T.C.
BAHÇEŞEHİR ÜNİVERSİTESİ

MLUE:
MULTIPLE LOOK-UP TABLE BASED EXPONENTIATION

M.S. Thesis

Hatice ŞAHİN

İstanbul, 2011

T.C.
BAHÇEŞEHİR ÜNİVERSİTESİ
The Graduate School of Natural and Applied Sciences
Electrical and Electronics Engineering

MLUE:
MULTIPLE LOOK-UP TABLE BASED EXPONENTIATION

M.S. Thesis

Hatice ŞAHİN

Supervisor: Asst. Prof. H. Fatih UĞURDAĞ

İstanbul, 2011

T.C.
BAHÇEŞEHİR ÜNİVERSİTESİ
The Graduate School of Natural and Applied Sciences
Electrical and Electronics Engineering

Title of Thesis: MLUE: Multiple Look-Up Table Based Exponentiation
Name/Last Name of the Student: Hatice ŞAHİN
Date of Thesis Defense: September 7, 2011

The thesis has been approved by the Graduate School of Natural and Applied Sciences.

Asst. Prof. Tunç BOZBURA
Director

This is to certify that we have read this thesis and that we find it fully adequate in scope, quality and content, as a thesis for the degree of Master of Science.

Asst. Prof. Dr. Devrim ÜNAY
Program Coordinator

Examining Committee Members:	Signature
Asst. Prof. H. Fatih UĞURDAĞ:	___________________
Assoc. Prof. Çiğdem Eroğlu ERDEM:	___________________
Asst. Prof. Yalçın ÇEKİÇ:	___________________

ACKNOWLEDGEMENTS

I am thankful to my advisor Asst. Prof. H. Fatih UĞURDAĞ for all of his support, insight, and invaluable help for this thesis as well as for making it possible for me to my master's work at Bahçesehir University.

I would also like to thank to my thesis committee members for their time. I would also like to thank all my peers, especially Fatih Temizkan, for their constructive discussions on subproblems in the scope of this thesis work. Then, I would like to thank TÜBİTAK for supporting me within BİDEB scholarship program in 2009 and 2010.

Finally, special thanks go to my parents for their love, support, and encouragement.

Before I conclude this text, I would also like to note that I also appreciate the inspiration that initiated this work, which came from Berkin Bilgiç and Assoc. Prof. Taylan Akdoğan. Berkin Bilgiç is an EECS graduate student at MIT at the moment. However, he was an undergraduate EE and Physics senior at the time he came up with the seed idea behind MLUE. Dr. Taylan Akdoğan is with the Physics Dept. at Boğaziçi University and holds a PhD from MIT.

Hatice ŞAHİN

İstanbul, September 2011

ABSTRACT

MLUE: MULTIPLE LOOK-UP TABLE BASED EXPONENTIATION

Şahin, Hatice

Electrical and Electronics Engineering
Thesis Advisor: Asst. Prof. H. Fatih Uğurdağ

September 2011, 44 Pages

Many scientific applications require computation of exponents. In this thesis, we are specifically interested in computing a constant to the power of a variable number (a^x), which can always be converted to 2^x through an extra multiplication without loss of generality. In real-time systems or any application where run-time matters, a^x is computed using a Look-Up Table (LUT). However, when the targeted precision is high, the table size blows up. Piecewise Polynomial Approximation (PPA) offers a tradeoff between speed and table size and is commonly used in the literature. Our contribution in this thesis is an alternative method, which can also offer a trade-off between speed and table size. It is called MLUE (short for Multiple Look-Up table based Exponentiation). MLUE partitions the input bits into segments. There is a LUT for each segment, and the result is the product of LUT outputs. While PPAs contain both method error and truncation error, MLUE has only truncation error. Although MLUE can be utilized in software implementations, we have looked at its performance when implemented in hardware – and specifically with combinational logic. Our claim is that MLUE offers smaller area for a reasonable target speed and precision. We wrote fully automated and parameterized design (RTL level Verilog) generators for both PPA and MLUE. We back up our claim through numerous results obtained with an automated regression script, which calls our generators. The contributions of this thesis also include regression methodology/scripts, a novel logic synthesis strategy/script, fully automated testing of the generated designs as well as automatic determination polynomial degree in PPA and number of MLUE bit partitions.

Keywords: Computer Arithmetic, HDL, Logic Synthesis, Polynomial Approximation, RTL Generation

ÖZET

MLUE: ÇOKLU SAYI-TABLOSU KULLANARAK ÜS ALMA

Şahin, Hatice

Elektrik-Elektronik Mühendisliği
Tez Danışmanı: Yrd. Doç. Dr. H. Fatih Uğurdağ

Eylül 2011, 44 Sayfa

Birçok bilimsel uygulama üs hesaplaması gerektirir. Bu tezde, biz özellikle bir sabit sayının değişken bir üssünü (a^x) almayla ilgileniyoruz. a^x her durumda ekstra bir çarpma kullanarak 2^x'e dönüştürülebilir. Gerçek-zamanlı sistemlerde veya işlem süresinin kritik olduğu tüm uygulamalarda, a^x bir sayı-tablosu (LUT: Look-Up Table) kullanarak hesaplanabilir. Ancak hedeflenen hesap hassasiyeti yüksek olduğunda, tablo boyutu aşırı büyür. Parçalı Polinom Yaklaşıklama (PPA: Piecewise Polynomial Approximation) hesaplama hızı ve tablo boyutu arasında bir dengeleme yapılmasına imkân verir. Bu tezin literatüre katkısı, aynı şekilde bir dengelemeyi mümkün kılan alternatif bir yöntemdir. Bu yöntemin ismi MLUE'dur (Çoklu Sayı-Tablosu Kullanarak Üs Alma'nın kısaltması). MLUE argümanın bitlerini segmanlara böler. Her segman için bir LUT oluşturulur ve işlem sonucu LUT çıkışlarını çarparak hesaplanır. PPA'de hem metot hem de kırpma hatası varken, MLUE'da sadece kırpma hatası vardır. MLUE'dan yazılım uygulamalarında da faydalanılabilir; ama biz MLUE'nun donanım olarak (birleşimsel lojikle) gerçeklendiğindeki performasını değerlendirdik. Hipotezimiz, MLUE'nun belli bir hız (aşırı olmayan) ve hassasiyet için rakiplerine göre daha az alanlı tasarımlar ürettiğidir. Hem PPA hem de MLUE için otomatik ve parametrize tasarım (RTL seviyesinde Verilog) üreteçleri yazdık. Hipotezimizi otomatik bir regresyon programı tarafından (Verilog üreteçlerimizi çağıran) üretilmiş birçok sentez sonucu ile destekliyoruz. Bu tezin literatüre katkıları arasında, regresyon metodolojisi/kodları, yeni bir lojik sentez stratejisi/kodu, üretilen tasarımların tam-otomatik testi ve PPA polinom derecesi ile MLUE bit segmanlarının sayısının otomatik olarak belirlenmesi de vardır.

Anahtar Kelimeler: Bilgisayar Aritmetiği, HDL, Lojik Sentezi, Polinom Yaklaşıklama, RTL Üreteçleri

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii
ABSTRACT........ iv
ÖZET........ v
LIST OF TABLES........ viii
LIST OF FIGURES ix
LIST OF ABBREVIATIONS x

1. INTRODUCTION 1

1.1 SCOPE........ 1
1.2 GOAL OF THE THESIS 2
1.3 OUTLINE OF THE THESIS 2

2. PREVIOUS WORK 3

3. POLYNOMIAL APPROXIMATION 10

3.1 POLYNOMIAL APPROXIMATION WITH TAYLOR SERIES 10
3.2 COMPUTING THE TERMS 13
3.3 REDUCING TABLE INPUT SIZE 16
3.4 ERROR ANALYSIS FOR POLYNOMIAL APPROXIMATION 17

4. MULTIPLE LOOK-UP TABLE BASED EXPONENTIATION (MLUE)........ 20

4.1 METHOD AND OVERVIEW 20
4.2 COMPUTING THE TERMS 21
4.3 INCREASING ACCURACY AND REDUCING TABLE SIZE 22
4.4 ERROR ANALYSIS 23

5. HDL GENERATORS AND SYNTHESIS - REGRESSION SCRIPTS 24

5.1 HDL GENERATORS 24
5.2 FILE GENERATION 29

6. SYNTHESIS RESULTS 31

6.1 REGRESSION SET UP 31
6.2 REGRESSION RESULTS 31

7. CONCLUSIONS AND FUTURE WORK 34

REFERENCES 35
APPENDIX A 38
CURRICULUM VITAE 44

LIST OF TABLES

Table 6.1: Area results for both methods……………………………………………32
Table A.1: Timing and area results for both methods (8-12 bits)..……......……………39
Table A.2: Timing and area results for both methods (13, 16, 22 bits)…...………..........40
Table A.3: Timing and area results for both methods (23, 32 bits)………...……….....41
Table A.4: Timing and area results for both methods (36, 37 bits)……...………..…..42
Table A.5: Timing and area results for both methods (40, 41 bits)…...……………..…..43

LIST OF FIGURES

Figure 2.1: Illustration of degree-1 interpolation...7
Figure 2.2: Design flow for polynomial approximation and interpolation hardware..7
Figure 2.3: Overview of the steps involved in polynomial approximation and interpolation...8
Figure 2.4: Block diagram of an elementary function generator....................................9
Figure 3.1: Segmentation of inputs for polynomial approximation...............................14
Figure 3.2: Overview of the steps involved in polynomial approximation…………......15
Figure 3.3: Illustration of defining precision according to error……………………..….19
Figure 4.1: Multiple look up table based computing the function f(x)….......................21
Figure 4.2: LUT total area for different partition number for MLUE………………...22
Figure 5.1: Segmentation of subwords for MLUE algorithm 1……………………..…25
Figure 5.2: Terms of intermediate steps for MLUE…………………………………....25
Figure 5.3: Segmentation of subwords for MLUE algorithm 2……………………..…25
Figure 5.4: Multiplier tree for second order polynomial approximation………………27
Figure 5.5: Known bits for second order polynomial approximation………………..….28
Figure 6.1: Area results for some different partition number using MLUE for 32 bits input...33
Figure 6.2: Area results for some different degree using polynomial approximation for 32 bits input...33

LIST OF ABBREVIATIONS

Look-Up Table	: LUT
Multiple Look-Up Table Based Exponentiation	: MLUE
Piecewise Polynomial Approximation	: PPA
Design Compiler	: DC
Hardware Description Language	: HDL
Least Significant Bit	: LSB
Read Only Memory	: ROM
Register Transfer Level	: RTL
Electronic Design Automation	: EDA

1. INTRODUCTION

This chapter gives a bird's eye view of the whole thesis report and gives the motivation behind this work.

1.1 SCOPE

Exponential (an elementary function) is widely used in engineering, scientific, and business applications such as 3-D computer graphics, artificial neural networks, digital signal processing, etc. (Paul, Jayakumar, and Khatri 2009).

In the literature so far, one can find many elementary functions implemented using Look-Up Tables (LUTs) in a diversity of applications. One of the earliest notable papers that presents a LUT approach is Brubaker (1975), which talks about evaluation of a function using a single LUT.

There are many purely LUT based methods for evaluating a function, such as polynomial approximation, interpolation, and combinations of them. Pure LUT method is sometimes used because of its simple design, reasonable delay, and uncomplicated error analysis, but table size grows exponentially with the number of input bits. Also accuracy of LUT approach depends absolutely on LUT's size.

Although memory density has had great progress in recent years, for large input data widths we cannot rely on that. For such cases we need novel approaches that are efficient. So in this thesis, we will introduce two new methods for efficient hardware implementation of the exponential function.

One of them has been inspired by the works in the literature: Piecewise Polynomial Approximation using Taylor Expansion (PPA-TE). The other is the contribution of this thesis work: Multiple Look-Up table based Exponentiation (MLUE). Note that polynomial approximation can be also done using Chebyshev or Remez algorithms. This work extends polynomial approximation by Taylor Expansion to arbitrary order.

In terms of hardware, our approaches were implemented in combinational logic (i.e., in single cycle) with ASIC standard cells (as opposed to FPGA) with goals of high speed and small area. In this work, because of using conversion of the exponential's base from

e to 2, the integer part of the power can be handled by a shift and hence is assumed to be zero (Jamro, Wiatr and Wielgosz 2007).

We have obtained our performance results (timing and area) through full automation. Regressions, synthesis, and design (i.e., Verilog RTL code) are completely automated through Perl and Tcl scripts as well as functional test of the generated Verilog designs. The Verilog generators generate a testbench and wrapper for proper synthesis (all in Verilog) besides the Verilog RTL.

1.2 GOAL OF THE THESIS

The main innovation of this thesis is using small look-up tables (LUTs) through partitioning of the input bits without compromising in logic speed and output precision. While computing the LUT for both methods PPA-TE and MLUE), we optimize the memory requirement for storing coefficients. And also we compare the two methods show that MLUE results in significantly lower area requirement for the same approximation errors and under equal speed constraints. By the way, our method scales very well with an increase in the required precision. We try to use the optimum table size and bit-length to realize a required precision.

1.3 OUTLINE OF THESIS REPORT

In Chapter 2, relevant previous works (literature survey) are described. In Chapter 3, the polynomial approximation method we put together is explained. In Chapter 4, our main contribution, namely, MLUE method is explained. In Chapter 5, the Verilog HDL generator, written for parameterized design, testbench, and wrapper generation, is explained both from an algorithmic as well as implementation perspective. In Chapter 6, the synthesis results of our method and the rival methods are presented for benchmarking of area and timing. We also present the methodology and scripts through which we obtained these results (i.e., regression and synthesis scripts). In Chapter 7, a conclusion and discussion about the thesis is given together with possible future research directions.

2. PREVIOUS WORK

In this chapter, a review of works in the literature review that have the same scope as us is presented. A quick look at the previous work points us to 2 main groups of work on function evaluation: (i) Iterative algorithms (such as CORDIC algorithm), (ii) Non-iterative algorithms (such as direct LUT, polynomial approx., or combination of the two).

We will now walk through individual papers in the literature and explain what they talk about.

Michard, Tisserand, and Veyrat-Charvillon (2005) implement powers of x, by a small number of additions instead of multiplication. Order 2 and 3 polynomial approximation is performed with at most 3-nonzero-bit coefficients. The coefficient of order zero term is stored as large as possible because it is the additive term. In this work, for the purpose of limiting the rounding error, extra guard bits are used for the intermediate computations. For fine-tuning a coefficient, an iterative algorithm is applied until non-zero bits are used or the desired accuracy is reached.

In the paper of Detrey and Dinechin (2005), function evaluation is performed using Taylor Expansion. In their method, the product terms are only stored in tables, so the design becomes faster because it becomes composed of table look-ups and additions. However, in contrast to traditional methods, they use a small multiplier, whose input and output sizes are much smaller than the input and output precision of the function to be evaluated. They also use piecewise polynomial approximation, where the input intervals are partitioned into sub-intervals of the same size, is used. Their piecewise polynomial approximation is focused on the interval A=[0,1), which, in turn, is divided into several sub-intervals $A_i=[i .2^{-a},(i+1) .2^{-a})$. These sub-intervals are addressed by the most significant bits of the input word, and they approximate the function on each of them by a degree n polynomial. Using only some most significant bits in the input word (i.e., LUT address) implies that smaller values will be computed for each interval. In polynomial approximation, order-0 term can be implemented a simple ROM because it doesn’t depend on sub-words. Total error is described in the paper and depends on factors such as table size, rounding, the polynomial expression, etc. To counter the

accumulation of error through various factors, guard bits are used in the output. According to the authors, greater than 16 bits of precision, the optimal solution is a 2nd order polynomial, whereas for 24 bits of precision the optimal solution is a 3rd order polynomial.

The approach of Paul et al. (2009) is based on efficient interpolation, without the need to perform multiplication or division. Also, log() and antilog() (i.e., exponentiation) operations use the same hardware architecture. The main idea in this approach is using LUTs along with linear or quadratic interpolation. They eliminate the multiplication required for interpolation using approximate log() and antilog() functions. According to this work, the most cost effective implementation is a LUT with a linear interpolation. The purpose is that using a smaller size LUT along with a simple linear function to interpolate between the table values and to obtain results quickly and with reduced error. This work uses a LUT based approach combined with a linear interpolation to generate the logarithm of a number. The multiplication required in this linear interpolation is avoided, resulting in an area and delay reduction. While computing the LUT size for this method, they optimize the memory requirement for storing the linear polynomial coefficients as follows: for the constant term stored in the LUT in 2^n locations, the first 3 bits of all 2^n locations are zero. So the first 3 bits can be avoided.

In the paper of Brisebarre, Muller, and Tisserand (2006), they state that two kinds of polynomial approximations are used: (i) Approximations that minimize the average error, called least squares approximations, and (ii) Approximations that minimize the worst-case error called least maximum approximations, or minimax approximations. The aim of both is to minimize the deviation of the fitted polynomial from the original function. Due to the nature of fixed-point arithmetic, the coefficients in the polynomial approximation are usually rounded. The goal is to find best fitted polynomial and yet have a reasonable computation cost. The paper deals with Chebyshev polynomials.

Tak and Tang (1991) state that table look-up based approximation is the most common way for calculating elementary functions from the point of view of speed and accuracy but the table size is very critical for efficient hardware realization. Their table look-up algorithm implementation has 3 steps. First one is reduction; for any input, the algorithm chooses a suitable “breakpoint” and carries out a “reduction transform”.

Second one is approximation; the algorithm fits a polynomial function over the approximated function. The last step is reconstruction; the algorithm combines all steps and calculates function values. They go over the computation of functions 2^x, log$x$, and sin$x$. In summary, their design is very fast in comparison to traditional methods and is very flexible aspect in terms of error analysis. Due to their small rounding errors in intermediate computations, their method is pretty accurate.

Brisebarre and Chevillard (2007) developed an algorithm to find high quality polynomial approximants with floating-point coefficients. The goal in this paper like most of the others is to optimize the error between the function and its polynomial approximation. However, they optimize the relative error ((polynomial approximation – function)/function) instead of the absolute error.

In the paper of Jamro and Wiatr (2007), they state that there are many methods for the purpose such as direct LUT-based, polynomial approximations, and combination of both. The bits of the input argument are divided into several sections. If the section bit-width increases, number of multiplication decreases but the LUT size exponentially increases. This paper converts e^x to 2^(c*x) just like us. They try to minimize the precision of constants especially to reduce area consumption. This paper also discusses the effect of using guard bits on the precision as well as hardware cost.

Hassler and Takagi (1995) state that their algorithm is based on decomposition of a function f(x) into a sum of functions (each with smaller input). So computation of functions is used by several parallel small LUTs and additions. The overall memory space required is much smaller compared to a single straight-forward look-up. They apply their technique to reciprocal, logarithm and exponentiation.

The work of Lee and Villasor (2007) observes that when designing hardware for a function determining the correct bit-widths throughout the whole design is critical. “Correct” means that no unnecessary extra bits or insufficient bits are allocated. This work includes a range analysis algorithm based on computing the roots of the derivatives of the signals for computing the required minimal number of integer bits. The technique is about checking the local minima, local maxima, the minimum and maximum input values at each signal in order to ensure bit-widths, and it is appropriate

for single polynomial and piecewise polynomials approximations. For each output signal, an analytical error expression is generated, and thus, the minimal number of fractional bits is determined.

In Wong and Goto (1995), the authors developed a method called ATA (Add-Table lookup-Add), which requires parallel adds, parallel subtracts, and parallel table look-up followed by a multi-operand summation. Like similar techniques, Taylor series approximation and large look-up tables are used in this method. And also the input word is split into smaller terms.

Kantabutra (1996) focuses on using low-precision elements to realize a high precision function computation. In this work, some of the coefficients are calculated on the fly instead of a direct table look-up. Unlike traditional methods, error is permitted to accumulate through some intermediate steps before precision is checked, and then it is corrected without too much work. Digit-by-digit method is the most commonly used method for hardware evaluation of elementary functions.

Lee et al. (2008) compare hardware implementations using piecewise polynomial approximations versus polynomial interpolations for precisions of up to 24 bits in terms of speed, area, and power consumption. They state that there are 3 sources of error: 1) Inherent error due to method, 2) Quantization error due to finite precision effects incurred when evaluating expressions. 3) Error of the final output rounding step. They try to optimize bit-width so that the total error is minimized. They also discuss advantages and disadvantages hardware architectures for approximation and interpolation methods. They specifically adjust the polynomial as in Figure 2.1 to minimize the first error source. They explain the design flow for hardware design and intermediate steps as in Figure 2.2 and Figure 2.3.

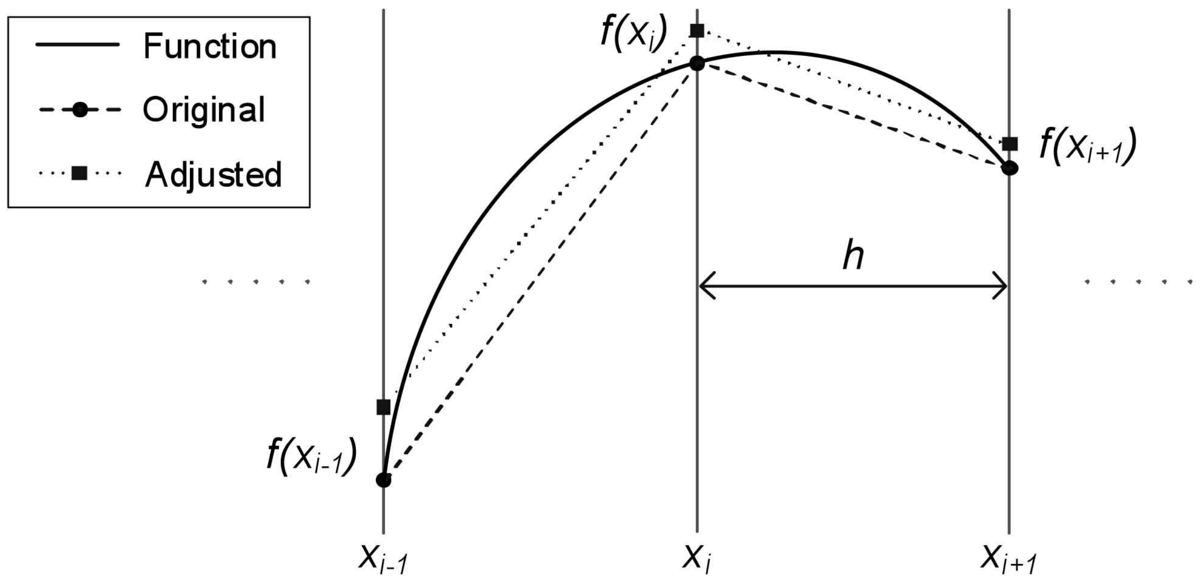

Figure 2.1: Illustration of degree-1 interpolation
Source: Lee et al. (2008)

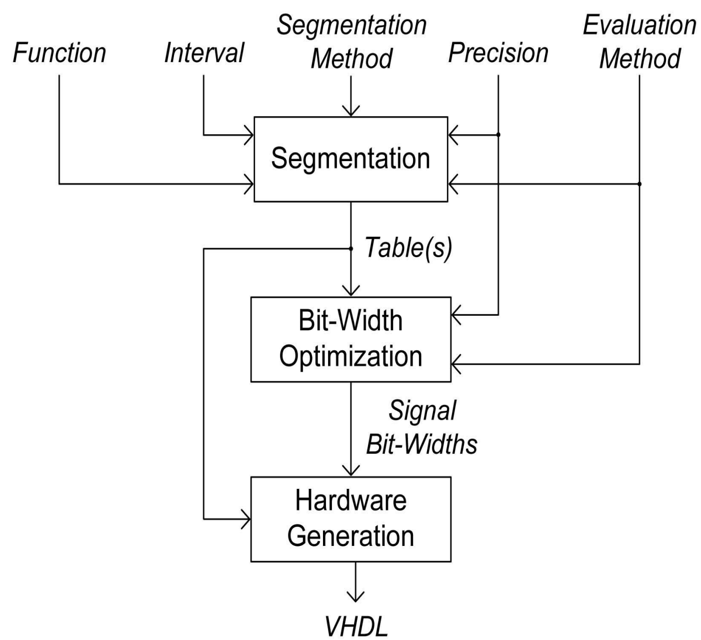

Figure 2.2: Design flow for polynomial approximation and interpolation hardware
Source: Lee et al. (2008)

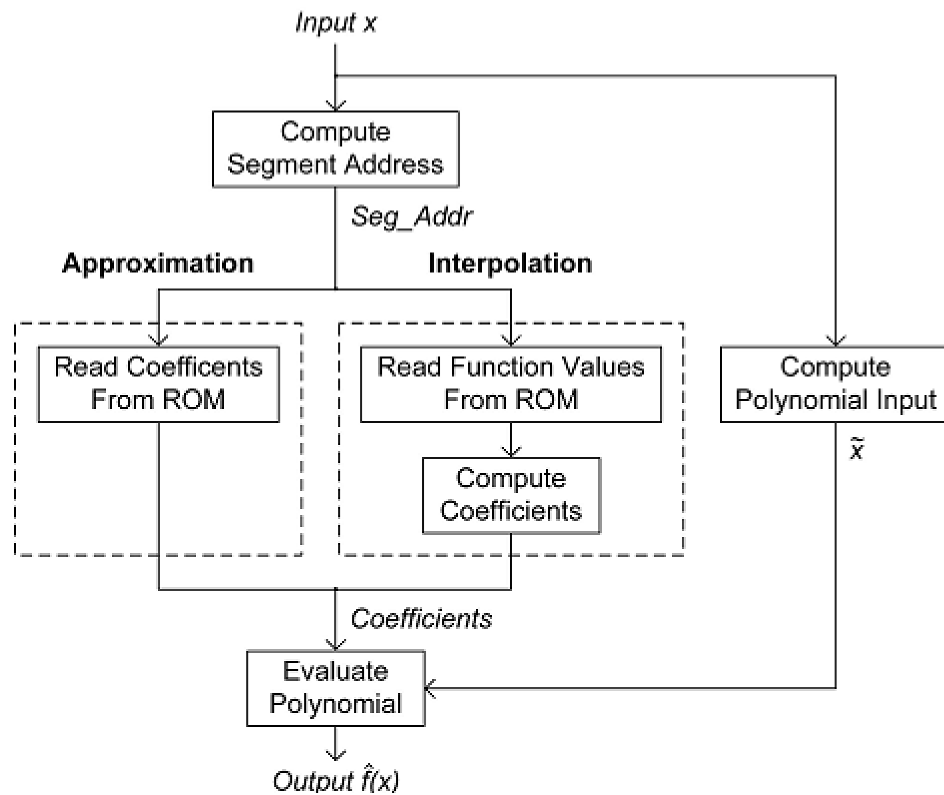

Figure 2.3: Overview of the steps involved in polynomial approximation and interpolation

Source: Lee et al. (2008)

Schulte and Swartzlander (1994) focus on some sort of a polynomial approximation whose coefficients are produced in parallel and then summed by a multi-operand adder. The method is based on partitioning of the input into equal size sub-words and different coefficients which are determined using Chebyshev series approximation. Figure 2.4 shows their idea. In the final stage, first of all the result is calculated by using full-precision (double-precision) arithmetic. Unless the result of this process reaches desired error precision, this routine iterates again until the desired error is obtained. The paper focuses on square root, 2^x, reciprocal, and $\log_2(x)$ computation because their calculation can be done in parallel and division is not required. However, their results show that their method does not work well for large word lengths.

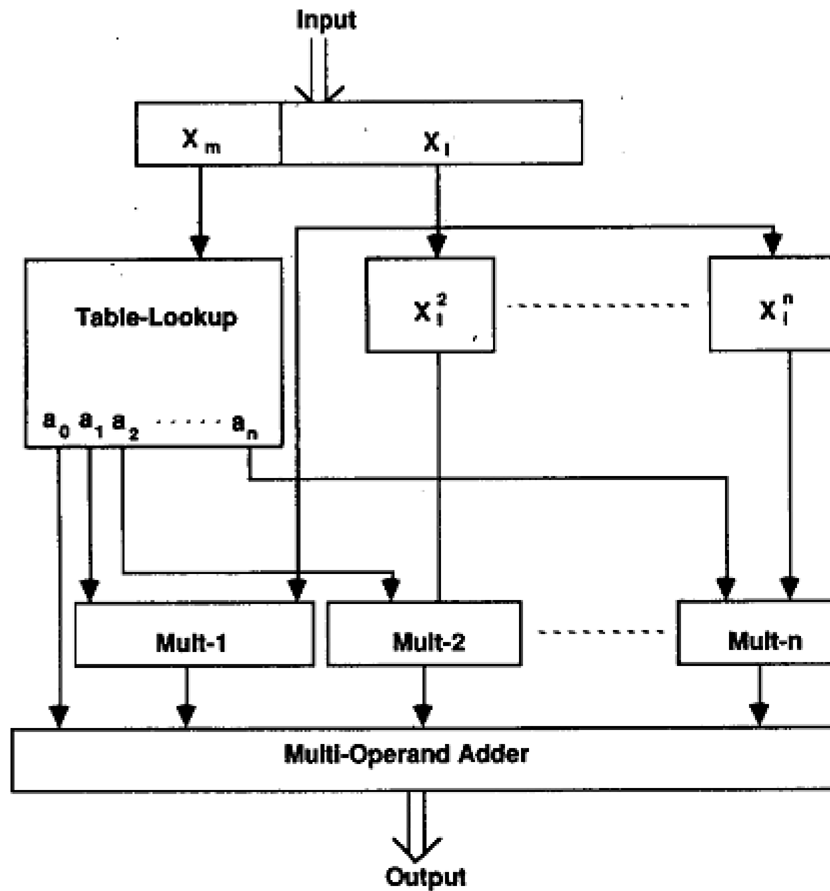

Figure 2.4: Block diagram of an elementary function generator
Source: Schulte and Swartzlander (1994)

Pineiro et al. (2004) present a different algorithm for computation of some elementary functions: 1) Digit-recurrence logarithm, 2) Left-to-right carry free (LRCF) multiplication and 3) Online exponential. A redundant number system is used. The selection in 1) and 3) is done by rounding except in the first iteration, when selecting by table look-up is necessary to guarantee the convergence of the recurrences. An analysis of the trade-offs between area and speed is necessary for determining which radix values result in the most efficient implementation. In terms of error, guard bits are used in each stage.

3. POLYNOMIAL APPROXIMATION

Specific methods, such as polynomial approximation, exist for implementing most of the elementary functions. In the literature, polynomial approximation method is done through Taylor series, Chebyshev series, or Remez algorithm. In our work, we implement polynomial approximation using Taylor series and our input range is [0,1).

In view of the previous work in the literature, our work could be categorized as the particular case of approximation by polynomial segments.

3.1 POLYNOMIAL APPROXIMATION WITH TAYLOR SERIES

Polynomials are just about the simplest mathematical functions that exist, requiring only multiplications and additions for their evaluation. The focus is on how closely the polynomial can follow the fitted function and especially on how small the maximum error can be made. The order of the polynomial can always be increased to boost accuracy of the fitting.

In general, a polynomial approximation has the following form:

$$f(x) \approx a_0 + a_1x^1 + a_2x^2 + \ldots + a_{n-1}x^{n-1} = \sum_{i=0}^{n-1} a_i x^i \qquad \textbf{(3.1)}$$

where f(x) is the function to be approximated, n is the number of terms in the polynomial approximation, a_i is the coefficient of the ith term.

Our aim is to find a polynomial that gives us a good approximation to exponentiation function. We find the desired polynomial approximation using a Taylor Series. A Taylor series is a representation of a function about a point with infinite sum of terms using function's derivatives at the point.

If we want a good approximation to the function in the region near $x = a$, we need to find the first, second, third (and so on) derivatives of the function and substitute the value of a. Then we need to multiply those values by corresponding powers of $(x - a)$, giving us the **Taylor Series expansion** of the function $f(x)$ about $x = a$:

$$f(x) = f(a) + f'(a)(x-a) + \frac{f''(a)}{2!}(x-a)^2 + ... + \frac{f^{(n)}(a)}{n!}(x-a)^n \quad \textbf{(3.2)}$$

We can write this more conveniently using summation notation as:

$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \quad \textbf{(3.3)}$$

The function can be approximated by adjusting the polynomial degree for a given target precision. The number of terms in polynomial approximations depend on the accuracy we are after.

For example, writing an order-5 Taylor expansion of *f* at x_0:

$$\begin{aligned} f(x) &= f(x_0) && (T_0) \\ &+(x-x_0)f'(x_0) && (T_1) \\ &+\frac{1}{2}(x-x_0)^2 f''(x_0) && (T_2) \\ &+\frac{1}{6}(x-x_0)^3 f'''(x_0) && (T_3) \\ &+\frac{1}{24}(x-x_0)^4 f^{(4)}(x_0) && (T_4) \\ &+\frac{1}{120}(x-x_0)^5 f^{(5)}(x_0) && (T_5) \\ &+\varepsilon_1 \end{aligned} \quad \textbf{(3.4)}$$

The error committed is

$$\varepsilon_1 = \frac{1}{720}[x_1 2^{-k} + x_2 2^{-2k} + x_3 2^{-3k} + x_4 2^{-4k}]^6 f^{(6)}(\zeta_1) < \frac{1}{720} 2^{-6k} \max\left|f^{(6)}\right| \quad \textbf{(3.5)}$$

In equation (3.4), we expand the term $(T_1) = (x-x_0)f'(x_0)$ as follows:

$$\begin{aligned} (T_1) &= (x-x_0)f'(x_0) \\ &= x_1 2^{-k} f'(x_0) + x_2 2^{-2k} f'(x_0) + x_3 2^{-3k} f'(x_0) + x_4 2^{-4k} f'(x_0) \end{aligned} \quad \textbf{(3.6)}$$

Now let us focus on terms $(T_2) = \frac{1}{2}(x-x_0)^2 f''(x_0)$, $(T_3) = \frac{1}{6}(x-x_0)^3 f'''(x_0)$,

(T$_4$)= $\frac{1}{24}(x-x_0)^4 f^{(4)}(x_0)$ and (T$_5$) = $\frac{1}{120}(x-x_0)^5 f^{(5)}(x_0)$ from equation (3.4). We have:

$$(T_2)=\frac{1}{2}x_1^2 2^{-2k} f''(x_0)+x_1x_2 2^{-3k} f''(x_0)+x_1x_3 2^{-4k} f''(x_0)+\frac{1}{2}x_2^2 2^{-4k} f''(x_0)$$
$$+x_2(1/2)2^{-5k} f''(x_0)+\frac{1}{2}x_1(1/2)2^{-5k} f''(x_0)+\varepsilon_2 \quad \textbf{(3.7)}$$

$$(T_3)=\frac{1}{6}x_1^3 2^{-3k} f'''(x_0)+\frac{1}{2}x_1^2x_2 2^{-4k} f'''(x_0)+\frac{1}{2}x_1^2x_3 2^{-5k} f'''(x_0)$$
$$+\frac{1}{2}x_1x_2^3 2^{-5k} f'''(x_0)+\varepsilon_3 \quad \textbf{(3.8)}$$

$$(T_4)=\frac{1}{24}x_1^4 2^{-4k} f^{(4)}(x_0)+\frac{1}{6}x_1^3x_2 2^{-5k} f^{(4)}(x_0)+\varepsilon_4 \quad \textbf{(3.9)}$$

$$(T_5)=\frac{1}{120}x_1^5 2^{-5k} f^{(5)}(x_0)+\varepsilon_5 \quad \textbf{(3.10)}$$

with $\varepsilon_2<\frac{1}{2}2^{-5k}\max|f''|$, $\varepsilon_3<\frac{1}{3}2^{-6k}\max|f'''|$, $\varepsilon_4<\frac{1}{24}2^{-6k}\max|f^{(4)}|$ and $\varepsilon_5<\frac{1}{120}2^{-6k}\max|f^{(5)}|$. We rewrite the equation (3.4) as follows:

$$f(x)=A(x_0,x_1)+B(x_0,x_2)+C(x_0,x_3)+D(x_0,x_4)$$
$$+x_2\times E(x_0,x_1)+x_3 2^{-k}\times E(x_0,x_1)+\varepsilon_f \quad \textbf{(3.11)}$$

where

$$A(x_0,x_1)=f(x_0)+x_1 2^{-k} f'(x_0)+\frac{1}{2}x_1^2 2^{-2k} f''(x_0)+\frac{1}{6}x_1^3 2^{-3k} f'''(x_0)+\frac{1}{24}x_1^4 2^{-4k} f^{(4)}(x_0)$$
$$+\frac{1}{120}x_1^5 2^{-5k} f^{(5)}(x_0)+\frac{1}{2}x_1(1/2)2^{-5k} f''(x_0) \quad \textbf{(3.12)}$$

$$B(x_0,x_2)=x_2 2^{-2k} f'(x_0)+\frac{1}{2}x_2^2 2^{-4k} f''(x_0)+(1/2)x_2 2^{-5k} f''(x_0) \quad \textbf{(3.13)}$$

$$C(x_0,x_3)=x_3 2^{-3k} f'(x_0) \quad \textbf{(3.14)}$$

$$D(x_{0,}x_4) = x_4 2^{-4k} f'(x_0) \quad \textbf{(3.15)}$$

$$E(x_{0,}x_1) = x_1 2^{-3k} f''(x_0) + \frac{1}{2} x_1^2 2^{-4k} f'''(x_0) + \frac{1}{6} x_1^3 2^{-5k} f^{(4)}(x_0) + \frac{1}{2} x_1 (1/2) 2^{-5k} f'''(x_0) \quad \textbf{(3.16)}$$

$$\varepsilon_f \le \varepsilon_1 + \varepsilon_2 + \varepsilon_3 + \varepsilon_4 + \varepsilon_5$$

$$\le \frac{1}{720} 2^{-6k} \max\left|f^{(6)}\right| + \frac{1}{2} 2^{-5k} \max\left|f''\right| + \frac{1}{3} 2^{-6k} \max\left|f'''\right| + \frac{1}{24} 2^{-6k} \max\left|f^{(4)}\right| + \frac{1}{120} 2^{-6k} \max\left|f^{(5)}\right|$$

$$\le 2^{-5k} \left[\frac{1}{2} \max\left|f''\right| + \frac{1}{3} \max\left|f'''\right| + \frac{1}{24} \max\left|f^{(4)}\right| + \frac{1}{120} \max\left|f^{(5)}\right| + \frac{1}{720} \max\left|f^{(6)}\right| \right] \quad \textbf{(3.17)}$$

Hence, *f(x)* can be obtained by performing 2 multiplications and adding 6 terms with an error less than ε_f. The size of the tables where all these terms are looked-up from, depend heavily of the function we consider.

3.2 COMPUTING THE TERMS

Our algorithm adjusts the LUT sizes and polynomial order for the desired precision. The challenge is to find the required intermediate precisions in the sum of products expression, given the input and output precision requirements.

The hardware organization required for the straight-forward interpolation by polynomial of degree d is shown in Figure 3.1. First k bit and $(k+1)^{th}$ bit used for calculating the address of memory unit to get the corresponding coefficients. To get address of the memory unit $(k+1)^{th}$ is added to first k bit. The memory size is 2^k+1 words. The (input_bit-k) approximating bits constitute the variable t. The t^h values, h=2, 3, …, d, are obtained in parallel with memory reads. The products are obtained in parallel multiple units, the bit-widths of which are usually less than that of the result. The products are combined in the final adder.

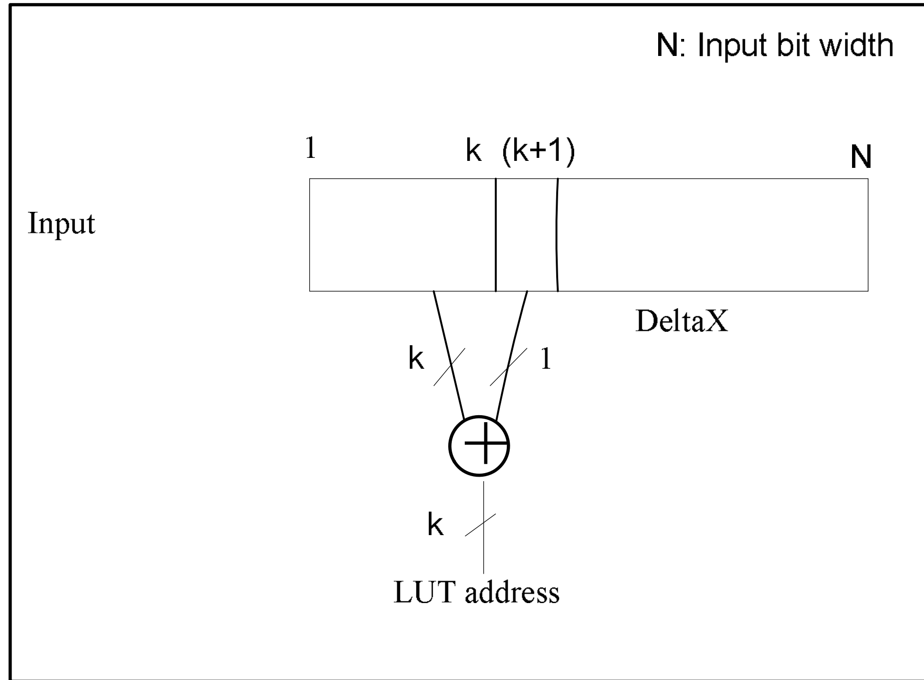

Figure 3.1: Segmentation of inputs for polynomial approximation

Coefficients are calculated as follows:

n: degree of polynomial, k: number of LUTs entry, x_k: points of the interval

$$c_{nk} = \frac{(-\ln 2)^n}{n!} 2^{-x_k} \qquad \textbf{(3.18)}$$

In our polynomial approximation technique, the evaluation interval is partitioned in to 2^k segments and an approximating function is specified for each segment. As shown in Figure 3.1, k bit LUT address identifies the segment. In an implementation, these bits will be used to address the storage unit holding the parameters of the approximating function. They will be called address bits. The next (input_bit-k) bits of x specify the point within the approximating segment at which the evaluation is to take place. These will be called interpolating bits (DeltaX).

Broadly stated (Figure 3.2), the polynomial approximating function evaluation method of the present invention comprises accessing a memory unit by specified bits of a function argument for the memory unit address, obtaining parameters of an approximating function from the memory unit, and evaluating, using other specified

bits of the argument, the specific approximating function whose parameters were obtained from the memory unit.

Memory is addressed using k MSB bits of the function argument. The memory unit contains the polynomial coefficients. The evaluation comprises of evaluating the degree d-polynomial (with fixed combinational logic) whose coefficients are obtained from the memory unit, using as the polynomial argument the bits of the function argument of lower order than the k bits used to address the memory unit.

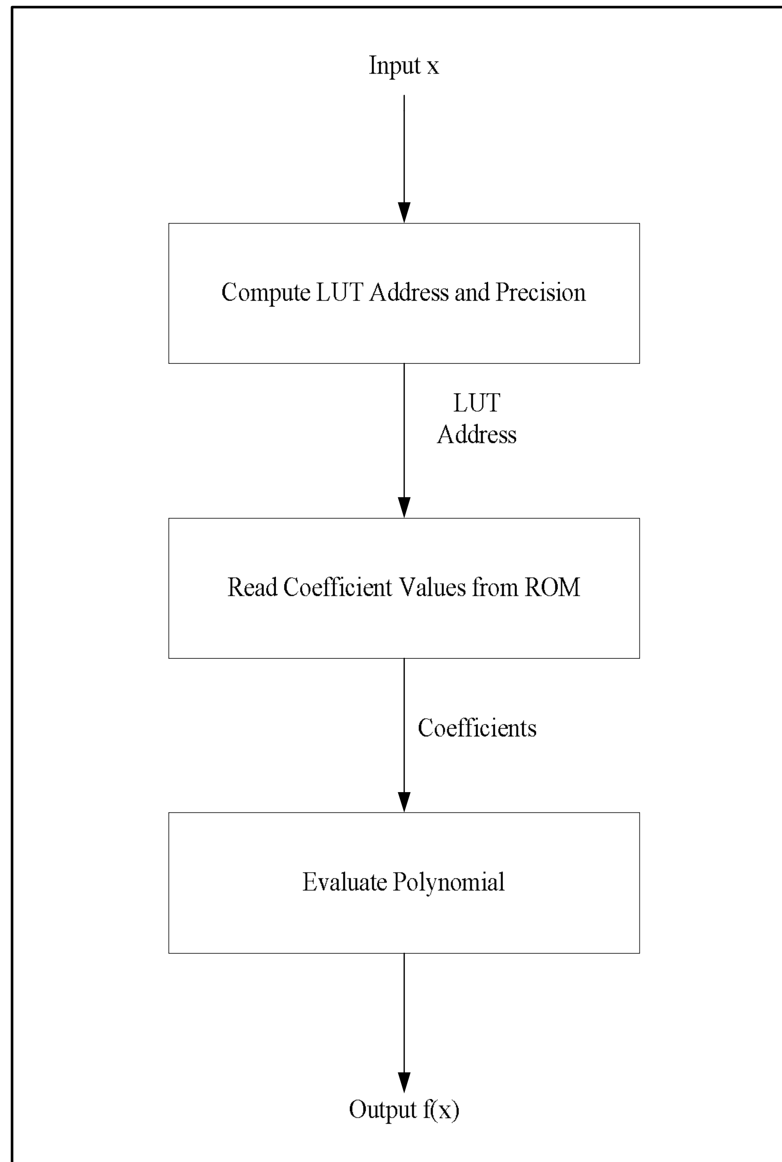

Figure 3.2: Overview of the steps involved in polynomial approximation

Evaluation of the polynomial is performed using the value t as an input, which is comprised of bits of the function argument to the right of the k bits (used as the address

of the memory unit). The quantities t^2, t^3, …, t^d are computed with a set of multipliers operating in parallel and then further multiplied with the coefficients obtained from the memory units and then summed.

The function evaluation includes using a binary representation of a number, addressing a memory with one part of the representation and supplying another part of the representation to combinational logic units in parallel with the memory, supplying coefficients from the memory to parallel multipliers and supplying values from the combinational logic to the multipliers, obtaining products, adding products and thereby producing a function evaluation.

3.3 REDUCING TABLE INPUT SIZE

One of the main challenges when designing hardware-based function evaluation units is the determination of the correct number of bits for the signals in the fixed-point data path. Excessive bit-width allocation will result in wasting valuable hardware sources, whereas insufficient bit-width allocation will introduce overflows and violate precision requirements. Thus, it is desirable to find the minimal bit-widths for all signals.

A method is presented here for determining the widths of data paths for the computation of *f(x),* given a specification of the maximum error (or the required precision) of the result. The data path widths determine the hardware required for the arithmetic logic and memory coefficients. Two’s complement representations is used in our arithmetic logic.

We used equation 3.18 to calculate the coefficients. In that eqution, there is $(-\ln 2)^n/n!$ operation. This equation creates coefficients with very small absolute value and hence many MSB bits are identical (i.e., sign-extension). For example, n=2 and $(-\ln 2)^n/n! = 0.2402$ for coefficient C_2. So, for all entry of the coefficient C_2 there are 3 bits that are zero (in binary) at the beginning of that coefficient. Therefore, we do not need to store these bits in the LUT. We can simply append them at the output of the LUT. In this way, we store $2^k * 3$ less bits at the look up table for coefficient C_2.

According to equation 3.18, coefficients can be positive or negative. So the assumed (i.e., not stored) bits at the beginning of the polynomial coefficient are 0 for positive and 1 for negative coefficients.

We have a special case for memory of polynomial approximation, so our memory size is 2^k+1 instead of 2^k. The special case is as follows:

Using DeltaX parameter, which is approximating parameter in Figure 3.1, the point argument x is figured out which segment close to that point. For this decision, if approximating parameters' LSB is equal to 0 in binary representation, the address remains the same. However, if the bit is equal to 1 in binary, '1 is added to the address. Unfortunately, overflow occurs for '11111….1' value for LUTs' address because of that situation we have to store an additional value in memory.

3.4 ERROR ANALYSIS FOR POLYNOMIAL APPROXIMATION

In hardware implementation, we do not have an infinite amount of resources, so we have to truncate or round numbers.

Lee and Villasor (2007) implemented polynomial approximation for several bit-widths and compared rounded and truncated results in term of area and timing. According to their results truncation gives better results for both parameters as rounding needs extra addition operation for each multiplication and addition. We took this into account and decided to use truncation for all operations.

So, in addition to the approximation errors considered in the previous section, this architecture involves several sources of truncation error:

- The tables have to be filled with fixed-point values which are the mathematical values truncated to some precision.
- Similarly, the outputs of the multipliers have to be truncated to the target precision.
- These approximation errors require us to compute with an internal precision, which is higher than the final required precision. We hence use some additional bits of precision, which are called guard bits in literature. The number of guard

bits that we use is 2. Because if we use 1 guard bit, the bit that has error may affect the correct bits. The number of guard bits is determined as the smallest value such that the sum of approximation error and all the truncation errors is smaller than an LSB of the final result (faithful truncation).

- The,n the results have to be truncated to the final precision after the final addition (with an error at worst of an half LSB).

In generally, approximation error is calculated as presented on equation 3.19.

D: Degree of polynomial approximation, E: Approximation error, S: Bit-width for LUT addresses

$$\frac{(\ln 2)^{D+1}}{(D+1)!} \times \frac{2^{-S(D+1)}}{2^{(D+1)}} = 2^{-E} \tag{3.19}$$

If approximation error decreases, LUT address bits decrease too.

In aspect of precision, input and output have the equal bits. But actually,

OutputBitWidth = InputBitWidth + 2 guard bits

For example, if input is 32 bits, output must be 34 bits. As shown in Figure 3.3, all parameters have 37 bits, which are adjusted for approximation with second order polynomial.

For second order polynomial approximation, $f(x) = c_0 + c_1 \Delta x + c_2 \Delta x^2$

●: bit, **x:** position of the max error

	1st bit	2nd bit	……	33th bit	34th bit	35th bit	36th bit	37th bit
c_0	●	●					**x**	●
$c_1\Delta x$	●	●					**x**	●
$c_2\Delta x^2$	●	●					**x**	●
Appr. error	●	●					**x**	●
Result	●	●			**x**	●	●	●
Output	●	●		**X**	●			

Figure 3.3: Illustration of defining precision according to error

The value of wrong bit position= $2^1+2^1+2^1+2^1=8$ → $\lceil \log_2 8 \rceil = 3$

So it is shifted left by 3 bit positions. After that 1 bit is shifted left for truncation error. It means that if output precision is 34 bits, all parameters have 38 bits in order that errors do not affect the achievement of the required precision.

4. MULTIPLE LOOK-UP TABLE BASED EXPONENTIATION (MLUE)

Many applications require the evaluation in hardware of a numerical function: trigonometric functions for digital signal processing algorithms or exponential and logarithmic function for some scientific computing applications but we especially focus on 2^{-x} where x is in [0, 1). The size and speed of the operator depends on the input and output precision, but also on function. We wrote circuit generators for 2^{-x}. These generators compute the optimal implementation and output circuit descriptions in the Verilog language, suitable for synthesis. The main motivation is to obtain efficient (timing and area-wise) hardware implementations of 2^{-x}.

4.1 OVERVIEW OF THE METHOD

Our proposed method synthesizes an implementation given a maximum error (the required precision) for the result. The hardware requirements of memory coefficients are all based on the specified error/precision.

An approximation algorithm has been presented which allows us to decompose 2^{-x} into a product of multiple 2^{-x}'s, each with smaller input. The general idea is to compute 2^{-x} by several parallel look-ups from small LUTs and multiply them (see Eq. 4.1). Hence, the overall memory space becomes is much smaller than the direct single table approach. The additional cost is a few multipliers.

The architecture is shown in Figure 4.1. The presented method uses smaller bit partitions of the function argument (x) to address a memory holding 2^{-x} in smaller ranges uses multipliers in parallel to compute the product.

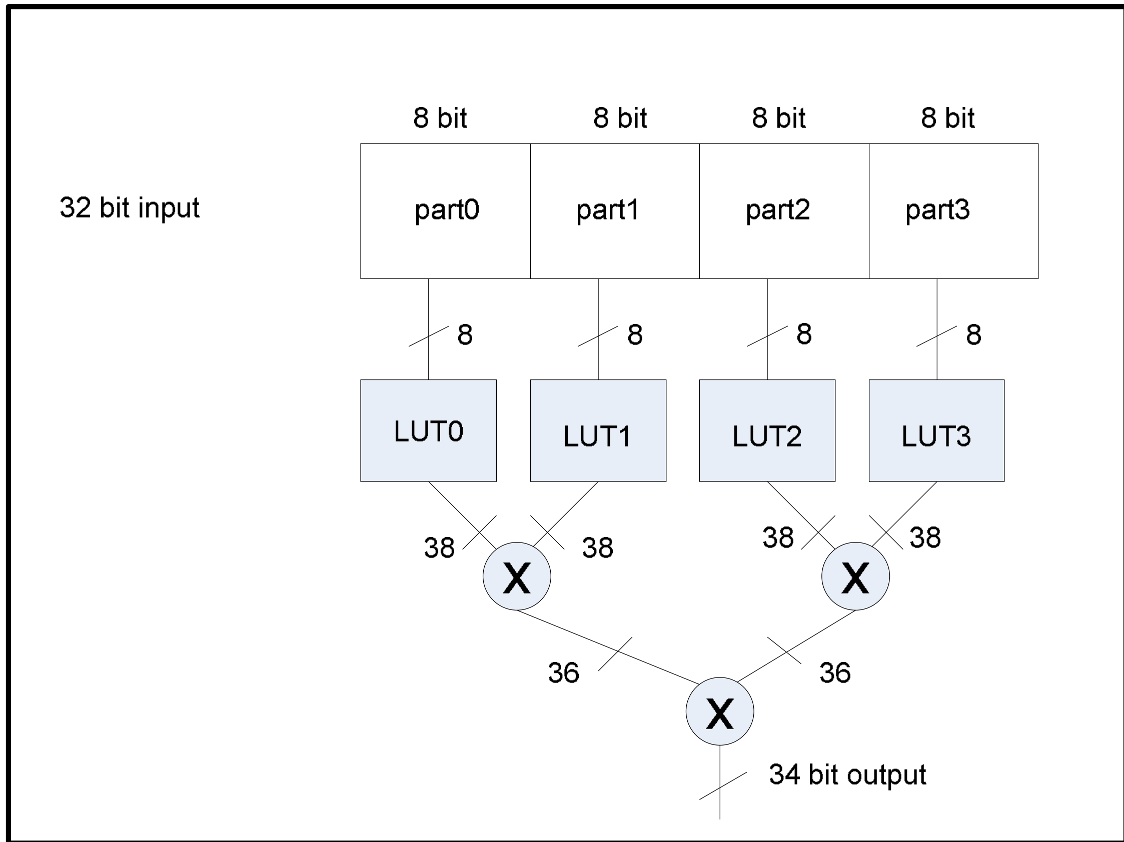

Figure 4.1: Multiple LUT based computing of 2^{-x}

This new multipartite representation can be easily programmed with efficient computer storage. To construct a module generator for LUTs and multiplier units, we need to compute all internal parameters (precision) given precision/range of the output. Internal parameters consist of the width and height of the LUTs and the number of bits for the multiplication.

4.2 COMPUTING THE TERMS

The input argument is divided into several equal (could actually be unequal) sections. Each section is used as an address to a separate LUT. Then, the LUT outputs are all multiplied. As shown in Figure 4.2, if the section bit-width increases, number of multiplications decreases but the sizes of LUTs increase exponentially.

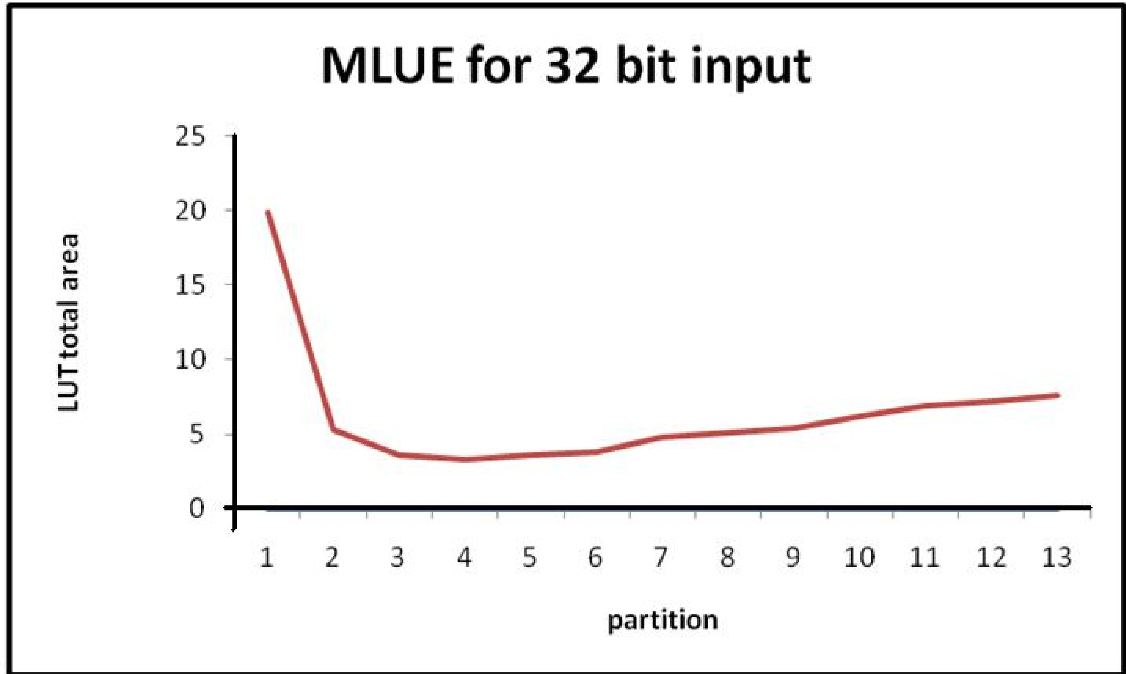

Figure 4.2: LUT total area for different partition number for MLUE

We express our MLUE method as:

$$2^{-0.p_1p_2p_3p_4p_5p_6} = 2^{(-0.p_1p_2 - 0.00p_3p_4 - 0.0000p_5p_6)} = 2^{-0.p_1p_2} \times 2^{-0.00p_3p_4} \times 2^{-0.0000p_5p_6} \qquad \textbf{(4.1)}$$

According to this partition, 3 different LUTs are generated, each with 4 entries.

4.3 INCREASING ACCURACY AND REDUCING TABLE SIZE

Another important step when computing an elementary function is range reduction. Most approximations of functions are valid in a small interval only. To compute *f(x)* for any input value *x*, one must first find a number *y* such that *f(x)* can be easily be deduced from *f(y)* (or more generally from an associated function *g(y)*), and such that *y* belongs to the interval where the approximation holds. This operation is called range reduction and *y* is called the reduced argument. For many functions (especially the sine, cosine and tangent functions), range reduction must be performed cautiously, it may be the most important source of errors.

While minimizing the total LUT size on one hand, we also reduce the bit-width of LUT entries. The output of LUT0 in Fig. 4.1 turns out to be in (0.5, 1) where the two ends are exclusive, provided we treat *x*=0 as a special case. Hence, the first digit to the right of the decimal point happens to be one at all times and is not stored in LUT0.

Similarly, LUT1 happens to have 4 ones, LUT2 8 ones, and so on. The number of ones in general is equal to the number of digits to the left of a LUT in the input argument (x). There is no need to keep these redundant bits in the LUTs and this gives us a lower memory requirement in overall.

4.4 ERROR ANALYSIS

Careful adjustment of bit-widths of coefficients and intermediate results is required for meeting a particular precision requirement, and that requires a detailed error analysis. The precision of a quantity is the number of bits in its binary representation. If bounds on the error of a quantity are known in design stage, hardware can be optimized by not implementing meaningless bits.

One of the most important properties of MLUE is that it does not have any “method error”. On the other hand, it has rounding error in LUT entries as well as truncation error in arithmetic computations.

Although rounding introduces a smaller error compared to truncation, it is more costly in terms of both timing and area. As a result, we prefer truncation in arithmetic computations since they are done with hardware resources. However, computation of LUT entries are done offline and hence rounding can be used for them.

The final output of the multipliers of MLUE is truncated to the target precision. We compute with an internal precision which is higher than the final required precision. We use some additional bits of precision, which are called guard bits in the literature. The number of guard bits that we use is 2. Because if we use 1 guard bit, the bit that has error may affect the correct bits. The number of guard bits is determined as the smallest value such that the sum of error and all the truncation errors is smaller than a LSB of the final result (faithful truncation).

5. HDL GENERATORS AND SYNTHESIS - REGRESSION SCRIPTS

In this chapter, scripts written for the purpose of generating the Verilog HDL files for the design part as well as testbench are explained in terms of their algorithm and implementation. These scripts are written using Perl. All generated Verilog files comply with Verilog 2001 standard.

5.1 HDL GENERATORS

Mlue.pl: This Perl script generates our MLUE design in Verilog. Its arguments are number of input bits and number of subwords. Based on arguments, error is estimated and bit-widths of intermediate steps are decided. It is run as follows:

```
$ Mlue.pl <input_width> <number_of_parts>
```

We partition the input into subwords. While partitioning, the following algorithm is used to decide bit-widths of the subwords. If the subword bit-width divides the input bit-width without a remainder, all subwords have equal number of bits. If not, all subwords have equal number of bits except the last subword. In the second scenario, all of the remainder bits are grouped spread to subwords starting from the very right one by one in order to optimize LUT area. Since the rightmost LUT has the lowest bit-width due to redundant ones on the left, it makes sense to give it one more bit if we have a remainder of one bit. However, if there is a remainder of 2, we need to give it to the next subword to the left of it because the effective width of the rightmost subword becomes doubled. Below, we consider other options and show why the above option is the best for the case of 16 bit inputs and 3 parts (subwords).

Segmentation of subwords for MLUE algorithm 1:

We type the following command line:

```
$ Mlue.pl 16 3
```

Figure 5.1: Segmentation of subwords for MLUE algorithm 1

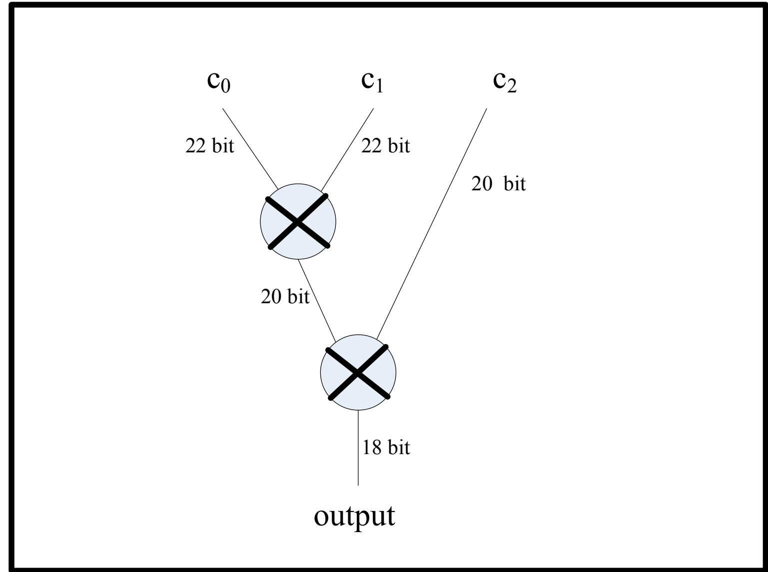

Figure 5.2: Terms of intermediate steps for MLUE

In MLUE, leftmost bit is always 1 for the first coefficient (LUT0) because output interval is always (0.5,1) , so we do not store this bit in the LUT, however it is included at the multiplication. According to Figure 5.1 and 5.2, coefficients which are kept in the LUTs are as follows: coefficient of Part 0 (c_0) is 21 bits, coefficient of Part 1 (c_1) is 16 bits and coefficient of Part 2 (c_2) is 9 bits.

So, total number of bits in the LUTs is:

$$= 2^6 \times 21b + 2^5 \times 16b + 2^5 \times 9b = 2144 \quad \textbf{(5.1)}$$

Segmentation of subwords for MLUE algorithm 2:

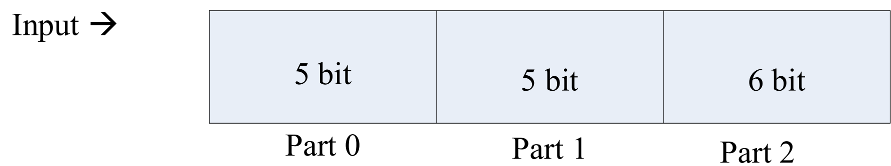

Figure 5.3: Segmentation of subwords for MLUE algorithm 2

According to Figure 5.2 and 5.3, coefficients which are kept in look up tables are like that: coefficient of Part 0 (c_0) is 21 bits, coefficient of Part 1 (c_1) is 17 bits and coefficient of Part 2 (c_2) is 10 bits.

So, total bit number in look up tables

$$= 2^5 \times 21b + 2^5 \times 17b + 2^6 \times 10b = 1856 \quad \textbf{(5.2)}$$

If both total storage capacities are compared using equation 5.1 and 5.2, algorithm 2 is more efficient in terms of memory size (in bits with 1856 versus 2144).

Poly_fitting.pl: We wrote this Perl script to generate our design (in Verilog) for the polynomial fitting. This design is the competition of our MLUE method. It is similar to works in the literature and is implemented to compare the performance of our MLUE to the existing art. It was critical that we write a generator for polynomial fitting because we need to synthesize MLUE and competition using the same tools and libraries.

Calculation of the bit-widths of LUTs and intermediate calculations for "poly" is also driven by the eventual desired error precision (just like MLUE). Input is divided into two parts which are LUT address bits and the term which is used for Taylor expansion coefficient (DeltaX). The value DeltaX can be negative and even in that situation we represent it with the minimal number of bits (through a fixed-point representation that has "negative" integer bits).

The arguments of this Perl script are the number of input bits and order of polynomial – very similar to the MLUE script.

Script gets two required inputs in the following way:

```
$ Poly_fitting.pl <input_width> <order_of_polynomial>
```

For example,

```
$ Poly_fitting.pl 32 2
```

is typed at the command line for 32 bit input and 2nd order polynomial:

$$f(x) = c_0 + c_1 \Delta x + c_2 \Delta x^2 \quad (5.3)$$

$$\Delta x \leq 2^{-(LUTaddress+1)=-11} \quad , \quad \Delta x^2 \leq 2^{-21} \quad , \quad c_2 \leq 2^{-2}$$

Using generators we compute the optimal implementation for a given precision, bit-width of coefficients are defined in Figure 5.4 shows the optimal bit-widths for a specific case. In this case, all multiplier outputs must have 36 bits for an eventual output width of 34 bits resulting in 32 bit precision plus 2 guard bits. Note that in the end all multiplier outputs are summed.

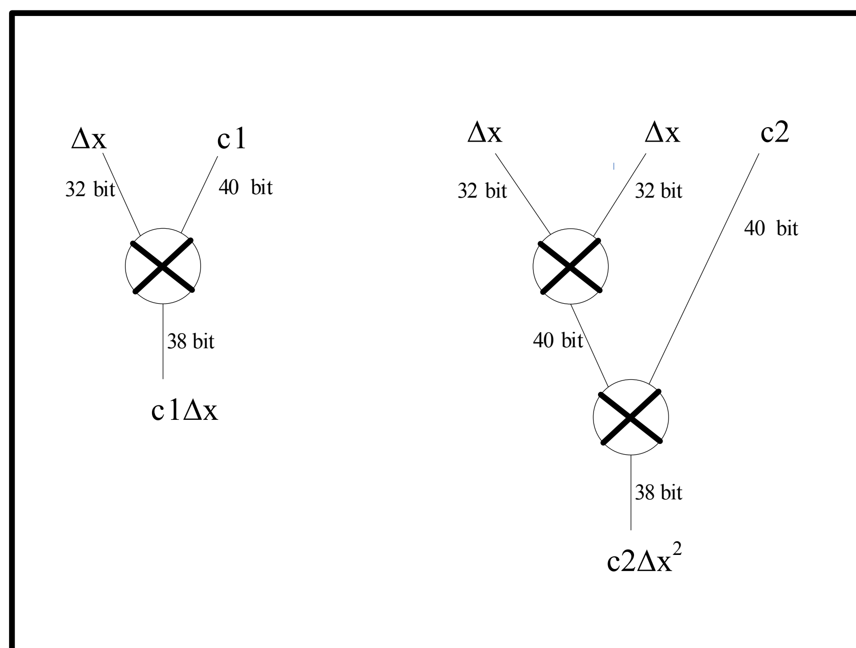

Figure 5.4: Multiplier tree for second order polynomial approximation

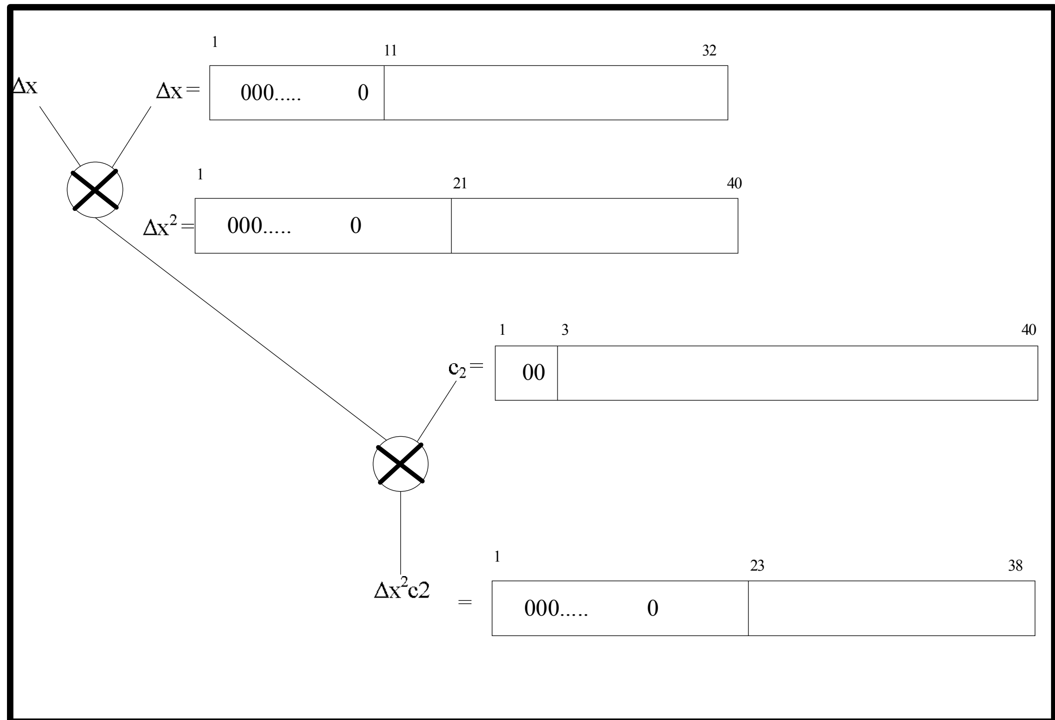

Figure 5.5: Known bits for second order polynomial approximation

In Poly_fitting.pl, number of known bits (i.e., repeated or sign-extension bits) at the beginning of polynomial coefficients are computed through Eq. 3.19 (replace D+1 with the order of the coefficient). Known bits at the beginning of the polynomial coefficient are 0 for positive or 1 for negative terms.

syn.pl: Synthesis is done through this Perl script. This Perl script simulates the design first (though optional) using test bench and top module files, and after that it synthesizes the design with Synopsys DC software if it passes verification (i.e., simulation). Its arguments are the number of synthesis iterations and job list, which are all contained in reglist.lst file.

Testbench: To verify the correctness of the design, $2^{inputBitWidth}$ is multiplied by difference between designed RTL output and real reference output. If this calculation result is smaller than 1, it means that there is no error in interval of input bit-width. If the result is bigger than 1, there is error in interval of input bit-width. If it is equal to1, there is an error at the position of input bit-width.

Wrapper Structure**:** In synthesis, we have a MUX set, which is a multiplixer chain instead of the LUTs. It is a place holder for the LUTs from a timing point of view. Note

that area of the LUTs are computed separately outside the synthesis. We have an optimization in the wrapper structure. For all subword parameters, we have a different MUX set because again for known bit values, the blocks which have less input bits are used.

Script arguments are like as follows:

$ syn.pl <job_list> <whether_to_resyn> <syn_with/without_tb>

reglist.lst: This file contains the job list. Every line in the file is a synthesis job. That is, Synopsys DC is run as many times as the number of lines in the file. There are some options for job list arguments. A job list line is as follows:

16b2 Mlue opt=300

The number before "b" is the bit-width x in 2^{-x}. The number after "b" is the degree of Taylor expansion if we are synthesizing for "Poly" or the number subwords if we are synthesizing MLUE. "Mlue" indicates we are synthesizing MLUE, and when there "Poly" instead of Mlue, then we are synthesizing polynomial approximation (i.e., Taylor expansion). (Note that syn.pl automatically calls Mlue.pl or Poly_fitting.pl.) The number after opt= is the length of clock period (in ns).

postProcessing.pl: This script is used to process data in all of log files generated by syn.pl and summarize the results. It writes out input bit-width, polynomial order or number of subwords, area and timing.

5.2 FILE GENERATION

Scripts generate several files for simulation, synthesis, testing, and information. The files with .v extension are Verilog files, and the .txt files are pure ASCII.

wrapper.v: It includes the wrapper that instantiates RTL of the design (Mlue or Poly) to add flip-flops to the design's input and output pins so that we can constrain timing through a clock.

Mlue.v: The top-level of the MLUE design. It is instantiated in "wrapper" module.

PolyMult.v: The top-level of the Poly design. It is instantiated in "wrapper" module.

TB.v: It includes the test bench, which verifies the correctness of the design. It contains a driver and a checker. The driver feeds stimuli to the input pins of MLUE or Poly modules, and the checker checks if the output pins produce the correct values. We input 1,000 random values for x, compute the reference output using real data types and 2**(–x) function in Verilog, and make sure the design's output matches 2**(–x) within the precision x. (The output of 2**(–x) is expected to have the same precision as x.)

MuxSet.v: It is a module with a chain of MUX2s. There is one for each LUT. A MuxSet represents a LUT in terms of timing (i.e., access delay). They are instantiated in the wrapper, not in the design (Mlue or Poly). That is because their area are not equal to the LUTs they represent, and we do not want their area to be included in the synthesis area report, which we collect from the design (Mlue or Poly).

LutTotalArea.txt: Like we said above, the area of the LUTs cannot be obtained from logic synthesis (by Synopsys DC). If we had memory compilers, we could the area of LUTs either from the memory compiler or the synthesis software. In our case, we compute total LUT area using a very simple formula. The LUT area is equal to the total number of LUT bits times two thirds of the area of a NAND2 with a 1X drive.

6. SYNTHESIS RESULTS

In this chapter, the experimental setup and experimental results for polynomial approximation and MLUE are presented. We used Synopsys DC version 2000.05-1. The standard-cell synthesis library used is TSMC .18um.

We tried 14 different input bit-widths between 8 and 41. The output precision targeted is equal to the input precision with 2 guard-bits. For ex., for 8-bit input (with decimal point on the very left), the output is a 10-bit number (with decimal point on the very left), and the error is on the bit to the immediate right of the LSB.

Appendix A lists all our results. Out of those results, which try several degrees/part numbers for each bit-width, we took the best MLUE and Poly results and listed them in the summary table (Table 6.1) below.

Our target in this work is area optimization with our novel algorithm/hardware MLUE over the traditional Poly method. 10 different bit-widths out of 14, MLUE has smaller area than Poly. MLUE has an average of 10% less area than Poly, while the maximum savings is 21%.

Fig. 6.1 and 6.2 show how area varies over different number partitions (i.e., subwords) and degrees for MLUE and Poly, respectively. Initially the area goes down then goes up because after a certain point LUT area becomes negligible but data path area (arithmetic logic's area) keeps increasing.

In Fig. 6.1, the number of MLUE partitions that yields the minimum area is 5, while in Fig. 6.2, the degree of Poly polynomial that yields the minimum area is 4. That is why in Table 6.1 we have /4 and /5 in the 2nd and 3rd columns.

With the below results at hand, we believe we accomplished what we set out to do. That was to come up with a new novel algorithm and combinational hardware architecture for the computation of exponentiation that is more area efficient than prior art.

Table 6.1: Area results for both methods

Input Bit-width	Poly Area/Degree	MLUE Area/Subwords	MLUE Area Gain
8	11k/1	12k/2	-9 %
9	14k/1	15k/2	-7 %
10	21k/1	19k/2	10 %
11	24k/1	26k/2	-8 %
12	36k/2	35k/3	3 %
13	41k/2	39k/3	5 %
16	60k/2	63k/3	-5 %
22	15k/2	13k/4	15 %
23	17k/3	15k/4	13 %
32	40k/4	33k/5	21 %
36	50k/4	44k/6	14 %
37	60k/4	47k/6	14 %
40	67k/4	57k/7	18 %
41	71k/4	61k/7	16 %

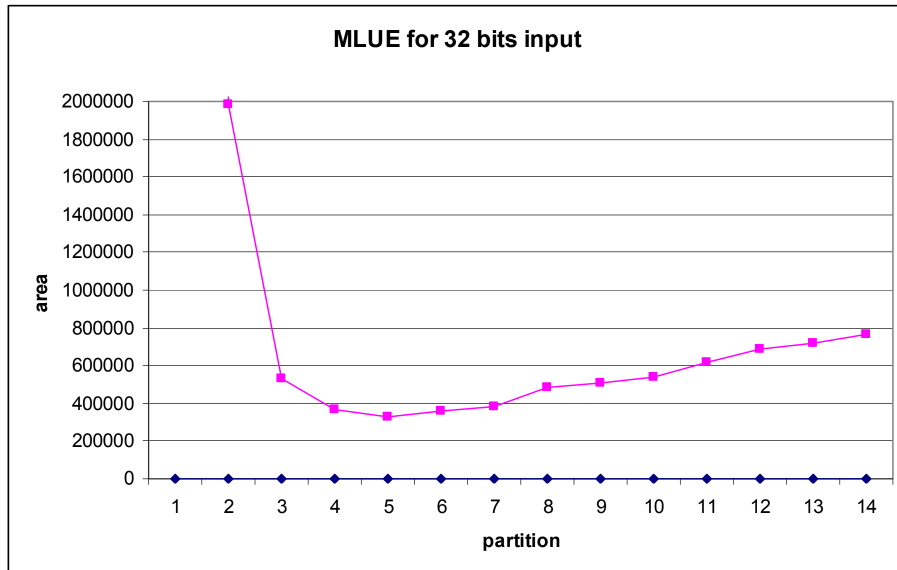

Figure 6.1: MLUE area for various numbers of partition for 32-bit input

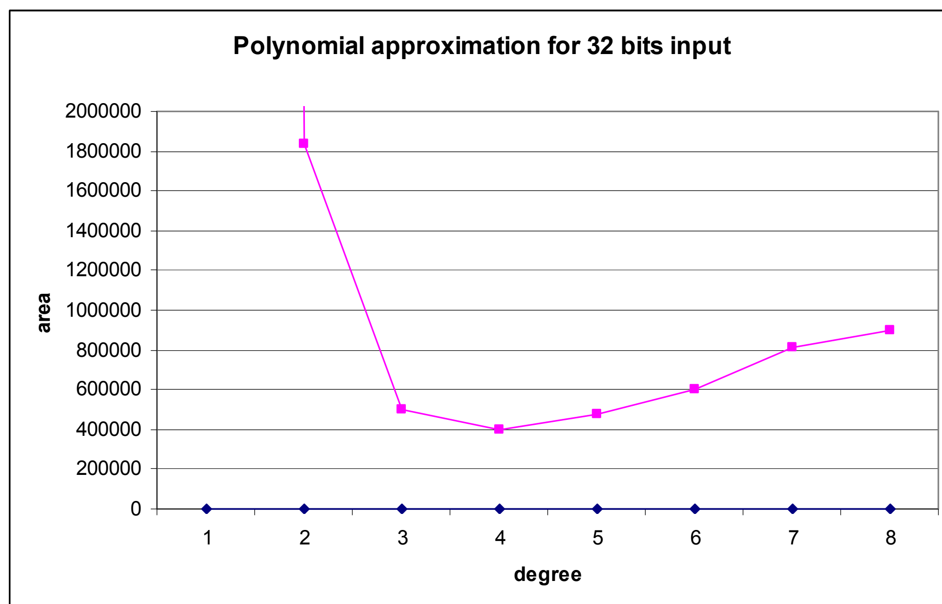

Figure 6.2: Poly area for various poly degrees for 32-bit input

7. CONCLUSIONS AND FUTURE WORK

This work presents a new hardware architecture that computes 2^{-x}. We call it MLUE for Multiple Look-Up table based Exponentiation. The interesting thing about this method is that it does not have any method error, also called approximation error. MLUE hardware architecture is a combinational and hence low-latency architecture. Its competition in the literature is "polynomial approximation" (Poly). We also implemented Poly. Our logic synthesis results show that MLUE offers implementations with smaller area compared to Poly, when timing is not pushed too much to extremely small latencies. The main advantage of MLUE with respect to Poly is in its smaller LUT sizes.

Example runs confirm that MLUE approach produces competitive designs for data widths (input and output) especially for more than 16 bits when compared with Poly. Some significant improvement is reported, up to 21% for circuit area reduction. The actual delays and area costs depend on the process technology used and on the actual memory compiler used. However, our memory area and timing models provide a good approximation to the actual LUT timings and area values.

In this work, our contributions are:

- A new method-error free and area efficient computation of 2^{-x}.
- Detailed and automated approximation and truncation error analysis.
- Automated exploration of the parameter space according to user specified criteria for both MLUE and Poly.
- Automated synthesis regressions.

Future work can be working on table compression techniques, applying the implemented Poly approach to different functions, evaluating MLUE and Poly on various FPGAs, optimizing and customizing guard-bits for each internal hardware signal, and using non-equal MLUE subwords even when we can use equal subwords.

REFERENCES

Brisebarre, N., Muller, J. M., and Tisserand, A., 2006. Computing Machine-Efficient Polynomial Approximations, *ACM Transactions on Mathematical Software,* Vol. 32, No. 2, pp. 236–256, June 2006.

Brisebarre, N. and Chevillard, S., 2007. Efficient polynomial $L\infty$-approximations, 18th IEEE Symposium on Computer Arithmetic (ARITH-18), pp. 169-176, June 2007.

Brubaker, T. A. and Becker, J. C., 1975. Multiplication Using Algorithms Implemented with Read-Only Memory, *IEEE Transactions on Computers*, Vol. C-24, No. 8, pp. 761-766, August 1975.

Defour, D., Dinechin, F. and Muller, J.M., 2002. A New Scheme for Table-based Evaluation of Functions, *Proc. 36th Asilomar Conf. Signals, Systems, and Computers,* November 2002.

Detrey, J. and Dinechin, F., 2004. Second Order Function Approximation Using a Single Multiplication on FPGAs, *14th International Conference on Field-Programmable Logic and Application,* pp. 221-230, 2004.

Detrey, J. and Dinechin, F., 2005. Table-based polynomials for fast hardware function evaluation, *IEEE Proceedings of the 16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)*, 2005.

Dinechin, D. and Detrey, J., 2002. Multipartite Tables in JBits for the Evaluation of Functions on FPGAs, *In Proceedings IEEE International Parallel and Distributed Symposium,* April 2002.

Dinechin, F. and Tisserand, A., 2001. Some Improvements on Multipartite Table Methods, *15th IEEE Symposium on Computer Arithmetic,* pp. 128-135, June 2001.

Hassler, H. and Takagi, N., 1995. Function Evaluation by table Look-up and Addition, *Proceedings of the12th IEEE Symposium on Computer Arithmetic*, pp. 10-16, July 1995.

Jamro, E. and Wiatr, K., 2007. FPGA Implementation of 64-Bit Exponential Function for HPC, *IEEE Proceedings of FPL 2007*, pp. 718-721, August 2007.

Kantabutra, V., 1996. On Hardware for Computing Exponential and Trigonometric Functions, *IEEE Transactions on Computers*, Vol. 45, No. 3, pp. 328-339, March 1996.

Lee, D. and Villasor, J. D., 2007. A Bit-Width Optimization Methodology for Polynomial-Based Function Evaluation. *IEEE Transactions on Computers*, Vol. 56, No. 4, pp. 567-571, April 2007.

Lee, D., Cheung, R. C. C., Luk, W., and Villasenor, J. D., 2008. Hardware Implementation Trade-Offs of Polynomial Approximations and Interpolations, *IEEE Transactions on Computers*, Vol. 57, No. 5, pp. 686-701, May 2008.

Michard, R., Tisserand, A., and Veyrat-Charvillon, N., 2005. Small FPGA polynomial approximations with 3-bit coefficients and low-precison estimations of the powers of x. *IEEE Proceedings of the 16th International Conference on Application-Specific Systems, Architecture and Processors (ASAP'05)*, 2005.

Paul, S., Jayakumar, N., and Khatri, S. P., 2009. A Fast Hardware Approach for Approximate, Efficient Logarithm and Antilogarithm Computations, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, Vol. 17, No. 2, February 2009.

Pineiro, J.-A., Ercegovac, M. D., and Bruguera, J. D., 2004. Algorithm and Architecture for Logarithm, Exponential, and Powering Computation, *IEEE Transactions on Computers*, Vol. 53, No. 9, pp. 1085-1096, September 2004.

Schulte, M. J. and Swartzlander, S. S., 1994. Hardware Designs for Exactly Rounded Elementary Functions, *IEEE Transactions on Computers*, Vol. 43, No. 8, pp. 964-973, August 1994.

Schulte, M. J. and Stine, J. E., 1997. Accurate Function Approximations by Symmetric Table Lookup and Addition, *In Proceedings ASAP '97 Proceedings of the IEEE International Conference on Application-Specific Systems, Architectures and Processors*, pp. 144-153.

Tak, P. and Tang, P, 1991. Table-Lookup Algorithms for Elementary Functions and Their Error Analysis, *Proceedings of the 10th Symposium on Computer Arithmetic*, pp. 232-236, 1991.

Walther, J. S., 1971. A Unified Algorithm for Elementary Functions, *Joint Computer Conference Proceedings AFIPS'71*, Vol. 38, pp. 379-385, Spring 1971.

Wong, W. F. and Goto, E., 1995. Fast Evaluation of the Elementary Functions in Single Precision, *IEEE Transactions on Computers* , Vol. 44, No. 3, March 1995.

APPENDIX A

Timing and area results are given in tables for both methods.

* is shown that the best timing for polynomial or MLUE

** is shown that the best result for that input bit number

Table A.1: Timing and area results for both methods (8-12 bits)

BitNumber	Partition/Degree	Method	Area	MeetTime	TargetTime	
8	1	Poly	10817,452	17,8	300	**
8	2	Poly	20044,886	22,98	300	
8	3	Poly	38383,33	33,26	300	
8	2	Mlue	11173,377	16,77	300	*
8	3	Mlue	20064,845	30,25	300	
8	4	Mlue	27266,501	30,01	300	
8	5	Mlue	46017,418	45,63	300	
-----	-----	-----	-----	-----	-----	-----
9	1	Poly	13917,657	19,13	300	**
9	2	Poly	24987,917	25,13	300	
9	3	Poly	45195,797	30,26	300	
9	4	Poly	56415,744	30,65	300	
9	2	Mlue	15165,057	18,27	300	*
9	3	Mlue	22506,422	31,62	300	
9	4	Mlue	31118,472	33,09	300	
9	5	Mlue	47211,595	52,55	300	
-----	-----	-----	-----	-----	-----	-----
10	1	Poly	21295,612	18,67	300	*
10	2	Poly	28756,728	26,07	300	
10	3	Poly	56655,245	32,85	300	
10	2	Mlue	19303,09933	19,36	300	**
10	3	Mlue	27269,82751	33,95	300	
10	4	Mlue	37555,05641	33,4	300	
10	5	Mlue	53022,81667	53,95	300	
-----	-----	-----	-----	-----	-----	-----
11	1	Poly	24488,95692	22,38	300	**
11	2	Poly	32738,42933	26,67	300	
11	3	Poly	60710,12733	32,73	300	
11	2	Mlue	26521,38734	21,29	300	*
11	3	Mlue	29901,00996	40,66	300	
11	4	Mlue	38912,22765	38,99	300	
11	5	Mlue	58571,25201	62,7	300	
-----	-----	-----	-----	-----	-----	-----
12	1	Poly	41473,55532	22,39	300	
12	2	Poly	35718,88357	30,96	300	*
12	3	Poly	62220,31302	32,73	300	
12	4	Poly	96342,52469	40,91	300	
12	2	Mlue	35306,40977	21,62	300	
12	3	Mlue	35286,4516	40,97	300	**
12	4	Mlue	43645,69491	39,34	300	

Table A.2: Timing and area results for both methods (13, 16, 22 bits)

BitNumber	Partition/Degree	Method	Area	MeetTime	TargetTime	
13	1	Poly	47597,45781	22,12	300	
13	2	Poly	40791,64374	29,64	300	*
13	3	Poly	67692,24096	43,13	300	
13	4	Poly	104728,3792	41,85	300	
13	2	Mlue	47903,48656	24,84	300	
13	3	Mlue	39411,1876	45,1	300	**
13	4	Mlue	47311,38775	45,16	300	
13	5	Mlue	70073,94331	62,52	300	
-----	-----	-----	-----	-----	-----	-----
16	1	Poly	176741,6114	27,81	300	
16	2	Poly	59529,25514	32,88	300	**
16	3	Poly	92869,76289	47,08	300	
16	4	Poly	144625,2214	47,56	300	
16	5	Poly	191440,9759	49,28	300	
16	2	Mlue	133831,0514	26,43	300	
16	3	Mlue	63188,29495	50,67	300	*
16	4	Mlue	64166,25669	50,17	300	
16	5	Mlue	90867,27005	70,58	300	
16	6	Mlue	106025,675	75,17	300	
16	7	Mlue	126902,1617	72,78	300	
-----	-----	-----	-----	-----	-----	-----
22	1	Poly	1705618,253	36,8	300	
22	2	Poly	146245,1771	42,79	300	*
22	3	Poly	154534,5668	52,72	300	
22	4	Poly	205591,4815	61,04	300	
22	5	Poly	288581,8368	60,55	300	
22	6	Poly	346630,8442	63,32	300	
22	2	Mlue	1250676,504	34,02	300	
22	3	Mlue	193094,1945	64,24	300	
22	4	Mlue	131266,398	62,16	300	**
22	5	Mlue	151231,4515	85,71	300	
22	6	Mlue	164460,5446	87,52	300	
22	7	Mlue	192212,7002	85,78	300	
22	8	Mlue	218241,7807	85,6	300	
22	9	Mlue	287021,7546	124,6	300	

Table A.3: Timing and area results for both methods (23, 32 bits)

BitNumber	Partition/Degree	Method	Area	MeetTime	TargetTime	
23	1	Poly	1800131,256	39,16	300	
23	2	Poly	230037,1932	42,59	300	
23	3	Poly	170085,487	54,93	300	*
23	4	Poly	219266,3121	62,84	300	
23	5	Poly	311171,4197	62,55	300	
23	6	Poly	374233,3117	65,42	300	
23	2	Mlue	1805260,565	34,75	300	
23	3	Mlue	244357,3451	67,17	300	
23	4	Mlue	148849,7485	60,64	300	**
23	5	Mlue	165016,0532	89,39	300	
23	6	Mlue	175081,7399	95,85	300	
23	7	Mlue	206602,707	92,95	300	
23	8	Mlue	228487,0928	95,81	300	
23	9	Mlue	302309,8893	122,2	300	
-----	-----	-----	-----	-----	-----	-----
32	1	Poly	73611309,34	45,06	300	
32	2	Poly	1838045,564	52,73	300	
32	3	Poly	502961,662	66,17	300	
32	4	Poly	400964,261	78,89	300	*
32	5	Poly	478788,718	79,57	300	
32	6	Poly	597976,958	81,86	300	
32	7	Poly	813540,987	88,69	300	
32	8	Poly	894891,427	103,88	300	
32	2	Mlue	54000887,37	42,15	300	
32	3	Mlue	1983655,398	81,23	300	
32	4	Mlue	527759,973	78,47	300	
32	5	Mlue	363545,585	118,06	300	
32	6	Mlue	329666,202	115,01	300	**
32	7	Mlue	358988,419	115,66	300	
32	8	Mlue	382416,255	119,64	300	
32	9	Mlue	485208,669	148,67	300	
32	10	Mlue	505572,89	149,97	300	
32	11	Mlue	536169,118	149,97	300	
32	12	Mlue	617359,89	156,63	300	
32	13	Mlue	689858,78	146,41	300	
32	14	Mlue	718043,367	151,35	300	
32	15	Mlue	764147,272	153,98	300	

Table A.4: Timing and area results for both methods (36, 37 bits)

BitNumber	Partition/Degree	Method	Area	MeetTime	TargetTime	
36	1	Poly	325731858,9	49,28	300	
36	2	Poly	3961120,365	55,76	300	
36	3	Poly	896478,109	71,98	300	
36	4	Poly	501953,766	86,5	300	*
36	5	Poly	626191,483	87,15	300	
36	6	Poly	701238,395	88,3	300	
36	7	Poly	960128,785	99,78	300	
36	8	Poly	1092393,105	113,21	300	
36	2	Mlue	239418484,9	50,31	300	
36	3	Mlue	5451949,643	82,37	300	
36	4	Mlue	1027268,829	83,53	300	
36	5	Mlue	543394,055	126,53	300	
36	6	Mlue	438163,391	121,85	300	**
36	7	Mlue	448678,143	129,77	300	
36	8	Mlue	488990,785	131,41	300	
36	9	Mlue	565321,687	173,02	300	
-----	-----	-----	-----	-----	-----	-----
37	1	Poly	337511103,7	56,76	300	
37	2	Poly	4120587,981	59,56	300	
37	3	Poly	924083,903	73,33	300	
37	4	Poly	595621,864	88,39	300	*
37	5	Poly	650218,071	89,09	300	
37	6	Poly	752138,969	89,05	300	
37	7	Poly	1006136,224	98,34	300	
37	8	Poly	1150375,584	114,21	300	
37	2	Mlue	337520520,7	50,52	300	
37	3	Mlue	6684387,496	90,25	300	
37	4	Mlue	1188256,61	90,72	300	
37	5	Mlue	607616,86	123,93	300	
37	6	Mlue	472884,354	126,34	300	**
37	7	Mlue	479011,585	123,62	300	
37	8	Mlue	510868,518	123,81	300	
37	9	Mlue	600218,95	165,06	300	

Table A.5: Timing and area results for both methods (40, 41 bits)

BitNumber	Partition/Degree	Method	Area	MeetTime	TargetTime	
40	1	Poly	1428379791	55,42	300	
40	2	Poly	8675154,736	62,6	300	
40	3	Poly	1711755,465	79,91	300	
40	4	Poly	671307,444	91,69	300	*
40	5	Poly	771245,809	95,08	300	
40	6	Poly	932669,353	96,04	300	
40	7	Poly	1121964,801	106,45	300	
40	8	Poly	1235411,675	120,18	300	
40	2	Mlue	1051692454	53,41	300	
40	3	Mlue	14119743,06	98,27	300	
40	4	Mlue	2073444,915	89,12	300	
40	5	Mlue	873695,596	133,29	300	
40	6	Mlue	606033,495	132,6	300	
40	7	Mlue	573388,206	131,05	300	**
40	8	Mlue	583783,207	132,26	300	
40	9	Mlue	700862,51	185,22	300	
-----	-----	-----	-----	-----	-----	-----
41	1	Poly	1475473212	57,36	300	
41	2	Poly	17320554,82	62,11	300	
41	3	Poly	1753268,937	78,8	300	
41	4	Poly	708037,553	93,34	300	*
41	5	Poly	806036,627	95,83	300	
41	6	Poly	982086,352	97,09	300	
41	7	Poly	1153123,19	106,78	300	
41	8	Poly	1289508,919	123,75	300	
41	2	Mlue	1475487113	52,23	300	
41	3	Mlue	18541120,75	91,51	300	
41	4	Mlue	2408736,056	91,9	300	
41	5	Mlue	959310,48	141,38	300	
41	6	Mlue	659312,444	150,09	300	
41	7	Mlue	607710,002	133,6	300	**
41	8	Mlue	622086,704	143,31	300	
41	9	Mlue	728588,054	176,9	300	

CURRICULUM VITAE

FULL NAME: Hatice Şahin

ADDRESS: Acıbadem Mah. Şehit Şükrü Sok. No: 5/15, Üsküdar / İSTANBUL

BIRTH PLACE / YEAR: Kayseri / 1986

LANGUAGE: Turkish (native), English

HIGH SCHOOL: Kayseri Sümer High School, 2004

BS: Electrical & Electronics Engineering, Erciyes University, 2008 (#2 in my class)

MS: Electrical & Electronics Engineering, Bahçeşehir University, 2011

Full support by both Bahçeşehir University (TA) & TÜBİTAK

NAME OF INSTITUTE: Natural and Applied Sciences

NAME OF PROGRAM: Embedded Video Systems - Chip Track

WORK EXPERIENCE: September 2008 – ongoing

Bahçeşehir University, Teaching Assistant