Hardware Algorithms for Division, Square Root and Elementary Functions

by

Sherif Amin Tawfik Naguib

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
July 2005

Hardware Algorithms for Division, Square Root and Elementary Functions

by

Sherif Amin Tawfik Naguib

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS

Under the Supervision of

Serag E.-D. Habib — Professor, Elec. and Com. Dept.
Hossam A. H. Fahmy — Assistant Professor, Elec. and Com. Dept.

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
July 2005

Hardware Algorithms for Division, Square Root and Elementary Functions

by

Sherif Amin Tawfik Naguib

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS

Approved by the
Examining Committee

_____
Prof. Dr. Serag E.-D. Habib, Thesis Main Advisor

_____
Prof. Dr. Elsayed Mostafa Saad

_____
Associate Prof. Dr. Ibrahim Mohamed Qamar

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
July 2005

Acknowledgments

I would like first to thank God for making everything possible and for giving me strength and good news at times of disappointment. I would like also to thank my advisers. Special thanks go to my second adviser Dr. Hossam Fahmy for his continuous advises, helpful notes and friendly attitude. I would like to thank also my family specifically my mother and two brothers for giving me support all my life and for having faith in my abilities.

Abstract

Many numerically intensive applications require the fast computation of division, square root and elementary functions. Graphics processing, image processing and generally digital signal processing are examples of such applications. Motivated by this demand on high performance computation many algorithms have been proposed to carry out the computation task in hardware instead of software. The reason behind this migration is to increase the performance of the algorithms since more than an order of magnitude increase in performance can be attained by such migration. The hardware algorithms represent families of algorithms that cover a wide spectrum of speed and cost.

This thesis presents a survey about the hardware algorithms for the computation of elementary functions, division and square root. Before we present the different algorithms we discuss argument reduction techniques, an important step in the computation task. We then present the approximation algorithms. We present polynomial based algorithms, table and add algorithms, a powering algorithm, functional recurrence algorithms used for division and square root and two digit recurrence algorithms namely the CORDIC and the Briggs and DeLugish algorithm.

Careful error analysis is crucial not only for correct algorithms but it may also lead to better circuits from the point of view of area, delay or power. Error analysis of the different algorithms is presented.

We made three contributions in this thesis. The first contribution is an algorithm that computes a truncated version of the minimax polynomial coefficients that gives better results than the direct rounding. The second contribution is about devising an algorithmic error analysis that proved to be more accurate and led to a substantial decrease in the area of the tables used in a powering, division

and square root algorithms. Finally the third contribution is a proposed high order Newton-Raphson algorithm for the square root reciprocal operation and a square root circuit based on this algorithm.

VHDL models for the powering algorithm, functional recurrence algorithms and the CORDIC algorithm are developed to verify these algorithms. Behavioral simulation is also carried out with more than two million test vectors for the powering algorithm and the functional recurrence algorithms. The models passed all the tests.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer Arithmetic is the science that is concerned with representing numbers in digital computers and performing arithmetic operations on them as well as computing high level functions.

1.1 Number Systems

Numbers systems are ways of representing numbers so that we can perform arithmetic operations on them. Examples of number systems include weighted positional number system WPNS, residue number system RNS and logarithmic number system LNS. Each of these systems has its own advantages and disadvantages. The WPNS is the most popular. We can perform all basic arithmetic operations on numbers represented in WPNS. Moreover the decimal system which we use in our daily life is a special case of the WPNS. The RNS has the advantage that the addition and multiplication operations are faster when the operands are represented in RNS system. However division and comparison are difficult and costly. LNS has the advantage that multiplication and division are faster when operands are represented in LNS system. However addition is difficult in this number system. From this point on we concentrate on the WPNS because it is the most widely used number system and because it is most suitable for general purpose processors.

In WPNS positive integers are represented by a string of digits as follows:

$$X = x_{L-1}x_{L-2}\ldots x_3x_2x_1x_0$$

such that

$$X = \sum_{i=0}^{i=L-1} x_i\mu^i \tag{1.1}$$

Where μ is a constant and it is called the radix of the system. Each digit x_i can take one of the values of the set $\{0, 1, 2, \ldots, \mu - 1\}$. The range of positive integers that can be represented with L digits is $[0, \mu^L - 1]$. It can be shown that any positive integer has a unique representation in this number system.

The decimal system is a special case of the WPNS in which the radix $\mu = 10$.

We can represent negative numbers by a separate sign that can be either $+$ or $-$. This is called sign-magnitude representation or we can represent negative numbers by adding a constant positive bias to the numbers hence the biased numbers that are less than the bias are negative while the biased numbers that are greater than the bias are positive. This is called biased representation. We can also represent negative integers using what is known as the radix complement representation. In such representation if X is a positive L digits number then $-X$ is represented by $RC(X) = \mu^L - X$. It is clear that $RC(X)$ is a positive number since $\mu^L > X$. $RC(X)$ is also represented in L digits. To differentiate between positive and negative numbers we divide the range of the L digits number which is $[0, \mu^L - 1]$ into two halves. We designate The first half to positive integers and the second half to negative integers in radix complement form. The advantage of using the radix complement representation to represent negative numbers is that we can perform subtraction using addition. Adding μ^L to an L digits number doesn't affect its L digits representation since μ^L is represented by 1 in the $L+1$ digit position. Hence when we subtract two L digits numbers $X_1 - X_2$ it is equivalent to $X_1 - X_2 + \mu^L = X_1 + RC(X_2)$. Hence to perform subtraction we simply add the first number to the radix complement of the second number. The radix complement $RC(X)$ can be easily obtained from X by subtracting each digit of X from the radix μ then we add 1 to the result.

We extend the WPNS to represent fractions by using either the fixed point representation or the floating point representation. In the former we represent numbers as follows:

$$X = x_{L_2-1}x_{L_2-2}\ldots x_3x_2x_1x_0.x_{-1}x_{-2}\ldots x_{-L_1} \tag{1.2}$$

such that

$$X = \sum_{i=-L_1}^{i=L_2-1} x_i\mu^i \text{ for positive numbers} \tag{1.3}$$

$$X = \sum_{i=-L_1}^{i=L_2-1} x_i\mu^i - \mu^{L_2} \text{ for negative numbers} \tag{1.4}$$

The digits to the right of the point are called the fraction part while the digits to the left of the point are called the integer part. Note that the number of digits of the fraction part as well as the integer part is fixed hence the name of the representation.

In Floating point representation we represent the number in the form:

$$X = \pm\mu^e 0.x_{-1}x_{-2}\ldots x_{-L}$$

such that

$$X = \pm\mu^e(\sum_{i=-L}^{i=-1} x_i\mu^i)$$

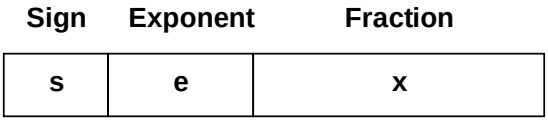
where e is called the exponent and $0.x_{-1}x_{-2}\ldots x_{-L}$ is called the mantissa of the floating point number. x_{-1} is non-zero. For the case of binary radix ($\mu = 2$) the leading zero of the mantissa can be a leading 1.

Negative exponents are represented using the biased representation in order to simplify the comparison between two floating point numbers. According to the value of the exponent the actual point moves to either the left or the right of its apparent position hence the name of the representation.

The gap of the representation is defined as the the difference between any two consecutive numbers. In case of fixed point representation the gap is fixed and is equal to the weight of the rightmost digit which is μ^{-L_1}. On the other hand the gap in floating point representation is not fixed. It is equal to μ^e multiplied by

the weight of the rightmost digit. Therefore the gap is dependent on the value of the exponent. The gap is small for small exponents and big for big exponents. The advantage of the floating point format over the fixed point format is that it increases the range of the representable numbers by allowing the gap to grow for bigger numbers.

In digital computers we use string of bits to represent a number. Each bit can be either 0 or 1. Some information in the representation is implicit such as the radix of the system. The explicit information about the number is formatted in a special format. The format specifies the meaning of every group of bits in the string. For example IEEE floating point numbers are represented as follows:



$$X = s \times 2^e \times 1.x$$

Where the sign s is represented by 1 bit $s = 0$ for positive sign and $s = 1$ for negative sign, e is the exponent that is represented by 8 bits for single precision format and 11 bits for double precision format and finally x is the fraction. It is represented by 23 bits for single precision format and 52 bits for the double floating format. The radix of the system which is 2 and the hidden 1 of the mantissa are implicit information and are not stored. More details about the floating IEEE standard can be found in [1].

The presence of such standard is necessary for information exchange between different implementation. Internally numbers can be represented in the computer in a different format but when we write the number to the memory it should be in the IEEE floating point standard in order to be easily transfered between the different platforms.

An important modification in the WPNS is the introduction of redundancy. We add redundancy by increasing the digit set. In the non-redundant representation each digit $x_i \in \{0, 1, 2, \ldots, \mu - 1\}$ and that caused each number to have a unique representation. In the redundant representation we increase the number of elements of the digit set and the digits can be negative. In such representation

a number can have more than one representation. This feature is used to speed up the arithmetic circuits. More details on redundant representation can be found in [2, 3]

1.2 Survey of Previous Work

The arithmetic circuits represent the bulk of the data path of microprocessors. They also constitute large parts of specific purpose digital circuits. These circuits process numbers represented by a certain format in order to compute the basic arithmetic operations as well as elementary functions.

In this thesis we are primarily interested in the arithmetic circuits that compute division, square root and elementary functions. Such circuits will use adders and multipliers as building blocks. Historically division, square root and elementary functions were computed in software and that led to a long computation time in the order of hundreds of clock cycles. Motivated by computationally intensive applications such as digital signal processing hardware algorithms have been devised to compute the division, square root and elementary functions at a higher speed than the software techniques.

The hardware approximation algorithms can be classified into four broad categories.

The first category is called digit recurrence techniques. The algorithms that belong to this category are linearly convergent and they employ addition, subtraction, shift and single digit multiplication operations. Examples of such algorithms are division restoring algorithm [4], division non-restoring algorithm [4], SRT division algorithm [4], CORDIC [5, 6],Briggs and DeLugish algorithm [7, 8], BKM [9] and online algorithms [10].

The second category is called functional recurrence. Algorithms that belong to this category employ addition, subtraction and full multiplication operations as well as tables for the initial approximation. In this class of algorithms we start by a given initial approximation and we feed it to a polynomial in order to obtain a better approximation. We repeat this process a number of times until we reach the desired precision. These algorithms converge quadratically or

better. Examples from this category include Newton-Raphson for division [11, 12], Newton-Raphson for square root [12] and high order Newton-Raphson for division [13].

The third category is the polynomial approximation. This category is a diverse category. The general description of this class is as follows: we divide the interval of the argument into a number of sub-intervals. For each sub-interval we approximate the elementary function by a polynomial of a suitable degree. We store the coefficients of such polynomials in one or more tables. Polynomial approximation algorithms employ tables, adders and multipliers. Examples of algorithms from this category are: Initial approximation for functional recurrence [14, 15, 12, 16]. The powering algorithm [17] which is a first order algorithm that employs a table, a multiplier and a special hardware for operand modification. This algorithm can be used for single precision results or as an initial approximation for the functional recurrence algorithms. Table and add algorithms can be considered a polynomial based approximation. These algorithms are first order polynomial approximation in which the multiplication is avoided by using tables. Examples of table add techniques include the work in [18] Bipartite [19, 20], Tripartite [21] and Multipartite [21, 22, 23] Examples of other work in polynomial approximation include [24, 25]. The convergence rate of polynomial approximation algorithms is function-dependent and it also depends to a great extent on the range of the given argument and on the number of the sub-intervals that we employ.

The fourth category is the rational approximation algorithms. In this category we divide the given interval of the argument into a number of sub-intervals. For each sub-interval we approximate the given function by a rational function. A rational function is simply a polynomial divided by another polynomial. It employs division operation in addition to tables, addition and multiplication operations. The rational approximation is rather costly in hardware due to the fact that it uses division.

Range reduction is the first step in elementary functions computation. It aims to transform the argument into another argument that lies in a small interval. Previous work in this area can be found in [26, 27]. In [26] an accurate algorithm for carrying out the modular reduction for trigonometric functions is presented

while in [27] useful theorems on range reduction are given from which an algorithm for carrying out the modular reduction on the same working precision is devised.

1.3 Thesis Layout

The rest of the thesis is organized as follows:

In chapter 2 We present range reduction techniques. We give examples for different elementary functions and we show how to make the range reduction accurate.

In chapter 3 we present the general polynomial approximation techniques. In these techniques we divide the interval of the argument into a number of sub-intervals and design a polynomial for each sub-interval that approximates the given function in its sub-interval. We then show how to compute the coefficients of such polynomials and the approximation error. We also give in this chapter three hardware architectures for implementing the general polynomial approximation. Another source of error is the rounding error. We present techniques for computing bounds of the accumulated rounding error. We also present an algorithm given in [25] that aims to truncate the coefficients of the approximating polynomials in an efficient way. The algorithm is applicable for second order polynomials only. Finally we present our algorithm for truncating the coefficients of the approximating polynomials in an efficient way and which is applicable for any order.

In chapter 4 we present table and add algorithms, a special case of the polynomial approximation techniques. In these algorithms we approximate the given function by a first order polynomial and we avoid the multiplication by using tables. In bipartite we use two tables. In tripartite we use three tables and in the general Multipartite we use any number of tables. A variant of the bipartite which is called the symmetric bipartite is also presented.

In chapter 5 we present a powering algorithm [17] that is a special case of the polynomial approximation. It is a first order approximation that uses one coefficient and operand modification. It uses smaller table size than the general first order approximation technique at the expense of a bigger multiplier hence

the algorithm is practical when a multiplier already exists in the system. We also present an error analysis algorithm [28] that gives a tighter error bound than the theoretical error analysis.

In chapter 6 we present the functional recurrence algorithms. These algorithms are based on the Newton-Raphson root finding algorithm. We present the Newton-Raphson algorithm and show how it can be used to compute the reciprocal and square root reciprocal functions using a recurrence relation that involves multiplication and addition. We present also the high order version of the Newton-Raphson algorithm for the division [13] and for the square root reciprocal [29]. A hardware circuit for the square root operation is presented. This circuit is based on the powering algorithm for the initial approximation of the square root reciprocal followed by the second order Newton-Raphson algorithm for the square reciprocal and a final multiplication by the operand and rounding. A VHDL model is created for the circuit and it passes a simulation test composed of more than two million test vectors.

In chapter 7 we present two algorithms from the digit recurrence techniques. The first is the CORDIC algorithm [5, 6] and the second is the Briggs and DeLugish algorithm.

Chapter 2

Range Reduction

We denote the argument by X and the elementary function by $F(X)$. The computation of $F(X)$ is performed by three main steps. In the first step we reduce the range of X by mapping it to another variable Y that lies in a small interval say $[a, b]$. This step is called the range reduction step. In the second step we compute the elementary function at the reduced argument $F(Y)$ using one of the approximation algorithms as described in the following chapters. We call this step the approximation step. In the third step we compute the elementary function at the original argument $F(X)$ from our knowledge of $F(Y)$. The third step is called the reconstruction step.

The range reduction step and the reconstruction step are related and they depend on the function that we need to compute.

The advantage of the range reduction is that the approximation of the elementary function is more efficient in terms of computation delay and hardware area when the argument is constrained in a small interval.

For some elementary functions and arithmetic operations the range reduction is straightforward such as in reciprocation, square root and log functions. For these functions the reduced argument is simply the mantissa of the floating point representation of the argument. The mantissa lies in a small interval $[1, 2[$. The reconstruction is also straightforward as follows:
The reciprocal function: $F(X) = \frac{1}{X}$

$$X = \pm 2^e 1.x \tag{2.1}$$

$$F(X) = \frac{1}{X} = 2^{-e}\frac{1}{1.x} \tag{2.2}$$

where e is the unbiased exponent and x is the fraction. Equation 2.2 indicates that computing the exponent of the result is simply performed by negating the exponent of the argument. Hence we only need to approximate $\frac{1}{Y}$ where Y is the mantissa of the argument X, $Y = 1.x$. To put the result in the normalized form we may need to shift the mantissa by a single bit to the left and decrement the exponent.

The log function: $F(X) = logX$

$$X = +2^e 1.x \tag{2.3}$$

$$F(X) = log(X) = e\ log(2) + log(1.x) \tag{2.4}$$

From equation 2.4 we only need to approximate $F(Y) = logY$ where $Y = 1.x$ and we reconstruct $F(X)$ by adding $F(Y)$ to the product of the unbiased exponent and the stored constant $log(2)$.

The trigonometric functions $F(X) = sin(X)$ and $F(X) = cos(X)$ and the exponential function $F(X) = exp(X)$ requires modular reduction of the argument X. That is we compute the reduced argument Y such that:

$$X = N \times A + Y \tag{2.5}$$
$$0 \leq Y < A$$

Where A is a constant that is function-dependent and N is an integer. For the trigonometric functions $sin(X)$ and $cos(X)$ $A = 2\pi$ or $A = \pi$ or $A = \frac{\pi}{2}$ or $A = \frac{\pi}{4}$. As A gets smaller the interval of the reduced argument Y becomes smaller and hence the approximation of $F(Y)$ will become more efficient at the expense of complicating the reconstruction step. For example if $A = \frac{\pi}{2}$ we reconstruct $sin(X)$ as follows:

$$sin(X) = sin(N \times \frac{\pi}{2} + Y) \tag{2.6}$$

$$sin(X) = sin(Y)\ ,\ \text{N mod } 4 = 0 \tag{2.7}$$

$$sin(X) = cos(Y)\ ,\ \text{N mod } 4 = 1 \tag{2.8}$$

$$sin(X) = -sin(Y) \text{ , N mod } 4 = 2 \tag{2.9}$$

$$sin(X) = -cos(Y) \text{ , N mod } 4 = 3 \tag{2.10}$$

On the other hand if $A = \frac{\pi}{4}$ we reconstruct $sin(X)$ as follows:

$$sin(X) = sin(N \times \frac{\pi}{4} + Y) \tag{2.11}$$

$$sin(X) = sin(Y) \text{ , N mod } 8 = 0 \tag{2.12}$$

$$sin(X) = \frac{cos(Y) + sin(Y)}{\sqrt{2}} \text{ , N mod } 8 = 1 \tag{2.13}$$

$$sin(X) = cos(Y) \text{ , N mod } 8 = 2 \tag{2.14}$$

$$sin(X) = \frac{cos(Y) - sin(Y)}{\sqrt{2}} \text{ , N mod } 8 = 3 \tag{2.15}$$

$$sin(X) = -sin(Y) \text{ , N mod } 8 = 4 \tag{2.16}$$

$$sin(X) = \frac{-sin(Y) - cos(Y)}{\sqrt{2}} \text{ , N mod } 8 = 5 \tag{2.17}$$

$$sin(X) = -cos(Y) \text{ , N mod } 8 = 6 \tag{2.18}$$

$$sin(X) = \frac{sin(Y) - cos(Y)}{\sqrt{2}} \text{ , N mod } 8 = 7 \tag{2.19}$$

The reconstruction of $cos(X)$ is similar.

For the exponential function $A = ln(2)$. The reconstruction step is as follows:

$$exp(X) = exp(N \times ln(2) + Y) \tag{2.20}$$

$$= 2^N \times exp(Y) \tag{2.21}$$

Since $0 \leq Y < ln(2)$ therefore $1 \leq exp(Y) < 2$. This means that $exp(Y)$ is the mantissa of the result while N is the exponent of the result.

From the previous examples it is clear that the range reduction step is performed so that the approximation step becomes efficient and at the same time the reconstruction step is simple.

To perform the modular reduction as in the last two examples we need to compute N and Y. We compute N and Y from X and the constant A as follows:

$$N = \left\lfloor \frac{X}{A} \right\rfloor \tag{2.22}$$

$$Y = X - N \times A \tag{2.23}$$

To carry out the above two equations we need to store the two constants A and $\frac{1}{A}$ to a suitable precision. We multiply the argument by the second constant $\frac{1}{A}$ and truncate the result after the binary point. The resulting integer is N. We then perform the equation 2.23 using one multiplication and one subtraction.

These two equations work fine for most arguments. However for arguments that are close to integer multiples of A a catastrophic loss of significance will occur in the subtraction in equation 2.23. This phenomenon is due to the errors in representing the two constants A and $\frac{1}{A}$ by finite precision machine numbers.

An algorithm given in [26] gives an accurate algorithm for carrying out the modular reduction for trigonometric functions. The algorithm stores a long string for the constant $\frac{1}{2\pi}$. It starts with the reduction process by a a subset of this string. If significance loss occurs the algorithm uses more bits of the stored constant recursively to calculate the correct reduced argument.

Useful theorems for range reduction are given in [27]. An algorithm based on these theorems is also given. The algorithm seeks the best representation for the two constants A and $\frac{1}{A}$ such that the reduction algorithm is performed on the same working precision. The theorems also give the permissible range of the given argument for correct range reduction.

Chapter 3

Polynomial Approximation

We denote the reduced argument by Y. It lies in the interval $[a, b]$. We denote the function that we need to compute by $F(Y)$.

One of the prime issues in polynomial approximation is the determination of the polynomial order that satisfies the required precision. If we approximate the given function in the given interval using one polynomial the degree of the resulting polynomial is likely to be large. In order to control the order of the approximating polynomial we divide the given interval into smaller sub-intervals and approximate the given function using a different polynomial for each sub-interval.

The rest of this chapter is organized as follows: We give the details of the interval division and the general formula of the polynomial approximation in section 3.1. Techniques for computing the coefficients of the approximating polynomials are given in section 3.2. Three possible implementation architectures are given in section 3.3. We present techniques for error analysis in section 3.4. An algorithm due to Muller [25] that aims at truncating the coefficients of the approximating polynomial is given in section 3.5. Finally we give our algorithm that has the same aim as that of Muller's in section 3.6.

3.1 Interval Division and General Formula

We divide the given interval $[a, b]$ uniformly into a number of divisions equal to J. Therefore every sub-interval has a width $\Delta = \frac{b-a}{J}$, starts at $a_m = a + m\Delta$

and ends at $b_m = a + (m+1)\Delta$ where $m = 0, 1, \ldots, J-1$ is the index of the sub-interval. For a given argument Y we need to determine the sub-interval index m in which it lies and the difference between the argument Y and the beginning of its sub-interval. We denote this difference by h as shown in figure 3.1.

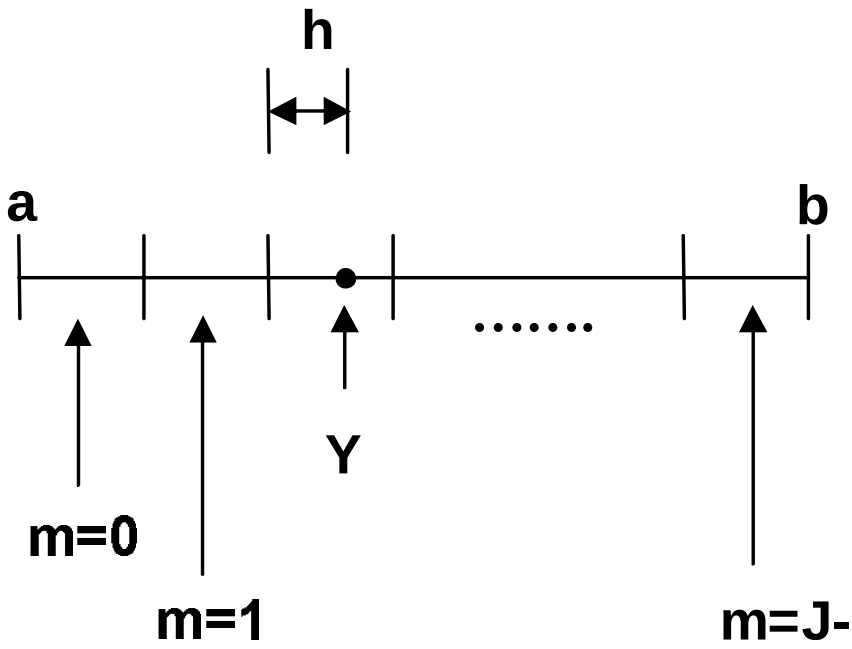


Figure 3.1: Dividing the given interval into J sub-intervals. From a given argument Y we need to determine the sub-interval index m and the distance to the start of the sub-interval h

We can determine m and h from a, b and J as follows:

$$\Delta = \frac{b-a}{J} \tag{3.1}$$

$$m = \lfloor \frac{Y-a}{\Delta} \rfloor \tag{3.2}$$

$$h = Y - a_m = Y - (a + m\Delta) \tag{3.3}$$

We usually set the width of the interval $[a, b]$ to be power of 2 and we choose J to be also power of 2. With such choices the computation of m and h becomes straightforward in hardware. For example if the interval $[a, b] = [0, 1[$ hence Y has the binary representation $Y = 0.y_1 y_2 \ldots y_L$ assuming L bits representation. If $J = 32 = 2^5$ then we calculate m and h as follows:

$$\begin{aligned} \Delta &= \frac{b-a}{J} \\ &= 2^{-5} \end{aligned} \tag{3.4}$$

$$
\begin{aligned}
m &= \lfloor \frac{Y-a}{\Delta} \rfloor \\
&= \lfloor (Y-0) \times 2^5 \rfloor \\
&= y_1 y_2 y_3 y_4 y_5 \qquad (3.5) \\
h &= Y - (a + m\Delta) \\
&= Y - (0 + m \times 2^{-5}) \\
&= 0.00000 y_6 y_7 \ldots y_L \qquad (3.6)
\end{aligned}
$$

For every sub-interval we design a polynomial that approximates the given function in that sub-interval. We denote such polynomial by $P_{mn}(h)$ where m stands for the index of the sub-interval while n denotes the order of the polynomial.

$$
\begin{aligned}
F(Y) &\approx P_{mn}(h) \\
&= c_{m0} + c_{m1}h + c_{m2}h^2 + \cdots + c_{mn}h^n \qquad (3.7)
\end{aligned}
$$

The coefficients are stored in a table that is indexed by m.

It is to be noted that as the number of sub-intervals J increases the width of every sub-interval and hence the maximum value of h decreases enabling us to decrease the order of the approximating polynomials. However more sub-intervals means larger coefficients table. Therefore we have a trade off between the number of sub-intervals and the order of the approximating polynomials. This trade off is translated in the iterative architecture to a trade off between area and delay and it is translated in the parallel and PPA architectures to a trade off between the area of the coefficients table and the area of the other units as we show in section 3.3.

3.2 Polynomial Types

We discuss in this section three techniques for computing the coefficients of the approximating polynomials. They are Taylor approximation, minimax approximation and interpolation.

Taylor approximation gives analytical formulas for the coefficients and the

approximation error. It is useful for some algorithms that we present in later chapters namely the Bipartite, Multipartite, Powering algorithm and functional recurrence.

Minimax approximation on the other hand is a numerical technique. It gives the values of the coefficients and the approximation error numerically. It has the advantage that it gives the lowest polynomial order for the same maximum approximation error.

Interpolation is a family of techniques. Some techniques use values of the given function in order to compute the coefficients while others use values of the function and its higher derivatives to compute the coefficients. Interpolation can be useful to reduce the size of the coefficients table at the expense of more complexity and delay and that is by storing the values of the function instead of the coefficients and computing the coefficients in hardware on the fly [30].

In the following three subsections we discuss the three techniques in detail.

3.2.1 Taylor Approximation

We review the analytical derivation of the Taylor approximation theory. We assume the higher derivatives of the given function exist. From the definition of the integral we have:

$$F(Y) - F(a_m) = \int_{t=a_m}^{t=Y} F'(t)dt \tag{3.8}$$

Integration by parts states that:

$$\int udv = uv - \int vdu \tag{3.9}$$

We let $u = F'(t)$ and $dv = dt$ hence we get $du = F''(t)dt$ and $v = t - Y$. Note that we introduce a constant of integration in the last equation. Y is considered as a constant when we integrate with respect to t. We substitute these results in equation 3.8 using the integration by parts technique to obtain:

$$F(Y) - F(a_m) = [(t-Y)F'(t)]_{t=a_m}^{t=Y} + \int_{t=a_m}^{t=Y} (Y-t)F''(t)dt \tag{3.10}$$

$$F(Y) = F(a_m) + (Y-a_m)F'(a_m) + \int_{t=a_m}^{t=Y} (Y-t)F''(t)dt \tag{3.11}$$

In equation 3.11 we write the value of the function at Y in terms of its value at a_m and its first derivative at a_m. The remainder term is function of the second derivative $F''(t)$.

We perform the same steps on the new integral involving the second derivative of $F''(t)$. We set $u = F''(t)$ and $dv = (Y-t)dt$ hence $du = F^{(3)}(t)dt$ and $v = -\frac{1}{2}(Y-t)^2$. We use the integration by parts on the new integral to obtain:

$$F(Y) = F(a_m) + (Y-a_m)F'(a_m) + \frac{1}{2}(Y-a_m)^2F''(a_m) + \frac{1}{2}\int_{t=a_m}^{t=Y} (Y-t)^2F^{(3)}(t)dt \tag{3.12}$$

Continuing with the same procedure we reach the general Taylor formula:

$$\begin{aligned} F(Y) &= F(a_m) + (Y-a_m)F'(a_m) + \frac{(Y-a_m)^2}{2!}F''(a_m) + \cdots \\ &+ \frac{(Y-a_m)^n}{n!}F^{(n)}(a_m) + \int_{t=a_m}^{t=Y} \frac{(Y-t)^n}{n!}F^{(n+1)}(t)dt \qquad (3.13) \\ R_n &= \int_{t=a_m}^{t=Y} \frac{(Y-t)^n}{n!}F^{(n+1)}(t)dt \qquad (3.14) \end{aligned}$$

The reminder given by equation 3.14 decreases as n increases. Eventually it vanishes when n approaches infinity. Using the mean value theorem the remainder can be also written in another form that can be more useful

$$R_n = \frac{(Y-a_m)^{n+1}}{(n+1)!}F^{(n+1)}(\zeta) \tag{3.15}$$

where ζ is a point in the interval $[a_m, Y]$. Since the point ζ is unknown to us we usually bound the remainder term by taking the maximum absolute value of $F^{(n+1)}$.

Equations 3.13 and 3.15 give the formula of Taylor polynomial and the approximation error respectively. To link the results of Taylor theorem with the

general formula given in the previous section we set a_m to the start of the sub-interval m in which the argument Y lies hence $Y - a_m = h$. Taylor polynomial and approximation error(remainder) can now be written as follows:

$$\begin{aligned} F(Y) &\approx P_{mn}(Y) = F(a_m) + F'(a_m)h + \frac{F''(a_m)}{2!}h^2 + \cdots \\ &+ \frac{F^{(n)}(a_m)}{n!}h^n \qquad (3.16) \\ \epsilon_a &= \frac{(h)^{n+1}}{(n+1)!}F^{(n+1)}(\zeta) \qquad (3.17) \end{aligned}$$

We use equation 3.16 to compute the coefficients of the approximating polynomials. By comparison with the general polynomial formula given in the previous section we get the coefficients

$$\begin{aligned} c_{mi} &= \frac{F^{(i)}(a_m)}{i!} \qquad (3.18) \\ i &= 0, 1, \ldots, n \end{aligned}$$

While equation 3.17 gives the approximation error.

3.2.2 Minimax Approximation

Minimax approximation seeks the polynomial of degree n that approximates the given function in the given interval such that the absolute maximum error is minimized. The error is defined here as the difference between the function and the polynomial.

Chebyshev Proved that such polynomial exists and that it is unique. He also gave the criteria for a polynomial to be a minimax polynomial[31]. Assuming that the given interval is $[a_m, b_m]$ Chebyshev's criteria states that if $P_{mn}(Y)$ is the minimax polynomial of degree n then there must be at least $(n+2)$ points in this interval at which the error function attains the absolute maximum value with alternating sign as shown in figure 3.2 for $n = 3$ and by the following equations:

$$\begin{aligned} & a_m \leq y_0 < y_1 < \cdots < y_{n+1} \leq b_m \\ & F(y_i) - P_{mn}(y_i) = (-1)^i E \qquad (3.19) \\ & i = 0, 1, \ldots, n+1 \end{aligned}$$

$$E = \pm \max_{a_m \leq y \leq b_m} |F(y) - P_{mn}(y)| \tag{3.20}$$

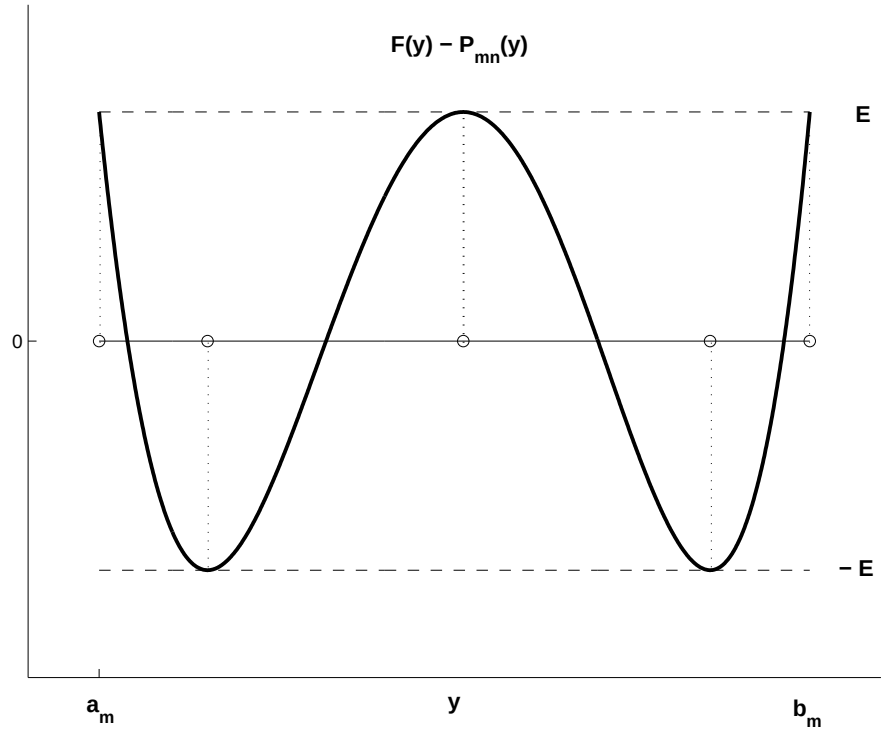


Figure 3.2: Example of a minimax third order polynomial that conforms to the Chebychev criteria

The minimax polynomial can be computed analytically up to $n = 1$. For higher order a numerical method due to Remez [32] has to be employed.

Remez algorithm is an iterative algorithm. It is composed of two steps in each iteration. In the first step we compute the coefficients such that the difference between the given function and the polynomial takes equal magnitude with alternating sign at $(n + 2)$ given points.

$$\begin{aligned} F(y_i) &- P_{mn}(y_i) = (-1)^i E && (3.21) \\ F(y_i) &- [c_{m0} + c_{m1}(y_i - a_m) + c_{m2}(y_i - a_m)^2 \\ &+ \cdots + c_{mn}(y_i - a_m)^n] = (-1)^i E && (3.22) \\ c_{m0} &+ c_{m1}h_i + \cdots + c_{mn}h_i^n + (-1)^i E = F(y_i) && (3.23) \\ i &= 0, 1, 2, \ldots, n+1 \end{aligned}$$

Equation 3.23 is a system of $(n + 2)$ linear equations in the $(n + 2)$ unknowns

$\{c_{m0}, c_{m1}, \ldots, c_{mn}, E\}$. These equations are proved to be independent [32] hence we can solve them using any method from linear algebra to get the values of the coefficients as well as the error at the given $(n+2)$ points.

The second step of Remez algorithm is called the exchange step. There are two exchange techniques. In the first exchange technique we exchange a single point while in the second exchange technique we exchange all the $(n+2)$ points that we used in the first step.

We start the second step by noting that the error alternates in sign at the $(n+2)$ points therefore it has $(n+1)$ roots, one root in each of the the intervals: $[y_0, y_1], [y_1, y_2], \ldots, [y_n, y_{n+1}]$. We compute these roots using any numerical method such as the method of chords or bisection. We denote these roots by $z_0, z_1, \ldots, z_n$. We divide the interval $[a_m, b_m]$ into the $(n+2)$ intervals: $[a_m, z_0], [z_0, z_1], [z_1, z_2], \ldots, [z_{n-1}, z_n], [z_n, b_m]$. In each of these intervals we compute the point at which the error attains its maximum or minimum value and denote these points by $y_0^*, y_1^*, \ldots, y_{n+1}^*$.

We can carry out the last step numerically by computing the root of the derivative of the error function if such root exists otherwise we compute the error at the endpoints of the interval and pick the one that gives larger absolute value for the error function.

We define k such that

$$k = \max_i |F(y_i^*) - P_{mn}(y_i^*)| \tag{3.24}$$

In the single point exchange technique we exchange y_k by y_k^* while in the multiple exchange technique we exchange all the $(n+2)$ points $\{y_i\}$ by $\{y_i^*\}$.

We use this new set of $(n+2)$ points in the first step of the following iteration. We repeat the two steps a number of times until the difference between the old $(n+2)$ points and the new $(n+2)$ points lies below a given threshold. Figure 3.3 illustrates the second step graphically for a third order polynomial. The initial $(n+2)$ points are selected arbitrarily.

It is to be noted that Remez algorithm gives also the value of the maximum absolute error from the computation of the first step represented by the variable E.

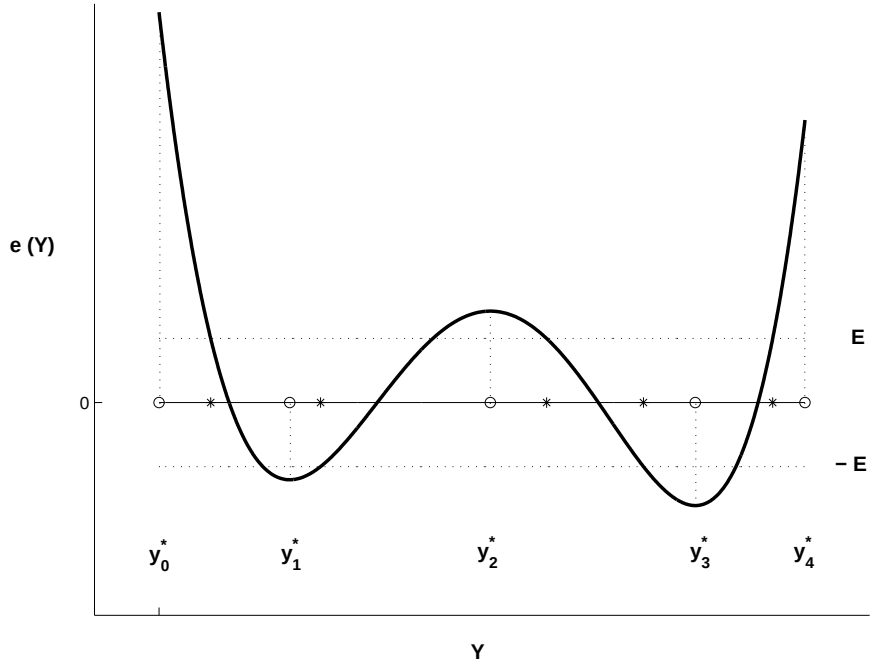


Figure 3.3: Illustration of second step of Remez algorithm

3.2.3 Interpolation

There are two main Interpolation techniques. The first technique makes use of the values of the given function at some points in the given interval in order to compute the coefficients of the approximating polynomial. The second technique makes use of the values of the function and its higher derivatives at some point in the given interval in order to compute the coefficients of the approximating polynomial. The second technique is actually a family of techniques because it has many parameters such as the derivative order to use, the number of points, etc. . ..

In the first Interpolation technique we select $n+1$ points arbitrarily (usually uniformly)in the given interval $[a_m, b_m]$. We force the approximating polynomial to coincide with the given function at these $n+1$ points

$$\begin{aligned} a_m &\leq y_0 < y_1 < \cdots < y_n \leq b_m \\ F(y_i) &= P_{mn}(y_i) \qquad (3.25) \\ &= c_{m0} + c_{m1}(y_i - a_m) + c_{m2}(y_i - a_m)^2 + \cdots \\ &+ c_{mn}(y_i - a_m)^n \qquad (3.26) \end{aligned}$$

Equation 3.26 is a system of $(n+1)$ linear equations in the $(n+1)$ unknowns $c_{m0}, c_{m1}, \ldots, c_{mn}$. The equations are independent and hence can be solved numerically by methods from linear algebra to yield the desired coefficients. There are many techniques in literature for computing the coefficients of the approximating polynomial from equation 3.26 such as Lagrange, Newton-Gregory Forward,Gauss Forward, Stirling, Bessel, etc... They all reach the same result since the solution is unique.

The approximation error is given by the following equation [33]:

$$\epsilon_a = \left|(Y-y_0)(Y-y_1)\cdots(Y-y_n)\frac{F^{(n+1)}(\zeta)}{(n+1)!}\right| \tag{3.27}$$
$$a_m \leq \zeta \leq b_m$$

The approximation error can also be computed numerically by noting that the error function is zero at the $n+1$ points $\{y_i\}$ hence it has a maximum or minimum value between each pair. We divide the interval $[a_m, b_m]$ into the intervals $[a_m, y_0], [y_0, y_1], [y_1, y_2], \ldots, [y_{n-1}, y_n]$. In each of these intervals we compute the extremum point and denote them by $\left\{y_0^*, y_1^*, \ldots y_{n+1}^*\right\}$. The approximation error is then given by $max\,|F(y_i^*) - P_{mn}(y_i^*)|$.

An algorithm given in [30] uses the second order interpolation polynomial. It computes the coefficients using the Newton-Gregory Forward method. Since $n = 2$ therefore we need three points in the interval in order to compute the three coefficients. The algorithm uses the two endpoints of the interval together with the middle point. Instead of storing the coefficients in a table we store the function value at the chosen three points and the coefficients are computed in hardware. The gain behind this architecture is that the end points are common between adjacent intervals thus we don't need to store them twice hence we reduce the size of the table to almost two thirds.

An algorithm that belongs to the second Interpolation technique is given in [34]. This interpolation algorithm is called MIP. It simply forces the approximating polynomial to coincides with the given function at the end points of the given interval and forces the higher derivatives of the approximating polynomial up to order $n-1$ to coincides with that of the given function at the starting point

of the given interval.

$$
\begin{aligned}
F(a_m) &= P_{mn}(a_m) = c_{m0} && (3.28)\\
F(b_m) &= P_{mn}(b_m) = c_{m0} + c_{m1}\Delta + c_{m2}\Delta^2 + \cdots + c_{mn}\Delta^n && (3.29)\\
F^{(i)}(a_m) &= p^{(i)}_{mn}(a_m) = (i!)c_{mi} && (3.30)\\
i &= 1, \ldots, n-1
\end{aligned}
$$

3.3 Implementation

In this section three different architectures that implements equation 3.7 in hardware are presented.

The first architecture is the iterative architecture. It is built around a fused multiply add unit. It takes a number of clock cycles equal to the order of the polynomial (n) in order to compute the polynomial.

The second architecture is the parallel architecture. It makes use of specialized powering units and multipliers to compute the terms of the polynomial in parallel and finally adds them together using a multi-operand adder. The delay of this architecture can be considered to be independent of the polynomial order. However as the order increases more powering units and multipliers and hence more area are needed.

The final architecture is the partial product array. In this architecture the coefficients and the operand are written in terms of their bits and the polynomial is expanded symbolically at design time. The terms that have the same numerical weight are grouped together and put in columns. The resulting matrix resembles the partial product array of the parallel multiplier and hence they can be added together using the multiplier reduction tree and carry propagate adder.

3.3.1 Iterative Architecture

We write equation 3.7 using horner method as follows:

$$
\begin{aligned}
P_{mn}(h) &= c_{m0} + c_{m1}h + c_{m2}h^2 + \cdots + c_{mn}h^n \\
&= c_{m0} + (\cdots + (c_{m(n-2)} + (c_{m(n-1)} + c_{mn}h)h)\cdots)h && (3.31)
\end{aligned}
$$

Equation 3.31 can be implemented iteratively using a fused multiply add unit (FMA) as follows:

$$c_{mn}h + c_{m(n-1)} \rightarrow T_0 \tag{3.32}$$

$$T_0h + c_{m(n-2)} \rightarrow T_1 \tag{3.33}$$

$$T_1h + c_{m(n-3)} \rightarrow T_2 \tag{3.34}$$

$$\vdots$$

$$T_{n-2}h + c_{m0} \rightarrow T_{n-1} \tag{3.35}$$

From the above equations we let $T_0, T_1, \ldots, T_{n-1}$ be stored in the same register that we denote by T and thus in every clock cycle we just read a coefficient from the coefficients table and add it to the product of T and h. After n clock cycles the value stored in T is the value of the polynomial that we seek to evaluate.

The first cycle is different from the rest in that we need to read two coefficients to add the $c_{m(n-1)}$ coefficient to the product of c_{mn} and h. To accomplish this special case we keep the c_{mn} coefficients in a separate table and use a multiplexer that selects either c_{mn} or the register T to be the multiplier of the FMA. The multiplexer selects c_{mn} at the first clock cycle and selects T at the remaining $(n-1)$ cock cycles.

Figure 3.4 depicts the details of this architecture. We note here that as the number of sub-intervals J increases the size of the coefficients table increases and vice versa. Also as the polynomial order n increases the number of cycles of this architecture increases and vice versa. Hence the trade off between the number of intervals J and the polynomial order n is mapped here as a trade off between the area of the tables and the delay of the circuit.

This architecture can be used to evaluate more than one function by simply adding coefficients tables for each function and selecting the output of the proper tables for evaluating the required function.

3.3.2 Parallel Architecture

The parallel architecture employs specialized powering units. The specialized powering units consumes less area than when implemented with multipliers and

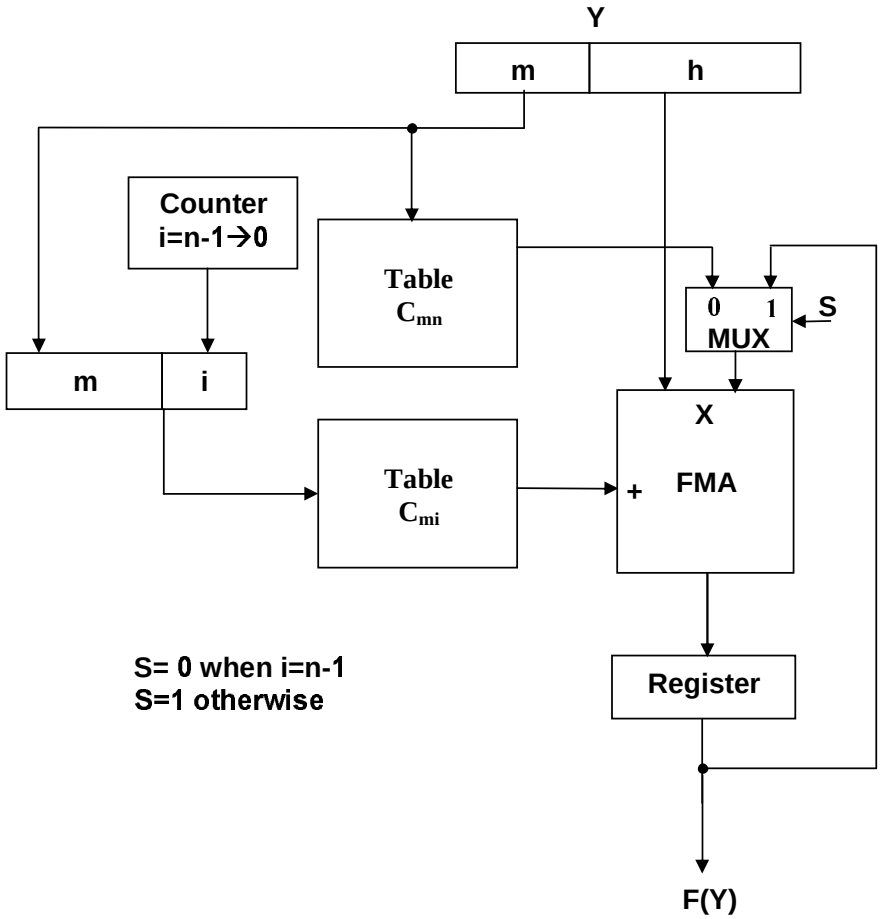


Figure 3.4: The Iterative Architecture

they are faster [35]. Furthermore the specialized powering units can be truncated.

We use the powering units to compute the powers of h then we multiply them by the coefficients in parallel and hence the name of the architecture. Finally we add the results using a multi operand adder as shown in figure 3.5.

The amount of hardware in this architecture can be prohibitive for higher order polynomials. It is only practical for low order polynomials. The speed of this architecture is independent of the polynomial order.

We note here as the number of sub-intervals J increases and consequently the polynomial order n for each sub-interval decreases the coefficients table increases in size and the number of powering units and multipliers decrease and vice versa. Thus the trade off between the number of sub-intervals and the polynomial order is mapped in this architecture as a trade off between the size of the coefficients table and the number of other arithmetic units and their sizes.

It is also to be noted that the arithmetic units used in this architecture can

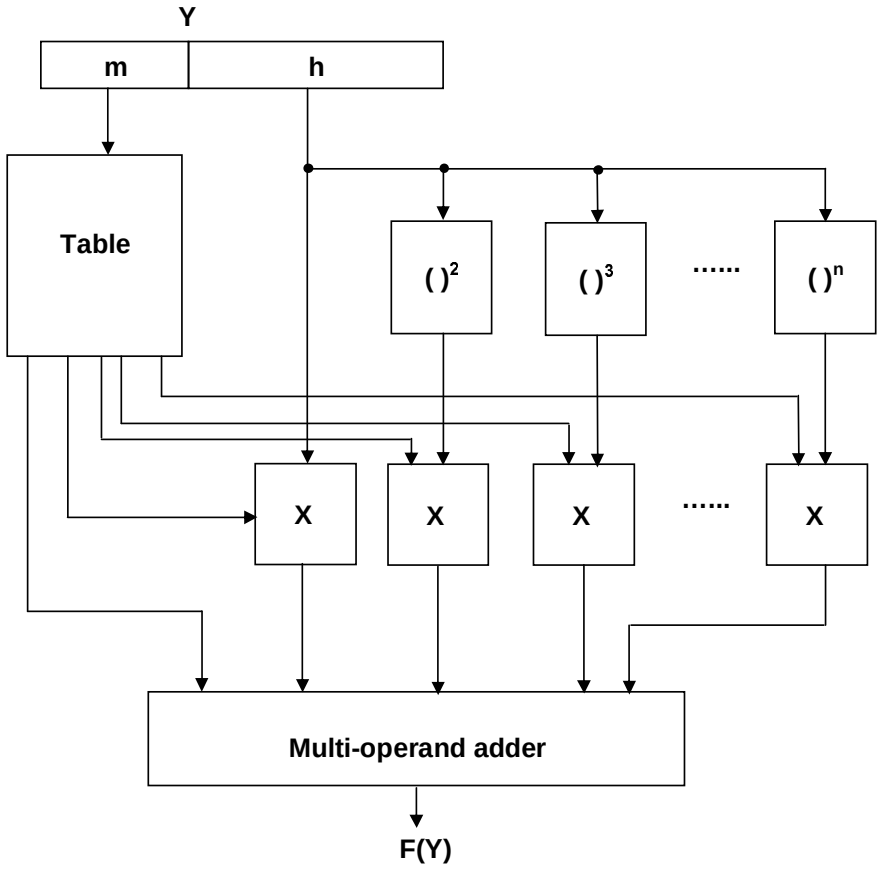


Figure 3.5: The Parallel Architecture

be shared over more than one function. For each function we only need a new coefficients table. In this case the optimal design is dependent on the number of the functions that we need to implement.

3.3.3 Partial Product Array

The partial product array technique was first proposed to evaluate the division operation [36].

The idea behind this method is that we write the coefficients and h in equation 3.7 in terms of their bits then we expand the polynomial symbolically. We next group the terms that have the same power of 2 numerical weight and write them in columns. The resulting matrix can be considered similar to the partial products that we obtain when we perform the multiplication operation. Hence we can add the rows of this matrix using the reduction tree and the carry propagate adder of the multiplier.

Figure 3.6 gives the general architecture of this technique.

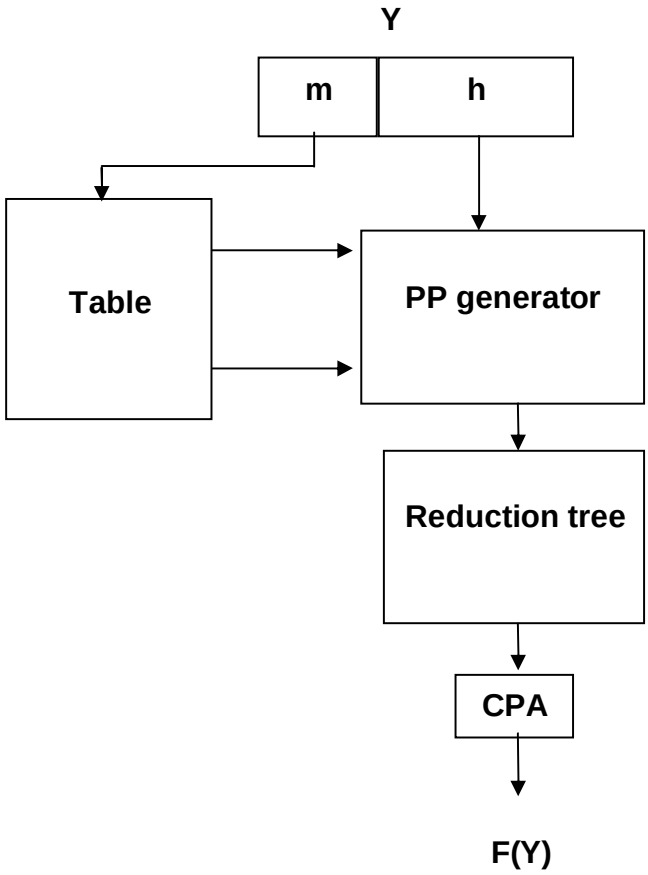


Figure 3.6: The Partial Product Array Architecture

We give a simple example to further illustrate the concept of this implementation technique. We assume $n = 2$ and that each of h, c_{m0}, c_{m1} and c_{m2} is represented by two bits as follows:

$$h = h_1 2^{-5} + h_2 2^{-6} \tag{3.36}$$

$$c_{m0} = c_{00} + c_{01} 2^{-1} \tag{3.37}$$

$$c_{m1} = c_{10} + c_{11} 2^{-1} \tag{3.38}$$

$$c_{m2} = c_{20} + c_{21} 2^{-1} \tag{3.39}$$

We next expand the polynomial $P_{m2}(h)$ symbolically and group the terms that have the same power of 2 numerical weight as follows:

$$\begin{aligned} P_{m2}(h) &= c_{00} + c_{01}2^{-1} + c_{10}h_1 2^{-5} + (c_{10}h_2 + c_{11}h_1)2^{-6} \\ &+ c_{11}h_2 2^{-7} + (c_{20}h_1 + c_{20}h_1h_2)2^{-10} + (c_{21}h_1 + c_{21}h_1h_2)2^{-11} \end{aligned}$$

$$+ \quad c_{20}h_2 2^{-12} + c_{21}h_2 2^{-13} \tag{3.40}$$

We next write them in the form of a partial product array as follows:

$$\begin{array}{cccccccccccccc} c_{00} & c_{01} & 0 & 0 & 0 & c_{10}h_1 & c_{10}h_2 & c_{11}h_2 & 0 & 0 & c_{20}h_1 & c_{21}h_1 & c_{20}h_2 & c_{21}h_2 \\ & & & & & & c_{11}h_1 & & & & c_{20}h_1h_2 & c_{21}h_1h_2 & & \end{array}$$

By adding these two vectors we obtain $P(Y)$. For larger problems the number of partial products will be larger and hence we need a reduction tree and a final carry propagate adder. The individual terms in the partial products are obtained using AND gates.

3.4 Error Analysis

The main goal of arithmetic circuits is the correctness of computation. Correctness in the context of the basic five arithmetic operations of the IEEE standard is defined as the rounded version of the infinite precision representation of the result. On the other hand in the context of elementary functions correctness is defined to be the case that the computed result is correct within one unit in the last place (1 ulp) of the infinite precision result which means that the computed result can be higher than the true result by no more than 1 ulp and it can be less than the true result by no more than 1 ulp.

In order to achieve this correctness goal careful error analysis has to be performed. There are two sources of error that need to be taken into consideration when performing error analysis. The first source is the approximation error. This error stems from the fact that we approximate the given function in a given interval by a polynomial. This error was given in section 3.2 along with the details of coefficients computation. The second source of error is the rounding error. It is implementation specific. Storing finite precision coefficients and rounding intermediate results are responsible for this error. The total error is the sum of the approximation error and rounding error. The basic principle used in estimating the rounding error is by bounding it as we see next.

We present techniques for bounding the rounding error that can be applied

to any architecture. All the hardware units used in the arithmetic circuits can be modeled by arithmetic operation specifically addition, multiplication or simple equality coupled with rounding of the result. We need to compute the bounds of the rounding error for any operation and most importantly we need to compute such bounds when the arguments themselves have suffered rounding from a previous step.

We need some notations before giving the bounding technique. We define the error as the difference between the true value and the rounded one. We denote the rounding error by ϵ_r. The error analysis seeks to find the bounds on the value of ϵ_r. Assuming that we have three variables, we denote their true values by $\hat{P_1}, \hat{P_2}$ and $\hat{P_3}$ and we denote their rounded values by P_1, P_2 and P_3 hence the rounding errors are $\epsilon_{r1} = \hat{P_1} - P_1$, $\epsilon_{r2} = \hat{P_2} - P_2$ and $\epsilon_{r3} = \hat{P_3} - P_3$.

In case that we add the two rounded variables P_1 and P_2 and introduce a new rounding error ϵ after the addition operation we need to determine the accumulated rounding error in the result. If we denote the result by P_3 then we can bound ϵ_{r3} as follows:

$$\begin{aligned} P_3 &= P_1 + P_2 - \epsilon && (3.41)\\ &= \hat{P_1} - \epsilon_{r1} + \hat{P_2} - \epsilon_{r2} - \epsilon && (3.42)\\ &= \hat{P_3} - \epsilon_{r1} - \epsilon_{r2} - \epsilon && (3.43)\\ \epsilon_{r3} &= \epsilon_{r1} + \epsilon_{r2} + \epsilon && (3.44) \end{aligned}$$

From Equation 3.44 we determine the bounds on the rounding error of P_3 from knowing the bounds on the rounding errors of P_1, P_2 and the rounding error after the addition operation.

$$\begin{aligned} min(\epsilon_{r3}) &= min(\epsilon_{r1}) + min(\epsilon_{r2}) + min(\epsilon) && (3.45)\\ max(\epsilon_{r3}) &= max(\epsilon_{r1}) + max(\epsilon_{r2}) + max(\epsilon) && (3.46) \end{aligned}$$

In case of multiplication, if we multiply P_1 and P_2 and introduce a rounding error after the multiplication equal to ϵ we need to bound the accumulated rounding error of the result. Again we denote the result by P_3 and its rounding

error that we seek by ϵ_{r3}.

$$\begin{aligned} P_3 &= P_1 \times P_2 - \epsilon && (3.47)\\ &= (\hat{P}_1 - \epsilon_{r1}) \times (\hat{P}_2 - \epsilon_{r2}) - \epsilon && (3.48)\\ &= \hat{P}_1 \times \hat{P}_2 - \hat{P}_2\epsilon_{r1} - \hat{P}_1\epsilon_{r2} + \epsilon_{r1}\epsilon_{r2} - \epsilon && (3.49)\\ &= \hat{P}_3 - \hat{P}_2\epsilon_{r1} - \hat{P}_1\epsilon_{r2} + \epsilon_{r1}\epsilon_{r2} - \epsilon && (3.50)\\ \epsilon_{r3} &= \hat{P}_2\epsilon_{r1} + \hat{P}_1\epsilon_{r2} - \epsilon_{r1}\epsilon_{r2} + \epsilon && (3.51) \end{aligned}$$

Using equation 3.51 we can bound the rounding error of the result from our knowledge of the bounds of the rounding errors of the inputs and the multiplication operation. Note that we can neglect the third term in equation 3.51 since it is smaller than the other terms and have opposite sign and that will result in a slight over estimation of the rounding error.

$$\begin{aligned} min(\epsilon_{r3}) &= min(\hat{P}_2) \times min(\epsilon_{r1}) + min(\hat{P}_1) \times min(\epsilon_{r2})\\ &- max(\epsilon_{r1}) \times max(\epsilon_{r2}) + min(\epsilon) && (3.52)\\ max(\epsilon_{r3}) &= max(\hat{P}_2) \times max(\epsilon_{r1}) + max(\hat{P}_1) \times max(\epsilon_{r2})\\ &- min(\epsilon_{r1}) \times min(\epsilon_{r2}) + max(\epsilon) && (3.53) \end{aligned}$$

What remains for complete rounding error analysis is the bounding of the rounding error that results from rounding a variable before storing in a table or after an arithmetic operation whether it is addition or multiplication. The rounding error depends on the sign of the variable and the rounding method. The most common rounding methods used are the truncation and the round to nearest.

If we truncate a positive variable after t bits from the binary point then the rounding error ϵ_r lie in the interval $[0, 2^{-t}]$. That is the rounding has a minimum value of 0 and a maximum value of 2^{-t}. If however the variable is negative then $\epsilon_r \in [-2^{-t}, 0]$.

Rounding to nearest after t bits from the binary point will cause a rounding error that is independent of the sign of the variable and lies in the interval $[-2^{-t-1}, 2^{-t-1}]$. The reason that makes the rounding error independent of the

sign of the variable is because the rounding can be either up or down for both cases.

3.5 Muller Truncation Algorithm

Muller [25] presents an algorithm to get the optimal second order approximating polynomial $P_{m2}(h) = c_{m0} + c_{m1}h + c_{m2}h^2$ with truncated coefficients. The decrease of the coefficients widths has an implementation advantage in the parallel architecture since it decreases the area of the multipliers. Compared to the direct rounding of the coefficients, Muller's algorithm gives results that are up to three bits more precise.

The algorithm is based on the observation that the maximum value of h is usually small and hence the maximum value of h^2 is even smaller. This observation leads to the conclusion that rounding c_{m1} has more effect on the final error than rounding c_{m2}. Based on this fact the algorithm first rounds c_{m1} to the nearest after t bits from the binary point and then seeks a new value for c_{m0} and c_{m2} to compensate part of the error introduced by rounding c_{m1}. We denote the new coefficients by $\hat{c}_{m0}, \hat{c}_{m1}$ and $\hat{c}_{m2}$. We then round $\hat{c}_{m0}$ and $\hat{c}_{m2}$ such that the resulting new rounding error is negligible compared to the total approximation error.

The goal of the algorithm is to make

$$\begin{aligned} c_{m0} \quad + \quad & c_{m1}h + c_{m2}h^2 \approx \hat{c}_{m0} + \hat{c}_{m1}h + \hat{c}_{m2}h^2 \\ h \quad \in \quad & [0, \Delta] \end{aligned} \tag{3.54}$$

That is we seek the polynomial that has truncated coefficients such that it is as close as possible to the original polynomial. After the first step of the algorithm we obtain $\hat{c}_{m1}$ by rounding c_{m1} to the nearest. We then rearrange equation 3.54 as follows:

$$(c_{m1} - \hat{c}_{m1})h \approx (\hat{c}_{m0} - c_{m0}) + (\hat{c}_{m2} - c_{m2})h^2 \tag{3.55}$$

We substitute $g = h^2$ in equation 3.55 to get:

$$\begin{aligned} (c_{m1} &- \hat{c}_{m1})\sqrt{g} \approx (\hat{c}_{m0} - c_{m0}) + (\hat{c}_{m2} - c_{m2})g \qquad (3.56) \\ g &\in [0, \Delta^2] \end{aligned}$$

If we approximate $\sqrt{g}$ by the first order minimax polynomial $\delta_0 + \delta_1 g$ we can obtain $\hat{c}_{m0}$ and $\hat{c}_{m2}$ as follows:

$$\hat{c}_{m0} = c_{m0} + (c_{m1} - \hat{c}_{m1})\delta_0 \qquad (3.57)$$

$$\hat{c}_{m2} = c_{m2} + (c_{m1} - \hat{c}_{m1})\delta_1 \qquad (3.58)$$

We can determine the values of δ_0 and δ_1 analytically using Chebychev minimax criteria given in section 3.2 directly without the need to run Remez algorithm. We define the error by $e(g) = \sqrt{g} - \delta_0 - \delta_1 g$. From the concavity of the square root function it is clear that $|e(g)|$ takes the maximum value at the three points $0, \Delta^2$ and a point in between that we denote by α as shown in figure 3.7.

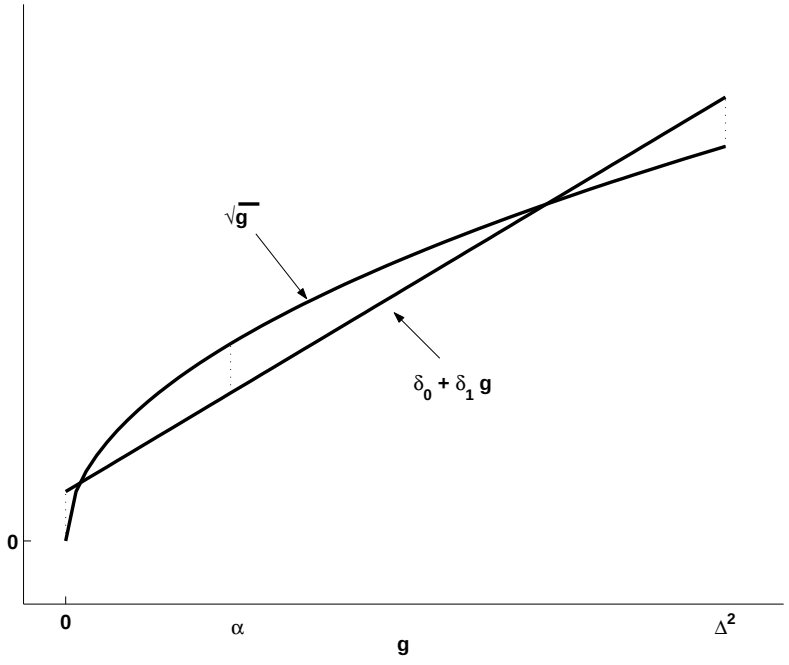


Figure 3.7: $\sqrt{g}$ and its first order minimax approximation $\delta_0 + \delta_1 g$

We can determine the value of α using calculus as follows:

$$\frac{\partial e(g)}{\partial g} = \frac{1}{2\sqrt{g}} - \delta_1 = 0 \,|_{g=\alpha} \tag{3.59}$$

$$\alpha = \frac{1}{(2\delta_1)^2} \tag{3.60}$$

For $\delta_0 + \delta_1 g$ to be a minimax approximation to $\sqrt{g}$ the absolute maximum error must be equal at the three points $0, \alpha$ and Δ^2 and alternating in sign i.e. $e(0) = -e(\alpha) = e(\Delta^2)$. Thus we have two equation in two unknowns δ_0 and δ_1. We solve the two equations as follows:

$$\begin{aligned} e(0) &= e(\Delta^2) \\ -\delta_0 &= \Delta - \delta_0 - \delta_1 \Delta^2 \\ \delta_1 &= \frac{1}{\Delta} \end{aligned} \tag{3.61}$$

$$\begin{aligned} e(0) &= -e(\alpha) \\ -\delta_0 &= -(\frac{1}{4\delta_1} - \delta_0) \\ \delta_0 &= \frac{1}{8\delta_1} = \frac{\Delta}{8} \end{aligned} \tag{3.62}$$

The maximum error in approximating $\sqrt{g}$ is given by the error at any of the three points a, α or Δ^2. The error is thus equal to $\frac{\Delta}{8}$.

We now substitute the values of δ_0 and δ_1 in equations 3.57 and 3.58 to get the values of the coefficients $\hat{c}_{m0}$ and $\hat{c}_{m2}$

$$\hat{c}_{m0} = c_{m0} + (c_{m1} - \hat{c}_{m1})\frac{\Delta}{8} \tag{3.63}$$

$$\hat{c}_{m2} = c_{m2} + (c_{m1} - \hat{c}_{m1})\frac{1}{\Delta} \tag{3.64}$$

The error added by this rounding algorithm is equal to the error in approximating $\sqrt{g}$ multiplied by the factor $(c_{m1} - \hat{c}_{m1})$. Therefore the total error ϵ is given by the following equation:

$$\epsilon = \epsilon_a + |(c_{m1} - \hat{c}_{m1})| \, \frac{\Delta}{8} \tag{3.65}$$

Where ϵ_a is the original approximation error before applying the truncation al-

gorithm.

If we use simple rounding of c_{m1} without modifying c_{m0} and c_{m2} then the error introduced by this rounding is equal to $|(c_{m1} - \hat{c}_{m1})|\, max(h)$ and since the maximum value of h is equal to Δ therefore the rounding error is equal to $|(c_{m1} - \hat{c}_{m1})|\, \Delta$ hence the total error is equal to $\epsilon_a + |(c_{m1} - \hat{c}_{m1})|\, \Delta$.

Comparing this error to the one we obtain from the algorithm it is clear that Muller's algorithm is better and for the case that the rounding error is larger than the approximation error Muller's algorithm gives a total error that is approximately one eighth of that of the direct rounding.

3.6 Our Contribution: Truncation Algorithm

In this section we present a similar algorithm to the one given in the previous section. The similarity lies in their goals but they differ in their approaches completely.

The motivation behind our algorithm is that we need to find an optimal approximating polynomial that has truncated coefficients but not necessarily restricted to the second order polynomial.

The algorithm we present is a numerical algorithm that is based on mathematical programming specifically Integer Linear Programming, ILP.

We modify the last iteration in Remez algorithm in order to put constraints on the widths of the coefficients. We modify the first step by first modeling equation 3.23 as a linear programming (LP) problem then we relax the requirement that the error at the given $(n + 2)$ points are equal in magnitude by introducing tolerance variables. This relaxation enables us to add precision constraints on the coefficients and the LP model becomes an ILP model. We solve the ILP model using the branch and bound algorithm to get the values of the coefficients. We then run the second step of Remez algorithm without modification in order to compute the maximum and minimum bounds of the approximation error.

Since we don't constrain the width of c_{m0} therefore we can make the maximum approximation error centered around the origin by adding the average of the maximum and minimum value of the approximation error to c_{m0}.

We modify equation 3.23 as follows

$$\begin{aligned} F(y_i) &- [c_{m0} + c_{m1}(y_i - a_m) + c_{m2}(y_i - a_m)^2 \\ &+ \cdots + c_{mn}(y_i - a_m)^n] = (-1)^i(q - s_i) \\ s_i &\geq 0 \\ i &= 0, 1, \ldots, n+1 \end{aligned} \tag{3.66}$$

Note that in the above equation we replaced E by q in order to avoid the tendency to assume that they are equal in magnitude.

In order to be as close as possible to the original equation we need to minimize the values of the $\{s_i\}$variables. We can do that by minimizing their maximum value since they are all positive and the resulting model is an LP model.
Minimize $max(s_i)$
Subject to

$$\begin{aligned} F(y_i) &- [c_{m0} + c_{m1}(y_i - a_m) + c_{m2}(y_i - a_m)^2 \\ &+ \cdots + c_{mn}(y_i - a_m)^n] = (-1)^i(q - s_i) \\ s_i &\geq 0 \\ i &= 0, 1, \ldots, n+1 \end{aligned} \tag{3.67}$$

This model is not in the standard LP form. We can convert it to the standard form by introducing a new variable s that is equal to $max(\{s_i\})$ hence $s \geq s_i, i = 0, 1, \ldots, n+1$. The model becomes:
Minimize s
Subject to

$$\begin{aligned} F(y_i) &- [c_{m0} + c_{m1}(y_i - a_m) + c_{m2}(y_i - a_m)^2 \\ &+ \cdots + c_{mn}(y_i - a_m)^n] = (-1)^i(q - s_i) \\ s_i &\geq 0 \\ s &\geq 0 \end{aligned}$$

$$\begin{aligned} s &\geq s_i \\ i &= 0, 1, \ldots, n+1 \end{aligned} \tag{3.68}$$

The presence of the tolerance variables permits us to add more precision constraints on the coefficients since without the tolerance variables the system has a unique solution and any more added constraints will render the system infeasible. The precision constraints are simply constraints on the number of bits after the binary point in the binary representation of the coefficients. We denote these number of bits by t. Such constraint can be seen as an integer constraint on the value of the coefficient multiplied by 2^t.

Hence the model becomes an ILP model as follows:
Minimize s
Subject to

$$\begin{aligned} & F(y_i) - [c_{m0} + c_{m1}(y_i - a_m) + c_{m2}(y_i - a_m)^2 \\ & + \cdots + c_{mn}(y_i - a_m)^n] = (-1)^i(q - s_i) \\ & s_i \geq 0 \\ & s \geq 0 \\ & s \geq s_i \\ & i = 0, 1, \ldots, n+1 \\ & c_{mk} \text{ has } t_k \text{ bits after the binary point} \\ & k = 1, 2, \ldots, n \end{aligned} \tag{3.69}$$

We solve the ILP model 3.69 using the branch and bound algorithm [37]. Some results of our algorithm and Muller's algorithm are given in table 3.1

It is clear from this table that our algorithm gives close results to that of Muller's algorithm. Both algorithms outperform the direct rounding. However our algorithm is applicable for high order.

3.7 Summary

This chapter presents polynomial approximation technique. In this technique we divide the interval of the reduced argument into a number of sub-intervals. For each sub-interval we design a polynomial of degree n that approximates the elementary function in that sub-interval. The coefficients of the approximating polynomials are stored in a table in the hardware implementation.

Techniques for designing the polynomial are given and compared. They are specifically Taylor approximation, Minimax approximation and Interpolation. For each of these polynomial types the approximation error is given.

Hardware Implementation is discussed using three hardware architectures, the iterative architecture, parallel architecture and the PPA architecture.

Another error arises from the rounding of the coefficients prior to storing them in tables and from rounding the intermediate variables during the computation. Techniques for bounding such errors are given. Such techniques are general and can be applied to any architecture.

Two algorithms for rounding the coefficients in an algorithmic method instead of direct rounding are presented. The first algorithm [25] is applicable for second order polynomials. It outperforms the direct rounding by up to 3 bits of precision in the final result. The second algorithm is applicable for any order. It gives close results to those of the first algorithm.

Function	Interval	order	J	t	error		
					Muller	Rounding	Our Work
Exp(x)	$[0,1[$	2	8	4	$2^{-10.8}$	2^{-8}	$2^{-10.97}$
		2	8	5	$2^{-11.75}$	$2^{-9.1}$	$2^{-11.88}$
		2	8	6	$2^{-13.1}$	$2^{-10.2}$	$2^{-13.1}$
		2	16	6	$2^{-13.8}$	2^{-11}	$2^{-13.94}$
		2	14	9	2^{-15}	$2^{-12.1}$	2^{-15}
		2	32	9	$2^{-17.9}$	2^{-15}	2^{-18}
		2	64	14	$2^{-23.27}$	$2^{-20.9}$	$2^{-23.33}$
		3	16	14	-	2^{-19}	2^{-23}
		4	8	15	-	$2^{-19.4}$	$2^{-24.3}$
Log(x)	$[1,2[$	2	8	6	$2^{-12.3}$	$2^{-9.87}$	$2^{-12.89}$
		2	32	10	$2^{-18.7}$	2^{-16}	$2^{-18.88}$
		2	64	14	$2^{-23.5}$	2^{-21}	$2^{-23.6}$
		3	16	14	-	$2^{-18.9}$	$2^{-23.1}$
		4	8	15	-	2^{-19}	2^{-24}
Sin(x)	$[0,1[$	2	8	5	$2^{-11.2}$	2^{-9}	$2^{-11.86}$
		2	16	7	$2^{-14.66}$	$2^{-12.54}$	$2^{-15.3}$
		2	64	14	$2^{-23.67}$	2^{-21}	$2^{-23.17}$
		3	16	14	-	2^{-19}	2^{-23}
		4	8	14	-	$2^{-18.35}$	2^{-23}
$\frac{1}{\sqrt{(x)}}$	$[1,2[$	2	8	6	$2^{-12.65}$	2^{-10}	2^{-13}
		2	16	10	$2^{-17.65}$	2^{-15}	$2^{-17.8}$
		2	64	14	$2^{-23.3}$	2^{-21}	$2^{-23.2}$
		3	16	15	-	2^{-20}	$2^{-23.8}$
		4	8	15	-	$2^{-19.3}$	2^{-24}

Table 3.1: Results of our algorithm for different functions, polynomial order, number of sub-intervals and coefficients precision in bits (t)

Chapter 4

Table and Add Techniques

We present in this chapter a class of algorithms called the Table and Add techniques. These are a special case of polynomial approximation. They are based on the first order polynomial approximation $P_{m1}(Y) = c_{m0} + c_{m1}h$ in which the multiplication in the second term is avoided. The multiplication is avoided by approximating the second term by a coefficient (Bipartite) or the sum of two coefficients (Tripartite) or the sum of more than two coefficients (Multipartite). We use a multi-operand adder to add c_{m0} and the other coefficients that approximate $c_{m1}h$. As the number of coefficients that approximate $c_{m1}h$ increase we approach the normal first order approximation that involves one multiplication and one addition since a multiplication can be viewed as a multi-operand addition. Hence the Multipartite is practical when the number of coefficients is small.

The rest of this chapter is organized as follows: In section 4.1 We present the Bipartite algorithm. In section 4.2 we present a variant of the Bipartite algorithm that makes use of symmetry to decrease the size of the table that hold the coefficient that approximates $c_{m1}h$. We present in section 4.3 the Tripartite algorithm as a step before we give the general Multipartite algorithm in section 4.4.

4.1 Bipartite

The Bipartite was first introduced in [19] to compute the reciprocal. A generalized Bipartite for computing other functions and formal description of the algorithm based on Taylor series was later given in [20, 21]. We present in this section the

formal description that is based on Taylor series.

Without loss of generality we assume the argument Y lies in the interval $[1, 2[$ and has the binary representation $Y = 1.y_1y_2y_3 \dots y_L$. We split the argument Y into three approximately equal parts m_1, m_2 and m_3 such that $m_1 = 1.y_1y_2y_3 \dots y_u$, $m_2 = 2^{-u}0.y_{u+1}y_{u+2} \dots y_{2u}$ and $m_3 = 2^{-2u}0.y_{2u+1}y_{2u+2} \dots y_L$. Hence $Y = m_1 + m_2 + m_3$. We approximate $F(Y)$ by the first order Taylor polynomial as follows:

$$F(Y) \approx F(m_1 + m_2) + F'(m_1 + m_2)(m_3) \tag{4.1}$$

$$\epsilon_{a1} = F''(\zeta_1)\frac{m_3^2}{2} \tag{4.2}$$

$$\zeta_1 \in [m_1 + m_2, Y]$$

$$\epsilon_{a1} \leq 2^{-4u-1} max(F'') \tag{4.3}$$

where ϵ_{a1} is the approximation error caused by retaining the first two terms of the Taylor series and discarding the other terms.

To compute $F(Y)$ using equation 4.1 we need one multiplication and one addition. In order to get rid of the multiplication we approximate $F'(m_1 + m_2)$ by the zero order Taylor expansion

$$F'(m_1 + m_2) \approx F'(m_1) \tag{4.4}$$

$$\epsilon_{a2} = F''(\zeta_2)(m_2) \tag{4.5}$$

$$\zeta_2 \in [m_1, m_1 + m_2]$$

$$\epsilon_{a2} \leq 2^{-u} max(F'') \tag{4.6}$$

where ϵ_{a2} is the error committed in such approximation. We substitute this result in equation 4.1 to get:

$$F(Y) \approx F(m_1 + m_2) + F'(m_1)(m_3) = c_1(m_1, m_2) + c_2(m_1, m_3) \tag{4.7}$$

$$\epsilon_a = \epsilon_{a1} + (m_3)\epsilon_{a2} \tag{4.8}$$

$$\epsilon_a \leq 2^{-4u-1} max(F'') + 2^{-3u} max(F'') \tag{4.9}$$

Using equation 4.7 we store the first term $c_1(m_1, m_2)$ in a table that is indexed by m_1 and m_2 and we store the second term $c_2(m_1, m_3)$ in another table that is indexed by m_1 and m_3. The resulting architecture is shown in figure 4.1. If the lengths of m_1, m_2 and m_3 are approximately equal and equal to $\frac{L}{3}$ then the number of address bits for each table is $\frac{2L}{3}$. This causes a significant decrease in the area of the tables over the direct table lookup that requires L address bits.

Another error arises from rounding c_1 and c_2. If we round both of c_1 and c_2 to the nearest after t fraction bits then the rounding error is given by

$$\epsilon_r \leq 2 \times 2^{-t-1} = 2^{-t} \tag{4.10}$$

Note that the second coefficient is small and less than $2^{-2u}F'(m_1)$ hence it will have a number of leading zeros or ones depending on its sign. Those leading zeros or ones needn't be stored. That will decrease the size of the second table. The exact number of leading zeros or ones depends on the function that we are approximating.

The total error is equal to the sum of the approximation error, coefficients rounding error and the rounding of the final result. The total error must lie within 1 ulp of the true value in order to be a faithful approximation.

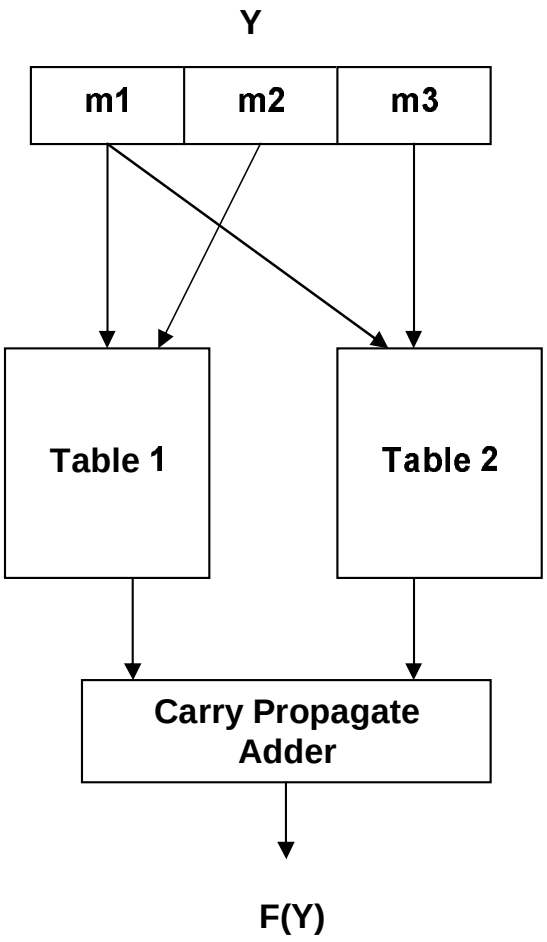


Figure 4.1: The architecture of the Bipartite algorithm

4.2 Symmetric Bipartite

A variant of the Bipartite algorithm is given in [20]. It is called the symmetric Bipartite. It makes use of symmetry in the second table in order to decrease its size to the half of its original size at the expense of adding simple logic gates before and after the second table.

Again we split the argument Y into three parts m_1, m_2 and m_3. To be more general we let the the three parts have unequal sizes u_1, u_2 and $L - u_1 - u_2$ bits.

$$\begin{aligned} m_1 &= 1.y_1 y_2 \dots y_{u_1} \\ m_2 &= 2^{-u_1} 0.y_{u_1+1} y_{u_1+2} \dots y_{u_1+u_2} \\ m_3 &= 2^{-u_1-u_2} 0.y_{u_1+u_2+1} y_{u_1+u_2+2} \dots y_L \end{aligned}$$

such that $Y = m_1 + m_2 + m_3$. We approximate $F(Y)$ using first order Taylor series as follows:

$$F(Y) \approx F(m_1 + m_2 + \lambda_2) + F'(m_1 + m_2 + \lambda_2)(m_3 - \lambda_2) \quad (4.11)$$

$$\lambda_2 = 2^{-u_1-u_2-1} - 2^{-L-1} \quad (4.12)$$

λ_2 is exactly halfway between the minimum and maximum of m_3. The error committed by this approximation is given by

$$\epsilon_{a1} = F''(\zeta_1)\frac{(m_3 - \lambda_2)^2}{2} \quad (4.13)$$

$$\zeta_1 \in [m_1 + m_2, Y] \quad (4.14)$$

$$\epsilon_{a1} \leq 2^{-2u_1-2u_2-3} max(F'') \quad (4.15)$$

We approximate $F'(m_1 + m_2 + \lambda_2)$ by the zero order Taylor expansion as follows:

$$F'(m_1 + m_2 + \lambda_2) \approx F'(m_1 + \lambda_1 + \lambda_2) \quad (4.16)$$

$$\lambda_1 = 2^{-u_1-1} - 2^{-u_1-u_2-1} \quad (4.17)$$

λ_1 is exactly halfway between the minimum and maximum of m_2. The error

committed by this approximation is given by

$$\begin{aligned} \epsilon_{a2} &= F''(\zeta_2)(m_2 - \lambda_1) && (4.18) \\ \zeta_2 &\in [m_1, m_1 + m_2] && (4.19) \\ \epsilon_{a2} &\leq 2^{-u_1-1} max(F'') && (4.20) \end{aligned}$$

By substituting in equation 4.11 we get:

$$\begin{aligned} F(Y) &\approx F(m_1 + m_2 + \lambda_2) + F'(m_1 + \lambda_1 + \lambda_2)(m_3 - \lambda_2) && (4.21) \\ \epsilon_a &= \epsilon_{a1} + (m_3 - \lambda_2)\epsilon_{a2} && (4.22) \\ \epsilon_a &\leq 2^{-2u_1-2u_2-3} max(F'') + 2^{-2u_1-u_2-2} max(F'') && (4.23) \end{aligned}$$

where ϵ_a is the total approximation error. The rounding error is the same as in the previous section.

Equation 4.21 defines the two coefficients of the symmetric Bipartite algorithm

$$\begin{aligned} F(Y) &= c_1(m_1, m_2) + c_2(m_1, m_3) && (4.24) \\ c_1(m_1, m_2) &= F(m_1 + m_2 + \lambda_2) && (4.25) \\ c_2(m_1, m_3) &= F'(m_1 + \lambda_1 + \lambda_2)(m_3 - \lambda_2) && (4.26) \end{aligned}$$

We store c_1 in a table that is indexed by m_1 and m_2. We can store c_2 in a table that is indexed by m_1 and m_3. However we can reduce the size of the table that stores c_2 to about the half by making use of the following symmetry properties

1. $2\lambda_2 - m_3$ is the one's complement of m_3

2. $c_2(m_1, 2\lambda_2 - m_3)$ is the negative of $c_2(m_1, m_3)$

To reduce the size of the table that stores c_2 we examine the most significant bit of m_3 if it is 0 we use the remaining bits to address the table and read the coefficient. If the most significant bit of m_3 is 1 we complement the remaining bits of m_3 and address the table and then complement the result.

Note here that we approximate the negative of the table output by its one's complement. The error of such approximation is equal to 1 ulp. This error is to be taken into consideration when computing the total error.

This way we decreased the number of address bits of the second table by 1 hence we decreased its size to the half. Figure 4.2 gives the resulting architecture.

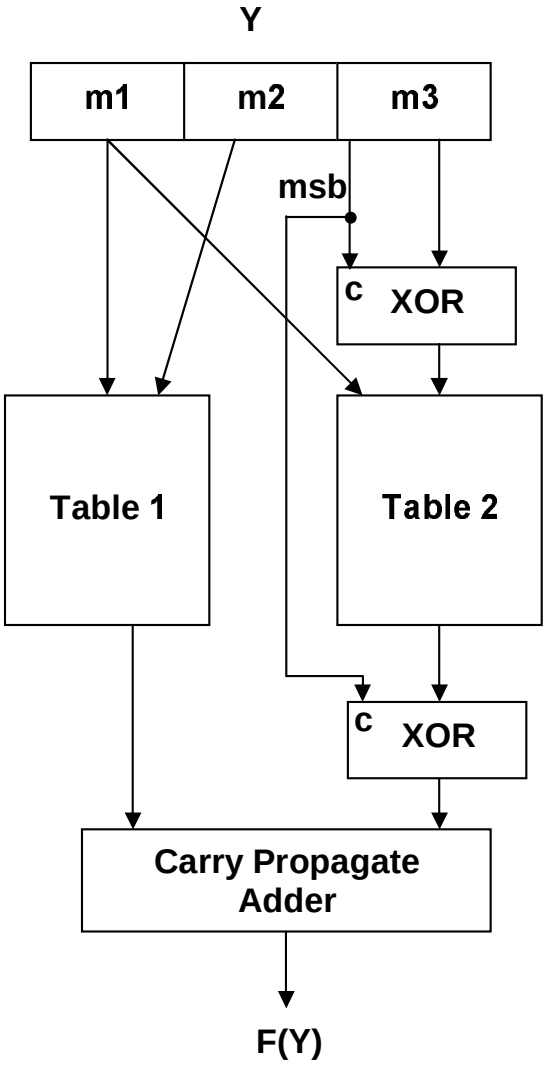


Figure 4.2: The architecture of the symmetric Bipartite algorithm

4.3 Tripartite

We generalize the Bipartite algorithm by splitting the argument Y to more than 3 parts. We present in this section the Tripartite algorithm [21] in which we divide the argument Y into 5 parts. In the following section we present the general case in which we divide the input argument into any odd number of parts. We divide the argument Y into the five parts of approximately equal number of bits:

$$m_1 = 1.y_1 y_2 \dots y_u$$

$$m_2 = 2^{-u} 0.y_{u+1} y_{u+2} \dots y_{2u}$$

$$m_3 = 2^{-2u} 0.y_{2u+1} y_{2u+2} \dots y_{3u}$$

$$m_4 = 2^{-3u} 0.y_{3u+1} y_{3u+2} \dots y_{4u}$$

$$m_5 = 2^{-4u} 0.y_{4u+1} y_{4u+2} \dots y_L$$

such that $Y = m_1 + m_2 + m_3 + m_4 + m_5$. We approximate $F(Y)$ using the first order Taylor expansion as follows:

$$F(Y) \approx F(m_1 + m_2 + m_3) + F'(m_1 + m_2 + m_3)(m_4 + m_5) \tag{4.27}$$

The approximation error caused by keeping the first two terms of Taylor expansion and discarding the rest is given by:

$$\begin{aligned} \epsilon_{a1} &= F''(\zeta_1)\frac{(m_4 + m_5)^2}{2} && (4.28)\\ \zeta_1 &\in [m_1 + m_2 + m_3, Y] \\ \epsilon_{a1} &\leq 2^{-6u-1} max(F'') && (4.29) \end{aligned}$$

We then split the second term of equation 4.27 into the two terms $F'(m_1 + m_2 + m_3)m_4$ and $F'(m_1 + m_2 + m_3)m_5$. We approximate them as follows:

$$\begin{aligned} F'(m_1 + m_2 + m_3)m_4 &\approx F'(m_1 + m_2)m_4 && (4.30)\\ F'(m_1 + m_2 + m_3)m_5 &\approx F'(m_1)m_5 && (4.31)\\ & && (4.32) \end{aligned}$$

The approximation error is given by

$$\begin{aligned}
\epsilon_{a2} &= F''(\zeta_2)m_3m_4 && (4.33)\\
\zeta_2 &\in [m_1+m_2, m_1+m_2+m_3]\\
\epsilon_{a2} &\leq 2^{-5u}max(F'') && (4.34)\\
\epsilon_{a3} &= F''(\zeta_3)(m_2+m_3)m_5 && (4.35)\\
\zeta_3 &\in [m_1, m_1+m_2+m_3]\\
\epsilon_{a3} &\leq (2^{-5u}+2^{-6u})max(F'') && (4.36)
\end{aligned}$$

By substituting in equation 4.27 we get

$$F(Y) \approx F(m_1+m_2+m_3) + F'(m_1+m_2)m_4 + F'(m_1)m_5 \quad (4.37)$$

and the total approximation error is given by

$$\begin{aligned}
\epsilon_a &= \epsilon_{a1}+\epsilon_{a2}+\epsilon_{a3} && (4.38)\\
\epsilon_a &\leq (2^{-6u-1}+2^{-5u}+2^{-5u}+2^{-6u})max(F'') \approx 2^{-5u+1}max(F'') && (4.39)
\end{aligned}$$

Equation 4.37 defines the three coefficients of the Tripartite algorithm which we denote by c_1, c_2 and c_3

$$\begin{aligned}
F(Y) &= c_1(m_1,m_2,m_3)+c_2(m_1,m_2,m_4)+c_3(m_1,m_5) && (4.40)\\
c_1(m_1,m_2,m_3) &= F(m_1+m_2+m_3) && (4.41)\\
c_2(m_1,m_2,m_4) &= F'(m_1+m_2)m_4 && (4.42)\\
c_3(m_1,m_5) &= F'(m_1)m_5 && (4.43)
\end{aligned}$$

The architecture is given in figure 4.3. The length in bits of each of m_1, m_2, m_3, m_4 and m_5 is approximately equal to $\frac{L}{5}$ hence the number of address bits for the first and second tables is $\frac{3L}{5}$ each and the number of address bits for the third table is $\frac{2L}{5}$. The second and third coefficients have a number of leading zeros or ones depending on the sign of the coefficients. These leading ones or zeros needn't be stored in the tables and they can be obtained by extending the sign bit. If we

round the three coefficients to the nearest after t fraction bits then the rounding error is given by

$$\epsilon_r = 3 \times 2^{-t-1} \tag{4.44}$$

The total error is thus given by

$$\epsilon = 2^{-5u+1} max(F'') + 3 \times 2^{-t-1} \tag{4.45}$$

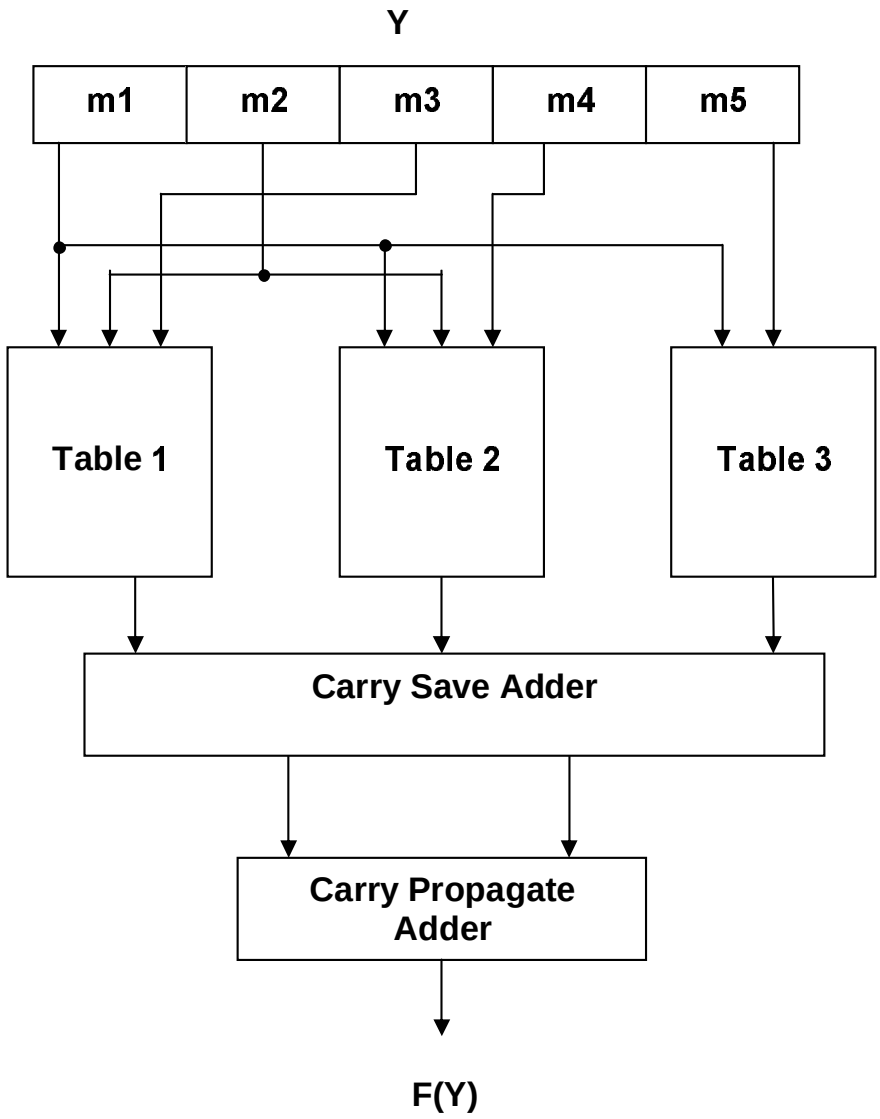


Figure 4.3: The architecture of the Tripartite algorithm

4.4 Multipartite

The previous algorithms can be generalized to what is called the Multipartite [21]. Similar work is found in [22] and it is called STAM or symmetric Table Addition Method. We present in this section the Multipartite algorithm.

We divide the argument Y into $2W+1$ parts which we denote by $m_1, m_2, \ldots m_{2W+1}$. Each part has u bits such that $L = (2W+1)u$. Hence $m_i \leq 2^{-(i-1)u}$ for $i = 2, 3, \ldots$.

$$Y = \sum_{i=1}^{i=2W+1} m_i$$

We approximate $F(Y)$ by the first order Taylor expansion as follows

$$F(Y) = F(\sum_{i=1}^{i=W+1} m_i) + F'(\sum_{i=1}^{i=W+1} m_i)(\sum_{i=w+2}^{i=2W+1} m_i) \tag{4.46}$$

We decompose the second term of the Taylor expansion into W terms and approximate each term as follows

$$\begin{aligned} &F'(\sum_{i=1}^{i=W+1} m_i)m_j \approx F'(\sum_{i=1}^{i=2W+2-j} m_i)m_j \\ &j = w+2, w+3, \ldots, 2w+1 \end{aligned} \tag{4.47}$$

Substituting in equation 4.46 we get

$$F(Y) = F(\sum_{i=1}^{i=W+1} m_i) + \sum_{j=w+2}^{j=2w+1} (F'(\sum_{i=1}^{i=2W+2-j} m_i)m_j) \tag{4.48}$$

Each term in equation 4.48 is stored in a separate table that is addressed by the bits of the argument that affect its value as follows:

$$c_1 = F(\sum_{i=1}^{i=W+1} m_i) \tag{4.49}$$

$$c_2 = F'(\sum_{i=1}^{i=W} m_i)m_{W+2} \tag{4.50}$$

$$c_3 = F'(\sum_{i=1}^{i=W-1} m_i)m_{W+3} \tag{4.51}$$

$$\vdots$$

$$c_{W+1} = F'(m_1)m_{2W+1} \tag{4.52}$$

The output of these tables are added by a multi-operand adder to give the approximation to $F(Y)$.

The approximation error is given by:

$$\epsilon_a \leq (2^{(-2W-2)u} + W2^{(-2W-1)u})max(F'') \tag{4.53}$$

If we round the coefficients to the nearest after t fraction bits then the rounding error is given by:

$$\epsilon_r = (W+1)2^{-t-1} \tag{4.54}$$

The total error is the sum of the approximation and rounding error in addition to the rounding of the final output.

4.5 Summary

This chapter presents Table and Add algorithms. Such algorithms are well suited for single precision approximation. They employ two or more tables and addition.

They are a special case of the polynomial approximation technique. They are a first order polynomial approximation based on Taylor approximation in which the multiplication involved in the second term is avoided by using one or more tables.

They are classified according to the number of tables they employ into Bipartite (two tables), Tripartite (three tables) and generally Multipartite (more than three tables).

More tables don't imply more area. In fact the more tables we use the less area we need for the tables at the expense of the delay of adding the outputs of such tables.

A variant of the Bipartite is presented. It is called the Symmetric Bipartite [20]. It makes use of a symmetry property in the second table in order to decrease its size to almost the half.

Chapter 5

A Powering Algorithm

In this chapter we present an algorithm that is given in [17]. We give the description of the algorithm in section 5.1, its theoretical error analysis in section 5.2. In section 5.3, we propose [28] a new algorithmic error analysis for the powering algorithm that gives a tighter bound on the maximum error.

5.1 Description of the Powering Algorithm

This powering algorithm is based on the first order Taylor approximation. It differs from the normal Taylor approximation in that instead of storing the two coefficients c_{m0} and c_{m1} it stores one coefficient that we denote by c_m and multiply it by a modified operand that we denote by $\tilde{Y}$. The modification of the operand is simply through rewiring and inversion of some bits for some cases and the use of a special booth recoder in other cases. The advantage of this algorithm over the ordinary first order approximation is that it uses smaller storage area since it stores one coefficient instead of two. Moreover it uses one multiplication instead of a multiplication and addition and that decreases the area and the delay of the hardware implementation. The multiplier size is larger than the ordinary first order approximation hence this algorithm is practical when we have an existing multiplier.

We assume without loss of generality that the argument Y lies in the interval $[1, 2[$ and therefore has the binary representation $Y = 1.y_1y_2y_3 \dots y_L$. The algorithm computes the function $F(Y) = Y^p$ where p is a constant that can take

the forms: $p = \pm 2^{k1}$ or $p = \pm 2^{k1} \pm 2^{k2}$ where $k1$ is an integer while $k2$ is a non-negative integer.

We split the argument Y into two parts $m = 1.y_1y_2y_3 \ldots y_u$ and $h = 2^{-u}0.y_{u+1}y_{u+2} \ldots y_L$ hence $Y = m + h$. Simple mathematical manipulations give:

$$Y^p = (m+h)^p = (m + 2^{-u-1} + h - 2^{-u-1})^p \quad (5.1)$$

$$= (m + 2^{-u-1})^p (1 + \frac{h - 2^{-u-1}}{m + 2^{-u-1}})^p \quad (5.2)$$

We expand the second factor in equation 5.2 using Taylor method to get

$$Y^p \approx (m + 2^{-u-1})^p (1 + p\frac{h - 2^{-u-1}}{m + 2^{-u-1}}) \quad (5.3)$$

$$Y^p \approx (m + 2^{-u-1})^{p-1} (m + 2^{-u-1} + p(h - 2^{-u-1})) \quad (5.4)$$

The first factor in equation 5.4 is the coefficient c_m which is stored in a table that is addressed by m while the second factor is $\tilde{Y}$ which can be computed from Y by simple modification. The resulting hardware architecture is shown in figure 5.1. There are five cases for the value of p for which we show how to compute $\tilde{Y}$ from Y.

case(1): $p = 2^{-k}$ where k is a non-negative integer. In this case $\tilde{Y} = (m + 2^{-u-1} + 2^{-k}h - 2^{-k-u-1}))$ Expanding $\tilde{Y}$ in bits gives:

$$\begin{aligned}
\tilde{Y} =& 1.y_1 \cdots y_u 1 \quad 0 \cdots \quad 0 \quad y_{u+1}y_{u+2} \cdots y_L \\
& -0.0 \cdots 0 \; 0 \quad 0 \cdots \quad 0 \quad 1 \quad 0 \quad \cdots 0 \\
\tilde{Y} =& 1.y_1 \cdots y_u 0 \quad 0 \cdots \quad 0 \quad y_{u+1}y_{u+2} \cdots y_L \\
& +0.0 \cdots 0 \; 0 \quad 1 \cdots \quad 1 \quad 1 \quad 0 \quad \cdots 0 \\
\tilde{Y} =& 1.y_1 \cdots y_u y_{u+1} \overline{y}_{u+1} \cdots \overline{y}_{u+1} \overline{y}_{u+1} y_{u+2} \cdots y_L
\end{aligned}$$

There are k $\overline{y}_{u+1}$ between y_{u+1} and y_{u+2}. It is clear that $\tilde{Y}$ can be obtained from Y by simple rewiring and inversion in this case.

case(2): $p = -2^{-k}$ where k is a non-negative integer. In this case $\tilde{Y} = (m + 2^{-u-1} - 2^{-k}h + 2^{-k-u-1}))$ Expanding $\tilde{Y}$ in bits gives:

$$
\begin{aligned}
\tilde{Y} =& 1.y_1 \quad \cdots y_u 1 \quad 0 \cdots \quad 0 \quad 1 \quad 0 \quad \cdots 0 \\
& -0.0 \quad \cdots 0 \ 0 \quad 0 \cdots \quad 0 \quad y_{u+1} y_{u+2} \cdots y_L \\
\tilde{Y} =& 1.y_1 \quad \cdots y_u 1 \quad 0 \cdots \quad 0 \quad 1 \quad 0 \quad \cdots 1 \\
& +1.1 \quad \cdots 1 \ 1 \quad 1 \cdots \quad 1 \quad \overline{y}_{u+1} \overline{y}_{u+2} \cdots \overline{y}_L \\
\tilde{Y} =& 1.y_1 \quad \cdots y_u \overline{y}_{u+1} y_{u+1} \cdots y_{u+1} y_{u+1} \overline{y}_{u+2} \cdots \overline{y}_L \\
& +2^{-L-k}
\end{aligned}
$$

There are k y_{u+1} bits between $\overline{y_{u+1}}$ and $\overline{y_{u+2}}$. We neglect the term 2^{-L-k} and therefore $\tilde{Y}$ can be obtained from Y by simple rewiring and inversion for some bits.

case(3): $p = 2^k$ where k is a positive integer. In this case $\tilde{Y} = (m + 2^{-u-1} + 2^k h - 2^{k-u-1}))$. We decompose $\tilde{Y}$ into two bit vectors $m + 2^{-u-1} = 1.y_1 y_2 \ldots y_u 1$ and $2^k h - 2^{k-u-1} = 2^{k-u}(0.\hat{y}_{u+1} y_{u+2} \ldots y_L)$ such that $\hat{y}_{u+1} = 0$ when $y_{u+1} = 1$ else $\hat{y}_{u+1} = -1$. We sum the two vectors using a special booth recoder described in [17]. The delay of this recoder is independent of L.

case(4): $p = -2^k$ where k is a positive integer. In this case $\tilde{Y} = (m + 2^{-u-1} - 2^k h + 2^{k-u-1}))$. We decompose $\tilde{Y}$ into two bit vectors $m + 2^{-u-1} = 1.y_1 y_2 \ldots y_u 1$ and $2^{k-u-1} - 2^k h = 2^{k-u}(0.\hat{y}_{u+1} \overline{y}_{u+2} \ldots \overline{y}_L) + 2^{-L-k}$ such that $\hat{y}_{u+1} = 0$ when $y_{u+1} = 0$ else $\hat{y}_{u+1} = -1$. By neglecting the 2^{-L-k} term We obtain $\tilde{Y}$ from Y by using inverters and a special booth recoder as in case(3). The delay of such modification is constant and independent on L.

case(5):$p = \pm 2^{k_1} \pm 2^{-k_2}$ where k_1 is an integer while k_2 is a non-negative integer. In this case $\tilde{Y} = (m + 2^{-u-1}) + (\pm 2^{k_1} \pm 2^{-k_2})(h - 2^{-u-1})$. We split $\tilde{Y}$ into two parts $(m + 2^{-u-1}) \pm 2^{-k_2}(h - 2^{-u-1})$ and $\pm 2^{k_1}(h - 2^{-u-1})$. The first part can be obtained as in case (1) or (2) while the second part can be added to the first using a special booth recoder.

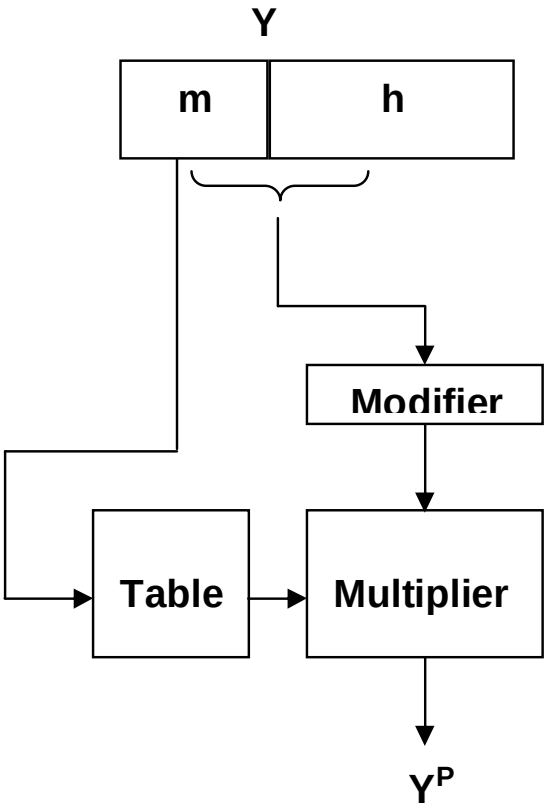


Figure 5.1: Architecture of the Powering Algorithm

5.2 Theoretical Error analysis

The error is defined as the difference between the true value and the computed value. There are two sources of error in this algorithm. The first error is the approximation error ϵ_a. It results from approximating the second factor in equation 5.2 by the first two terms of its Taylor expansion. The second error is the rounding error ϵ_r that results from truncating the coefficient c_m to t bits after the binary point. The total error ϵ is the sum of these two errors.

The error in Taylor approximation is given by the first omitted term evaluated at an unknown point ζ. We bound the approximation error by choosing the value of ζ that maximizes the error expression.

The approximation error is thus given by the third term of Taylor expansion of the second factor of equation 5.2 multiplied by the first factor and the value of h and m are chosen to maximize the resulting expression.

$$\epsilon_a = \frac{p(p-1)}{2}(h-2^{-u-1})^2(m+2^{-u-1})^{p-2} \tag{5.5}$$

$$|\epsilon_a| \leq \left|\frac{p(p-1)}{2}2^{-2u-2}(m+2^{-u-1})^{p-2}\right| \tag{5.6}$$

$$|\epsilon_a| \leq \left|\frac{p(p-1)}{2}2^{-2u-2}\right| \ ,p \leq 2 \tag{5.7}$$

$$|\epsilon_a| \leq \left|\frac{p(p-1)}{2}(2^{-2u-2})(2^{p-2})\right| \ ,p > 2 \tag{5.8}$$

$$|\epsilon_a| \leq \left|\frac{p(p-1)}{2}(2^{-2u-2})max(1,2^{p-2})\right| \tag{5.9}$$

ϵ_a is negative when p lies in the interval [0,1] and it is positive otherwise. Therefore

$$\epsilon_a \in [\frac{p(p-1)}{2}(2^{-2u-2})max(1,2^{p-2}),0] \ ,0 < p < 1 \tag{5.10}$$

$$\epsilon_a \in [0,\frac{p(p-1)}{2}(2^{-2u-2})max(1,2^{p-2})] \ ,p < 0 or p > 1 \tag{5.11}$$

The approximation error can be improved as described in [17] by adjusting the value of the coefficient c_m. When c_m is given by the following equation:

$$c_m = m^{p-1} + (p-1)2^{-u-1}m^{p-2} + (p-1)(3P-4)2^{-2u-4}m^{p-3} \tag{5.12}$$

The absolute value of the approximation error becomes half its old value however the approximation error in this case can be positive or negative. It is not directed in one side of the origin as in the previous case.

The rounding error is caused by rounding the coefficient c_m. If we truncate c_m after t bits from the binary point then the maximum rounding error in representing c_m is 2^{-t} and since we multiply c_m by $\tilde{Y}$ to get the final result therefore the maximum rounding error is equal to 2^{-t} multiplied by the maximum value of $\tilde{Y}$ which is 2 hence the interval of the rounding error is given by:

$$\epsilon_r \in [0, 2^{-t+1}] \tag{5.13}$$

If we round c_m to the nearest after t bits from the binary point then the rounding error is given by:

$$\epsilon_r \in [-2^{-t}, 2^{-t}] \tag{5.14}$$

If we round c_m up after t bits from the binary point then the rounding error is given by:

$$\epsilon_r \in [-2^{-t+1}, 0] \tag{5.15}$$

The total error ϵ is the sum of the approximation error ϵ_a and the rounding error ϵ_r hence the interval in which ϵ lies is equal to the sum of the two intervals of ϵ_a and ϵ_r.

5.3 Our Contribution: Algorithmic Error Analysis

The error analysis in the previous section is not tight. It gives a bound to the total error of the algorithm. In this section, we propose an algorithm given in [28] that computes a tight bound to the total error of this powering method.

Since we define the error as the difference between the true value and the

computed value therefore the error can be given by the following equation:

$$\epsilon = Y^p - c_m\tilde{Y} \tag{5.16}$$

$$\epsilon = (m+h)^p - c_m(m + 2^{-u-1} + ph - p2^{-u-1}) \tag{5.17}$$

Where c_m depends on m, p and u and is given by the first factor of equation 5.4 or the modified version given by equation 5.12.

In order to find the maximum value of ϵ we may think of using exhaustive search for all the combinations of m and h. This technique is not feasible for large operand width because of the excessive time this search takes. However if we manage to remove h from the equation of ϵ we can use exhaustive search on m since the number of combinations of m is small. We can remove h from the equation of ϵ by finding the value of h that maximizes the magnitude of ϵ analytically as follows:

$$\frac{\partial \epsilon}{\partial h} = p(m+h)^{p-1} - c_m p \tag{5.18}$$

$$\frac{\partial^2 \epsilon}{\partial h^2} = p(p-1)(m+h)^{p-2} \tag{5.19}$$

In case that p lies in the interval $[0, 1]$ then the value of the second derivative of ϵ with respect to h is negative hence the extreme point obtained from equating equation 5.18 to zero is a local maximum. On the other hand when p lies outside the interval $[0, 1]$ the second derivative is positive hence the extreme point is a local minimum. The error function ϵ is a convex function versus h hence we need to consider only the endpoints of h: 0 and $2^{-u} - 2^{-L}$ as well as the extreme point (local minimum or local maximum) $c^{\frac{1}{p-1}} - m$. Substituting these three values in equation 5.17 we get three different error functions that we denote by ϵ_1, ϵ_2 and ϵ_3. These error functions are independent of h. We perform exhaustive search on the value of m in these three functions and pick the minimum and the maximum to be the endpoints of the error interval.

Note that the error is in one direction from the origin when $p \in [0, 1]$ and c_m is rounded up or when $p \notin [0, 1]$ and c_m is truncated as shown in figures 5.2 and 5.3. Otherwise the error can be either + or −.

In the case that $p \in [0,1]$ and c_m is rounded up the error is negative and thus the maximum error is 0 and we don't need to consider the local maximum point.

In the case that $p \notin [0,1]$ and c_m is truncated the error is positive hence the minimum error is 0. Therefore in this case we don't need to consider the local minimum point.

Hence in these two cases we need only consider the endpoints of h. In other cases we need to consider the endpoints of h as well as the extreme point.

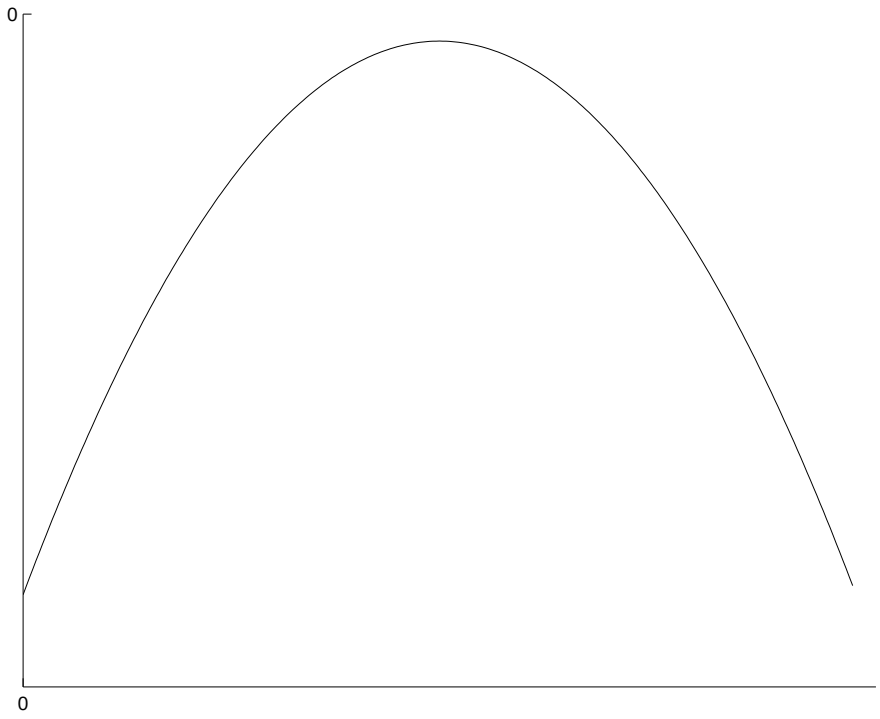


Figure 5.2: The error function versus h when $p \in [0, 1]$ and we round c_m up

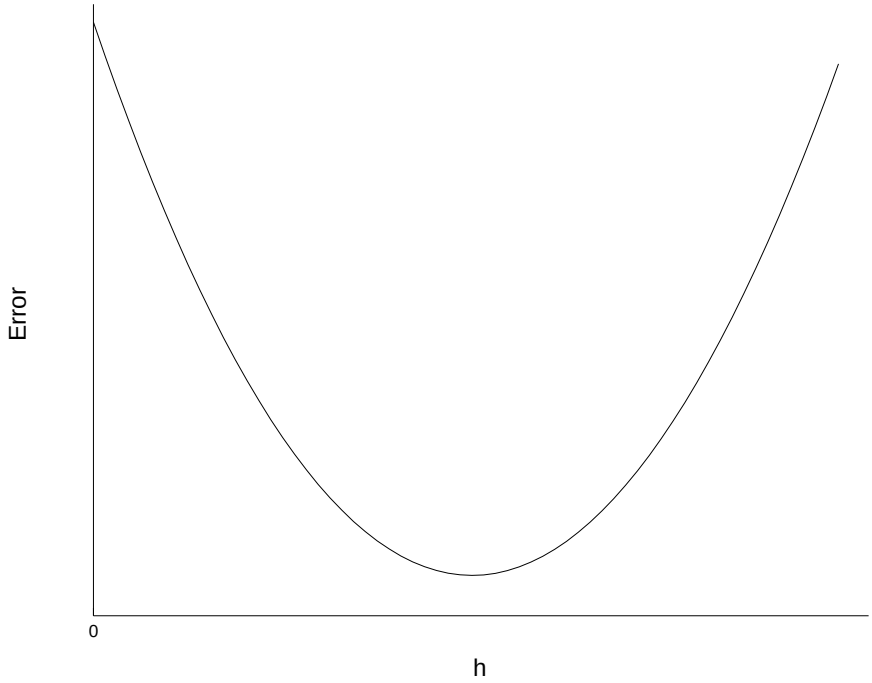


Figure 5.3: The error function versus h when $p \notin [0, 1]$ and we truncate c_m

p	u	t	Coefficient Rounding	Theoretical Bound	Algorithmic Bound
-0.5	8	21	Trunc	$[0, 2^{-18.67}]$	$[0, 2^{-19}]$
-0.5	9	23	Trunc	$[0, 2^{-20.67}]$	$[0, 2^{-21}]$
-0.5	7	16	Nearest	$[-2^{-16}, 2^{-15.54}]$	$[-2^{-16}, 2^{-16}]$
-1	7	16	Trunc	$[0, 2^{-14.4}]$	$[0, 2^{-15}]$
-1	7	16	Nearest	$[-2^{-16}, 2^{-15}]$	$[-2^{-16}, 2^{-15.7}]$
0.5	7	20	Up	$[-2^{-18}, 0]$	$[-2^{-18.54}, 0]$

Table 5.1: Comparison between the theoretical error analysis and our algorithmic error analysis. The coefficient is not computed with the modified method.

In table 5.1 we give some results of our algorithm compared to the theoretical error analysis given in the previous section. It is clear from this table that our algorithm gives a tighter bound to the total error of the powering algorithm. The difference is in the order of half a bit. Although the difference is not so big yet it can cause significant reduction in the size of the coefficient table in some cases especially when the output of the powering algorithm is used as an initial approximation to a functional recurrence algorithm.

5.4 Summary

This chapter presents a powering algorithm for special powers. The algorithm is based on the first order Taylor approximation with one modification. It uses one coefficient instead of two coefficients and it modifies the argument before multiplying it by the coefficient. The modification of the argument requires simple hardware that has constant delay.

The reduction in the number of used coefficients from two to one reduced the area of the used tables significantly. Such reduction in the size of the tables comes at the expense of using a larger multiplier. Hence this algorithm has an advantage over the usual first order polynomial approximation when there is an existing multiplier in the system. It also has an advantage when more than one function is implemented and share the multiplier.

A complete theoretical error analysis is presented and a proposed algorithmic error analysis is also presented. The two error analysis schemes are compared. It is shown that the algorithmic error analysis gives a tighter bound on the total

error.

Such more accurate error analysis leads to a significant reduction in the size of the table especially when the output of the powering algorithm is used as an initial approximation for a functional recurrence algorithm.

Chapter 6

Functional Recurrence

In this chapter we present a class of algorithms called functional recurrence. In this class of algorithms we start by an initial approximation to the elementary function that we need to compute and apply this initial approximation to a recurrence function. The output of this recurrence function is a better approximation to the elementary function. We can then reapply this new better approximation to the recurrence function again to obtain yet a better approximation and so on. The initial approximation can be obtained by a direct table lookup or by polynomial approximation or by any other approximation technique.

Functional Recurrence techniques are based on the Newton-Raphson root finding algorithm. The convergence rate of these techniques is quadratic i.e. after every iteration on the recurrence function the number of correct bits of the result is approximately doubled. The functional recurrence techniques can be generalized to what is known as high order Newton-Raphson. For these techniques the convergence rate can be cubic, quadruple or better.

The rest of this chapter is organized as follows: In section 6.1 we describe the Newton-Raphson algorithm and how it can be used to compute the reciprocal and square root reciprocal functions . In section 6.2 we present the high order version of the Newton-Raphson algorithm for the reciprocal function while in section 6.3 we present the high order Newton-Raphson algorithm for the square root reciprocal function.

6.1 Newton Raphson

Newton-Raphson algorithm is a numerical algorithm that aims to find the root of a given function $\phi(\alpha)$ from a starting crude estimate. We denote the initial estimate by α_i. The basic idea of the algorithm is to draw a tangent to the function at the initial estimate. This tangent will intersect the horizontal axis at a closer point to the root that we seek as shown in figure 6.1. We denote this new estimate by α_{i+1}. We repeat the above procedure with α_{i+1} taking the role of α_i. After several iterations we reach the root at the desired precision.

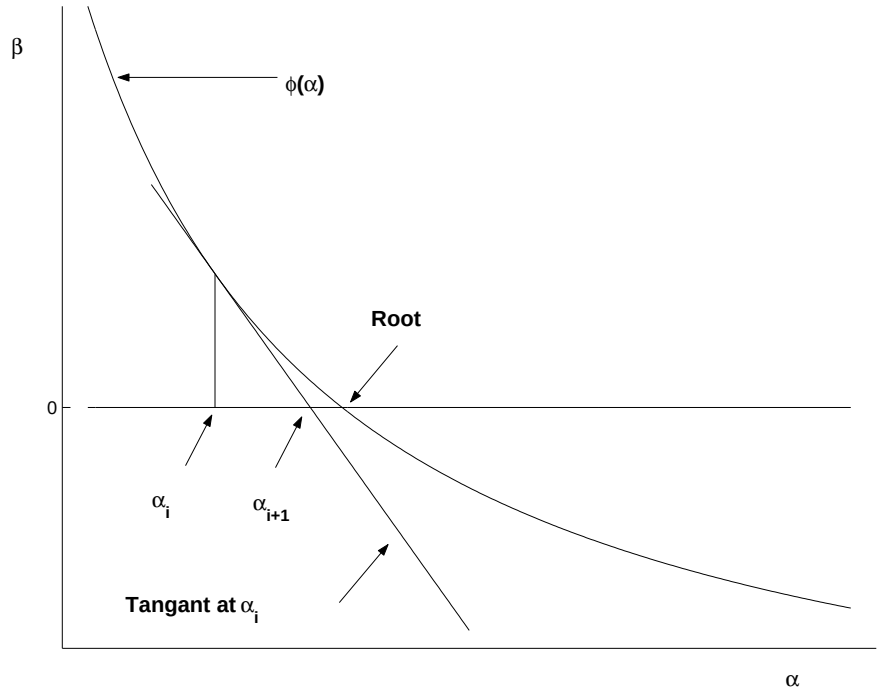


Figure 6.1: Illustration of Newton-Raphson root finding algorithm

The recurrence function can be obtained by determining the equation of the tangent and then finding the intersection point between the tangent and the horizontal axis. The tangent has a slope that is equal to the first derivative of $\phi(\alpha)$ evaluated at α_i and it passes through the point $(\alpha_i, \phi(\alpha_i))$ hence if we denote the vertical axis by β then the equation of the tangent is given by:

$$\frac{\beta - \phi(\alpha_i)}{\alpha - \alpha_i} = \phi'(\alpha_i) \tag{6.1}$$

The tangent intersects the horizontal axis at the point $(\alpha_{i+1}, 0)$ hence we can get

α_{i+1} by setting $\beta = 0$ in the tangent equation.

$$\alpha_{i+1} = \alpha_i - \frac{\phi(\alpha_i)}{\acute{\phi}(\alpha_i)} \tag{6.2}$$

Equation 6.2 is the recurrence relation from which we get α_{i+1} from α_i.

The convergence of the Newton-Raphson algorithm is guaranteed if:

$$\left| \frac{\phi(\alpha)\phi''(\alpha)}{(\phi'(\alpha))^2} \right| < 1 \tag{6.3}$$

We can use this root finding algorithm to compute the reciprocal and the square root reciprocal by choosing $\phi(\alpha)$ suitably.

If we choose $\phi(\alpha)$ such that $\phi(\alpha) = \frac{1}{\alpha} - Y$ then the root that we seek is $\alpha = \frac{1}{Y}$. Therefore we can compute the reciprocal function using such a choice for $\phi(\alpha)$. We can also compute the division operation by multiplying the dividend by the reciprocal of the divisor. Substituting in equation 6.2 the recurrence relation becomes:

$$\alpha_{i+1} = \alpha_i(2 - Y\alpha_i) \tag{6.4}$$

In this case we compute the reciprocal of Y using a recurrence function that involves addition and multiplication only specifically two multiplications and one addition. Hence every iteration on the recurrence relation takes two clock cycles when we use a parallel multiplier.

Assuming that α_i is less than the root $\frac{1}{Y}$ by a value ϵ_i that is $\alpha_i = \frac{1}{Y} - \epsilon_i$. By substituting this value of α_i in the recurrence relation above we get

$$\begin{aligned}
\alpha_{i+1} &= (\frac{1}{Y} - \epsilon_i)(2 - Y(\frac{1}{Y} - \epsilon_i)) \\
&= (\frac{1}{Y} - \epsilon_i)(1 + Y\epsilon_i) \\
&= \frac{1}{Y} - Y\epsilon_i^2 \qquad (6.5) \\
\epsilon_{i+1} &= Y\epsilon_i^2 \qquad (6.6)
\end{aligned}$$

From equation 6.6 if $Y \in [0.5, 1]$ then $\epsilon_{i+1} \leq \epsilon_i^2$ therefore the error decreases

quadratically after every application of the recurrence relation. For example if $\epsilon_i = 2^{-8}$ then after applying the recurrence relation once the error becomes $\epsilon_{i+1} = 2^{-16}$ and after another application of the recurrence relation the error becomes $\epsilon_{i+2} = 2^{-32}$. Hence the number of correct bits in representing $\frac{1}{Y}$ doubles after each application of the recurrence relation. The initial error ϵ_0 and the final precision will dictate the number of iterations on the recurrence relation.

By Choosing $\phi(\alpha)$ such that $\phi(\alpha) = \frac{1}{\alpha^2} - Y$ the root will become equal to $\frac{1}{\sqrt{Y}}$. Therefore we can compute the reciprocal square root function with such choice for $\phi(\alpha)$. We can also compute the square root from the square root reciprocal by multiplying the latter by the argument Y. Substituting in equation 6.2 the recurrence relation becomes:

$$\alpha_{i+1} = \frac{\alpha_i}{2}(3 - \alpha_i^2 Y) \tag{6.7}$$

The recurrence relation thus involves only addition and multiplication specifically three multiplications and one addition. Hence every iteration on the recurrence relation takes three clock cycles when we use a parallel multiplier.

If α_i is less than $\frac{1}{\sqrt{Y}}$ by a value ϵ_i that is $\alpha_i = \frac{1}{\sqrt{Y}} - \epsilon_i$ then by substituting this relation in the recurrence relation above we get

$$\begin{aligned}
\alpha_{i+1} &= \frac{\frac{1}{\sqrt{Y}} - \epsilon_i}{2}(3 - (\frac{1}{\sqrt{Y}} - \epsilon_i)^2 Y) \\
\alpha_{i+1} &= \frac{1}{\sqrt{Y}} - \frac{3}{2}\epsilon_i^2\sqrt{Y} + \frac{1}{2}\epsilon_i^3 Y \qquad (6.8) \\
\epsilon_{i+1} &= \frac{3}{2}\epsilon_i^2\sqrt{Y} - \frac{1}{2}\epsilon_i^3 Y \qquad (6.9)
\end{aligned}$$

Equation 6.9 gives the error after applying the recurrence relation in terms of the error before applying the recurrence relation. The error decreases quadratically.

In general $\phi(\alpha)$ can take other forms in order to compute other functions recursively. For practical consideration $\phi(\alpha)$ is constrained to certain forms that gives recurrence relations that involves addition and multiplication only in order to be suitable for hardware implementation.

6.2 High Order NR for Reciprocal

The first order Newton-Raphson algorithm converges quadratically at the cost of two clock cycles per iteration for the case of reciprocation. The delay cost of the first order Newton-Raphson for reciprocation is thus an integer multiple of two clock cycles.

In this section we present the high order version of the Newton-Raphson algorithm for computing the reciprocal [13]. The advantage of the higher order NR is that the delay cost can vary at a finer step of one clock cycle thus the designer has more freedom in selecting the delay cost of the algorithm and hence its area cost.

The recurrence relation for high order NR algorithm for the reciprocal function can be derived as follows: We define $D = 1 - \alpha_0 Y$ where α_0 is the initial approximation to $\frac{1}{Y}$.

$$\begin{aligned} D &= 1 - \alpha_0 Y && (6.10)\\ \frac{1}{Y} &= \frac{\alpha_0}{1-D} && (6.11)\\ \frac{1}{Y} &= \alpha_0(1 + D + D^2 + \cdots) && (6.12)\\ \alpha_r &= \alpha_0(1 + D + D^2 + \cdots + D^r) && (6.13) \end{aligned}$$

Equation 6.13 is the recurrence relation for the high order NR reciprocation algorithm. We compute D in one clock cycle and we compute α_r using horner formula in r clock cycles as follows [12]:

$$\begin{aligned} D \times D + D &\rightarrow T_0\\ T_0 \times D + D &\rightarrow T_1\\ T_1 \times D + D &\rightarrow T_2\\ &\vdots\\ T_{r-2} \times \alpha_0 + \alpha_0 &\rightarrow T_{r-1} \end{aligned}$$

We perform only one iteration of the recurrence relation. From the given error in α_0 we choose the value of r so that the final error in α_r lies within the desired

precision. If $\alpha_0 = \frac{1}{Y} - \epsilon_0$ then by substitution in equation 6.13 we get:

$$
\begin{aligned}
D &= 1 - (\frac{1}{Y} - \epsilon_0)Y = \epsilon_0 Y && (6.14)\\
\alpha_r &= (\frac{1}{Y} - \epsilon_0)(1 + (\epsilon_0 Y) + (\epsilon_0 Y)^2 + \cdots + (\epsilon_0 Y)^r) \\
\alpha_r &= \frac{1}{Y} - Y^r \epsilon_0^{r+1} && (6.15)\\
\epsilon_r &= Y^r \epsilon_0^{r+1} && (6.16)
\end{aligned}
$$

Equation 6.16 indicates that the algorithm is (r+1)-convergent i.e. when $r = 1$ it is quadratically convergent and when $r = 2$ it is cubically convergent and so on. The delay cost is $(r + 1)$ clock cycles. We can choose r to be any integer starting from 1 and above. Therefore the delay in clock cycles can vary at a step of one clock cycle.

6.3 Our Contribution

High Order NR for Square Root Reciprocal

In this section we present the high order version of the NR algorithm for the square root reciprocal function. It has the same advantage as in reciprocation and that is the fine steps in the area delay curve. The delay in the high order NR can increase at a step of one clock cycle instead of three as in the first order NR. Hence the high order NR gives the designer more freedom in choosing the number of clock cycles of the algorithm.

We derive the recurrence relation for the high order NR algorithm for the square root reciprocal as follows: We define $D = 1 - \alpha_0^2 Y$ where α_0 is an initial approximation to $\frac{1}{\sqrt{Y}}$.

$$
\begin{aligned}
D &= 1 - \alpha_0^2 Y && (6.17)\\
\frac{1}{\sqrt{Y}} &= \frac{\alpha_0}{\sqrt{1 - D}} && (6.18)\\
\frac{1}{\sqrt{Y}} &= \alpha_0(1 + \frac{1}{2}D + \frac{3}{8}D^2 + \cdots) && (6.19)\\
\alpha_r &= \alpha_0(1 + \frac{1}{2}D + \frac{3}{8}D^2 + \cdots + \frac{(1)(3)(5)\cdots(2r-1)}{2^r r!}D^r) && (6.20)
\end{aligned}
$$

Equation 6.20 is the recurrence relation for the high order Newton-Raphson algorithm for square root reciprocal function. We compute D in two clock cycles and then we compute α_r in $(r+1)$ clock cycles. For the case that $r = 1$ or $r = 2$ we compute α_r in r clock cycles by an additional relatively small hardware [29].

We perform only one iteration of the recurrence relation. From the given error in α_0 we choose the value of r so that the final error in α_r lies within the desired precision. If $\alpha_0 = \frac{1}{\sqrt{Y}} - \epsilon_0$ then by substitution in equation 6.20 we get:

$$D = 1 - (\frac{1}{\sqrt{Y}} - \epsilon_0)^2 Y = 2\epsilon_0\sqrt{Y} - \epsilon_0^2 Y \quad (6.21)$$

$$\alpha_r = \frac{1}{\sqrt{Y}} - O(Y^r \epsilon_0^{r+1}) \quad (6.22)$$

$$\epsilon_r = O(Y^r \epsilon_0^{r+1}) \quad (6.23)$$

Equation 6.23 indicates that the algorithm is (r+1)-convergent i.e. when $r = 1$ it is quadratically convergent and when $r = 2$ it is cubically convergent and so on. The delay cost is $(r + 3)$ clock cycles for $r > 2$ and can be $(r + 2)$ clock cycles for $r = 1$ or $r = 2$. We can choose r to be any integer starting from 1 and above. Therefore the delay in clock cycles can vary at a step of one clock cycle.

6.3.1 A Proposed Square Root Algorithm

In this sub-section we present a square root algorithm that is based on the second order NR algorithm for square root reciprocal [29].

The algorithm computes the square root for the double precision format. The algorithm employs the powering method for the initial approximation of the square root reciprocal. It then performs one iteration of the second order Newton-Raphson algorithm followed by a multiplication by the operand to get an approximation to the square root. Finally the algorithm rounds this result according to one of the four IEEE Rounding modes. The initial approximation parameters u and t have the values 8 and 21 respectively.

Fig. 6.2 presents the proposed architecture. The box labeled 'T' is a temporary register to hold the intermediate results of the calculation. The main block in the figure is the fused multiply add (FMA) unit. The given double precision floating point number is $X = s2^e 1.x$. We denote $Y = 1.x$. Y is split into two parts m and

h such that $Y = m + h$, the steps of the algorithm for calculating the mantissa of the result are:

1. From m and the lookup table $\rightarrow c_m$,
   from $Y \rightarrow \tilde{Y}$,
   $c_m\tilde{Y} \rightarrow \alpha_0$

2. $\alpha_0 \times \alpha_0 \rightarrow T_1$

3. $1 - \alpha_1 \times T_1 \rightarrow D$

4. $\frac{3}{8}D + \frac{1}{2} \rightarrow T_2$

5. $1 + D \times T_2 \rightarrow T_3$

6. $\alpha_0 \times T_3 \rightarrow T_4$

7. $T_4 \times Y \rightarrow T_5$. For RZ or RM modes, round T_5 to the nearest value using the guard bits and jump to step 8. For RP mode round T_5 to the nearest value using the guard bits and jump to step 9. For RN mode truncate the guard bits of T_5 and jump to step 10.

8. If $T_5^2 - Y > 0$, then result $= T_5 - 1ulp$. Otherwise, result $= T_5$.

9. If $T_5^2 - Y < 0$, then result $= T_5 + 1ulp$. Otherwise, result $= T_5$.

10. If $(T_5 + 0.5ulp)^2 - Y > 0$, then result $= T_5$. Otherwise, result $= T_5 + 1ulp$.

In steps 8, 9 and 10 the FMA acts as a multiply subtract unit by adding the two's complement of the third operand.

The complete error analysis of the proposed square root algorithm can be found in [28]

Note that in the proposed algorithm we compute $\frac{3}{8}D^2 + \frac{1}{2}D + 1$ in two clock cycles. We can compute the same expression in one clock cycle at the expense of adding a small hardware in the FMA reduction tree. The idea is as follows: we multiply D times D and reduce them into two partial products right before the carry propagate adder of the FMA. We denote these two partial products by z_1

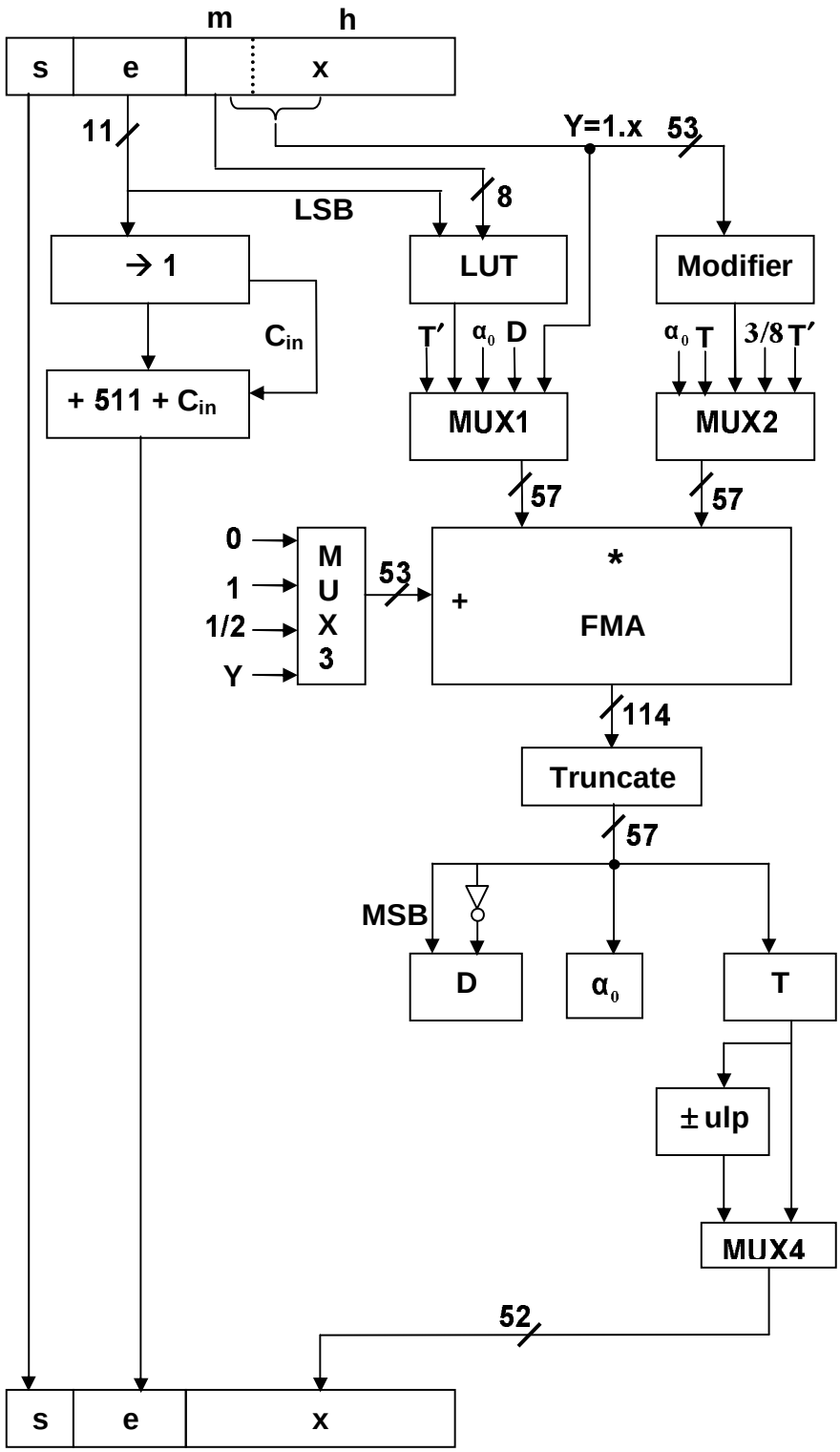


Figure 6.2: Hardware Architecture of the Proposed Square Root Algorithm.

and z_2 that is $D^2 = z_1 + z_2$. The expression $\frac{3}{8}D^2$ can be expanded as follows:

$$\frac{3}{8}D^2 = (\frac{1}{4} + \frac{1}{8})(z_1 + z_2) \quad (6.24)$$

$$= \frac{1}{4}z_1 + \frac{1}{4}z_2 + \frac{1}{8}z_1 + \frac{1}{8}z_2 \quad (6.25)$$

Therefore we need to reduce the five partial products $\frac{1}{4}z_1$, $\frac{1}{4}z_2$, $\frac{1}{8}z_1$, $\frac{1}{8}z_2$ and $1+D$ into two partial products that are then fed to the carry propagate adder of the FMA in order to evaluate the desired expression above.

A VHDL model is created for this circuit and a behavioral simulation is carried out with more than two million test vectors. The circuit passes these tests successfully.

6.4 Summary

This chapter presents functional recurrence algorithms. Such algorithms are based on Newton-Raphson root-finding algorithm. They are also called Newton-Raphson algorithm for this reason. They start by a crude approximation and refine it recursively by evaluating a recursive relation.

Such algorithms are applicable for a certain class of arithmetic operations and elementary functions for which the recurrence relation involves addition and multiplication only in order to be implemented in hardware efficiently.

The reciprocal and square root reciprocal are examples of arithmetic operations which can be efficiently implemented using the functional recurrence algorithm. The details of using NR algorithm for evaluating reciprocal and square root reciprocal is presented.

High order NR algorithm for reciprocal and square root reciprocal are presented. The high order version of the NR algorithm offers the designer more freedom in selecting the number of clock cycles of the resulting hardware circuit.

A hardware circuit for evaluating the square root operation is presented. It is based on the second order NR algorithm for square root reciprocal followed by multiplication by the operand and one of the IEEE rounding modes. The initial approximation is based on the powering algorithm that is presented in chapter 5.

A VHDL model of the circuit was simulated with two million random test vectors and passed all the tests with no errors.

Chapter 7

Digit Recurrence

In this chapter we present digit recurrence algorithms. These algorithms are also called shift and add algorithms because they employ adders and shifters in their hardware implementation. They converge linearly i.e. they recover a constant number of bits each iteration.

Examples of algorithms that belong to this class include the division restoring and non-restoring algorithms, SRT algorithm, online algorithms, CORDIC, an algorithm due to Briggs and DeLugish for computing exponential and logarithm and finally the BKM algorithm that generalizes the last two algorithms. In section 7.1 we present the CORDIC algorithm while in section 7.2 we present the Briggs and DeLugish algorithm

7.1 CORDIC

The CORDIC algorithm was invented by Volder [5]. The name stands for COordinate Rotation Digital Computer. There are two modes of operation for the CORDIC algorithm. The first mode is called the rotation mode. In this mode we are given a vector $(\alpha 1, \beta 1)$ and an angle θ and we are required to rotate this vector by the angle θ to obtain the new vector $(\alpha 2, \beta 2)$. The second mode is called the vectoring mode. In the vectoring mode we are given a vector $(\alpha 1, \beta 1)$ and we are required to compute the subtended angle between this vector and the horizontal axis (ρ) as shown in figure 7.1.

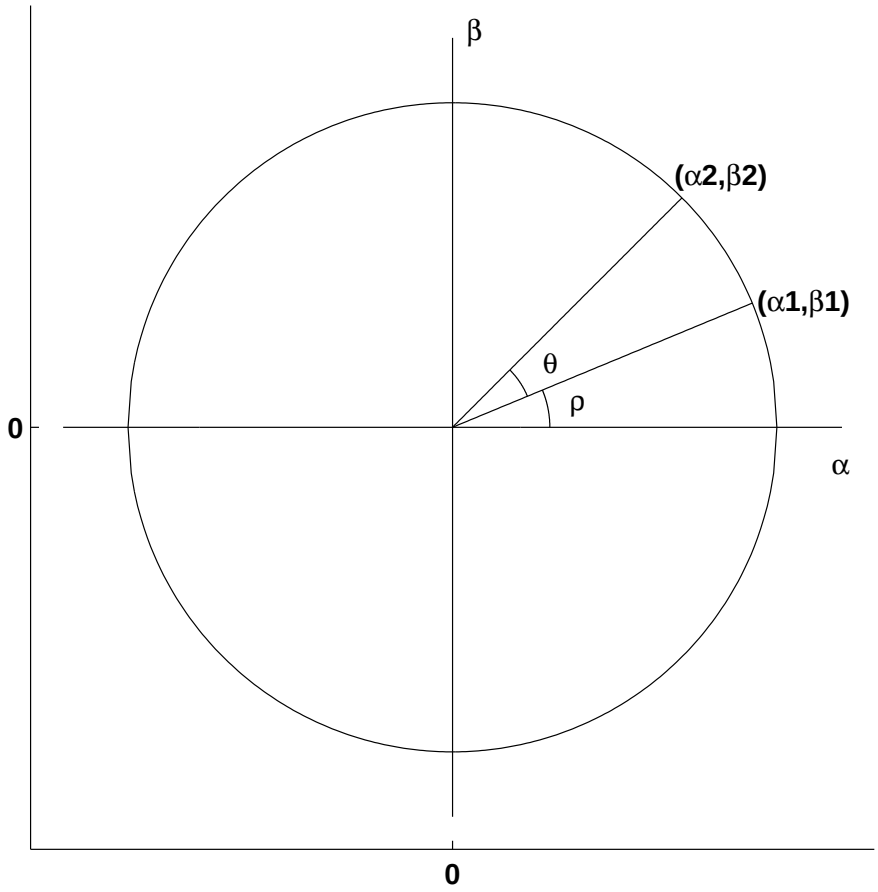


Figure 7.1: Two vectors and subtended angle. In rotation mode the unknown is the second vector while in vectoring mode the unknown is the subtended angle between the given vector and the horizontal axis

7.1.1 Rotation Mode

The equations that relates the two vectors $(\alpha 1, \beta 1)$ and $(\alpha 2, \beta 2)$ and the angle θ are :

$$\alpha 2 = \alpha 1 cos(\theta) - \beta 1 sin(\theta) \tag{7.1}$$

$$\beta 2 = \alpha 1 sin(\theta) + \beta 1 cos(\theta) \tag{7.2}$$

These two equations can be written as:

$$\alpha 2 = cos(\theta)(\alpha 1 - \beta 1 tan(\theta)) \tag{7.3}$$

$$\beta 2 = cos(\theta)(\alpha 1 tan(\theta) + \beta 1) \tag{7.4}$$

The basic idea of the CORDIC algorithm is to decompose the angle θ into a

sum of weighted basis angles as follows:

$$\theta = \sum_{i=0}^{i=L-1} d_i \omega_i \tag{7.5}$$

$$\tag{7.6}$$

such that the basis angles ω_i are computed as follows

$$\omega_i = tan^{-1}(2^{-i}) \tag{7.7}$$

and $d_i \in \{-1, 1\}$ Note that the basis angles form a decreasing set.

The rotation process is thus decomposed into L steps. In each step we rotate by ω_i in the clockwise or anti-clockwise direction according to the value of d_i. By substituting θ by $d_i\omega_i$ in equations 7.3 and 7.4 we get the basic equations of each step:

$$\alpha_{i+1} = \frac{1}{\sqrt{1+2^{-2i}}}(\alpha_i - d_i\beta_i 2^{-i}) \tag{7.8}$$

$$\beta_{i+1} = \frac{1}{\sqrt{1+2^{-2i}}}(\beta_i + d_i\alpha_i 2^{-i}) \tag{7.9}$$

Note that in these two previous equation the external factor does not depend on the value of d_i hence we can lump the external factors of all the steps and account for them at the beginning or the end of the algorithm. The lumped factor which we denote by fa is given by the following equation:

$$fa = \prod_{i=0}^{i=L-1} \frac{1}{\sqrt{1+2^{-2i}}} \tag{7.10}$$

$fa = 0.60725293500888$ for $L = 52$. The modified equations without the external factor become:

$$\alpha_{i+1} = (\alpha_i - d_i\beta_i 2^{-i}) \tag{7.11}$$

$$\beta_{i+1} = (\beta_i + d_i\alpha_i 2^{-i}) \tag{7.12}$$

These two equations are implemented in hardware using two shifters and two adders as shown in figure 7.2

We need to determine how to compute d_i. We define a new variable γ such that γ is equal to the difference between θ and the sum of the rotated basis angles till step i. We need to force γ to be zero which is identical to forcing the sum of the weighted basis angles to be equal to θ. We initialize γ to the angle θ i.e $\gamma_0 = \theta$. If γ_i is positive then we set $d_i = 1$ in order to rotate in the anti-clockwise direction and subtract ω_i from γ_i to get γ_{i+1}. On the other hand if γ_i is negative we set $d_i = -1$ in order to rotate in the clockwise direction and add ω_i to γ_i to get γ_{i+1}. In both cases we force the value of γ to approach zero. Hence we force the difference between θ and $\sum_{i=0}^{i=L-1} d_i\omega_i$ to approach zero as well.

$$\begin{aligned} \gamma_0 &= \theta && (7.13)\\ d_i &= 1 \text{ if } \gamma_i >= 0 \\ &= -1 \text{ if } \gamma_i < 0 && (7.14)\\ \gamma_{i+1} &= \gamma_i - d_i\omega_i && (7.15) \end{aligned}$$

The complete equations of the rotation mode are as follows:

$$\begin{aligned} \alpha_0 &= fa \times \alpha 1 && (7.16)\\ \beta_0 &= fa \times \beta 1 && (7.17)\\ \gamma_0 &= \theta && (7.18)\\ d_i &= 1 \text{ if } \gamma_i >= 0 \\ &= -1 \text{ if } \gamma_i < 0 && (7.19)\\ \alpha_{i+1} &= (\alpha_i - d_i\beta_i 2^{-i}) && (7.20)\\ \beta_{i+1} &= (\beta_i + d_i\alpha_i 2^{-i}) && (7.21)\\ \gamma_{i+1} &= \gamma_i - d_i\omega_i && (7.22)\\ i &= 0, 1, 2, \ldots, L-1 \end{aligned}$$

We initialize α , β, γ and d then we run the last four equations for L iterations. Note that in the above equations we only use addition operation and multiplication by 2^{-i} which is a right shift operation.

A direct application to the CORDIC algorithm rotation mode is the com-

putation of the two elementary functions cos and sin. We compute cos(Y) and sin(Y) simultaneously using the CORDIC algorithm by setting $\alpha 1 = 1$, $\beta 1 = 0$ and $\theta = Y$ (see equations 7.1 and 7.2). Therefore $\alpha_0 = fa$, $\beta_0 = 0$ and $\gamma_0 = Y$. At the end of the L iterations of the algorithm $\alpha_L = cos(Y)$ and $\beta_L = sin(Y)$.

7.1.2 Vectoring Mode

In vectoring mode we start with the given vector and rotate it in the direction of the horizontal axis. In iteration i we rotate with the basis angle ω_i in the direction of the horizontal axis and accumulate the angles with which we rotate. At the end of the algorithm we reach the horizontal axis and the accumulated basis angles give the angle between the given vector and the horizontal axis ρ.

At the horizontal axis $\beta = 0$ hence to approach the horizontal axis we rotate in the clockwise direction when the vector is above the horizontal axis ($\beta_i > 0$) and we rotate in the anti-clockwise otherwise.

The equations that govern the vectoring mode are thus given as follows:

$$\begin{aligned}
\alpha_0 &= \alpha 1 && (7.23)\\
\beta_0 &= \beta 1 && (7.24)\\
\gamma_0 &= 0 && (7.25)\\
d_i &= 1 \text{ if } \beta_i <= 0 &&\\
&= -1 \text{ if } \beta_i > 0 && (7.26)\\
\alpha_{i+1} &= (\alpha_i - d_i \beta_i 2^{-i}) && (7.27)\\
\beta_{i+1} &= (\beta_i + d_i \alpha_i 2^{-i}) && (7.28)\\
\gamma_{i+1} &= \gamma_i - d_i \omega_i && (7.29)\\
i &= 0, 1, 2, \ldots, L-1 &&
\end{aligned}$$

Note that we don't account for the scale factor in the vectoring mode because we are only seeking the value of the angle and not the vector and because we compare the vertical component of the vector β_i with zero hence the absence of the scale factor doesn't affect the comparison result. We may need to account for the scale factor in case we need to compute the length of the given vector

$(\alpha 1, \beta 1)$ since at the end of the algorithm $\alpha_L = \frac{1}{fa}\sqrt{\alpha 1^2 + \beta 1^2}$.

In vectoring mode we compute the angle between the given vector and the horizontal axis. This angle is equal to $tan^{-1}(\frac{\beta 1}{\alpha 1})$. Therefore we can compute the elementary function $F(Y) = tan^{-1}(Y)$ by setting $\beta 1 = Y$ and $\alpha 1 = 1$.

7.1.3 Convergence Proof

In the rotation mode we decompose the angle θ into a sum of weighted basis angles ω_i as in equation 7.5 and this decomposition is carried out sequentially by nulling the variable γ which is initialized to θ. In the vectoring mode on the other hand we start with the vector $(\alpha 1, \beta 1)$ that has an angle ρ and we rotate this vector in the direction of the horizontal axis by nulling β_i hence we are also decomposing ρ into a sum of the basis angles ω_i. We need to prove that this decomposition converges. The following two conditions on the basis angles are necessary and sufficient to guarantee such convergence.

$$|\theta \text{ or } \rho| \leq \omega_0 + \omega_1 + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.30}$$

$$\omega_i \leq \omega_{i+1} + \omega_{i+2} + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.31}$$

From the first condition we get the limits on the value of θ that can be used in the algorithm. For $L = 24$, $|\theta| \leq 1.74328662047234$ radian.

We prove the above statement for the rotation mode and the same procedure can be applied to the vectoring mode by replacing θ by ρ. In each step of the rotation mode we rotate in the direction that will decrease the absolute value of the variable γ by the amount of the current basis angle that is $|\gamma_{i+1}| = ||\gamma_i| - \omega_i|$ $\gamma_0 = \theta$ hence from the first condition we have

$$|\gamma_0| \leq \omega_0 + \omega_1 + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.32}$$

$$-\omega_0 \leq |\gamma_0| - \omega_0 \leq \omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.33}$$

From the second condition we get

$$\omega_0 \leq \omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.34}$$

$$-\omega_0 \geq -(\omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1}) \tag{7.35}$$

By using the last inequality in inequality 7.33 we get

$$\begin{aligned} -(\omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1}) &\leq |\gamma_0| - \omega_0 \\ &\leq \omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1} \qquad (7.36) \\ ||\gamma_0| - \omega_0| &\leq \omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1} \qquad (7.37) \\ |\gamma_1| = ||\gamma_0| - \omega_0| &\leq \omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1} \qquad (7.38) \end{aligned}$$

After one step the value of γ_1 is bounded as in the inequality 7.38. By repeating the same procedure for the other steps we get

$$|\gamma_L| \leq \omega_{L-1} \tag{7.39}$$

And since ω_i is a decreasing sequence as given by equation 7.7 and for large L $\omega_{L-1} \leq 2^{-L+1}$ hence $|\gamma_L| \leq 2^{-L+1}$ and this concludes the proof that γ approaches zero as L increases provided that the basis angles ω_i satisfy the two convergence conditions. It can be proved that the basis angles as given by equation 7.7 satisfy the two convergence conditions hence the CORDIC algorithm converges.

7.1.4 Hardware Implementation

The hardware Implementation is given in figure 7.2. We use two adders-subtractors and two shifters for the two variables α_i and β_i. We use a table to store the basis angles ω_i and we use an adder-subtractor for the variable γ_i. d_i depends on the sign of either γ_i or β_i for rotation and vectoring modes respectively and it controls the adders-subtractors whether to add or subtract. The given architecture is called the sequential architecture. Another architecture called the unfolded architecture repeats the hardware units of the sequential architecture in order to accomplish more than one step in the same clock cycle at the expense of increased delay of each clock cycle.

The CORDIC algorithm is generalized to compute other elementary functions such as hyperbolic functions, multiplication and division [6]

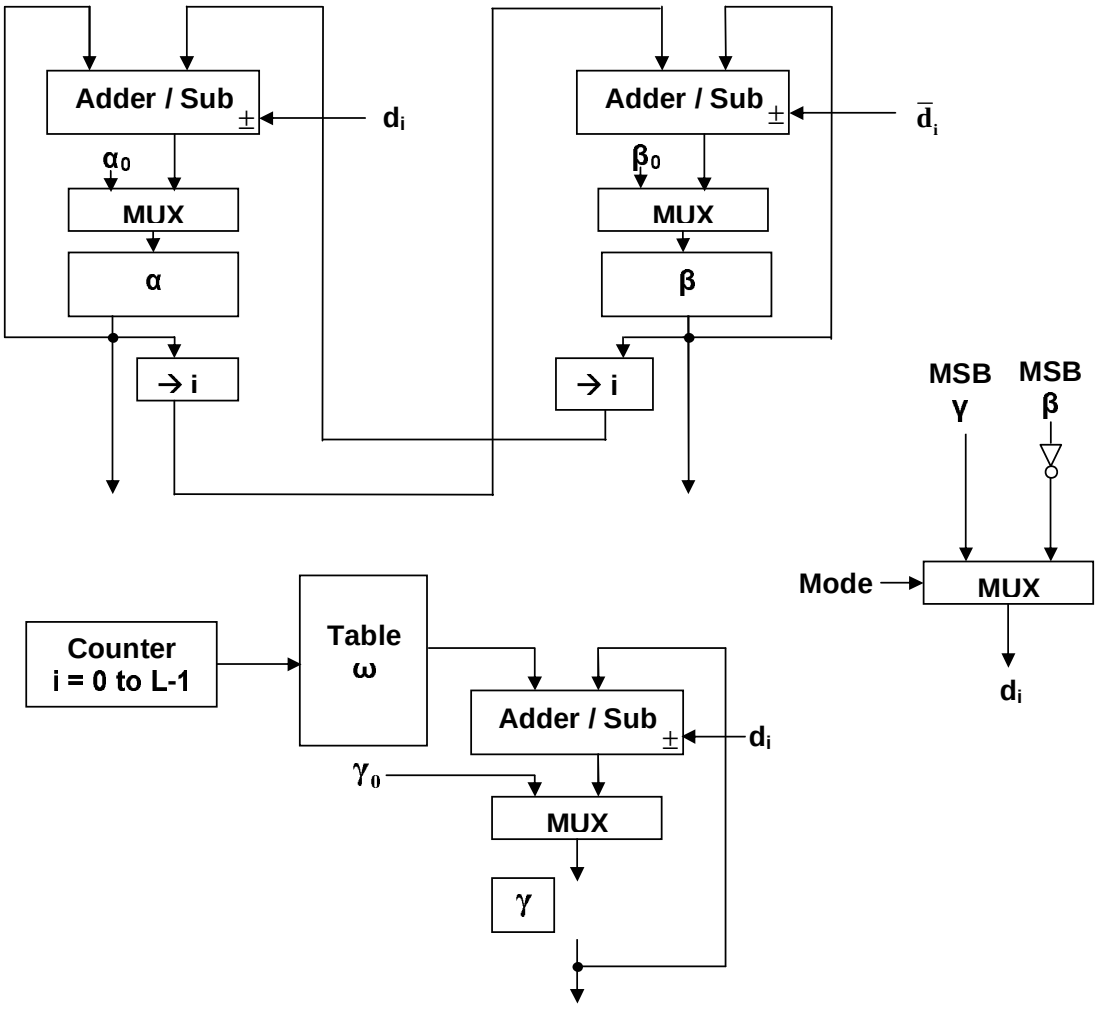


Figure 7.2: The CORDIC Architecture

7.2 Briggs and DeLugish Algorithm

This section presents an algorithm [7] that computes the exponential and logarithm functions.

7.2.1 The Exponential Function

We are given the reduced argument Y and we need to compute $exp(Y)$. We decompose Y as a sum of weighted basis values as follows:

$$Y = \sum_{i=0}^{i=L-1} d_i \omega_i \tag{7.40}$$

such that

$$\omega_i = ln(1 + 2^{-i}) \tag{7.41}$$

and $d_i \in \{0, 1\}$. The basis weights given by equation 7.41 form a decreasing set that is $\omega_0 > \omega_1 > \cdots > \omega_{L-1}$

With such decomposition $exp(Y) = \prod_{i=0}^{i=L-1}(1 + 2^{-i})^{d_i}$. We can thus compute $exp(Y)$ sequentially as follows

$$\alpha_0 = 1 \tag{7.42}$$

$$\alpha_{i+1} = \alpha_i + 2^{-i}\alpha_i \text{ if } d_i = 1 \tag{7.43}$$

$$\alpha_{i+1} = \alpha_i \text{ if } d_i = 0 \tag{7.44}$$

$$i = 0, 1, \ldots, L-1$$

Note that in the above recursive equation we use only addition and shift operations.

The above algorithm is incomplete. We still need to devise a method to compute d_i. We can carry out this task by defining a new variable β that we initialize to zero. In each step if when we add ω_i to β the sum is less than the argument then we set $d_i = 1$ and add ω_i to β otherwise we set $d_i = 0$ and leave

β unchanged. We denote the variable β in step i by β_i.

$$\beta_0 = 0 \tag{7.45}$$

$$d_i = 1 \text{ if } \beta_i + \omega_i \leq Y \tag{7.46}$$

$$d_i = 0 \text{ otherwise} \tag{7.47}$$

$$\beta_{i+1} = \beta_i + d_i\omega_i \tag{7.48}$$

$$i = 0, 1, \ldots, L-1$$

The necessary and sufficient conditions for the convergence of this algorithm are:

$$0 \leq Y \leq \omega_0 + \omega_1 + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.49}$$

$$\omega_i \leq \omega_{i+1} + \omega_{i+2} + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.50}$$

From the first condition we get the convergence range of Y. For $L = 24$ we get $0 \leq Y \leq 1.56202383321850$.

Since $\beta_0 = 0$ hence from the first condition we get

$$0 \leq Y - \beta_0 \leq \omega_0 + \omega_1 + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.51}$$

if $\beta_0 + \omega_0 \leq Y$ then $\beta_1 = \beta_0 + \omega_0$. From inequality 7.51 we get

$$0 \leq Y - \beta_1 = Y - \beta_0 - \omega_0 \leq \omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.52}$$

On the other hand if $\beta_0 + \omega_0 > Y$ then $\beta_1 = \beta_0$. From the second convergence condition we get

$$0 \leq Y - \beta_1 = Y - \beta_0 \leq \omega_0 \leq \omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.53}$$

Hence after the first step of the algorithm we have

$$0 \leq Y - \beta_1 \leq \omega_1 + \omega_2 + \cdots + \omega_{L-1} + \omega_{L-1} \tag{7.54}$$

If we repeat the same procedure for the remaining $L-1$ steps we reach:

$$0 \leq Y - \beta_L \leq \omega_{L-1} \tag{7.55}$$

Therefore the variable β approaches the given argument Y and the difference between the argument Y and the variable β after L steps is bounded by ω_{L-1}. In each step of the algorithm $\alpha_i = exp(\beta_i)$ hence the variable α approaches $exp(Y)$. We define the approximation error to be $\epsilon_a = exp(Y) - \alpha_L$. Hence from inequality 7.55 we get:

$$0 \leq Y - \beta_L \leq \omega_{L-1} \tag{7.56}$$

$$\beta_L \leq Y \leq \beta_L + \omega_{L-1} \tag{7.57}$$

$$exp(\beta_L) \leq exp(Y) \leq exp(\beta_L + \omega_{L-1}) \tag{7.58}$$

$$\alpha_L \leq exp(Y) \leq \alpha_L(1 + 2^{-L+1}) \tag{7.59}$$

$$\epsilon_a = exp(Y) - \alpha_L \tag{7.60}$$

$$0 \leq \epsilon_a \leq \alpha_L 2^{-L+1} \tag{7.61}$$

Hence the relative error is bounded as follows:

$$0 \leq \frac{exp(Y) - \alpha_L}{exp(Y)} \leq 2^{-L+1} \tag{7.62}$$

Therefore we conclude that at the $(L+1)$ step the algorithm computes L significant bits correctly without taking into account the accumulated rounding error.

ω_i as given by equation 7.41 satisfies the convergence conditions [38] hence the algorithm converges to the desired function value $exp(Y)$.

7.2.2 The Logarithm Function

The same algorithm is used to compute the logarithm function $F(Y) = ln(Y)$. We assume we know the value of the function and we compute its exponential which is obviously the argument Y The modified algorithm is as follows:

$$\alpha_0 = 1 \tag{7.63}$$

$$\beta_0 = 0 \tag{7.64}$$

$$d_i = 1 \text{ if } \beta_i + \omega_i \leq ln(Y)$$

$$d_i = 0 \text{ otherwise} \tag{7.65}$$

$$\alpha_{i+1} = \alpha_i + d_i \alpha_i 2^{-i} \tag{7.66}$$

$$\beta_{i+1} = \beta_i + d_i \omega_i \tag{7.67}$$

$$i = 0, 1, \ldots, L-1$$

The above algorithm computes the exponential of $ln(Y)$ hence α approaches Y and since $\alpha_i = exp(\beta_i)$ therefore β approaches $ln(Y)$. However in computing d_i in the above algorithm we use the value of $ln(Y)$ that we don't know. therefore we use an equivalent comparison by taking the exponential of both sides of the inequality. the resulting algorithm becomes:

$$\alpha_0 = 1 \tag{7.68}$$

$$\beta_0 = 0 \tag{7.69}$$

$$d_i = 1 \text{ if } \alpha_i + \alpha_i 2^{-i} \leq Y$$

$$d_i = 0 \text{ otherwise} \tag{7.70}$$

$$\alpha_{i+1} = \alpha_i + d_i \alpha_i 2^{-i} \tag{7.71}$$

$$\beta_{i+1} = \beta_i + d_i \omega_i \tag{7.72}$$

$$i = 0, 1, \ldots, L-1$$

In the above algorithm α_L approaches the argument Y. The invariant in the above algorithm is that $\alpha_i = exp(\beta_i)$ hence β_i approaches the logarithm of the given argument $ln(Y)$.

An algorithm due to Muller [9] which is called the BKM algorithm generalizes both the CORDIC algorithm and the Briggs algorithm presented in this chapter. The BKM algorithm computes the complex exponential and logarithm. It has the advantage that in each step d_i can be either $-1, 0$ or 1 hence it is can be implemented using redundant number systems and that reduces the delay of each step.

7.3 Summary

This chapter presents two algorithms from the Shift and Add techniques. The first algorithm is the CORDIC while the second algorithm is the Briggs and DeLugish algorithm.

The CORDIC algorithm computes the functions: $sin(Y)$, $cos(Y)$ and $tan^{-1}(Y)$ directly. It is generalized to compute the hyperbolic functions $sinh(Y)$, $cosh(Y)$ and $tanh^{-1}(Y)$ and multiplication and division. It is used also to compute other elementary functions indirectly. The Briggs and DeLugish algorithm on the other hand is used to compute the exponential and the logarithm functions.

For both algorithms we decompose the argument into a sum of weighted basis values. The decomposition is carried out sequentially. In each step we employ shift and add operations only.

These algorithms converge linearly that is the number of sequential steps is proportional to the number of correct bits in the final output.

The derivation of these two algorithms is presented with the convergence proof. The hardware implementation for the CORDIC algorithm is presented.

Chapter 8

Conclusions

8.1 Summary

We have presented in this thesis a survey of hardware algorithms for computation of division, square root and elementary functions.

The computation of these functions is performed by three steps. The first is the range reduction step that is usually based on a modular reduction technique. The second step is the approximation step which can be performed by digit recurrence algorithms or polynomial approximation algorithms or functional recurrence algorithms. The third and last step is the reconstruction step.

The range reduction and reconstruction steps are related and they were presented in chapter 2. Careful range reduction is crucial for correct computation.

Approximation algorithms were given in the other chapters. We presented general polynomial approximation techniques, interval division, coefficient computation, hardware implementation and error analysis.

We presented some work in this area such as the work in [25] that aims at truncating the coefficients of the second order polynomial efficiently.

We also presented the table and add algorithms which are a special case from the polynomial approximation technique. They employ tables to avoid multiplication.

Another special type of the polynomial approximation techniques which is the powering algorithm [17] was also presented.

We also presented functional recurrence algorithms for the computation of the

reciprocal and square root reciprocal both the first order and high order versions.

Finally we presented the digit recurrence algorithms specifically the CORDIC algorithm and Briggs and DeLugish algorithm.

8.2 Contributions

We made three contributions in this thesis.

The first contribution [39] is an algorithm for truncating the coefficients of the approximating polynomials in an efficient manner. The proposed algorithm modifies the last iteration of Remez algorithm and employs integer programming. Compared to the direct rounding our algorithm gives results that are more precise by 3 to 5 bits. Compared to Muller's algorithm presented in [25] our algorithm gives comparable results, slightly better in some cases and slightly worse in other cases. The advantage of our algorithm over Muller's algorithm is that our algorithm is applicable for any polynomial order while Mulller's algorithm is applicable for second order only.

The second contribution [28] is an algorithmic error analysis for the powering algorithm [17] and for a novel square root algorithm. When compared with the theoretical error analysis it was shown that the theoretical error analysis overestimated the error by about half a bit. The more accurate algorithmic error analysis can lead to a significant decrease in the memory requirement of the algorithm specially when we use the output of the powering algorithm as a seed approximation for the functional recurrence algorithm. The square root circuit presented in subsection 6.3.1 is based on the powering algorithm for initial approximation. The use of the more accurate algorithmic error analysis we were able to reduce the size of the table used in the powering algorithm to half of its value had we relied on the theoretical error analysis.

The third contribution [29] is the high order Newton-Raphson algorithm for the square root reciprocal. The advantage of using high order Newton-Raphson algorithm is that it gives the designer more freedom in selecting the number of clock cycles of the circuit. In the first order Newton-Raphson the number of clock cycles of the circuit is an integer multiple of three.

We also gave a complete design for the square root circuit in which we use the powering algorithm for the initial approximation then we the second order Newton-Raphson algorithm to approximate the square root reciprocal then multiply it by the operand to get the square root and finally we round the result according to one of the IEEE rounding modes. The rounding algorithm was also presented.

The use of the powering algorithm in the initial approximation leads to a decrease in the table size used in the initial approximation stage. The table size of the powering algorithm is approximately two thirds of that of the first order Taylor approximation. That reduction in the table size comes at the expense of increasing the size of the required multiplier however since we have an existing multiplier in the FMA that we use in the functional recurrence stage therefore the total area of the square root circuit is reduced when using the powering algorithm for initial approximation.

8.3 Recommendations for Future Work

There are still many areas to explore in computer arithmetic in general and in elementary functions in specific.

One important field is the power aware designs. It became quite important recently to reduce the power consumption of the digital circuits for battery-operated devices such as the mobile phone or mp3 player. It is also important to reduce the power consumption so that the overall heat dissipation of the complicated chips is reduced and that can lead to a further increase in the integration density of the chips.

Another interesting area is the more exact modeling of the area and delay functions. Wiring area and delay should be taken into account. This direction is justified by the increasing operating frequencies and the increasing packing density. The former results in more wiring delay while the latter results in more interconnections and thus increased wiring area.

Application specific processors are gaining attention because we have more optimization freedom in their design. Examples of application specific processors

include graphics processor and processing circuits for embedded systems. For such specific applications statistical analysis can be performed to discover the frequency of using specific high level functions. Efficient Algorithms for implementing the frequently used functions in hardware should be devised to increase the overall performance of the systems.

The emerging technologies open new areas for migrating the present computer arithmetic algorithms to cope with such technologies.

References

[1] *IEEE Standard for Binary Floating-Point Arithmetic, (ANSI/IEEE Std 754-1985).* New York, NY: IEEE press, Aug. 1985.

[2] I. Koren, "Hybrid signed-digit number systems: A unified framework for redundant number representations with bounded carry propagation chains," *IEEE Transactions on Computers*, vol. 43, pp. 880–891, Aug. 1994.

[3] B. Parhami, "Generalized signed-digit number systems: A unifying framework for redundant number representations," *IEEE Transactions on Computers*, vol. 39, pp. 89–98, Jan. 1990.

[4] A. A. Liddicoat, *High-Performance Arithmetic For Division And Elementary Functions.* Ph.D thesis, Stanford University, Stanford, CA, USA, Feb. 2002.

[5] J. E. Volder, "The CORDIC trignometric computing technique," *IRE Transactions on electronic Computers*, pp. 330–334, Sept. 1959.

[6] J. S. Walther, "A unified algorithm for elementary functions," in *Proceedings of the AFIPS Spring Joint Computer Conference*, pp. 379–385, 1971.

[7] W. H. Specker, "A class of algorithms for ln x, exp x, sin x, cos x, $tan^{-1}x$, $cot^{-1}x$," *IEEE Transactions on Electronic Computers*, vol. EC-14, pp. 85–86, Feb. 1965.

[8] B. G. DeLugish, "A class of algorithms for automatic evaluation of certain elementary functions in a binary computer," Tech. Rep. 399, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1970.

[9] S. K. Jean-Claude Bajard and J.-M. Muller, "BKM: A new hardware algorithm for complex elementary functions," *IEEE Transactions on Computers*, vol. 43, pp. 955–963, Aug. 1994.

[10] P. K.-G. Tu, *On-line Arithmetic Algorithms for Efficient Implementation*. Dissertation, University of California, Los Angeles, Los Angeles, CA, USA, 1990.

[11] M. J. Flynn, "On division by functional iteration," *IEEE Transactions on Computers*, vol. C-19, pp. 702–706, Aug. 1970.

[12] M. Ito, N. Takagi, and S. Yajima, "Efficient initial approximation and fast converging methods for division and square root," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, England*, pp. 2–9, July 1995.

[13] A. A. Liddicoat and M. J. Flynn, "High-performance floating point divide," in *Proceedings of Euromicro Symposium on Digital System Design , Warsaw, Poland*, pp. 354–361, Sept. 2001.

[14] D. D. Sarma and D. W. Matula, "Measuring the accuracy of ROM reciprocal tables," *IEEE Transactions on Computers*, vol. 43, no. 8, pp. 932–940, 1994.

[15] D. D. Sarma and D. W. Matula, "Faithful interpolation in reciprocal tables," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA*, pp. 82–91, July 1997.

[16] P. Kornerup and J.-M. Muller, "Choosing starting values for Newton-Raphson computation of reciprocals, square-roots and square-root reciprocals," in *presented at RNC5, Lyon, France*, Sept. 2004.

[17] N. Takagi, "Generating a power of an operand by a table look-up and a multiplication," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA*, pp. 126–131, July 1997.

[18] H. Hassler and N. Takagi, "Function evaluation by table look-up and addition," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, England*, pp. 10–16, July 1995.

[19] D. D. Sarma and D. W. Matula, "Faithful bipartite ROM reciprocal tables," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, England*, pp. 17–28, July 1995.

[20] M. J. Schulte and J. E. Stine, "Symmetric bipartite tables for accurate function approximation," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA*, pp. 175–183, July 1997.

[21] J.-M. Muller, "A few results on table-based methods," *Research Report*, vol. 5, Oct. 1998.

[22] M. J. Schulte and J. E. Stine, "The symmetric table addition for accurate function approximation," *Journal of VLSI Signal Processing*, vol. 21, no. 2, pp. 167–177, 1999.

[23] A. T. Florent de Dinechin, "Some improvements on multipartite table methods," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado, USA*, pp. 128–135, June 2001.

[24] P. T. P. Tang, "Table-lookup algorithms for elementary functions and their error analysis," in *Proceedings of the 10th IEEE Symposium on Computer Arithmetic, Grenoble, France*, pp. 232–236, June 1991.

[25] J.-M. Muller, "Partially rounded small-order approximation for accurate, hardware-oriented, table-based methods," in *Proceedings of the 16th IEEE Symposium on Computer Arithmetic, Santiago de Compostela, Spain*, June 2003.

[26] M. H. Payne and R. N. Hanek, "Radian reduction for trignometric functions," *SIGNUM Newsletter*, no. 18, pp. 19–24, 1983.

[27] S. B. Ren-Cang Li and M. Daumas, "Theorems on effcient argument reductions," in *Proceedings of the 13th IEEE Symposium on Computer Arithmetic, Asilomar, California, USA*, July 1997.

[28] S. A. Tawfik and H. A. H. Fahmy, "Error analysis of a powering method and a novel square root algorithm," in *Proceedings of the 17th IMACS World*

Congress, Scientific Computation, Applied Mathematics and Simulation , Paris, France, pp. 126–131, July 2005.

[29] S. A. Tawfik and H. A. H. Fahmy, "Square root and division: An improved algorithm and implementation," *To be published*, 2005.

[30] J. C. Jun Cao, Belle W. Y. Wei, "High-performance architectures for elementary function generation," in *Proceedings of the 15th IEEE Symposium on Computer Arithmetic, Vail, Colorado,USA*, June 2001.

[31] N. L. Carothers, *A Short Course on Approximation Theory*. http://personal.bgsu.edu/∼carother/Approx.html, 1998.

[32] L. Veidinger, "On the numerical determination of the best approximations in the Chebychev sense," *Numerische Mathematik*, vol. 2, pp. 99–105, 1960.

[33] E. Kreyszic, *Advanced Engineering Mathematics*. Wayne Anderson, 1993.

[34] V. K. Jain and L. Lin, "High-speed double precision computation of nonlinear functions," in *Proceedings of the 12th IEEE Symposium on Computer Arithmetic, Bath, England*, pp. 107–114, July 1995.

[35] A. A. Liddicoat and M. J. Flynn, "Parallel square and cube computations," in *Proceedings of the 34th Asilomar Conference on Signals, Systems, and Computers, California, USA*, Oct. 2000.

[36] E. M. Schwarz and M. J. Flynn, "Hardware starting approximation for the square root operation," in *Proceedings of the 11th IEEE Symposium on Computer Arithmetic, Windsor, Ontario, Canada*, pp. 103–111, July 1993.

[37] R. J. Vanderbei, *Linear Programming: Foundations and Extensions*. http://www.princeton.edu/∼rvdb/LPbook/index.html, 2001.

[38] N. Revol and J.-C. Yakoubsohn, "Accelerated shit-and-add algorithms for an hardware implementation of elementary functions," tech. rep., École Normale Supérieure de Lyon, Mar. 1999.

[39] S. A. Tawfik and H. A. H. Fahmy, "Algorithmic truncation of minimax polynomial coefficients," *To be published*, 2005.