary for the result. Then, the calculated value divides $a^i_{x,y}$ to complete the normalization procedure. The equation uses parameter constant k to avoid possible singularities, such as dividing by zero. In the equation, i is the target channel and j is the neighbor channel.

The equation for intra-Channel LRN shows as below,

$$b^k_{x,y} = a^k_{x,y}/(k + \alpha \sum_{i=max(0,x-\frac{n}{2})}^{min(W,x+\frac{n}{2})} \sum_{j=max(0,y-\frac{n}{2})}^{min(H,y+\frac{n}{2})} (a^k_{i,j})^2)^\beta \tag{2.3}$$

In the equation above, the parameter (k, α, β, n) for intra-Channel LRN equation are identical to the parameters of inter-Channel LRN equation. The (W, H) are the width and height of the original map, in the example figure 2.2, the $(W, H) = (6, 5)$. In the equation, k is the target unit, and i and j are the neighbor units surround. The difference between inter-Channel LRN and intra-Channel LRN is how it defines its neighbor. In the Inter-Channel LRN, the neighbors are the same pixels across different channels, and in the intra-Channel LRN, the neighbors are the pixels around the under-selected pixel. The AlexNet paper uses inter-Channel LRN equation to calculate the desired LRN value.

2.6 AsAP3

This thesis uses AsAP3 as the platform to test and implement the functions. AsAP stands for Asynchronous Array of Simple Processors, and AsAP3 is its third generation[2]. The processor array contains 1000 independent processors and 12 memory modules with 768 KB shared memory[9]. Since it has 1000 processors, KiloCore is another name for the processor array. There is a packet router in each processor to route the input and output signals. Each router has a 45.5Gb/s maximum reading speed and a 9.1Gb/s maximum reading speed per port[10]. The processor supports two 16 bits signed or unsigned inputs with each input having an independent clock[11], and seven 16 bits unsigned outputs. Thus, the range for input is either -32767 to 32767 or 0 to 65535 and the range for output is 0 to 65535.

Each processor supports 128 instructions and 72 instruction types including addition, subtraction, multiplication, and shifting. The processor supports carry operations and partial accumulations for data types larger than 16 bits[12]. Although the input and output range for the processor is 16 bits, the AsAP3 supports 32-bit fixed point and floating point calculation through the software.

Running a thousand cores together is power-consuming for a chip. To save power, the KiloCore processor has globally asynchronous locally synchronous(GALS) clocking styles. Each module has its local programmable clock oscillator, and the oscillator can change its frequency based on the requirement of the running applications[13]. To maintain reliability in transferring data, the GALS system has synchronization circuits between each clock domain. Therefore, the cores can operate or idle separately to reduce power usage[14]. Since there are 1000 cores, 12 memory modules, and 1000 packet routers connecting with each module, the total number of the GALS clocks for a single KiloCore chip is 2012[15]. By using the GALS, the processor can optimize the energy efficiency by 5% to 42% compared with traditional core usage optimization[16].

Chapter 3

The Implemented Functions

This chapter introduces the functions implemented in this paper. It first introduces the basic division function this paper uses. Then, it introduces the implemented functions. The functions include Trigonometric, Exponential, Natural Logarithm, and LRN functions. This chapter also presents the derivation process of the CORDIC algorithm.

3.1 Shift Division

The shift calculation realizes the function of division by left shift and addition. It serves as a basic division function in this paper. Since shifting left by 1 bit represents a multiplication of 2, the final quotient is the sum of powers of 2. The following flowchart presents the process of shift division. The functions in the following sections all utilize shift division as the division method to replace the arithmetic division.

The primary use of shift division in the Taylor series is to calculate the quotient for Taylor series terms. Each term uses the shift division once. Another use of shift division in the Taylor series is calculating each series' input values. The following subsections of different Taylor series further explain the use of the shift algorithm. The CORDIC function uses shift division in calculating the final output for the Arctangent function. LRN calculation applies the shift division in calculating the normalization of the neurons. The input and output range for shift division are 16 bits unsigned fixed-point numbers.

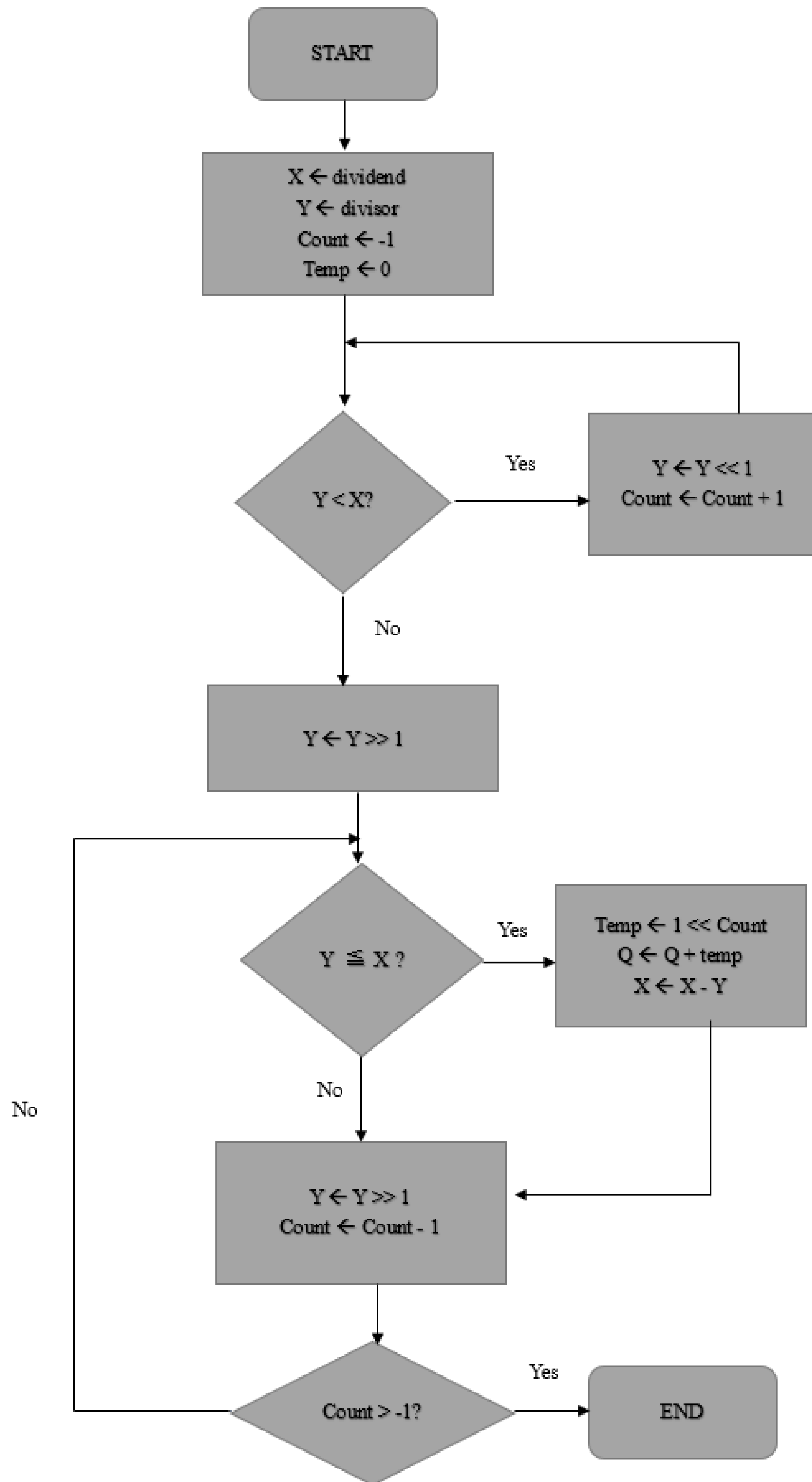

Figure 3.1: The flowchart of shift division

3.2 Trigonometric functions

3.2.1 Taylor Series Algorithm

This paper implements trigonometric functions including Sine, Cosine, and Arctangent. The implementation utilizes both the Taylor series and CORDIC algorithms. For the Taylor series, each simulation has different input ranges. In the Sine and Cosine functions using the Taylor series, inputs are from 0 to 4096, representing inputs from 0 to 2π. To convert from 2π to 4096, the conversion equation is

$$x = \frac{input * 2\pi}{4096} \tag{3.1}$$

The output for Sine and Cosine are signed values with the format of $s1.15$. Thus, the output range is from -32767 to 32767.

The equations for Sine and Cosine present as below,

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + ... \tag{3.2}$$

$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + ... \tag{3.3}$$

For Arctangent, inputs are from 0 to 4095 representing inputs from 0 to 4. The conversion equation is

$$x = \frac{input}{1024} \tag{3.4}$$

The outputs for Arctangent are unsigned values with the format of $s1.15$. Thus, the output range is from 0 to 65535. However, since the Arctangent function converges at $\frac{\pi}{2}$, the actual output range is from 0 to $51472(\frac{\pi}{2} * 32768)$.

The equations for Arctangent show as below. The Arctangent functions contain four equations. The following chapters presents the explanation for choosing the four equations.

$$arctan(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + ...(x < 0.78125) \tag{3.5}$$

$$arctan(x) = 0.663 + 0.621 * (x - 0.78125) - 0.301 * (x - 0.78125)^2$$
$$+0.0663 * (x - 0.78125)^3 + 0.0453(x - 0.78125)^4 \quad (3.6)$$
$$-0.0599(x - 0.78125)^5 ... \ (0.78125 \leq x < 1)$$

$$arctan(x) = \frac{\pi}{4} + \frac{x-1}{2} - \frac{(x-1)^2}{4} + \frac{(x-1)^3}{12} - \frac{(x-1)^5}{40} + \frac{(x-1)^6}{48} ...(1 \leq x < 1.5625) \quad (3.7)$$

$$arctan(x) = \frac{\pi}{2} - \frac{1}{x} + \frac{1}{3x^3} - \frac{1}{5x^5} + ...(x \geq 1.5625) \quad (3.8)$$

3.2.2 CORDIC Algorithm

For the CORDIC algorithm, the Sine and Cosine functions use degrees as inputs instead of radians in the CORDIC function. Thus, inputs range from 0° to 360° instead of 0 to 2π. For Arctangent, the CORDIC algorithm takes the value of opposite and adjacent sides as input. In the Arctangent function, this paper sets the adjacent side to 1024 and calculates the Arctangent value by varying the opposite side. The function sets 1024 as 1. Thus, the inputs ranging from 0 to 4095 represents inputs from 0 to 4.

The output for Sine and Cosine are signed values with the format of $s1.15$ and the output range is from -32767 to 32767. The outputs for Arctangent are unsigned values with the format of $s1.15$ and the output range is from 0 to 51472.

3.3 Exponential

This paper uses the Taylor series to calculate the Exponential function. The inputs of the exponential function range from 0 to 4095 and each input divides by 2048 to represent input ranges from 0 to 2. The outputs for exponential are unsigned values with the format of $s3.13$. Thus, the output range is from 0 to 65535.

The equation for the Exponential function shows as below,

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + ... \quad (3.9)$$

3.4 Natural Logarithm

This paper uses the Taylor series to calculate the Natural Logarithm function. The inputs for the Natural Logarithm function range from 0 to 4095, and each input divides by 2048 to represent $-1 < x \leq 1$. The outputs for the Natural Logarithm are signed values with the format of $s2.14$. Thus, the output range is from -32767 to 32767.

The equations for the Natural Logarithm function show as below. The functions contain two equations, one equation takes input value greater than -1 and less than 0, and one equation takes input values greater than 0 and less than 1.

$$ln(1 + x) = -x - \frac{x^2}{2} - \frac{x^3}{3} - \frac{x^4}{4} - ...(-1 < x \leq 0) \quad (3.10)$$

$$ln(1 + x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4}...(0 < x \leq 1) \quad (3.11)$$

3.5 Square Root

This paper uses binary search algorithm to accomplish the square root function. The square root calculation can take input value up to 16 bits unsigned value with an 8-bit integer part and an 8-bit decimal part. Thus, the input range is from 0 to 65535. Since the AsAP3 platform supports 32-bit multiplication calculation, the output of the square root function is still 16 bits unsigned value with an 8-bit integer part and 8-bit decimal part, and the range is from 0 to 65535.

3.6 LRN

This paper uses the square root function and shift division function to calculate the LRN function. This paper follows the AlexNet and uses the inter-Channel LRN equation

as the equation to compute the LRN function. The inter-Channel LRN equation shows as below,

$$b^i_{x,y} = a^i_{x,y}/(512 + \sum_{j=max(0,i-\frac{5}{2})}^{min(N-1,i+\frac{5}{2})} (a^j_{x,y})^2)^{0.75} \tag{3.12}$$

In AlexNet paper, the constants (k, α, β, n) are set to $(2, 10^{-4}, 0.75, 5)$. In this paper, the constants are set to $(512, 1, 0.75, 5)$ to be compatible with fixed number calculation. The maximum possible number of channels involved in the calculation is 5. Thus, $5 * 127 * 127 + 512 = 81157$, and it creates an overflow for 16-bit calculations, whereas $5 * 63 * 63 + 512 = 20357$ does not. Therefore, the input range for the LRN function is set from 0 to 63, and the output range is from 0 to 65535.

Chapter 4

Algorithms

This chapter explains the algorithms used in the following calculations. First, it describes the Taylor series algorithm. It also presents the CORDIC algorithm with equations, flowchart, and tables displaying the ith iteration value for $arctan(2^{-i})$ and 2^{-i} in the calculating equations for the CORDIC algorithm. Lastly, it shows the binary search algorithm.

4.1 Taylor series

The Taylor series is a mathematical tool that represents a function as an infinite sum of terms. The following equation shows the general form of the Taylor series,

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 ... \tag{4.1}$$

The accuracy of the Taylor series depends on the number of terms in the equation. Generating a Taylor series equation requires four steps. The first step is to find a center of expansion. In this paper, the centers of expansion for most of the Taylor series equations are 0. For the Arctangent function, it has two more centers of expansion at 0.78125 and 1.5625. The second step is to decide the number of terms for the series and calculate the derivatives of the function for each term. In this paper, the different function has a different number of equation terms to ensure accuracy. For Sine and Cosine, they have

5 terms. For Arctangent, different equations have different terms. Equation 3.5 has 10 terms, equation 3.6 and equation 3.7 have 5 terms, and equation 3.8 has 7 terms. For Exponential, it has 8 terms. For Natural Logarithm, equations have 7 terms. The third step is to substitute the calculated derivatives into the Taylor series and simplify the series into a recognizing pattern. The fourth step is to determine the convergence to ensure the series has an accurate approximation within a specific range. For Sine, Cosine, and Exponential functions, the interval of convergence is all real numbers. For Arctangent, equation 3.5 converges for interval $-1 \leq x \leq 1$, and equation 3.8 converges for $x > 1$. Natural Logarithm converges between $0 < x \leq 2$.

4.2 CORDIC

4.2.1 Derivation of CORDIC

Suppose a system rotates the original angle β in θ degree. Therefore, the point (X_{in}, Y_{in}) rotates to (X_R, Y_R). The rotating point (X_R, Y_R) can calculate as follows,

$$X_R = X_{in} cos(\theta) - Y_{in} sin(\theta) \tag{4.2}$$

$$Y_R = X_{in} sin(\theta) + Y_{in} cos(\theta) \tag{4.3}$$

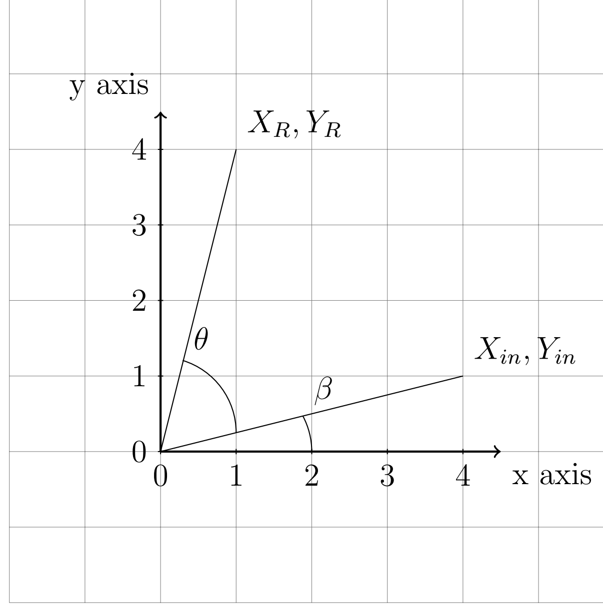

Figure 4.1: Rotating the input vector by θ

The equation 2.1 and 2.2 can be transformed to the following matrix multiplication.

$$\begin{bmatrix} X_R \\ Y_R \end{bmatrix} = cos(\theta) \begin{bmatrix} 1 & -tan(\theta) \\ tan(\theta) & -1 \end{bmatrix} \begin{bmatrix} X_{in} \\ Y_{in} \end{bmatrix} \tag{4.4}$$

Rotation of an angle can divide into the addition of smaller angles, either positive or negative, and $\theta = arctan(2^{-i})$ for $i = 0, 1, 2, ...n$ can reduce the calculation complexity and use bit-wise shift for calculation. Therefore, equation 2.3 can simplify as follows,

$$\begin{bmatrix} X_0 \\ Y_0 \end{bmatrix} = cos(45^\circ) \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} X_{in} \\ Y_{in} \end{bmatrix} \tag{4.5}$$

$$\begin{bmatrix} X_0 \\ Y_0 \end{bmatrix} = cos(-45^\circ) \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} X_{in} \\ Y_{in} \end{bmatrix} \tag{4.6}$$

For the second rotation, equation 2.4 becomes

$$\begin{bmatrix} X_1 \\ Y_1 \end{bmatrix} = cos(45^\circ)cos(26.57^\circ) \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} 1 & -2^{-1} \\ 2^{-1} & 1 \end{bmatrix} \begin{bmatrix} X_0 \\ Y_0 \end{bmatrix} \tag{4.7}$$

As rotation continues, $cos(45^\circ) * cos(26.57^\circ) * ...cos(arctan(2^{-i}))$ becomes a fixed value, and the value is approximately 0.6072. The equations after n^{th} rotation are

$$X_{i+1} = X_i - \alpha_i(2^{-i}Y_i) \tag{4.8}$$

$$Y_{i+1} = Y_i + \alpha_i(2^{-i}X_i) \tag{4.9}$$

$$Z_{i+1} = Z_i - \alpha_i\theta^i \tag{4.10}$$

where α is the direction of nth rotation and Z_i is the accumulated angle of nth rotations. CORDIC uses equations 4.8, 4.9, and 4.10 as three basic equations for calculation.

4.2.2 CORDIC Algorithm

The equation 4.8, equation 4.9, and equation 4.10 present the three equations that calculate Sine, Cosine, and Arctangent values using the CORDIC algorithm.

The flowchart below shows the algorithm flow for the CORDIC algorithm for Sine and Cosine.

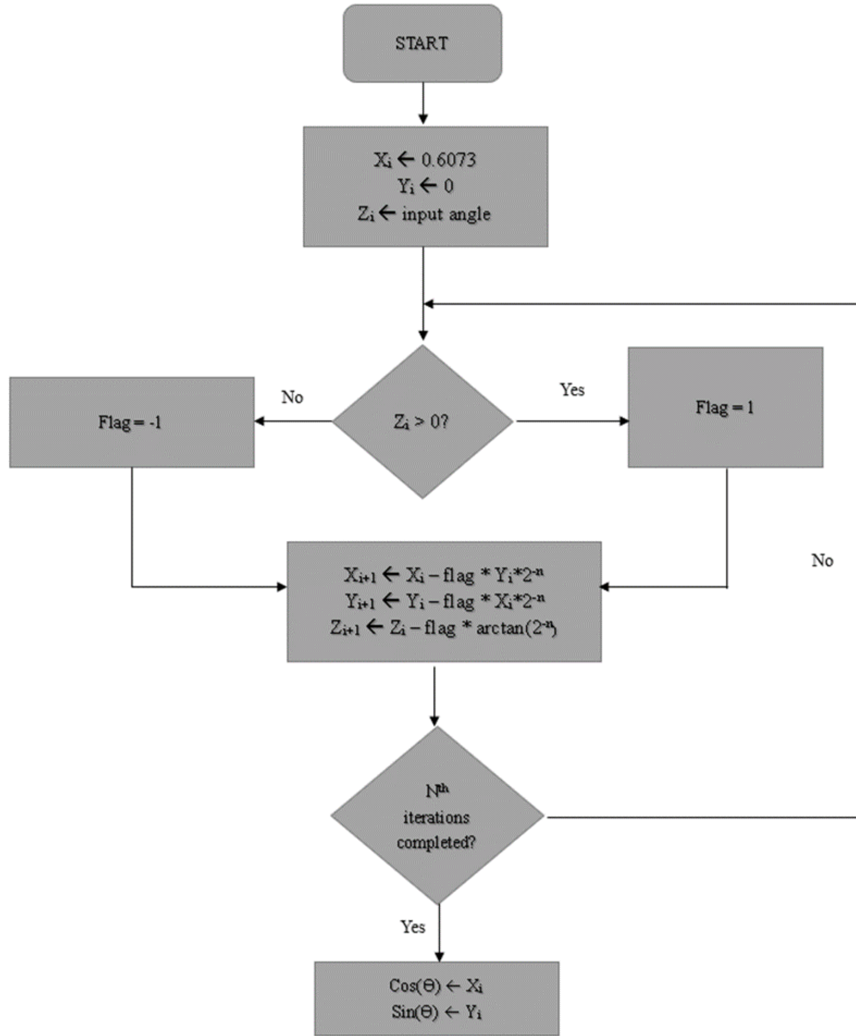

Figure 4.2: The flowchart of CORDIC algorithm

Figure 4.2 displays the flowchart for the CORDIC algorithm. The calculation uses $\theta^i = arctan(2^{-i})$ to calculate the rotation angle in each iteration. This paper uses 16 iterations for each calculation to increase the accuracy. α_i serves as a flag to decide the rotation direction. In the calculating procedure of Sine and Cosine, z^i decides the rotation direction. A positive value means a counterclockwise rotation, whereas a negative z^i value indicates a clockwise rotation. For the Arctangent function, y^i determines the direction of the next movement. A positive y^i means a clockwise rotation and a negative value means a counterclockwise rotation.

Table 4.1 and 4.2 present the calculated $arctan(2^{-i})$ and 2^{-i}.

Iteration i	$\theta_i(arctan(2^{-i}))$
0	45
1	26.565
2	14.036
3	7.125
4	3.576
5	1.790
6	0.895
7	0.448
8	0.224
9	0.112
10	0.0560
11	0.0280
12	0.0140
13	0.00699
14	0.00350
15	0.00175

Table 4.1: $arctan(2^{-i})\, value\, for\, i^{th}\, iteration$

Iteration i	2^{-i}
0	1
1	0.5
2	0.25
3	0.125
4	0.0625
5	0.03125
6	0.015625
7	0.0078125
8	0.00390625
9	0.001953125
10	0.0009765625
11	0.00048828125
12	0.000244140625
13	0.000122070312
14	0.000061035156
15	0.000030517578

Table 4.2: $2^{-i}\, value\, for\, i^{th}\, iteration$

4.3 Binary Search

Binary search is an algorithm used to find a specific element in a sorted array. It compares the number by repeatedly dividing the search interval in half. The algorithm has a time complexity of $O(logn)$ and a space complexity of $O(1)$ for locating an element in the array of size n.

This paper uses binary search in calculating the square root function. The algorithm first sets a $start$ value and an end value, normally the $start$ value equals 0 and the end value equals the largest possible number. Then, the algorithm sets a mid value by calculating the average value of the $start$ and end values. Next, it compares the mid value with the input value. If mid is larger than the input, the end value updates to $mid - 1$, if mid is smaller than the input, the $start$ updates to $mid + 1$. The algorithm continues until the mid value is the closest possible value to the input.

Chapter 5

Implement Functions on AsAP3

This chapter displays the implementing of the functions on AsAP3 in the previous chapter. It explains the procedures of the implementation. It also presents the pseudo-code for each implemented function.

5.1 Shift Division

The pseudo-code below illustrates this calculation begins with a comparison of a dividend x and a divisor y. If y is less than x, y starts the left shift, and it shifts n times to ensure y is less than or equal to x, while one more time left shift makes y larger than x. Then x subtracts y to get the first remainder. Since y left shifts n times from the original divisor, x subtracts $y * 2^n$. Thus, the result for the first loop is 2^n, which adds to the final quotient. Then, the next loop checks whether x is greater than y. If x is larger than y, the above algorithm continues calculating at $n-1$ loop. The algorithm skips this loop if x is less than y in the $n-1$ loop. The algorithm stops when the remainder is less than y.

For decimal divisions, the basic logic is the same. In this case, the dividend x left shifts n times, and 2^{-n} adds to the previous quotient. In the last loop, the algorithm uses rounding if the remainder of x is larger or equal to half of y.

The following pseudo-code shows the shift division algorithm precisely.

Algorithm 1 Pseudo-code for shift division

```
1: function DIVISION(x,y)                              ▷ Dividendxanddivisory
2:     count ← −1
3:     temp_result ← 0
4:     while y < x do
5:         y ← y ≪ 1
6:         count ← count + 1
7:     end while
8:     y ← y ≫ 1
9:     while count > −1 do
10:        if x ≥ y then
11:            temp_result ← 1 ≪ count
12:            q ← q + temp_result
13:            x ← x − y
14:        end if
15:        y ← y ≫ 1
16:        count ← count − 1
17:    end while
18:    remainder ← y
19:    check ← 9                                       ▷ This is to make a 2^{-10} precision
20:    if remainder < y ∨ x < y then
21:        while check ≥ −1 do
22:            y ← y ∗ 2
23:            while x ≥ y do
24:                q ← q + 2^{check−10}
25:                x ← x − y
26:            end while
27:            check ← check − 1
28:        end while
29:        if x ≤ (y ≫ 1) then
30:            q ← q + 2^{−10}
31:        end if
32:    end if
33: return q
34: end function
```

5.2 Trigonometric Functions

5.2.1 Taylor Series

For $sin(x)$ and $cos(x)$, the calculation function can take input from 0 to 2π. As inputs get larger towards 2π, the calculation deviates from the desired values. The function only calculates the value from 0 to $\frac{\pi}{2}$ to decrease the inaccuracy, and the algorithm transforms the calculation from $\frac{\pi}{2}$ to 2π to 0 to $\frac{\pi}{2}$ because Sine and Cosine functions have symmetrical characteristic. For example, the algorithm transforms $\frac{3\pi}{4}$ to $\frac{\pi}{4}$ since $cos(\frac{3\pi}{4})$ is the negative of $cos(\frac{\pi}{4})$ and $sin(\frac{3\pi}{4})$ is equal to $sin(\frac{\pi}{4})$. Thus, to get the Sine and Cosine value of $\frac{3\pi}{4}$, the algorithm only calculates the Sine and Cosine value of $\frac{\pi}{4}$ and converts the Cosine value into negative. For the input value from $\frac{\pi}{2}$ to π in the second quadrant, the conversion equation is

$$new_input = \pi - input \tag{5.1}$$

For input value from π to $\frac{3\pi}{2}$ in the third quadrant, the conversion equation is

$$new_input = input - \pi \tag{5.2}$$

For input value from $\frac{3\pi}{2}$ to 2π in the fourth quadrant, conversion equation is

$$new_input = 2\pi - input \tag{5.3}$$

Similarly, the output of the algorithm changes as well. For the input in the second quadrant, the $sin(x)$ value remains the same, while the $cos(x)$ value negates itself. The output in the third quadrant negates the $sin(x)$ value and $cos(x)$ value. The output in the fourth quadrant keeps the $cos(x)$ value the same, meanwhile negating the $sin(x)$ value. The following pseudo-code shows the implemented algorithm of the Taylor series for Sine and Cosine.

Algorithm 2 Pseudo-code for Sine and Cosine Taylor series - part 1

Input: $radian\, r$

Output: sin_val, cos_val

1: **for** $r = 0; r < \frac{\pi}{2}; r++$ **do**

2: $a \leftarrow r$

3: $cos_val = 1 - division(a^2, 2!) + division(a^4, 4!) - division(a^6, 6!) + division(a^8, 8!) - division(a^{10}, 10!)$

4: $sin_val = a - division(a^3, 3!) + division(a^5, 5!) - division(a^7, 7!) + division(a^9, 9!) - division(a^{11}, 11!)$

5: **end for**

6: **for** $r = \frac{\pi}{2}; r < \pi; r++$ **do**

7: $a \leftarrow \pi - r$

8: $cos_val = 1 - division(a^2, 2!) + division(a^4, 4!) - division(a^6, 6!) + division(a^8, 8!) - division(a^{10}, 10!)$

9: $sin_val = a - division(a^3, 3!) + division(a^5, 5!) - division(a^7, 7!) + division(a^9, 9!) - division(a^{11}, 11!)$

10: **end for**

11: **for** $r = \pi; r < \frac{3\pi}{2}; r++$ **do**

12: $a \leftarrow r - \pi$

13: $cos_val = 1 - division(a^2, 2!) + division(a^4, 4!) - division(a^6, 6!) + division(a^8, 8!) - division(a^{10}, 10!)$

14: $sin_val = a - division(a^3, 3!) + division(a^5, 5!) - division(a^7, 7!) + division(a^9, 9!) - division(a^{11}, 11!)$

15: **end for**

16: **for** $r = \frac{3\pi}{2}; r < 2\pi; r++$ **do**

17: $a \leftarrow 2\pi - r$

18: $cos_val = 1 - division(a^2, 2!) + division(a^4, 4!) - division(a^6, 6!) + division(a^8, 8!) - division(a^{10}, 10!)$

19: $sin_val = a - division(a^3, 3!) + division(a^5, 5!) - division(a^7, 7!) + division(a^9, 9!) - division(a^{11}, 11!)$

20: **end for**

Algorithm 3 Pseudo-code for Sine and Cosine Taylor series - part 2

```
1: if r < π/2 then
2:     sin_val = sin_val
3:     cos_val = cos_val
4: end if
5: if π/2 < r < π then
6:     sin_val = sin_val
7:     cos_val = −cos_val
8: end if
9: if π < r < 3π/2 then
10:    sin_val = −sin_val
11:    cos_val = −cos_val
12: end if
13: if 3π/2 < r < 2π then
14:    sin_val = −sin_val
15:    cos_val = cos_val
16: end if
```

For Arctangent, the ratio test shows a radius of convergence is 1. Thus, at $x = 1$ and $x = -1$, the series converges. As x grows, the Taylor series slowly diverges and becomes a harmonic series. When equation 3.5 is used in the calculation, at $x < 1$ and $x > -1$, x^n limits the growth of $arctan(x)$ because x^n cannot be greater than 1. When $|x| > 1$, x^n grows rapidly as x increases. Therefore, equation 3.5 fails to work for $|x| > 1$ and can only apply to the situation when $|x| \leq 1$. Equation 3.5 calculates the output from 0 to $\frac{\pi}{4}$. The algorithm introduces equation 3.8 to extend the calculation range beyond 1. Since the limit of $arctan(x)$ is at $\pm\frac{\pi}{2}$, the second equation can calculate the output from $\frac{\pi}{4}$ to $\frac{\pi}{2}$. This equation can reduce the growth of each series term since x^n is the denominator. Equation 3.6 and equation 3.7 are the two extra equations used to increase the accuracy from 0.78125 to 1.5625. The next chapter explains the reason for adding the two equations. The following pseudo-code shows the algorithm of the Taylor series for Arctangent.

Algorithm 4 Pseudo-code for Arctangent Taylor series

Input: $radian\, r$, $limit$

Output: $atan_val$

1: **for** $r = 0; r < 0.78125; r++$ **do**

2: $a \leftarrow r$

3: $atan_val = division(a, 1) - division(a^3, 3) + division(a^5, 5) - division(a^7, 7) + division(a^9, 9)$

4: **end for**

5: **for** $r = 0.78125; r < 1; r++$ **do**

6: $a \leftarrow r$

7: $atan_val = 0.663+0.621(a-0.78125)-0.301(a-0.78125)^2+0.0663(a-0.78125)^3+ 0.0453(a - 0.78125)^4 - 0.0599(a - 0.78125)^5$

8: **end for**

9: **for** $r = 1; r < 1.5625; r++$ **do**

10: $a \leftarrow r$

11: $atan_val = \frac{\pi}{4} + division(2, a-1) - division(4, (a-1)^2) + division(12, (a-1)^3) - division(40, (a-1)^5) + division(48, (a-1)^6)$

12: **end for**

13: **for** $r > 1.5625; r < limit; r++$ **do**

14: $a \leftarrow r$

15: $atan_val = division(\pi, 2) - division(1, a) + division(3, 3a^3) - division(5, 5a^5) + division(7, 7a^7) - division(9, 9a^9)$

16: **end for**

5.2.2 CORDIC

The two pseudo-codes, Sine and Cosine, and Arctangent below show the procedure for calculating the CORDIC of Sine, Cosine, and Arctangent. The codes calculate 16 iterations. Therefore, the value of $cos(45°)*cos(26.57°)*...cos(arctan(2^{-15}))$ is 0.6072529, and in $s1.15$, it is 19898. In both pseudo-codes, the $arctan(2^{-i})$ on the 11th line refers to

the values in table 4.1.

Algorithm 5 Pseudo-code for Sine and Cosine in CORDIC - part 1

Input: $degree$

Output: sin_val, cos_val

1: $flag \leftarrow 1$

2: $x_val \leftarrow 0$

3: $y_val \leftarrow 0$

4: $x_temp \leftarrow 19898$

5: $y_temp \leftarrow 0$

6: $z_val \leftarrow degree$

7: $result \leftarrow 0$

8: **for** $i = 0; i < 16; i++$ **do**

9: $\quad x_val \leftarrow x_temp - flag * (y_temp \gg i)$

10: $\quad y_val \leftarrow y_temp + flag * (x_temp \gg i)$

11: $\quad result \leftarrow z_val - flag * arctan(2^{-i})$ ▷ $arctan(2^{-i})$ refers to the values in table 4.1

12: $\quad$ **if** $result > 0$ **then**

13: $\qquad y_temp \leftarrow y_val$

14: $\qquad x_temp \leftarrow x_val$

15: $\qquad z_val \leftarrow result$

16: $\qquad flag \leftarrow 1$

17: $\quad$ **else**

18: $\qquad y_temp \leftarrow y_val$

19: $\qquad x_temp \leftarrow x_val$

20: $\qquad z_val \leftarrow result$

21: $\qquad flag \leftarrow -1$

22: $\quad$ **end if**

23: **end for**

Algorithm 6 Pseudo-code for Sine and Cosine in CORDIC - part 2

```
1: if degree < π]/2 then
2:     sin_val = y_val
3:     cos_val = x_val
4: end if
5: if π/2 < degree < π then
6:     sin_val = y_val
7:     cos_val = −x_val
8: end if
9: if π < degree < 3π/2 then
10:    sin_val = −y_val
11:    cos_val = −x_val
12: end if
13: if 3π/2 < degree < 2π then
14:    sin_val = −y_val
15:    cos_val = x_val
16: end if
```

Algorithm 7 Pseudo code for Arctangent in CORDIC - part 1

Input: a ▷ x and y are two edges of the triangle

Output: $atan_val$

```
1: flag ← −1
2: x_val ← 0
3: y_val ← 0
4: x_temp ← 1024
5: y_temp ← a
6: z_val ← 0
7: result ← 0
8: for i = 0; i < 16; i + + do
9:     x_val ← x_temp − flag ∗ y_temp ∗ 2^{−i}
10:    y_val ← y_temp + flag ∗ x_temp ∗ 2^{−i}
11:    result ← z_val − flag ∗ arctan(2^{−i}) ▷ arctan(2^{−i}) refers to the values in table 4.1
12:    if y_val < 0 then
13:        y_temp ← y_val
14:        x_temp ← x_val
15:        z_val ← result
16:        flag ← 1
17:    else
18:        y_temp ← y_val
19:        x_temp ← x_val
20:        z_val ← result
21:        flag ← −1
22:    end if
23: end for
24: atan_val ← z_val
```

5.3 Exponential Function

Equation 3.9 is the Taylor series for e^x. Since the Taylor series calculates the summation of each term, the accuracy of the series bases on the number of terms. When the number of terms is small, the result of the equation becomes inaccurate even if the input value is small. Thus, more amount of terms increase the accuracy of output. This paper uses ten series terms in the function to increase the accuracy. However, ten series terms can only have a guaranteed accuracy when input is around 2. The deviation is still significant when input is over 2. Increasing the number of terms can solve the problem, but it also creates an overflow in calculating higher powers. Thus, this paper chooses to use ten series terms in the calculation. The following pseudo-code shows the algorithm of the Taylor series for the exponential function.

Algorithm 8 Pseudo-code for Exponential function Taylor series

Input: r, $limit$

Output: exp_val

1: $count \leftarrow -1$

2: **for** $r = 0; r < limit; r++$ **do**

3: $a \leftarrow r$

4: $exp_val = 1 + division(a, 1) + division(a^2, 2!) + division(a^3, 3!) + division(a^4, 4!) + division(a^5, 5!) + division(a^6, 6!) + division(a^7, 7!) + division(a^8, 8!) + division(a^9, 9!) + division(a^{10}, 10!)$

5: **end for**

5.4 Natural Logarithm Function

Equation 3.10 and equation 3.11 are the Taylor series for natural logarithm $ln(1+x)$. The Taylor series for natural logarithm diverges at $|x| > 1$, therefore, it only approximates the natural log at $-1 < x \leq 1$. $-1 < x < 0$ and $0 \leq x < 1$ have different Taylor

series. Since $ln(1 + x)$ approaches negative infinity when x approaches -1, the results for 3.11 become increasingly inaccurate, while equation 3.7 approaches positive infinity as x grows. Thus, the result for equation 3.10 becomes inaccurate as x approaches 1. The following pseudo-code shows the two algorithms of the Taylor series for the natural logarithm function.

Algorithm 9 Pseudo-code for Natural log function Taylor series

Input: r, $limit$

Output: log_val

1: **for** $r = -1; r < 0; r++$ **do**

2: $a \leftarrow r$

3: $log_val = -division(a, 1) - division(a^2, 2) - division(a^3, 3) - division(a4, 4) - division(a^5, 5) - division(a^6, 6) - division(a^7, 7) - division(a^8, 8)$

4: **end for**

5: **for** $r = 0; r \leq 1; r++$ **do**

6: $a \leftarrow r$

7: $log_val = division(a, 1) - division(a^2, 2) + division(a^3, 3) - division(a4, 4) + division(a^5, 5) - division(a^6, 6) + division(a^7, 7) - division(a^8, 8)$

8: **end for**

5.5 Square Root

In this paper, the square root function uses binary search as the basic algorithm. Since the input range is from 0 to 65535, the start position is set to 0 and the end position is set to 65535 initially. The AsAP3 platform supports 32-bit arithmetic operation and the maximum possible bits for $mid * mid$ is 32 bits. Thus, to maintain the same number of bits, the input value also multiplies by 65535 before the comparison between the input and $mid * mid$. The implementation only compares the first 16 bits. Therefore, 16-bit

shifts apply to both $mid * mid$ and input value. The following pseudo-code shows the algorithm for the square root function,

Algorithm 10 Pseudo-code for Square root

Input: *output*

Output: *input*

```
1:  start ← 0
2:  end ← 65535
3:  mid ← 0
4:  ans ← 0
5:  while start <= end do
6:      mid ← (start + end) >> 1
7:      if (mid >> 1) * (mid >> 1) >> 14 == (input * 65535) >> 16 then
8:          ans ← mid
9:          break
10:     end if
11:     if (mid >> 1) * (mid >> 1) >> 14 < (input * 65535) >> 16 then
12:         ans ← start
13:         start = mid + 1
14:     end if
15:     if (mid >> 1) * (mid >> 1) >> 14 > (input * 65535) >> 16 then
16:         end ← mid − 1
17:         ans ← end
18:     end if
19: end while
```

Note: for $mid * mid$ part, the actual code is $(mid >> 1) * (mid >> 1) >> 14$ instead of $mid * mid >> 16$. It happens because the former code can improve the final output accuracy compared with the later code.

5.6 LRN

In this paper, β is set to 0.75, which is a $\frac{3}{4}^{th}$ power of the value. This paper uses square root to calculate the $\frac{3}{4}^{th}$ power. Based on $a^{\frac{3}{4}} = a^{\frac{1}{2}} * a^{\frac{1}{4}}$, this paper first calculates $a^{\frac{1}{2}}$, takes the square root of $a^{\frac{1}{2}}$ to compute $a^{\frac{1}{4}}$, and multiplies them together to get $a^{\frac{3}{4}}$. The following pseudo-code shows the algorithm for the LRN function,

Algorithm 11 Pseudo-code for Local Response Normalization

Input: $3D\, array\, A, channel, height, width$

Output: $3D\, array\, B$

1: $size \leftarrow height * width * channel$
2: $local_size \leftarrow 5$
3: $alpha \leftarrow 1$
4: $beta \leftarrow 0.75$
5: $bias \leftarrow 512$
6: **for** $i = 0; i < channel; i++$ **do**
7:     **for** $j = 0; j < height; j++$ **do**
8:         **for** $l = 0; l < width; k++$ **do**
9:             **for** $l = max(0, local_size >> 1); l \leq min(channel - 1, (i + local_size) >> 1); l++$ **do**
10:                 $position \leftarrow height * width * l + width * j + k$
11:                 $value \leftarrow value + A[position]^2$
12:             **end for**
13:             $B[position] \leftarrow division((A[position], bias + alpha * value)^{beta})$
14:         **end for**
15:     **end for**
16: **end for**

Chapter 6

Analysis of Implemented Functions

This chapter records and analyzes the simulation output for each implemented function. It lists the simulation statistics of the implemented functions including Trigonometric functions, Exponential function, Natural Logarithm function, square root function and LRN function in several tables. The tables contain the difference between simulation value and reference value, SNR value, and throughput for each implemented function. It also makes comparison graphs for the simulated results and reference values for each function. Furthermore, this chapter presents some more tests and researches for some of the simulated functions.

This thesis has a test set for each implemented function. Each test set includes two functions: the simulated and the standard functions. This paper first writes the programs on a C file and then transfers them into the KiloCore C file. The KiloCore chip generates the simulated outputs. The standard functions are $sin(x)$, $cos(x)$, $atan(x)$, $exp(x)$, $log(x)$ and square root in C. For the LRN function, the standard function is written in C. This paper uses the generated outputs from standard functions as a golden reference. This paper uses MATLAB to compare the simulated results with the golden reference. Each test records the maximum difference and Signal-to-Noise Ratio(SNR). The maximum difference is the largest difference between the simulation and reference

values existing in each test set. SNR calculates as follows,

$$SNR = 10 * log_{10}(\frac{energy}{energydiff}) \tag{6.1}$$

In this case,

$$energy = \sum_{n=1}^{i} output_i^2 \tag{6.2}$$

$$energy_{diff} = \sum_{n=1}^{i} (expected_i - output_i)^2 \tag{6.3}$$

The SNR value represents the bit accuracy for a function and is an important value to determine the accuracy of the simulation output in this paper. Each extra bit increases the range of SNR by 6.02. Therefore, for the 16 bit output model, the theoretical range for SNR is from 0 to 96.32.

This thesis also records the number of throughput for each function. It measures the volume of data that passes through the function per second, and the unit of measurement for throughput is $MWords/s$.

6.1 Trigonometric Functions

The tables below show the comparison results for Trigonometric functions. Table 6.1 presents the differences and SNR value for Sine, Cosine, and Arctangent in the Taylor series, and table 6.2 shows the differences and SNR value for Trigonometric functions in CORDIC.

	Sine	Cosine	Arctan
Max diff	1.534	1.533	1356.07
SNR	92.725	92.725	46.96
Throughput (MWords/s)	7.545	7.545	3.083

Table 6.1: Results of Trigonometric functions in Taylor series

	Sine	Cosine	Arctan
Max diff	5.56	5.58	2.656
SNR	80.873	80.87	85.47
Throughput (MWords/s)	3.553	3.553	3.717

Table 6.2: Results of Trigonometric functions in CORDIC

6.1.1 Analysis for Trigonometric functions in Taylor series

The graphs below in Figure 6.1 present an intuitive view of the comparison among the simulated outputs and reference outputs of the Sine function. Figure 6.1 includes 4 graphs, and they display the simulated Sine value, reference Sine value, the difference between simulation and reference, and the ratio between simulation and reference. For the ratio graph, a value closer to 1 shows a satisfactory ratio between the simulated and reference values.

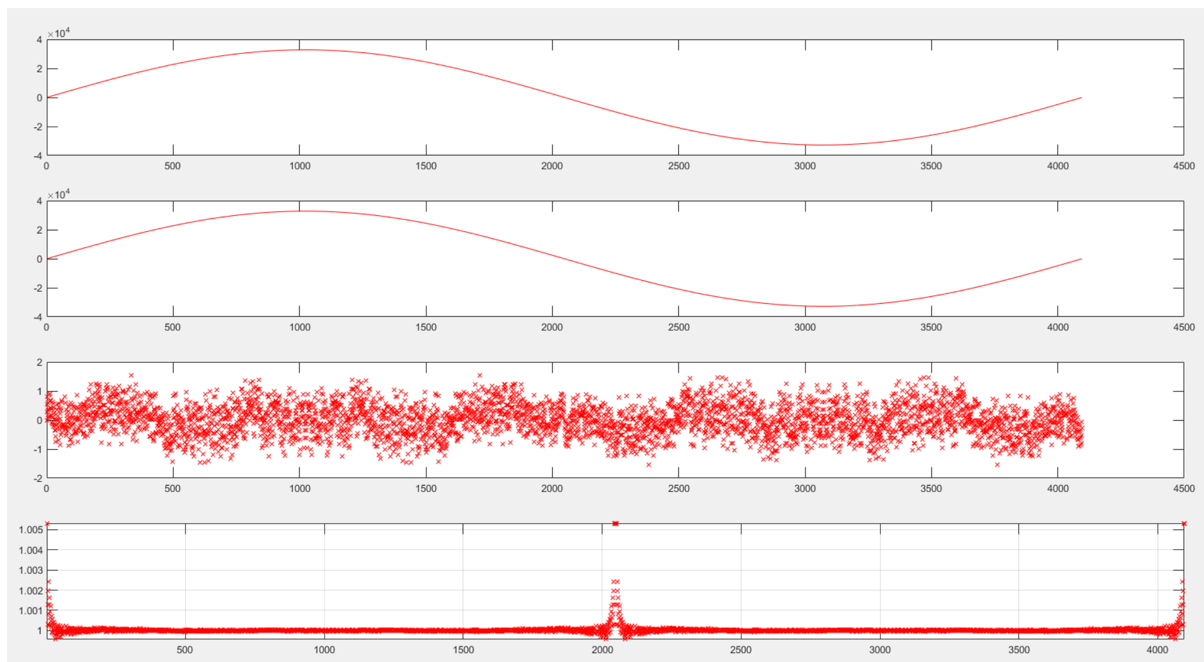

Figure 6.1: Graph for Taylor series, Sine and Cosine

Since a 90° phase shift of a Sine wave is a Cosine wave, this paper only presents the simulation output for the Sine function as a representation for both Sine and Cosine

functions. Sine and Cosine functions have small differences and ratios between the simulated value and reference value. As inputs transform and limit to 0 to $\frac{\pi}{2}$, the deviation from the desired value decreases when input increases. The SNR value indicates a 14 to 15 bits accuracy, and this proves the reliability of the two Taylor series functions.

The graphs below are the 4 Arctangent graphs generated by MATLAB. The generated graphs use only equation 3.5 and equation 3.8.

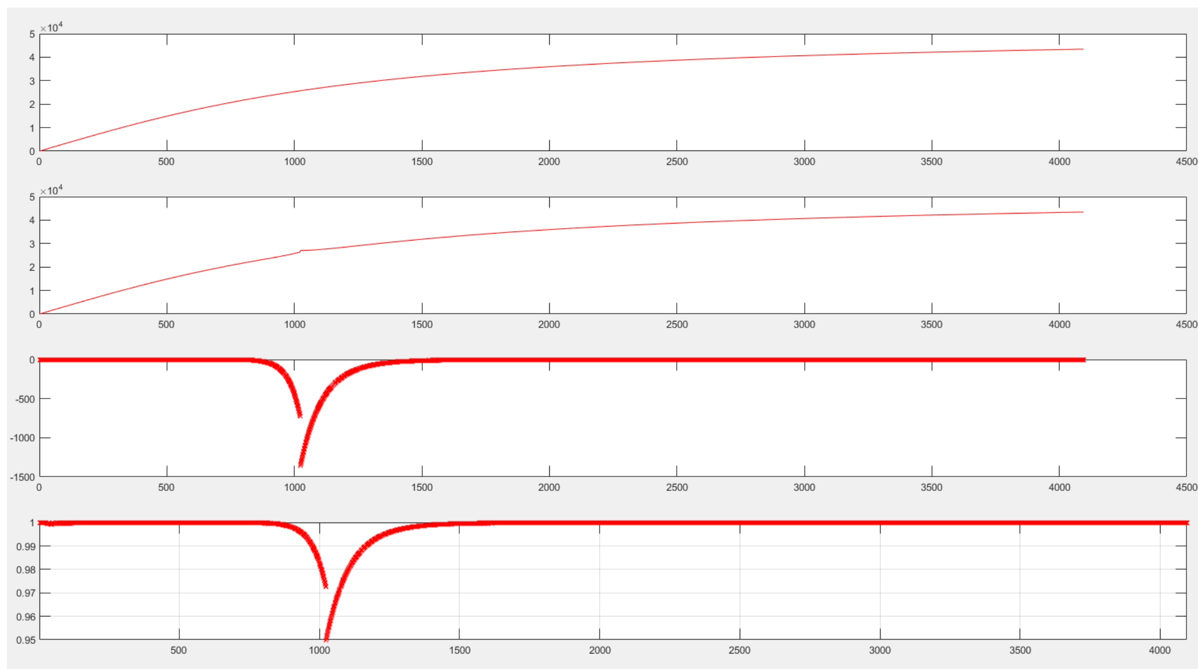

Figure 6.2: Graph for Taylor series, Arctangent

For Arctangent, the average difference, and maximum difference are much higher than the other trigonometric functions, while the SNR value is much lower. The SNR value shows the bit accuracy is only 7 to 8 bits. From the graphs, it is obvious that there is a clear gap around $X = 1024$. The gap starts at approximately $x = 800$ and ends after $x = 1400$. The possible reason for the large difference is the deviation of the Arctangent functions at the edge of the X intervals. To do a deep study, another simulation for Arctangent completes with different input ranges. The new simulation removes the input from $X = 800$ to $X = 1600$, and the graphs below show the new graphs generated by MATLAB.

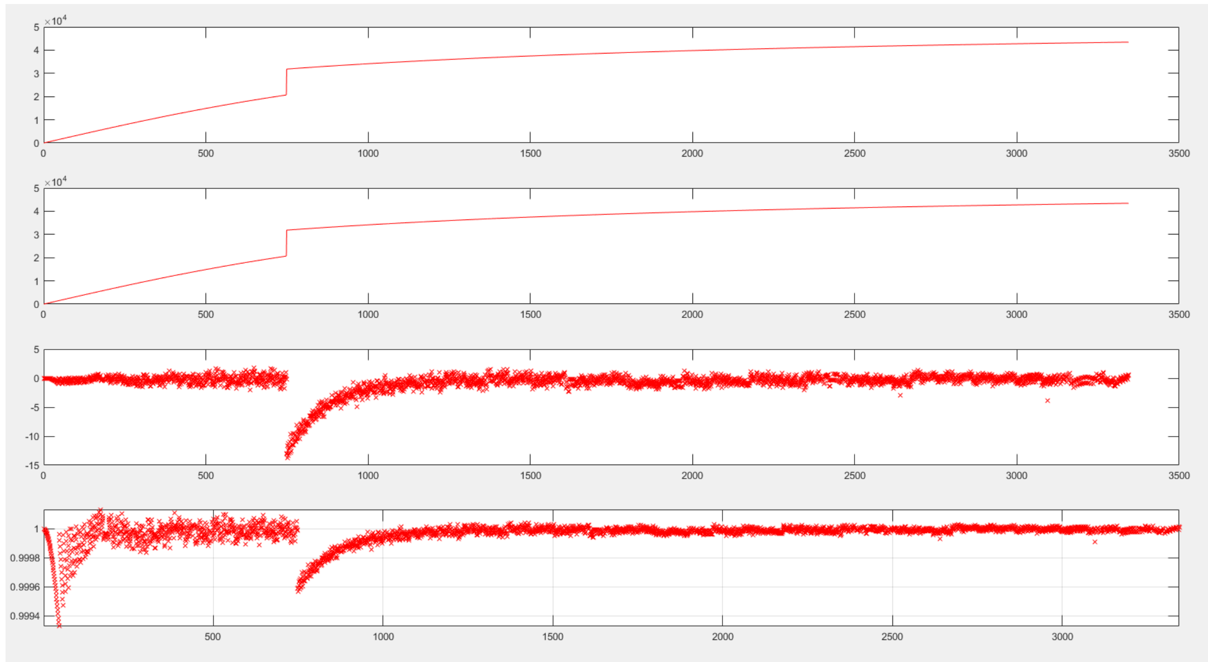

Figure 6.3: Graph for Taylor series, Arctangent

For the new simulation, the maximum difference is 13.782 and the SNR value is 85.361. Therefore, without the inputs range from $X = 800$ to $X = 1600$, the bit accuracy can increase to 12 to 14 bits. Thus, to obtain a better bit accuracy for Arctangent, this paper performs another two Taylor series expansions at $X = 800$ and $X = 1024$. The two new Taylor series equations are equation 3.6 and equation 3.7. With these two new equations, new simulation graphs for the Arctangent function present as below,

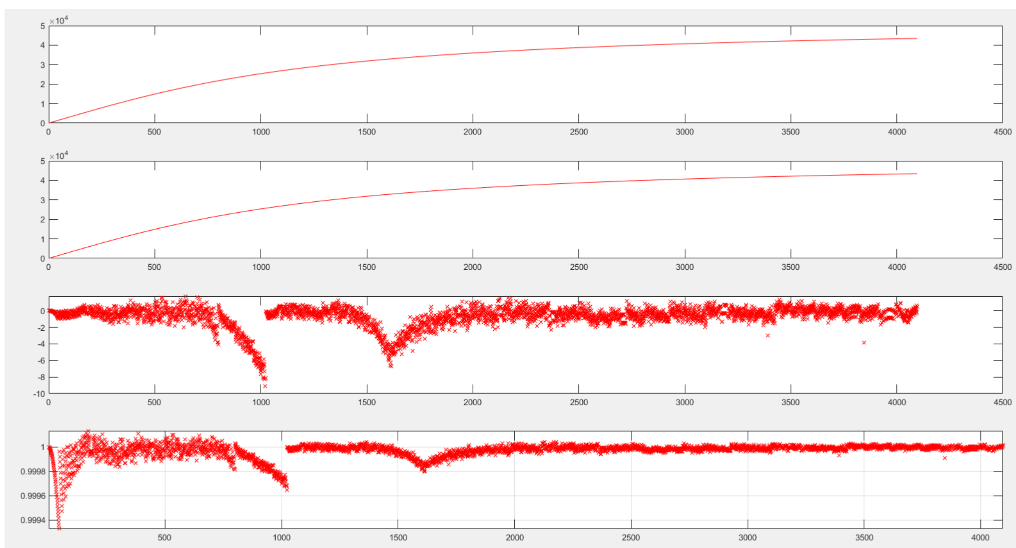

Figure 6.4: Graph for Taylor series, Arctangent

For the new simulation, the maximum difference decreases to 9.104 and the SNR value increases to 87.234. Thus, with the two newly added equations, the bit accuracy increases to 14 to 15 bits. The high SNR value shows the 4 piece-wise equations can provide an acceptable accuracy in numerical calculation for the Arctangent function. Therefore, this paper uses the 4 piece-wise equations in computing the Arctangent value for the Taylor series.

6.1.2 Analysis for Trigonometric functions in CORDIC

The graphs below present the difference and ratio comparison between the simulated and reference values for the CORDIC function of Sine.

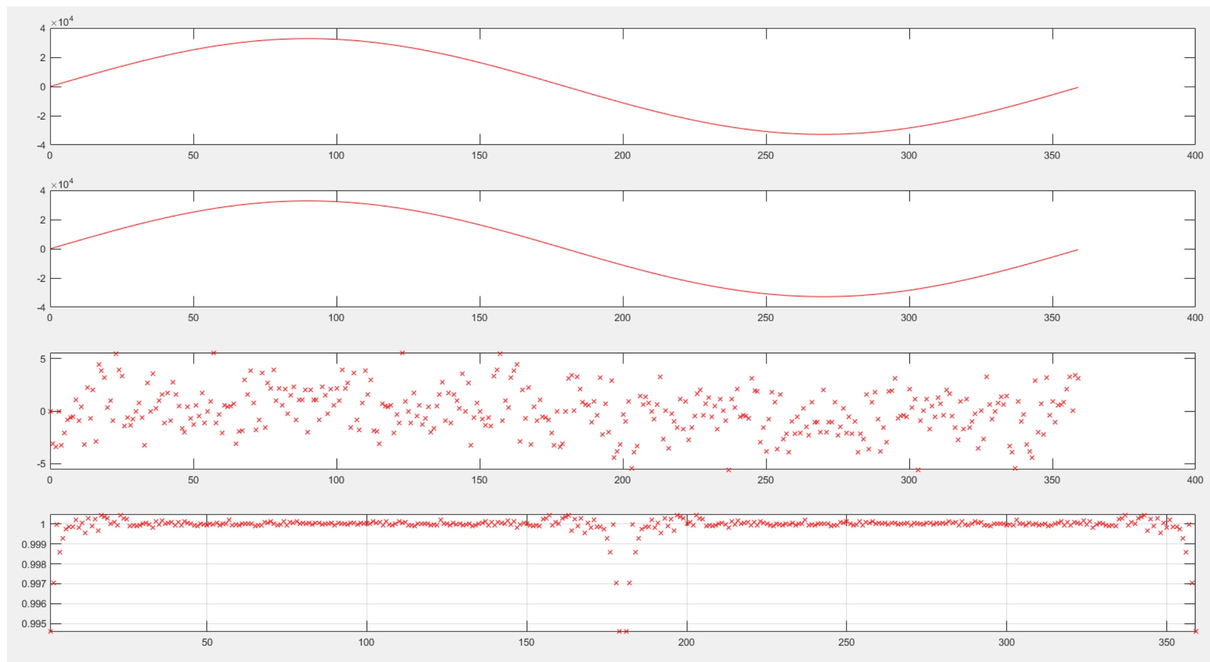

Figure 6.5: Graph for CORDIC, Sine

The Sine function for CORDIC has a small difference and ratio between the simulated value and reference value. The maximum difference of 5.56 and SNR value of 80.87 present a relatively accurate model for the Sine and Cosine functions. The bit accuracy for both CORDIC functions is 13 bits. The relatively small maximum difference and high SNR value prove that the CORDIC functions for Sine and Cosine are accurate.

The graphs below present the difference and ratio comparison for the CORDIC function of Arctangent.

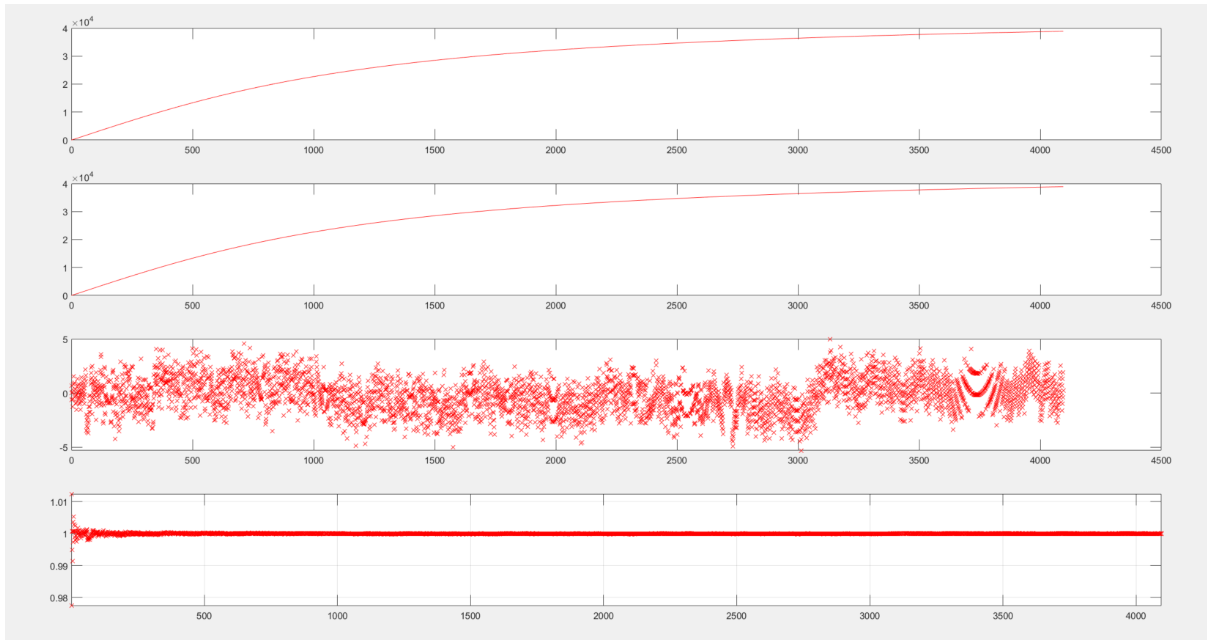

Figure 6.6: Graph for CORDIC, Arctangent

The maximum difference of 2.656 and SNR value of 85.47 presents an accurate model for the CORDIC function of Arctangent. The bit accuracy for the Arctangent function can reach 14 bits. The high SNR value shows an accurate model for calculating the Arctangent function.

6.1.3 Data Analysis for Trigonometric Functions

This section compares the performance of the Taylor series and CORDIC function among Sine, Cosine, and Arctangent functions. The tables below present the collected data for Sine and Arctangent functions in the previous chapter. Since the Cosine function is similar to the Sine function, this section uses only the Sine function for comparison.

	Sine	
	Taylor series	CORDIC
Max difference	1.534	5.56
SNR	92.725	80.873
Number of input	4096	360
Throughput (MWords/s)	7.545	3.553

Table 6.3: Comparison for Sine

	Arctangent	
	Taylor series	CORDIC
Max difference	9.104	2.656
SNR	87.234	85.47
Number of input	4096	4096
Throughput (MWords/s)	3.083	3.717

Table 6.4: Comparison for Arctangent

Table 6.3 demonstrates Taylor series has a better performance than the CORDIC function in both maximum difference and SNR value. For the efficiency comparison, the Taylor series function also has a higher number of throughput in Mwords per second. Thus, the Taylor series has a better performance than CORDIC in calculating Sine and Cosine functions.

From table 6.4, it is clear that the CORDIC function has a smaller maximum difference, and the SNR value for both the Taylor series and the CORDIC function is close to each other. The throughput per second for the Taylor series is smaller than CORDIC. The complicated equation in computing the Arctangent value for the Taylor series is the possible cause for the smaller throughput per second. Equation 3.8 has x values in the denominator, and it slows down the speed of division calculation.

Both functions have the same number of inputs and the input range is also the same from 0 to 4. For the amount of throughput per second, the Taylor series has a better throughput per second in Sine and Cosine, and CORDIC has a better throughput per second in Arctangent.

6.2 Exponential Function

The table below displays the results for the Exponential function in the Taylor series algorithm.

Max difference	69.544
SNR	64.671
Throughput (MWords/s)	4.891

Table 6.5: Results of Exponential function in Taylor series

The graph below shows the graphs generated by MATLAB.

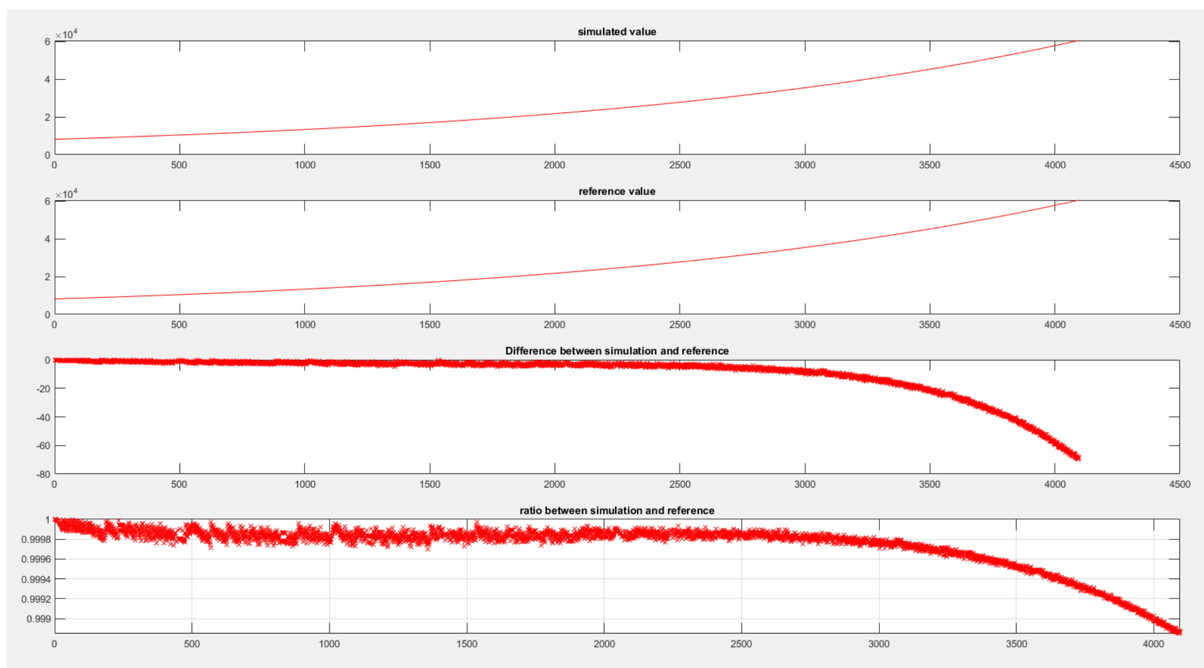

Figure 6.7: Graph for Taylor series, Exponential

For the exponential function, the average difference and maximum difference are

smaller than the Arctangent function, whereas the SNR value is larger than Arctangent. The SNR value for the exponential function shows a bit accuracy of 11 bits. The difference graph shows that the difference between simulation and reference results increases when the input gets larger. The maximum difference, 69.544, appears at $X = 4095$. The divergence of the Taylor series equation possibly causes the increasing difference. The throughput per second is 4.891. In the Taylor series, this throughput is slower than Sine and Cosine but faster than the Arctangent function.

To study the effect of input value, this paper completes another simulation for a smaller range of inputs. The new input range is from 0 to 1, and the number of inputs is still 4096. The new simulation graphs for the exponential function show as below,

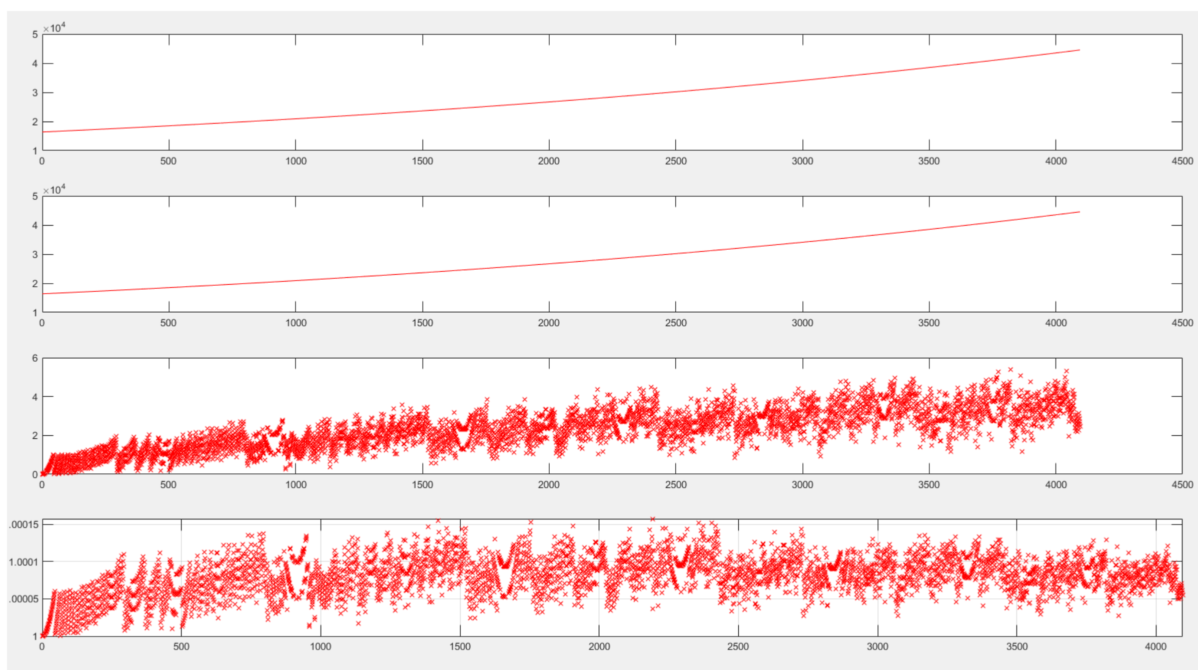

Figure 6.8: Graph for Taylor series, Exponential

From the above graph, it is clear that the difference between the simulated and reference results still increases as the input grows larger. However, as the input range decreases to 0-1, the maximum difference decreases and the SNR value increases. The new maximum difference, 2.323, and the new SNR value, 81.239, show a 13 to 14 bit accuracy. The increase in the difference between the simulated and reference values demonstrates that the Taylor series equation for the exponential function is accurate and reliable when the input value is small enough.

6.3 Natural Logarithm Function

The tables below show the results for the Natural Logarithm function in the Taylor series algorithm.

	Natural log	
	starts at x = 1	starts at x = 500
Max difference	80658.432	876.544
SNR	9.971	34.646

Table 6.6: Results of Natural Logarithm function in Taylor series

	ln(1+x)	ln(1-x)
number of inputs	2048	2048
Throughput (MWords/s)	2.155	2.163

Table 6.7: Throughput Results of Natural Logarithm function in Taylor series

The Taylor series of natural logarithm function collects two data sets for analysis. The first data set collects at $X = -1$ and the second data set collects at $X = -0.75$. For the $X = -1$ data set, the maximum difference of 80658 and SNR value of 9.971 show an unreliable simulation model. For the input value starting at $X = -0.75$, the maximum difference of 876.544 and SNR value of 34.646 indicate an improved but still unreliable model compared with the first data set. The graphs below show the difference and ratio for the simulated and reference values.

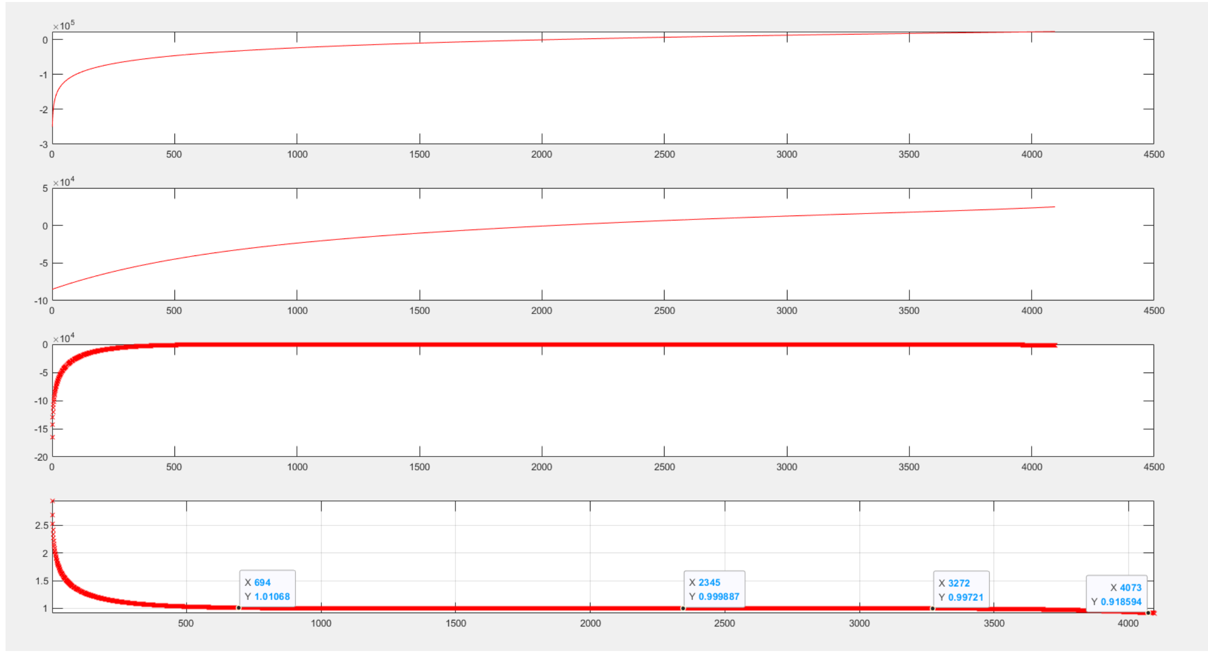

Figure 6.9: Graph for Taylor series, Natural Logarithm

From the above figure, it is obvious that there is a huge difference between the simulated value and reference value from input $X = 0$ to $X = 500$. As stated, $ln(1 + x)$ approaches negative infinity as x approaches -1. Therefore, the Taylor series is hard to simulate the expected value when the input is minimal. From some specific data points in the ratio graph shown above, the ratio between simulated output and reference value decreases as the input value becomes larger. Therefore, the difference grows larger again as the input value increases. The table 6.7 shows the throughputs per second for the natural log equation $ln(1 + n)$ and $ln(1 - n)$. The throughputs per second for both equations are close to each other and slower than other Taylor series functions. This is probably caused by the unoptimized code of shift division.

To investigate the effect of the input range, this paper chooses a new input range for the natural logarithm function. The graphs below show the difference and ratio of simulated and reference values for the new input range.

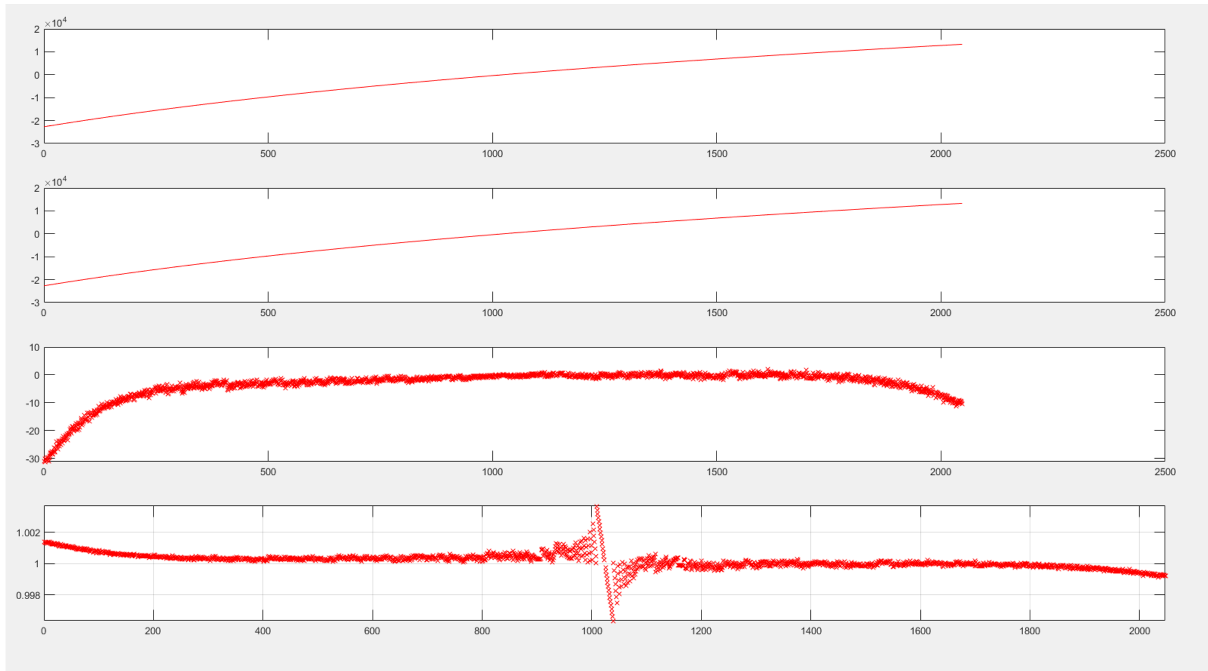

Figure 6.10: Graph for Taylor series, Natural Logarithm with input from $X = -0.5$ to $X = 0.5$

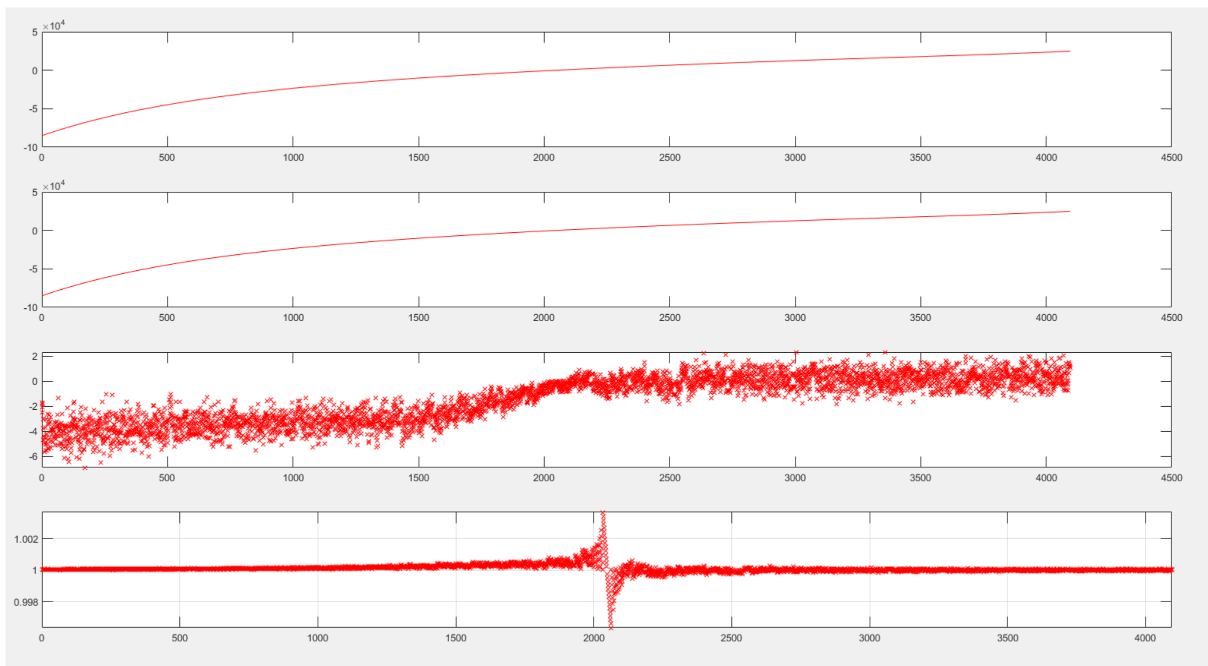

Figure 6.11: Graph for Taylor series, Natural Logarithm with 4096 inputs from $X = -0.5$ to $X = 0.5$

The new data points collect from $X = -0.5$ and end at $X = 0.5$. The above graphs demonstrate the difference and ratio of simulated value and reference value decrease significantly compared with the graph 6.9. The new input range for 2048 inputs has an

SNR value of 64.17, the accuracy reaches 10 to 11 bits. In addition, the new input range for 4096 inputs has an SNR value of 81.465, and the bit accuracy is close to 14 bits.

To investigate the effect of the number of terms on the natural logarithm function, this paper completes two more simulations based on the number of terms. The table below shows the SNR values for different number of terms. The input range for the new simulations is from 0 to 4095.

number of term	SNR value
7 terms	9.971
13 terms	12.67
20 terms	14.97

Table 6.8: SNR results for Natural logarithm function with different number of terms

The above table shows that the SNR value can gain a small increase with an increasing number of Taylor polynomial terms. The increase in SNR values from the above tests shows that the accuracy of the natural logarithm function depends on the input range as well as the number of Taylor polynomial terms. However, adding more terms does not have as much effect on SNR value as narrowing down the input range. Therefore, for the natural logarithm function, the simulation result is accurate if the input value is closer to 0.

6.4 Square Root

Table 6.9 shows the difference and SNR value for the square root function.

Max difference	0.998
SNR	89.951
Throughput (MWords/s)	1.485

Table 6.9: Results of Square Root function

The graphs below present the difference and ratio comparison for the Square Root function.

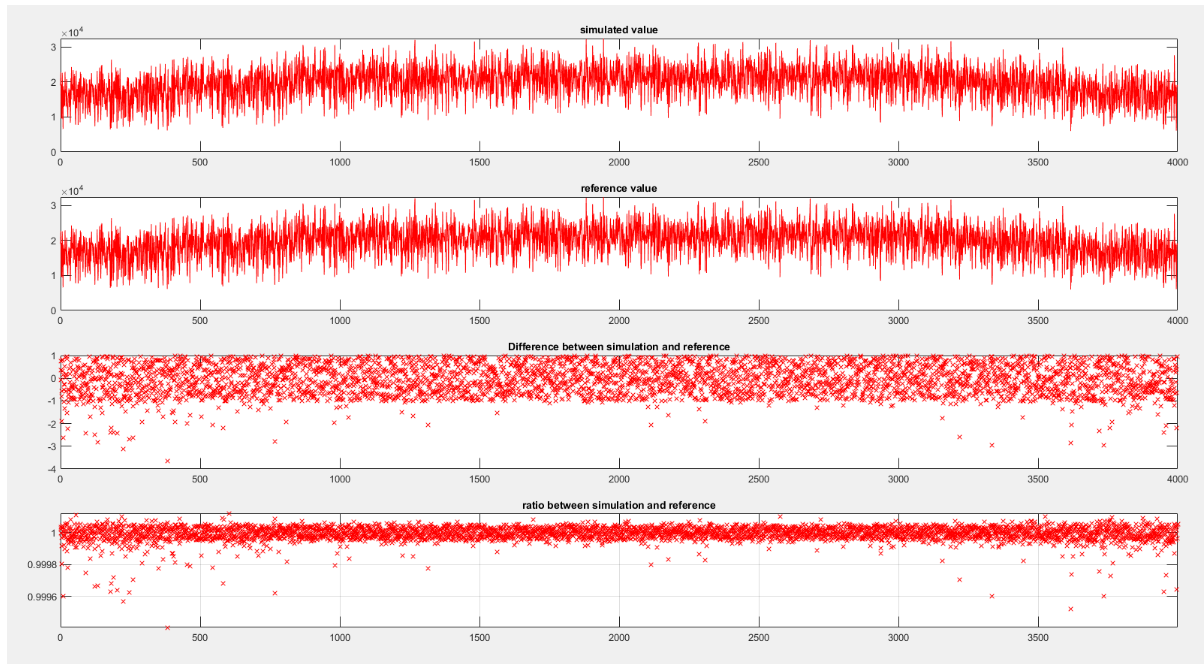

Figure 6.12: Graph for Square Root function

The graphs above in Figure 6.12 provide the four graphs for a visualized comparison between the simulated results and golden reference. Table 6.9 and figure 6.12 show the square root function provides an accurate calculation. The maximum difference is 0.998 and the SNR is 89.951. It means that the square root function has a 15-bit accuracy. The number of throughput per second is slower than other implemented functions presented previously. The use of binary search in the function is a possible reason for the decrease in speed. The lengthy process of binary search impedes the computing speed of the Square Root function.

For using $(mid >> 1) * (mid >> 1) >> 14$ instead of $mid * mid >> 16$ in the actual calculation for $mid * mid$, this paper records the SNR value for both calculating methods. The former method has an SNR value of 89.951, while the latter method has an SNR value of 85.813. Therefore, SNR value increases when using the $(mid >> 1) * (mid >> 1) >> 14$ calculation method. Thus, this paper uses this method to calculate the square root.

6.5 LRN

Table 6.10 presents the differences and SNR value for the simulated results. This thesis conducts two simulations with different numbers of inputs.

	1000 inputs	4000 inputs
Average difference	0.976	0.897
Max difference	16.759	28.537
SNR	77.702	78.252
Throughput (MWords/s)	1.015	1.015

Table 6.10: Results for LRN function

The testing parameters for 1000 inputs are $channel = 10$, $width = 10$ and $height = 10$. For 4000 inputs, the testing parameters are $channel = 10$, $width = 20$, and $height = 20$. The testing program randomly generates input values from 0 to 63, and it uses the $srand()$ function from C to generate retrievable data randomly. The random seed for the function is 5.

The graphs below present the difference and ratio comparison for 4000 inputs of the LRN function.

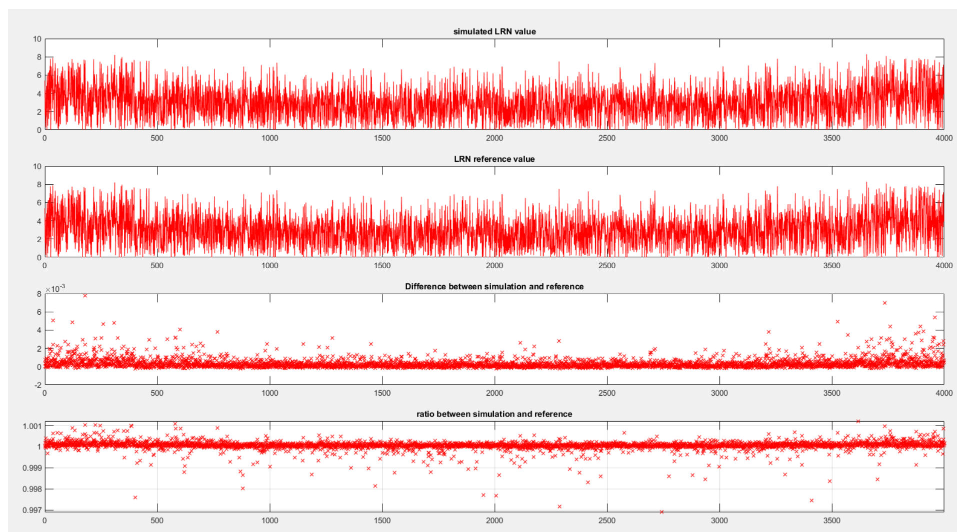

Figure 6.13: Graph for LRN

For 1000 inputs, the maximum difference is 16.759 and the SNR value is 77.702. For 4000 inputs, the maximum difference is 28.537 and the SNR value is 78.252. The high SNR value presents an accurate model for the LRN function, the bit accuracy for the LRN function is 13 bits. The calculating procedure contains a square root calculation and a fourth root calculation. Therefore, a 13 accurate bit is acceptable for a fixed-point arithmetic function. The number of throughput per second for the LRN function is the slowest among the implemented functions. The primary cause of the slow speed lies in the utilization of the square root within the LRN function.

Chapter 7

Thesis Summary and Future Work

7.1 Thesis Summary

This thesis summarizes and implements several arithmetic functions in basic numerical calculations. These functions include Trigonometric functions, Exponential function, Natural Logarithm function, Square Root function, and LRN function. It reviews backgrounds for these functions and it also introduces the AsAP3 as the implementing platform for these functions.

Furthermore, the thesis displays the algorithms for the functions and implements shift division in these functions. It designs tests for each algorithm to check the accuracy of the implemented function and analyzes the simulated results for each test. The analysis uses maximum difference, SNR value, and throughput as parameters to compare the performance of each function.

The simulated results demonstrate that the Sine, Cosine, and Exponential functions in the Taylor series have satisfactory simulation results. The Arctangent equations have a less accurate model when using only two original equations. After adding two new Taylor series equations, the simulated results of Arctangent become acceptable. In comparison, the Natural Logarithm function in the Taylor series does not have a good performance if the input range begins at $X = -1$, the simulated value is accurate when the input range narrows down and approaches 0. This thesis also studies the precision

of the CORDIC algorithm and compares the accuracy of Taylor series and CORDIC. The performance of the Taylor series exceeds the CORDIC function in both accuracy and efficiency. Lastly, this paper analyzes the performance of the Square Root and LRN functions. The performance is adequate in producing accurate simulated results.

7.2 Future works

The implementation of the functions has acceptable accuracy. However, the current versions of functions use the primal design of the division and square root calculating functions. The current division is the shift division, and the square root uses binary search to complete the calculation. There needs to have an upgrade to both functions to increase the efficiency of the functions.

Finding a more advanced model for the division is a solution to increase the number of throughput per second. The current implementation takes many operations on branch searches due to the use of the shift division method. A division method with less branch search can increase the efficiency of the calculation. The Sweeney-Robertson-Tocher (SRT) division is a feasible method to be considered.[17]. Sweeney, Robertson[18] and Tocher[19] designed the algorithm. The algorithm uses redundant binary representation for the quotient. Thus, the choice of the quotient digits is fault-tolerant, and this allows a small space for choosing the quotient digits. Since it does not require a full subtraction in calculating the quotient, the calculation speed increases. It means the precision of calculating results can be accomplished with more decimal values. These division methods do not require much branch operations, thus, the number of operations in each calculation reduces. This reduction in the operation number does not affect the accuracy of the calculation.

Another possible improvement for the LRN function is a more advanced square root calculation method. Goldschmidt's algorithm is an algorithm to calculate square root [20]. This algorithm usually calculates square roots in floating-point numbers, but it also has the functionality of calculating fixed-point numbers. [21]. This algorithm

develops on Newton-Raphson's algorithm and it can calculate both the square root and inverse of square root at the same time. This method can take a $32 - bit$ fixed-point input value with a $16 - bit$ decimal value. The method requires fewer branch checks and operations than the binary search method. Therefore, it can reduce the number of operations during the calculation.

Bibliography

[1] B.M. Baas. A parallel programmable energy-efficient architecture for computationally-intensive dsp systems. In *The Thrity-Seventh Asilomar Conference on Signals, Systems & Computers, 2003*, volume 2, pages 2185–2189 Vol.2, 2003.

[2] Bevan Baas, Zhiyi Yu, Michael Meeuwsen, Omar Sattari, Ryan Apperson, Eric Work, Jeremy Webb, Michael Lai, Tinoosh Mohsenin, Dean Truong, and Jason Cheung. Asap: A fine-grained many-core platform for dsp applications. *IEEE Micro, 27(2):34–45*, March 2007.

[3] Sajib Mitra and Ahsan Raja Chowdhury. Optimized logarithmic barrel shifter in reversible logic synthesis. *Proceedings of the IEEE International Conference on VLSI Design*, 2015:441–446, 02 2015.

[4] Steve Arar. An introduction to the cordic algorithm, May 31, 2017. Last accessed 11 November 2022.

[5] Aqeel Anwar. Difference between local response normalization and batch normalization., 2019, June 18. Last accessed 11 November 2022.

[6] Xu Pan, Luis Gonzalo Sanchez Giraldo, Elif Kartal, and Odelia Schwartz. Brain-inspired weighted normalization for cnn image classification. *bioRxiv*, 2021.

[7] Lorenzo Putzu, Luca Piras, and Giorgio Giacinto. Convolutional neural networks for relevance feedback in content based image retrieval: A content based image retrieval system that exploits convolutional neural networks both for feature extraction and for relevance feedback. *Multimedia Tools and Applications*, 79, 10 2020.

[8] Prateek Joshi. What is local response normalization in convolutional neural networks, APRIL 5, 2016. Last accessed 11 November 2022.

[9] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A 32-nm 1000-processor computational array. *IEEE Journal of Solid-State Circuits (JSSC)*, 52(4):891–902, April 2017.

[10] Aaron Stillmaker, Brent Bohnenstiehl, and Bevan Baas. The design of the kilocore chip. In *ACM/IEEE Design Automation Conference*, Austin, TX, Jun. 2017.

[11] Ryan Apperson, Zhiyi Yu, Michael Meeuwsen, Tinoosh Mohsenin, and Bevan Baas. A scalable dual-clock FIFO for data transfers between arbitrary and haltable clock domains. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 15(10):1125–1134, October 2007.

[12] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. KiloCore: A fine-grained 1,000-processor array for task parallel applications. *IEEE Micro*, 37(2):63–69, March 2017.

[13] Zhiyi Yu and Bevan M. Baas. High performance, energy efficiency, and scalability with gals chip multiprocessors. *IEEE Transactions on Very Large Scale Integration Systems (TVLSI)*, 17(1):66–79, January 2009.

[14] Zhiyi Yu and Bevan M. Baas. Implementing tile-based chip multiprocessors with gals clocking styles. In *IEEE International Conference of Computer Design (ICCD)*, October 2006.

[15] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. Kilocore: A 32 nm 1000-processor array. In *IEEE HotChips Symposium on High-Performance Chips*, August 2016.

[16] Bin Liu. *Energy-Efficient Computing with Fine-Grained Many-Core Systems*. PhD thesis, University of California, Davis, Davis, CA, USA, September 2016. `http://vcl.ece.ucdavis.edu/pubs/theses/2016-1/`.

[17] Antonio Lloris Ruiz, Encarnación Castillo Morales, Luis Parrilla Roure, Antonio García Ríos, and María José Lloris Meseguer. *Division*, pages 287–315. Springer International Publishing, Cham, 2021.

[18] James E. Robertson. A new class of digital division methods. *IRE Transactions on Electronic Computers*, EC-7(3):218–222, 1958.

[19] K. D. Tocher. Techniques of multiplication and division for automatic binary computers. *Quarterly Journal of Mechanics and Applied Mathematics*, 11:364–384, 1958.

[20] Peter Markstein. Software division and square root using goldschmidt's algorithms. *Real Numbers and Computers'6, 146–157*, 2004.

[21] Michael Morris. Computing fixed-point square roots and their reciprocals using goldschmidt algorithm.