Methodologies for Approximation of Unary Functions and Their Implementation in Hardware

Erik Hertz
Methodologies for Approximation of Unary Functions and Their Implementation in Hardware
© Erik Hertz
Halmstad University Dissertations No. 21
Publisher: Halmstad University Press, 2016 | www.hh.se/hup
Printer: Media-Tryck, Lund

Cover:
Approximation technology from clay tablet to chip.
Applications in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics and many other areas have evolved to become very computation intensive. Algorithms are becoming increasingly complex and require higher accuracy in the computations. In addition, software solutions for these applications are in many cases not sufficient in terms of performance. A hardware implementation is therefore needed. A recurring bottleneck in the algorithms is the performance of the approximations of unary functions, such as trigonometric functions, logarithms and the square root, as well as binary functions such as division. The challenge is therefore to develop a methodology for the implementation of approximations of unary functions in hardware that can cope with the growing requirements. The methodology is required to result in fast execution time, low complexity basic operations that are simple to implement in hardware, and – since many applications are battery powered – low power consumption. To ensure appropriate performance of the entire computation in which the approximation is a part, the characteristics and distribution of the approximation error are also things that must be possible to manage.

The new approximation methodologies presented in this thesis are of the type that aims to reduce the sizes of the look-up tables by the use of auxiliary functions. They are founded on a synthesis of parabolic functions by multiplication – instead of addition, which is the most common. Three approximation methodologies have been developed; the two last being further developments of the first.

For some functions, such as roots, inverse and inverse roots, a straightforward solution with an approximation is not manageable. Since these functions are frequent in many computation intensive algorithms, it is necessary to find very efficient implementations of these functions. New methods for this are also presented in this thesis. They are all founded on working in a floating-point format, and, for the roots functions, a change of number base is also used. The transformations not only enable
simpler solutions but also increased accuracy, since the approximation algorithm is performed on a mantissa of limited range.

Tools for error analysis have been developed as well. The characteristics and distribution of the approximation error in the new methodologies are presented and compared with existing state-of-the-art methods such as CORDIC. The verification and evaluation of the solutions have to a large extent been made as comparative ASIC implementations with other approximation methods, separately or embedded in algorithms. As an example, an implementation of the logarithm made using the third methodology developed, Harmonized Parabolic Synthesis (HPS), is compared with an implementation using the CORDIC algorithm. Both implementations are designed to provide 15-bit resolution. The design implemented using HPS performs 12 times better than the CORDIC implementation in terms of throughput. In terms of energy consumption, the new methodology consumes 96% less. The chip area is 60% smaller than for the CORDIC algorithm. In summary, the new approximation methodologies presented are found to well meet the demanding requirements that exist in this area.
Matematikkens absolute nøjagtighed består i den principielle unøjagtighed at se bort fra unøjagtigheder.

Piet Hein
Acknowledgment

Many are the ancestors who paved the path before me and upon whose shoulders I stand. I also dedicate this in memory of my parents. A special thanks, I want to give to Anna and Rolf Abdon for their unreserved interest and support in my work.

To my primary supervisor, Professor Bertil Svensson, I wish to express my deepest gratitude for his unwavering support, excellent comments and not least for all the excellent support in the writing of this thesis.

Then I also want to show my sincere appreciation to my assistant supervisor, Professor Peter Nilsson, for making it possible for me to do my research and making it possible to continue my PhD studies. Without his unselfish support and guidance I would never have reached the point where I am today.

The creative injections and extensive explanations I got in the meetings with my assistant supervisor, Doctor Eric Järpe, I feel tremendous gratitude for.

To all students, I would like to express my deepest and sincere gratitude for all help in my research.

To my support committee, Doctor Bengt Lindoff, Professor Walid Taha, Doctor Stefan Byttner and my supervisors, I want to express my deepest gratitude for unusually constructive and inspiring meetings.

I would like to extend my thanks to Doctor Jörgen Carlsson, the leader of the Computing and Communication Laboratory (CC-lab), for allocating the complementary financing for my research towards the doctorate.

My greatest appreciation goes to the administrative staff, who have gone far beyond expectations in assisting me with numerous things in my research and many other things. Especially, am I very grateful for the excellent and foresighted help that I got in various ways from former Research Secretary Eva Nestius. In the same manner,
I want to thank Administrative Coordinator Pia Bruhn at the Department of Electrical and Information Technology at Lund University.

To Professor Emeritus Christer Svensson and Magnus Wahlin I want to give my unrestricted thanks for their sound advice leading me to continue towards my doctorate.

To Iren Wallin, Arbetsförmedlingen in Ängelholm, I stand in a huge debt of gratitude for making it possible for me to continue to my doctorate.

In addition to this, I would like to thank all colleagues at Halmstad and Lund Universities who have been involved in this process. I cannot mention you all by name, but I will always remember your patience and help. Thank you.

To all my friends, I thank you for your support and for never giving up on our friendship even though I have for long periods forsaken you.

In memory of my supervisor and friend Professor Peter Nilsson, who passed away in February 2016.

The first time Peter and I met was in the late 1980s. Peter was then a PhD student at the Department of Electrical and Information Technology at Lund University. At this time, the CAD tools for electronic design were in their early development. The department in Lund collaborated with UC Berkeley in the development, which Peter and I took part in. Over the years, Peter and I went to conferences together, of which the highlight for us was a conference in San Francisco with a visit to UC Berkeley. Peter was later to be a Technical Program Committee member for this conference, the IEEE International Solid State Circuits Conference. When I decided to pursue my PhD studies in 2004, I asked Peter if he could be my supervisor, which he said yes to. The collaboration between Peter and me was exceptionally good; Peter came up with what is needed and I found solutions to it. Peter had a funny way to trigger me by saying in an annoying way, “You cannot ...” and of course he tricked me to start trying to solve it.

I feel very honored to have had Peter as my supervisor and friend.

Halmstad, May 25, 2016
Erik Hertz
List of Publications


Other Publications


Preface

This thesis describes original work in the field of implementation of approximations of unary functions and algorithms in hardware. My interest for digital implementations of approximations and algorithms was founded with my Master thesis work in developing signal processing algorithms for a medical ultrasonic application. The interest was strengthened during my time at the Swedish Defence Research Agency because I saw an increasing digitalization in many industrial applications. It was also where I got the initial ideas to the work presented in this thesis. When I got the ideas I also understood that, because of some components needed in the methodology, I had to await the downscaling of silicon technology. With my move to Ericsson Research, where I worked with mobile phones, I increasingly came to realize that the foundation in future mobile communication systems is going to be digital signal processing. In these systems, the overall trend is moving towards higher complexity and speed of the calculations. The trend that many applications are battery powered adds requirements on low power consumption.

With these insights in my mind and since the technology development is starting to catch up with my ideas I started the further development. I presented the new ideas that I got to my supervisor, Professor Peter Nilsson at Lund University, and immediately we started to develop them. A complication, however, was that I had now moved on from Ericsson and worked with combustion engines for hyper sport cars. My research therefore had to be carried out in my spare time, after work.

It was a giant leap forward for me and my research when I, at Halmstad University, got the opportunity to focus only on my research. The focus promptly gave more ideas and insights about implementation of algorithms in hardware. A steady stream of Master thesis projects that were carried out at Lund University, under the guidance of Professor Peter Nilsson and me, enabled verification and further development of my ideas. The focus yielded that three research areas crystallized, approximation methodologies, dedicated algorithms of functions for implementa-
tion, and distribution of the error in the computations of the algorithms. The two first research areas aim to reduce chip area, computation time, and power consumption. The last research area aims for optimization of the computations in the algorithms which in itself also can reduce chip area, computation time, and power consumption.

My research will continue with developing algorithms using the approximation methodologies and dedicated algorithms of functions for implementation in different applications. Furthermore the tools for analyzing the error distribution will be further developed.
Contents

CHAPTER 1. Introduction................................................................. 1
  1.1. Motivation........................................................................... 1
  1.2. Problem formulation and objectives ................................... 2
  1.3. Approach ........................................................................... 3
  1.4. Research contributions ..................................................... 4
  1.5. Organization of the thesis.................................................. 5
  1.6. Summary of appended papers .......................................... 6

CHAPTER 2. Approximation Methods......................................... 11
  2.1. Look-up table ................................................................. 11
  2.1.1. Direct and indirect table look-up ................................... 11
  2.1.2. Look-up table reduction by auxiliary function ................. 12
  2.2. Interpolation .................................................................... 12
  2.3. Polynomial approximation ............................................... 13
  2.4. Piecewise approximation.................................................. 14
  2.5. Newton-Raphson method................................................. 15
  2.6. Weighted sum of bit-product approximation ....................... 15
  2.7. CORDIC ........................................................................... 16
  2.7.1. General concept ......................................................... 16
  2.8. Research situation for approximation methods.................. 18

CHAPTER 3. Parabolic Synthesis Methodologies ....................... 21
  3.1. Parabolic Synthesis .......................................................... 21
  3.1.1. First sub-function........................................................ 22
  3.1.1.1. Developing the first sub-function ............................... 24
  3.1.2. Second sub-function .................................................... 24
6.1.5. Root mean square ................................................................. 55
6.1.6. Decibel .................................................................................. 55
6.1.7. Decibel and binary numbers ................................................. 56
6.1.8. Distribution of the error ......................................................... 57

CHAPTER 7. Algorithms for roots, inverse and inverse roots ...... 59

7.1. Algorithms for roots, inverse and inverse roots ................. 59
7.1.1. Algorithms for computing roots ....................................... 60
7.1.1.1. Algorithm for computing the square root ................. 61
7.1.2. Algorithms for computing the inverse ......................... 63
7.1.3. Algorithms for computing inverse roots ...................... 64

CHAPTER 8. Conclusions ................................................................. 67

CHAPTER 9. Future Work ................................................................. 71

References 73
1 Introduction

This thesis presents and analyzes a new methodology for approximation of unary functions. The new methodology, called Parabolic Synthesis, is shown to be useful for all interesting functions used in engineering and science and suitable for hardware implementation. The functions that can be approximated include trigonometric functions, logarithms, square root and other roots, as well as the inverse function. Characteristic features of the methodology are efficient hardware implementation, high computation rate and low power consumption. In addition to this, the new technique also offers the possibility to tailor the approximation error to have favorable statistical characteristics.

1.1. Motivation

In the history of science, interpolation theory was first used in the context of mathematical astronomy. The rudimentary solutions of interpolation problems that have been revealed date back to early antiquity [1]. Examples of interpolation techniques originally conceived by ancient Babylonian as well as early medieval Chinese, Indian and Arabic astronomers and mathematicians can be linked to the classical interpolation techniques later developed in Western countries from the 17th until the 19th century. For the classical interpolation theory it is justified to say that there is no single person who did as much for this field as Isaac Newton [2]. Actually, Newton deserves the credit for having put classical interpolation theory on a solid foundation. Another important development from the late 1800s is the rise of approximation theory. In 1885, Weierstrass [3] justified the use of approximations by establishing the so-called approximation theorem, which states that every continuous function on a closed interval can be approximated uniformly to any prescribed accuracy by a polynomial.

With the scientific advancement and physics discoveries in the late 19th century and forward followed a need to compute complex algorithms with high speed. Approximations are an essential part in these algorithms, and, with the technological devel-
opments, demands have increased for both high speed and high precision in the calculations. The invention of the computer and its later miniaturization by means of transistors and integrated circuits have enabled an exponential development in areas such as baseband for wireless/wireline communication, computer graphics, digital signal processing, robotics, astrophysics, fluid physics and many other application areas. Until recently this development has been nearly unhindered due to the rapid development in semiconductor technology following Moore’s law [4]. The stagnation in this development began in the early 2000s [5], when the benefits of miniaturization progressively started to go down. Since the development in many applications needs higher performance, and since the semiconductor technology can no longer provide this in terms of higher clock speeds, interest in efficient implementations of algorithms has increased significantly. On top of this, a continuous sophistication of algorithms for implementation in digital hardware makes approximations of unary functions, e.g. trigonometric functions, logarithms and the square root, as well as binary functions, such as division, essential. The drive behind this is twofold. One is the continuous refinement of algorithms in the technology development in, e.g., image processing and telecommunications. The other is the mobility motivated development towards battery powered devices, which requires advanced, fast and power efficient algorithms.

1.2. Problem formulation and objectives

Developments in areas such as telecommunications are moving towards greater system complexity. The extensive complexity of the systems has an immense impact on the computation performance that is needed, both in speed and accuracy. Analysis of these algorithms reveals bottlenecks in the computation of approximations of unary functions. Software solutions are not sufficient for high speed applications as well as in low power systems, and a hardware implementation is therefore needed. In surveying existing approximation methods for hardware implementation, it is found that they have reached the limits of what they can achieve when it comes to speed combined with accuracy. For the performance issue, it has previously been possible to rely on technology development in terms of faster circuitry. With the end of the speed enhancements following Moore’s law, the focus with regard to improving performance is moved to the design and implementation of algorithms. The first objective of this study is therefore to develop new methodologies for constructing approximations of unary functions for implementation in digital hardware in such a way that it leads to small chip area, short critical path and low power consumption.

A related problem in the development of algorithms is to find efficient algorithms to implement specific unary functions. Algorithms that include matrices usually involve a massive amount of computations of a small number of functions. Efficient algorithms for computing specific unary functions are therefore essential in these
implementations. The second objective of this study is therefore to develop algorithms for implementation of specific unary functions, again with the requirement of small chip area, short critical path and low power consumption.

It is important to keep the data path in the algorithm as narrow as possible when implementing an algorithm. Due to a shorter word length in the computations, a narrow data path will indirectly reduce chip area, critical path delay and power consumption. A narrow data path is facilitated if the approximation error is evenly distributed and has a small standard deviation, since this will reduce the probability of an accumulation of large errors in the computation. The last objective of this study is therefore to investigate how the characteristics and distribution of the error affect the performance of an approximation or algorithm. This implies finding tools that can characterize the error. These tools are essential in the development and when comparing different approximation methodologies.

1.3. Approach

The new approximation methodologies for implementation of unary functions in hardware, that are presented in this thesis have all been developed with the objectives of achieving efficient hardware implementation, a high degree of parallelism, power efficient hardware architecture and adjustable error characteristics and distribution.

The architectures of the methodologies developed are founded on a synthesis of parabolic functions by multiplication instead of addition, which is the most common. Multiplication is used in the synthesis since it enhances the convergence compared to using addition. The efficiency of the hardware implementation of the methodologies is also supported by the fact that the only operations used are addition, shift and multiplication. To ensure a short critical path, a computation tree with few branches has been strived for in the architecture. A side effect of the area efficient and the short critical path is that it will lead to high power efficiency.

A goal in the development of the architecture of the methodologies is also the ability to tailor the characteristics and the distribution of the error. Advantageous characteristics and distribution of the error will not only be beneficial for the performance of the approximation but also for the following parts in the algorithm. Beneficial error performance will contribute to an efficient implementation of the approximation regarding chip area, short critical path and low power.

Three new methodologies for performing approximations of unary functions in hardware are presented in this thesis. The first is founded on a synthesis of parabolic functions in such a way that the accuracy of the approximation is given by the number of parabolic functions used. In the second and third methodologies the parabolic functions are replaced by one parabolic function and one second-degree inter-
polation. The accuracy for these methodologies is determined by choosing the number of intervals in the interpolation. The difference between the second and third methodologies lies in how the development is done. While each function is developed separately in the second methodology, the development in the third takes a more holistic approach. The approach of the second and third methodologies leads to a drastic reduction of chip area, computation time and power consumption compared to the first approach. Using second-degree interpolation also enables greater possibilities to modify the characteristics and the distribution of the error and to perform approximations on a wider range of unary functions.

In addition to the above, algorithms for computing roots, the inverse and inverse roots have been developed. Characteristic for these algorithms is that they all are founded on floating-point number representation. The advantage of using the floating-point number representation lies in its structure, based on the fact that the exponent is scaling the mantissa. The scaling makes it possible to perform the computation on a number with a very limited range instead of the full range of the number, as in the fixed point case. The transformations of the exponent when using these algorithms are very simple to perform. The conversion to and from floating-point numbers is also easily manageable, and therefore the expense of this is very low.

To evaluate the methodologies, several implementations of unary functions and algorithms on ASICs and FPGAs have been made. To evaluate the performance of the new methodologies, comparisons have been made with implementations using current state-of-the-art methodologies.

### 1.4. Research contributions

The main scientific contributions of the thesis are the following:

- The Parabolic Synthesis methodology, an approximation methodology for implementing unary functions in hardware, has been developed. The methodology is characterized by a high degree of parallelism in the architecture and that it is multiplicative and not additive as in the case of the majority of corresponding methodologies.

- In the Parabolic Synthesis methodology (and all its variations), the squaring function, including its partial products, is a frequently occurring operation. Therefore a special squaring architecture computing the square and its partial products has been developed.

- Combining Parabolic Synthesis and Second-Degree Interpolation is a further development of the Parabolic Synthesis methodology. The introduction of second-degree interpolation implies that increased accuracy can be reached at
lower cost in terms of the resulting chip area, critical path delay and energy consumption. The combined methodology also has more extensive opportunities to achieve desired characteristics and distribution of the error. The range of functions that can be approximated is also increased.

- The Harmonized Parabolic Synthesis methodology is a further development of the Parabolic Synthesis Combined with Second-Degree Interpolation. Characteristic of the Harmonized Parabolic Synthesis methodology is the holistic approach when developing a design. This approach will further improve performance by reducing chip area, computation delay and power consumption without harming the favorable characteristics and distribution of the error. Furthermore, it increases the possibility to tailor the characteristics and the distribution of the error and extends the number of unary functions that approximations can be performed upon.

- Algorithms have been developed for computing roots of the order of natural numbers from 2 and up. The algorithms are founded on using floating-point number representation and manipulation of the number base. The algorithms are characterized by a very high efficiency compared to existing methods.

- An algorithm for computing the inverse has been developed. It is founded on using floating-point number representation and is characterized by a very high efficiency compared to existing methods.

- Algorithms have been developed for computing inverse roots of the order of natural numbers from 2 and up. Like the above, they are founded on using floating-point number representation and manipulation of the number base and show a very high efficiency compared to current methods.

- An initial set of tools for analyzing the characteristics of the error of an algorithm implemented in hardware has been developed. A greater understanding has been gained of the problems associated with how the characteristics and the distribution of the error affect the computation and the different parameters of a hardware design.

- Finally, investigation and improvement of other approximation methods, such as polynomial approximation, the Newton-Raphson method and CORDIC, has also been made.

1.5. Organization of the thesis

The remaining part of this thesis is organized as follows: Chapter 2 briefly describes a selection of approximation methodologies, look-up table, interpolation, polynomial approximation, piecewise approximation, such as the Newton-Raphson method, weighted sum of bit-product approximation and CORDIC, followed by a
review of the research situation for approximation methods; Chapter 3 describes the new approximation methodologies: Parabolic Synthesis, Parabolic Synthesis Combined with Second-Degree Interpolation, and Harmonized Parabolic Synthesis; Chapter 4 describes the different architectures resulting from the different Parabolic Synthesis methodologies; Chapter 5 describes a special hardware architecture for computing the squaring result and all partial squaring results. Chapter 6 describes a set of tools for analyzing the characteristics of the error; Chapter 7 describes algorithms for implementing the functions roots, inverse and inverse roots; and the thesis closes with Chapter 8 with conclusions and Chapter 9 with future work. Nine papers are appended to the thesis; references to these for further details will appear throughout the thesis.

1.6. Summary of appended papers

**Paper A**

Summary
This paper introduces a parabolic synthesis methodology for developing approximations of unary functions, like trigonometric functions and logarithms, which are specialized for efficient hardware mapped VLSI design. The advantages with the methodology are, short critical path, fast computation and high throughput enabled by a high degree of architectural parallelism.

**Paper B**

Summary
This paper shows the feasibility of the Parabolic Synthesis methodology by presenting an efficient implementation of the sine function in hardware.

**Paper C**

Summary
This book chapter is a more detailed description of the Parabolic Synthesis methodology. The high degree of architectural parallelism of the methodology provides a short critical path that enables fast computation and high throughput. A squaring
operation occurs in every branch of the architecture. Since the products of squaring operations in the different branches are a partial product of the squaring of the incoming signal, a special squaring operation has been developed that computes all the partial products in one hardware block. This special squaring operation is described in detail in the book chapter. Last, detailed descriptions are given of implementations of some of the unary and binary functions for which the methodology is feasible.

**Paper D**

**Summary**
High performance implementations of unary functions are important in many applications, e.g., in the wireless communication area. This paper shows the development and VLSI implementation of unary functions such as the logarithmic and exponential function by using a novel approximation methodology based on parabolic synthesis, which in the paper is compared to the well known CORDIC algorithm. Both designs are synthesized and implemented on an FPGA and as an ASIC. The results of the implementations are compared with metrics such as performance and area. In a 65 nm Standard-$V_T$ ASIC implementation, the parabolic architecture is shown to exceed the CORDIC architecture in terms of speed by a factor of 4.2.

**Paper E**

**Summary**
This paper presents hardware implementations of the Taylor series. The focus is on the exponential function, but the methodology is applicable for any unary function. Two different architectures are investigated, one original, straightforward and one modified structure. The outcomes are higher performance, smaller area and lower power consumption for the modified architecture compared to the original.

**Paper F**
Summary
This paper shows a novel methodology to improve unrolled CORDIC architectures. The methodology is based on removing adder stages, starting from the first stage. As an example, a 19-stage CORDIC is used but the methodology is applicable to CORDICs with an arbitrary number of stages. The CORDIC is implemented, simulated and synthesized into hardware. In the paper, the performance is shown to be increased by 23% and the dynamic power to be reduced by 27%.

**Paper G**

Summary
In this paper, in order to further improve the performance of Parabolic Synthesis based designs, the methodology is combined with Second-Degree Interpolation. The paper shows that the methodology provides a significant reduction in chip area, computation delay and power consumption with preserved characteristics and distribution of the error. To evaluate this, the logarithm function has been implemented, as an example, using the Parabolic Synthesis methodology in comparison to the Parabolic Synthesis methodology combined with Second-Degree Interpolation. To further demonstrate the feasibility of both methodologies, they have been compared with the CORDIC methodology. The comparison is made on the implementation of the fractional part of the logarithm with a 15-bit resolution. The designs implemented using the Parabolic Synthesis methodology - with and without the Second-Degree Interpolation - perform 4x and 8x better, respectively, than the CORDIC implementation, both in terms of throughput. In terms of energy consumption, the CORDIC implementation consumes 140% and 800% more energy, respectively. The chip area is also smaller in the case when the Parabolic Synthesis methodology combined with Second-Degree Interpolation is used.

**Paper H**

Summary
The Harmonized Parabolic Synthesis methodology is a further development of the Parabolic Synthesis methodology for approximation of unary functions. Without harming the favorable characteristics and distribution of the error, it is possible to significantly enhance performance in terms of reducing chip area, computation delay and power consumption. Furthermore, it increases the possibility to tailor the characteristics and distribution of the error, which improves conditions for the sub-
sequent calculations. It also extends the number of unary functions that approximations can be performed upon since the possibilities to elaborate with the characteristics and the distribution of the error increase. To evaluate the proposed methodology, the fractional part of the logarithm has been implemented, and its performance is compared to the Parabolic Synthesis methodology. The comparison is made with 15-bit resolution. The design implemented using the Harmonized Parabolic Synthesis methodology performs 3x better than the original Parabolic Synthesis implementation in terms of throughput. In terms of energy consumption, the new methodology consumes 90% less energy. The chip area is 70% smaller than for the original Parabolic Synthesis methodology. In summary, the new technology presented in this paper further increases the advantages of the parabolic synthesis approach to approximation.

Paper I

Summary
For some functions, such as roots, inverse and inverse roots, a straightforward solution with an approximation is not manageable. Since these functions are frequent in many computation intensive algorithms, it is necessary to find very efficient implementations of them. In this paper, new methods for implementation of these functions are presented. The methods are all founded on working in a floating-point format, and, for the roots functions, a change of number base is also used. The transformations not only enable simpler solutions but also increased accuracy, since the approximation algorithm is performed on a mantissa of limited range. As a study of the usability of the methods, implementations of the inverse square root function using the Harmonized Parabolic Synthesis methodology and the Newton-Raphson method have been carried out.
Approximation Methods

An unavoidable problem when implementing signal and image processing algorithms in hardware is to find realizations of elementary function approximations in hardware. With the increasing digitalization, there is a growing need for hardware realization methods of each approximation. It is natural in view of the increasing relevance that the subject of developing hardware realization methods of approximations gets more and more attention. This chapter will give a brief overview of existing hardware realization methods of approximations of mathematically defined elementary functions.

2.1. Look-up table

Computation by look-up table is an attractive VLSI realization method since this simplifies the implementation of a logic function compared to a random logic implementation [6]. Also, with increasing memory density, multimegabit look-up tables can become more practical in some applications. A benefit of look-up tables is that they reduce the costs of hardware development for design, validation and testing. They also provide more flexibility for design changes and reduce the number of different building blocks or modules required when implementing arithmetic system designs.

When storing tables in read-only memories, the benefit is also improved reliability since memories are more robust than combinational logic circuits. Moving to read/write memories and reconfigurable peripheral logic will facilitate the evaluation of different functions and simplify the maintenance and repair of an implementation.

2.1.1. Direct and indirect table look-up

The straightforward usage of a look-up table is a direct table look-up. When given an $m$-variable function $f(x_m, x_{m-1}, \ldots, x_1, x_0)$, the evaluation of $f$ when the input values are $u$-bit and the desired result is $v$-bit, the required table size is $2^{m-u\cdot v}$ bits. The concatenated $u$-bit string is then used as an address into the table, with the read
out \( v \)-bit value from the table directly forwarded to the output. This arrangement gives a high degree of flexibility but is not practical in most cases. When implementing unary functions, the size of the table can be manageable when the input operand is up to 12 to 16 bits, which gives a table size of 4K to 64K words. To implement binary functions, such as \( xy \), \( x \mod y \) or \( x^y \), in look-up tables, each input operand has to be shortened to half of this size to be manageable. With increasing size, the logic latency of the memory will also be a problem for many applications.

To reduce the consequences of exponential growth of the table size, preprocessing steps of the operands and postprocessing steps [7] of the values’ read out from the tables can be introduced, resulting in an indirect table look-up. When implementing this hybrid scheme, the elements in the pre- and postprocessing parts are both simpler and faster and may also be more cost effective than either a pure table look-up approach or a pure logic circuit implementation based on an algorithm.

### 2.1.2. Look-up table reduction by auxiliary function

An approach to reduce the table size is to develop a binary function as an auxiliary unary function. With different complexity of the auxiliary function, the size of the look-up table can be reduced to different extents. These auxiliary functions are used in both the preprocessing, i.e., before table look-up, and following the table look-up [7]. Generally, in comparison to other approximation methods, table look-up based methods are faster since they need less computation in the approximation step. They can also be made more accurate and are more susceptible to shaping the error. The reduction of the table size for a given resolution can be rather significant and can also allow pipelining of the design, which increases the throughput.

### 2.2. Interpolation

A simple method to reduce the size of the look-up table is to use linear interpolation when implementing a function [6] [8]. When a function \( f(x) \) has a known value for \( x = x_{lo} \) and \( x = x_{hi} \), where \( x_{lo} < x_{hi} \), the values for \( x \) in the interval \([x_{lo}, x_{hi}]\) can be computed from \( f(x_{lo}) \) and \( f(x_{hi}) \) by interpolation. The simplest method to perform an interpolation is to use linear interpolations where \( f(x) \) for \( x \) in \([x_{lo}, x_{hi}]\) is computed according to (2-1).

\[
f(x) = f(x_{lo}) + (x - x_{lo}) \cdot \frac{[f(x_{hi}) - f(x_{lo})]}{x_{hi} - x_{lo}} \tag{2-1}
\]

When implementing linear interpolation in hardware, two look-up tables are needed, one for the starting point of the interpolation \( a = f(x_{lo}) \) and one for the direction coefficient \( b = [f(x_{lo}) - f(x_{hi})]/(x_{hi} - x_{lo}) \), as shown in Figure 2-1 for the initial linear
approximation. When choosing $x_{lo}$ and $x_{hi}$ it is beneficial to express them as numbers in the form of $2$ to the power of $z$, since the addressing of the look-up tables and computation of the linear interpolation can be simplified. This will imply that the addressing of the look-up tables will be in the most significant part of $x$ and the subtraction for $\Delta x = (x - x_{lo})$ will be in the least significant part of $x$. By this, the hardware realization of the approximation can be simplified to $a + b \cdot \Delta x$, where $a$ and $b$ are fetched from memories for the interval in which the computation is to be performed.

![Figure 2-1. Linear interpolation for computing $f(x)$.](image)

To minimize the absolute and the relative error in the worst case, an improved linear approximation strategy, as shown in Figure 2-1, can be chosen. As shown in Figure 2-1, the starting point $a$ and the direction coefficient $b$ are chosen to minimize the error of the interpolation in the interval.

### 2.3. Polynomial approximation

Since polynomials only involve additions, subtractions, multiplications and comparisons, it is natural to approximate elementary functions with polynomials. To achieve computation efficiency, the multiplier is the most crucial part of the implementation. It is therefore important to choose a fast multiplier for the efficiency of the computation of the polynomial approximation. A number of polynomial schemes are available for polynomial approximations. The polynomial scheme chosen affects the number of terms included for a given precision and thus the computational complexity.

When developing polynomial approximations, the challenge is to develop an efficient approximation that conforms to the function to be approximated in the desired interval. Two development strategies are available in developing an approximation,
one to minimize the average error, called least squares approximation, and one to minimize the worst case error, called least maximum approximation [9]. The strategy to choose depends on the requirements of the design. When the requirement of the design is that the approximation gets the best fitting to the function to be approximated, least squares approximations is favorable. An example of when least squares approximations are favorable is when the approximation is used in a series of computations. Least maximum approximations are favorable when it is important that the maximum error to the function to be approximated is to be kept small. An example when least maximum approximations are favorable is when the error from the approximation has to be within a limit from the function to be approximated.

A list of commonly known approximation schemes is given in Table 2-1. For additional information the reader can consult Muller’s book [9].

### 2.4. Piecewise approximation

Piecewise polynomial approximation [6] is a more flexible method for approximating a function \( f(x) \) since the interval on which to perform an approximation is divided into a number of subintervals, on which approximations are implemented as piecewise approximations by polynomials of a low degree. These piecewise polynomial approximations are called splines, and the endpoint of a subinterval is known as knots. An example of a linear spline approximation is shown in Figure 2-2.

The definition of a spline of degree \( n, n \geq 1 \), is a polynomial function of the degree \( n \) or less in each subinterval and has a prescribed degree of smoothness. The spline is expected to be continuous and also to have a continuous derivative of the order up to \( k, 0 \leq k < n \).

In real-time applications, performing elementary functions is often made in dedicated hardware, since software routines are too slow and computationally intensive [10]. When using piecewise approximation methods in hardware implementations, the degree of spline used is therefore often limited to first-order approximations, also called linear approximations.

<table>
<thead>
<tr>
<th>Taylor Polynomial</th>
<th>Maclaurin Polynomial</th>
<th>Legendre Polynomial</th>
<th>Chebyshev Polynomial</th>
<th>Jacobi Polynomial</th>
<th>Laguerre Polynomial</th>
</tr>
</thead>
</table>

Table 2-1. Approximation Schemes.
Figure 2-2. Example of a linear spline approximation.

Much effort is put into reducing the delay of the computation and the chip area when developing hardware design. It is therefore interesting to exclude multipliers. An example of this is shown in [11], where the multipliers have been replaced with configurable shifts.

2.5. Newton-Raphson method

The Newton-Raphson method [12] [13] is an iterative method that by iteration, \( n \), better approximations to the roots of a unary function \( f(x) = 0 \) are gradually found. The general Newton-Raphson method is given as the iteration shown in (2-2).

\[
x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}
\]

(2-2)

The method is initiated by guessing an initial value, \( x_0 \). The better the initial guess is, the fewer iterations are needed to compute the root. A look-up table with a suitable initial value of \( x_0 \) is often therefore used to reduce the number of iterations. A drawback of this method is its structure, shown in (2-2), which is only feasible for implementation in hardware for a limited number of functions.

2.6. Weighted sum of bit-product approximation

The approximation method of weighted sum of bit-products [14] can be beneficial for implementation of elementary functions in hardware since it can give an area efficient implementation with a high throughput and reasonable accuracy. A general formulation of the approach is that it starts with a given function, \( f(X) \), where \( X \) is composed of \( N \) bits, as shown in (2-3).
This gives the function $2^N$ function values, $f(X)$, where $0 \leq X \leq 2^N - 1$. The function’s values are then computed as a weighted sum of bit-products, as shown in (2-4).

$$f(X) = \sum_{i=0}^{2^N-1} c_j p_j$$

In (2-4), $c_j$ corresponds to the weight and $p_j$ to the bit-products, where each bit-product, $p_j$, is composed of the bits $x_j$ that are one in the binary representation of $j$.

### 2.7. CORDIC

The CORDIC (COordinate Rotation DIgital Calculation) algorithm is an iterative algorithm able to approximate a number of elementary functions using only simple mathematical operations such as additions, shifts and table look-ups. The algorithm is therefore very appealing for hardware implementation [15] [16]. It can be implemented in many ways, such as an iterative single stage method. Since the same hardware is used for all iterations, the implementations will be very efficient. The same hardware can also be used for computing many different functions and for this reason is ideal for applications where the emphasis is on reducing cost over speed. The iterative computation process results in that the computation time is drawn out, which is a drawback of the method.

The CORDIC algorithm has received a special status when implementing approximations of unary functions in hardware. This is backed up by the fact that CORDIC is the default method for approximation in most hardware development tools.

#### 2.7.1. General concept

The general concept of the CORDIC algorithm [17] [18] is vector rotation. Since vector rotation with an arbitrary angle is non-trivial, two methods are used to simplify the computation. The first method is to perform “pseudo-rotations” instead of performing rotations, since this is easier to compute. The second method is to construct the desired angle $\theta$ from the sum of special angles $\alpha_i$. Figure 2-3 shows rotations and pseudo-rotations of a vector of the length $R_i$, an angle of $\alpha_i$ from the origin.

$$x_{i+1} = x_i - y_i \cdot \tan(\alpha_i)$$
$$y_{i+1} = y_i + x_i \cdot \tan(\alpha_i)$$
$$\theta_{i+1} = \theta_i - \alpha_i$$

(2-5)
The co-ordinates following a pseudo-rotation are shown in (2-5).

A drawback of the algorithms in (2-5) is illustrated in Figure 2-3. The drawback is that, over a sequence of \( n \) pseudo-rotations, the vectors will grow by a factor of \( K \), given by (2-6).

\[
K = \prod_{i=0}^{n-1} \sqrt{1 + \tan^2(\alpha_i)}
\]

If the angles \( \alpha_i \) are always the same, then \( K \) is a constant and can be accounted for later. There are two criteria for choosing the set of angles \( \alpha_i \). The first criterion is that any angle can be constructed from a sum of all of them and with the appropriate signs. The second criterion is that every \( \tan(\alpha_i) \) is a power of 2, hence that the multiplication can be performed as a simple shift.

\[
\begin{align*}
\alpha_1 &= \tan\left(\frac{1}{2^1-1}\right) = 45^\circ \\
\alpha_2 &= \tan\left(\frac{1}{2^2-1}\right) = 26.57^\circ \\
\alpha_3 &= \tan\left(\frac{1}{2^3-1}\right) = 14.04^\circ \\
\alpha_4 &= \tan\left(\frac{1}{2^4-1}\right) = 7.13^\circ 
\end{align*}
\]
In (2-7) the first four rotation angles are shown to fulfill the two criteria. An example estimating a vector rotation of the given input angle, $\theta = 30^\circ$, is shown in Figure 2-4. The result of the estimation of the vector rotation in Figure 2-4 is shown in (2-8).

$$30 \approx 45.0 - 26.57 + 14.04 - 7.13 = 25.34$$  \hspace{1cm} (2-8)

More steps are used to improve the accuracy of the approximation.

### 2.8. Research situation for approximation methods

Broadly speaking, not one approximation method is generally best. The different methods have different advantages in terms of performing approximations of various functions or algorithms. Similarly, the various methods yield different performance at an implementation in hardware. This implies that choosing the most appropriate approximation method for a certain implementation can mean that you have to weigh together many different parameters. The fundamental hardware implementation parameters are chip area, critical path delay and power consumption. Essential parameters are also the characteristics and the distribution of the error, since these will affect both the hardware implementation and the calculations in the algorithm in which the approximation is included. Nominating one approximation method as the best must therefore be done with regard to the context in which it is a part.

Research on implementation of approximations of unary functions in hardware is mainly done on improving the performance of various existing methods. Examples
are the improvements described in [8] [19] [20] [21] as in [Paper E] and [Paper F] in this thesis. An exception is the Weighted Sum of Bit-Products methodology [14] presented 2005. In this methodology, the function is rewritten as a sum of weighted bit-products. This provides an efficient implementation for reasonable accuracies and will then also yield a high throughput.
3 Parabolic Synthesis Methodologies

The Parabolic Synthesis methodologies are specialized for implementing approximations of unary functions, such as trigonometric functions, logarithms and the square root, in hardware. A common and simple way to implement approximations of unary functions is the use of look-up tables. The drawback of the look-up tables is that, with increasing accuracy of the approximation, they become both chip space demanding and slow. Another approach when developing approximations of unary functions is to develop algorithms such as the CORDIC algorithm. The disadvantage of this approach is that, with increasing accuracy of the approximation, they tend to become increasingly more computationally intensive. By combining these two methods, the benefits of both methods can create effective solutions for approximations of unary functions with higher accuracy. The Parabolic Synthesis methodologies can be seen as a combination of these approaches. The Parabolic Synthesis methodologies will reduce the table size by developing functions as auxiliary functions to the look-up table. For a given resolution, the Parabolic Synthesis methodologies have proven to considerably reduce the table size. This has been accomplished while also obtaining small chip area, short computation time, low power consumption and the ability to tailor the characteristics and the distribution of the error.

3.1. Parabolic Synthesis

The Parabolic Synthesis methodology, [Papers A, C, and G], is founded on multiplications of sub-functions, $s_n(x)$, $n = 1, 2, \ldots$, each sub-function being a second-order parabolic function. The function to perform the approximation on is called the original function, $f_{org}(x)$. When multiplying an infinite number of these sub-functions, as shown in (3-1), the original function, $f_{org}(x)$, is obtained.

$$f_{org}(x) = s_1(x) \cdot s_2(x) \cdot \ldots \cdot s_\infty(x) \quad (3-1)$$
In reality, of course, the number of sub-functions is not infinite. Instead, the number of sub-functions used is related to the desired accuracy of the approximation. Note that, in the Parabolic Synthesis methodology the approximation is based on multiplication of factors, distinguishing it from many other methodologies that are based on summation of terms.

The first step in the methodology is to define the first sub-function, \( s_1(x) \). This is described in Section 3.1.1 below. The procedure used to obtain the following sequence of sub-functions includes the development of help functions, from which sub-functions are then developed. To compute the first help function, \( f_1(x) \), the ratio function between the original function, \( f_{\text{org}}(x) \), and the first sub-function, \( s_1(x) \), is computed. This division generates the first help function, \( f_1(x) \), as shown in (3-2).

\[
f_1(x) = \frac{f_{\text{org}}(x)}{s_1(x)} = s_2(x) \cdot s_3(x) \cdot \ldots \cdot s_\infty(x) \quad (3-2)
\]

In the same manner, the following help functions, \( f_n(x) \), are generated, as shown in (3-3).

\[
f_n(x) = \frac{f_{n-1}(x)}{s_n(x)} = s_{n+1}(x) \cdot s_{n+2}(x) \cdot \ldots \cdot s_\infty(x) \quad (3-3)
\]

In (3-4), the relationship for the calculation in (3-3) of the help functions, \( f_n(x) \), is clarified.

\[
f_{\text{org}}(x) = s_1(x) \cdot s_2(x) \cdot s_3(x) \cdot \ldots \cdot s_\infty(x)
\]

\[
f_1(x) = \frac{s_2(x) \cdot s_3(x) \cdot s_4(x) \cdot \ldots \cdot s_\infty(x)}{s_1(x) \cdot s_2(x) \cdot s_3(x) \cdot \ldots \cdot s_\infty(x)}
\]

\[
f_2(x) = \frac{s_3(x) \cdot s_4(x) \cdot s_5(x) \cdot \ldots \cdot s_\infty(x)}{s_2(x) \cdot s_3(x) \cdot s_4(x) \cdot \ldots \cdot s_\infty(x)}
\]

\[\vdots\]

3.1.1. First sub-function

The first sub-function, \( s_1(x) \), (3-1) can be looked upon as a rough estimate of the original function, \( f_{\text{org}}(x) \). It consists of a linear part, which is dependent on a constant together with a linear function, and a non-linear part, which is a parabolic function. When developing the first sub-function, \( s_1(x) \), the linear part, \( s_{1\_lin}(x) \), is first developed. The initial look of the linear part of the first sub-function, \( s_{1\_lin}(x) \), is shown in (3-5).

\[
s_{1\_lin}(x) = l_1 + k_1 x \quad (3-5)
\]
To reduce the hardware for realizing the first sub-function, coefficient $l_1$ in (3-5) is set to 0, which removes the addition in (3-5). Similarly, the coefficient $k_1$ in (3-5) is set to 1, which removes the multiplication. The linear part of the first sub-function is shown in (3-6).

$$s_{1\text{ lin}}(x) = x$$  \hspace{1cm} (3-6)

To obtain a simple implementation of the non-linear part as a parabolic function of the second degree, the range of the interval on the $x$-axis is restricted to $0 \leq x < 1$. Thus, the function to be approximated must be normalized and must satisfy the requirement that the values are in the interval $0 \leq x < 1.0$ on the $x$-axis and $0 \leq y < 1.0$ on the $y$-axis and have the starting point in $(0,0)$. The normalization of the function to be approximated creates the original function, $f_{\text{org}}(x)$, as shown in Figure 3-1.

![Figure 3-1. Normalization into the original function.](image)

This implies that the first sub-function, $s_1(x)$, can be expressed as in (3-7), where coefficient $c_1$ determines the height of the non-linear part.

$$s_1(x) = x + c_1(x-x)^2$$  \hspace{1cm} (3-7)

To satisfy the demands of the methodology, the original function, $f_{\text{org}}(x)$, has to fulfill three additional criteria.

1. The original function, $f_{\text{org}}(x)$, has to be strictly concave or convex through the interval since the approximation used is a second-order parabolic function.
2. The original function, $f_{\text{org}}(x)$, after being divided by the first sub-function, $s_1(x)$, must have a limit value when $x$ goes towards 0.
If the function has no limit value, it implies that a help function, \( f_1(x) \), can not be defined when \( x = 0 \).

3. The limit value in criterion 2 subtracted with 1 must be smaller than or equal to 1 or larger than or equal to -1.

If the limit value in criterion 2 is outside this interval, the gradient of the first sub-function, \( s_1(x) \), will not be positive through the entire interval and therefore not deployable as an approximation of the original function. Fortunately, for almost all interesting functions, criterion 3 is fulfilled. Examples of original functions that do not fulfill criterion 3 are higher order roots than cube roots.

### 3.1.1.1. Developing the first sub-function

The first sub-function, \( s_1(x) \), is a second-order parabolic function as shown in (3-7). When developing \( s_1(x) \), the criterion for the second sub-function, \( s_2(x) \), has to be taken into account. The criterion for \( s_2(x) \) is that it shall start in \((0,1)\) and end in \((1,1)\). Based on the start criterion, the coefficient \( c_1 \) in (3-7) can be computed from the limit in (3-8).

\[
1 = \lim_{x \to 0} \frac{f_{\text{org}}(x)}{x + c_1(x - x^2)} \tag{3-8}
\]

The limit in (3-8) is rewritten according to (3-9) since the \( x^2 \) term in (3-8) goes faster towards 0 than \( x \) and the \( x^2 \) term can therefore be excluded, as shown in (3-9).

\[
1 = \lim_{x \to 0} \frac{f_{\text{org}}(x)}{(1 + c_1)x} \tag{3-9}
\]

The limit in (3-9) is rewritten according to (3-10), and the coefficient \( c_1 \) can be calculated from this.

\[
c_1 = \lim_{x \to 0} \frac{f_{\text{org}}(x)}{x} - 1 \tag{3-10}
\]

### 3.1.2. Second sub-function

The second sub-function, \( s_2(x) \), is developed as an approximation of the first help function, \( f_1(x) \). The computing of \( f_1(x) \) is performed according to (3-2) and the result of this operation is a concave or convex function. The appearance of this function is similar to a parabolic function, as illustrated in Figure 3-2 or its mirror on the x-axis. As an approximation of the first help function, \( f_1(x) \), the second sub-function, \( s_2(x) \), is chosen as a second-order function [6] for which the start value, \( l_{2,\text{start}} \), is 1 and
the end value, $l_{2,\text{end}}$, is 1, as shown in Figure 3-2. Since the interval on the $x$-axis is normalized to 1, as shown in Figure 3-2, the denominator when computing the gradient, $k_2$, is 1. Gradient $k_2$ is computed as $k_2 = l_{2,\text{end}} - l_{2,\text{start}}$. From Figure 3-2 it is found that both $l_{2,\text{end}}$ and $l_{2,\text{start}}$ are 1, which leads to the gradient $k_2$ being equal to 0. This implies that the second-order function can be reduced according to (3-11).

$$s_2(x) = l_{2,\text{start}} + k_2 x + c_2 (x - x^2) = 1 + c_2 (x - x^2)$$  \hspace{1cm} (3-11)

In (3-11) the coefficient $c_2$ is chosen so that the quotient between the first help function, $f_1(x)$, and the second sub-function, $s_2(x)$, is equal to 1 for $x = 0.5$, according to (3-12), which is also shown in Figure 3-2.

$$c_2 = 4(f_1(0.5) - 1)$$  \hspace{1cm} (3-12)

\[0.00 0.25 0.50 0.75 1.00 \quad f_1(x), s_2(x)\]

Figure 3-2. Example of the first help function, $f_1(x)$, compared with the second sub-function, $s_2(x)$.

### 3.1.3. Sub-functions, $s_n(x)$, when $n > 2$

When calculating the second help function, $f_2(x)$, according to (3-3), the result is a pair of opposite, strictly concave or convex functions, as shown in Figure 3-3. The third sub-function, $s_3(x)$, is an approximation of the second help function, $f_2(x)$. It has two parts; the first part is restricted to the interval $0 \leq x < 0.5$ and the second part is thus restricted to the interval $0.5 \leq x < 1.0$. When the third sub-function, $s_3(x)$, is developed, it is divided into two partial functions, one for each interval.
A general approach when developing help functions, \( f_n(x) \), when \( n > 1 \), is, in each interval, to develop a strictly concave or strictly convex function as a partial help function. When developing the higher-order sub-functions, \( s_n(x) \), when \( n > 2 \), each partial strictly concave or convex function is divided into two separate parabolic functions. A parabolic sub-function is developed for each sub-interval as an approximation of the help function, \( f_n(x) \), in the sub-interval. To show the subinterval for which the partial functions are valid, the subscript index is increased with the index \( m \), which gives the partial help function, \( f_{n,m}(x) \), according to (3-13). In equation (3-13) it is shown how the help function, \( f_n(x) \), is divided into partial help functions, \( f_{n,m}(x) \), when \( n > 1 \).

\[
f_n(x) = \begin{cases} 
  f_{n,0}(x), & 0 \leq x < \frac{1}{2^{n-1}} \\
  f_{n,1}(x), & \frac{1}{2^{n-1}} \leq x < \frac{2}{2^{n-1}} \\
  \vdots & \\
  f_{n,2^{n-1}-1}(x), & \frac{2^{n-1}-1}{2^{n-1}} \leq x < 1 
\end{cases} \tag{3-13}
\]

As shown in (3-13), the number of partial help functions is doubled for each order of \( n > 2 \), i.e., the number of partial functions is \( 2^{n-1} \).
The corresponding sub-functions are developed from these partial functions. Analogous to the help function, \( f_n(x) \), the sub-function, \( s_{n+1,m}(x) \), will also have partial sub-functions, \( s_{n+1,m}(x) \). Equation (3-14) shows how the sub-function, \( s_n(x) \), is divided into partial functions when \( n > 2 \).

\[
s_n(x) = \begin{cases} 
  s_{n,0}(x_n), & 0 \leq x < \frac{1}{2^{n-2}} \\
  s_{n,1}(x_n), & \frac{1}{2^{n-2}} \leq x < \frac{2}{2^{n-2}} \\
  \ldots \\
  s_{n,2^{n-1}-2}(x_n), & \frac{2^{n-2}-1}{2^{n-2}} \leq x < 1 
\end{cases} \tag{3-14}
\]

Note that, in (3-14), input variable \( x \) has been changed to \( x_n \). The change to \( x_n \) is a normalization to the corresponding interval, which simplifies the hardware implementation of the parabolic function.

To simplify the normalization of the interval of \( x_n \), it is chosen as an exponentiation by 2 of \( x \) where the integer part is removed. The normalization of \( x \) is therefore done by multiplying \( x \) with \( 2^{n-2} \), which in hardware is \( n - 2 \) left shifts. The integer part is thus dropped, which gives \( x_n \) as a fractional part of \( x \), as shown in (3-15).

\[
x_n = \text{frac}(2^{n-2}x) \tag{3-15}
\]

The dropped integer part from the normalization is used to code the interval in which the sub-function is performed, which is equal to the index \( m \) in the sub-function.

As in the second sub-function, shown in (3-11), the second-order function is used as an approximation in each interval of a partial help function, \( f_{n-1,m}(x) \). The start value for each partial help function is 1, which implies that the constant term, \( l_{n,m} \), in the sub-function (3-16) is 1. The end value in each partial help function’s interval is also 1, which corresponds to the gradient, \( k_{n,m} \), being 0 for each interval in the sub-function, \( s_{n,m}(x) \), (3-16). The range of \( x_n \) in each interval of the partial help function, \( f_{n-1,m}(x) \), is 1, which allows the sub-functions, \( s_{n,m}(x) \), to be reduced according to (3-16).

\[
s_{n,m}(x_n) = l_{n,m} + k_{n,m}x_n + c_{n,m}(x_n - x_n^2) = 1 + c_{n,m}(x_n - x_n^2) \tag{3-16}
\]

In (3-16) the coefficients, \( c_{n,m} \), are chosen so that the quotients between the help functions, \( f_{n-1,m}(x) \), and the sub-sub-functions, \( s_{n,m}(x) \), are equal to 1 when \( x_n \) is equal to 0.5. This is given by (3-17).
3.1.4. Summary

The Parabolic Synthesis methodology is founded on multiplications of sub-functions, \( s_n(x) \), \( n = 1, 2, \ldots \), each sub-function being a second-order parabolic function. The first sub-function, \( s_1(x) \), can be viewed as a rough estimate of the original function, \( f_{\text{org}}(x) \), and the number of remaining sub-functions relates to the desired accuracy of the approximation. Note that in the Parabolic Synthesis methodology the approximation is based on multiplication of factors as shown in (3-1). Figure 3-4 is a visualization that describes the behavior of the sub-functions in an implementation with four sub-functions.

\[
c_{n,m} = 4 \left( f_{n-(1,m)} \left( \frac{2(m+1) - 1}{2^n - 1} \right) - 1 \right) \tag{3-17}
\]

Figure 3-4. Visualization of the behavior of four sub-functions in an implementation.

The graphs in Figure 3-4 show the behavior of the four sub-functions and the resulting approximation. Note that the range in the graphs on the \( x \)-axis is the same for all graphs. The heights of the graphs have no mutual relevance, however, and should only be seen as an illustration of the functions’ behaviors. From the graphs of the sub-functions in Figure 3-4 it can be seen that the number of periods of the function in the graphs is doubled with an increasing index on the sub-functions, as shown in (3-7), (3-11) and (3-14). In the graph of the first sub-function, \( s_1(x) \), the length of the interval \([0,1]\) is a quarter of a period, whereas in the graph of the fourth sub-function, \( s_4(x) \), the length of the interval is two periods. The heights of the sub-functions drop
significantly with an increasing sub-function index. The drop in height between the sub-functions is so large that their behavior would not have been visible using the same scale factor as of the first sub-functions.

### 3.2. Parabola Synthesis Combined with Second-Degree Interpolation

A consequence of the Parabolic Synthesis methodology is that, to increase the accuracy of the approximation, the number of sub-functions must be increased, which will increase the size of the hardware. A new methodology has been developed to mitigate this. By combining Parabolic Synthesis and Second-Degree Interpolation [Paper G] the structure is similar to the Parabolic Synthesis methodology in that a synthesis of sub-functions is done (3-1). However, never more than two sub-functions are required (3-18). The sub-function developed with Parabolic Synthesis gives a rough approximation, while the approximation is enhanced by increasing the number of intervals in the sub-function developed with Second-Degree Interpolation.

\[ f_{\text{org}}(x) = s_1(x) \cdot s_2(x) \]  

Other benefits are:

- The Second-Degree Interpolation part relaxes criteria 1 and 3 described in Section 1.1.1. These are demands on the original function, \( f_{\text{org}}(x) \), that must be fulfilled to enable an approximation. However, it is still an advantage if the criteria are fulfilled since this will imply a more efficient implementation.
- The reduction to two sub-functions enables a reduction in hardware, and with that follows a reduction of the power consumption.
- With the combination of the methodologies, a reduction of the length of the critical path can be foreseen.
- An advantage of using the Second-Degree Interpolation is that a rough internal error compensation can be accomplished to improve the characteristics and the distribution of the error.

#### 3.2.1. Second sub-function

As in the Parabolic Synthesis methodology, the second sub-function, \( s_2(x) \), is developed as Second-Degree Interpolations, as shown in (3-19) and Figure 3-4.

\[ s_2(x) = l_2, i + k_2, i x_w + c_2, i (x_w - x_w^2) \]  

(3-19)
To simplify the coding of the interval numbers, \( i \), in hardware, the number of equal range intervals in the second sub-function, \( I \), is chosen as 2 to the power of \( w \), where \( w \) is a natural number (3-20).

\[
I = 2^w
\]  

(3-20)

To simplify the normalization of the interval of \( x_w \), the interval is selected as an exponentiation by 2 of \( x \) where the integer part is removed. The normalization of \( x \) is therefore done by multiplying \( x \) with \( 2^w \), which in hardware is \( w \) left shifts. Hereafter, the integer part is dropped, which gives \( x_w \) as a fractional part of \( x \), as shown in (3-21).

\[
x_w = \text{frac}(2^w x)
\]  

(3-21)

This truncation performs normalization to the interval, as shown in Figure 3-4. The dropped integer part from the normalization is used to code the number of the interval in which the second sub-function is performed, which is therefore synonymous with the index \( i \) in the sub-function, as shown in Figure 3-5.

![Figure 3-5. Description of the development of the Second Degree Interpolation.](image)

The index \( i \) is defined as the number of the interval, starting with 0 and ending with \( I - 1 \). In (3-19), \( l_{2,i} \) is the starting point of an interval of the interpolation. This is computed by inserting the starting point value of the interval, \( x_{\text{start},i} \), in the help function, \( f_1(x) \), (3-2), as shown in (3-22) and Figure 3-5.

\[
l_{2,i} = f_1(x_{\text{start},i})
\]  

(3-22)
In (3-19), \( k_{2,i} \) is the gradient for an interpolation interval. The gradient \( k_{2,i} \) for an interval is computed as the end point value of the help function, \( f_1(x_{end,i}) \), subtracted with the start point value of the help function, \( f_1(x_{start,i}) \). Since the interval is normalized to 1, this excludes the denominator, as shown in (3-23) and Figure 3-5.

\[
\begin{align*}
  k_{2,i} &= f_1(x_{end,i}) - f_1(x_{start,i}) \quad (3-23)
\end{align*}
\]

In (3-19), \( c_{2,i} \) is pre-computed so that the second sub-function for an interval, \( s_{2,i}(x_w) \), cuts the help function, \( f_1(x) \), in the middle of interval \( i \) when \( x_w = 0.5 \), which satisfies the point \( x_{middle,i} \) for the help function, \( f_1(x) \), as shown in (3-24) and Figure 3-5.

\[
\begin{align*}
  c_{2,i} &= 4\left(f_1(x_{middle,i}) - l_{2,i} - \frac{k_{2,i}}{2}\right) \quad (3-24)
\end{align*}
\]

The sub-function in (3-18) can be simplified according to (3-25).

\[
\begin{align*}
  s_{2,i}(x) &= l_{2,i} + j_{2,i} x_w - c_{2,i} x_w^2 
\end{align*}
\]

In (3-25), \( j_i \) is pre-determined according to (3-26).

\[
\begin{align*}
  j_{2,i} &= k_{2,i} + c_{2,i} \quad (3-26)
\end{align*}
\]

Equation (3-27) shows how the sub-function, \( s_2(x) \), is divided into partial functions.

\[
\begin{align*}
  s_2(x) &= \begin{cases} 
    s_{2,0}(x_n), & 0 \leq x < \frac{1}{2w} \\
    s_{2,1}(x_n), & \frac{1}{2w} \leq x < \frac{2}{2w} \\
    \vdots \\
    s_{2,L-1}(x_n), & \frac{L-1}{2w} \leq x < 1 
  \end{cases} 
\end{align*}
\]

(3-27)

Note that in (3-27), \( x \) has been changed to \( x_w \); the change to \( x_w \) is made because the intervals for the sub-sub-functions, \( s_{2,i}(x) \), in (3-27) have equal ranges.

**3.3. Harmonized Parabolic Synthesis**

The Harmonized Parabolic Synthesis methodology described in **Paper H** is a further evolution of the Parabolic Synthesis Combined with Second-Degree Interpolation methodology. The improvement compared to the previous methodology lies in the development of the algorithm for approximation. As for Parabolic Synthesis
Combined with Second-Degree Interpolation, it is founded on the product of the two sub-functions in (3-28).

\[ f_{\text{org}}(x) = s_1(x) \cdot s_2(x) \quad (3-28) \]

In (3-28) the first factor is called the first sub-function, \( s_1(x) \), and is derived from the Parabolic Synthesis methodology described in Section 3.1.1. The second factor in (3-28) is called the second sub-function, \( s_2(x) \), and is a second-degree interpolation derived from the Parabolic Synthesis Combined with Second-Degree Interpolation methodology.

The essential difference between Harmonized Parabolic Synthesis and Parabolic Synthesis Combined with Second-Degree Interpolation is in the development of the sub-functions. In the previous methodology, the two sub-functions are developed separately, whereas in the Harmonized Parabolic Synthesis methodology the sub-functions are developed as one unit. The intention with this design strategy is to have a holistic view of the design process and thereby improve performance such as chip area, critical path delay and power consumption. This design strategy results in that the range of unary functions suitable for the approximation methodology is extended even further. This is because the development methodology allows a higher degree of flexibility and adaptability in the design process.

### 3.3.1. The sub-functions

The first sub-function, \( s_1(x) \), is a second-order parabolic function, which in conjunction with the second sub-function, \( s_2(x) \), results in an approximation of the original function, \( f_{\text{org}}(x) \). The second sub-function is a second-degree interpolation \([6] \quad [22] \quad [23]\) specifically shown in Paper H, which is the reason for the higher degree of flexibility and adaptability in finding an optimum for the design. The number of intervals in the second sub-function determines the final accuracy of the approximation and allows the characteristics and the distribution of the error to be tailored. When developing an approximation with the proposed methodology, an empirical design methodology is the only feasible approach because the two sub-functions are designed as a complete unity to fulfill the design criteria. This design process makes it possible to a higher extent to weigh various design parameters against each other and thereby achieve a design that is closer to optimal.

### 3.3.1.1. Developing the first sub-function

The first sub-function, \( s_1(x) \), is a second-order parabolic function as shown in (3-29).

\[ s_1(x) = l_1 + k_1x + c_1(x - x^2) \quad (3-29) \]
It can be seen that the two terms on the left form a linear equation with a constant $l_1$ and a gradient $k_1$, while the rightmost term is the nonlinear part. From Figure 3-6 it is then found that the starting point for the first sub-function, $s_1(x)$, is in $(0,0)$. This gives the start value, $l_1$, to be 0. Furthermore, the gradient, $k_1$, is 1 since the gradient starts in $(0,0)$ and ends in $(1,1)$ and the range of $x$ therefore is 1. The first sub-function, $s_1(x)$, can therefore be rewritten according to (3-30).

$$s_1(x) = x + c_1(x - x^2)$$

(3-30)

### 3.3.1.2. The second sub-function

![Figure 3-6. Description of the development of the second-degree interpolation.](image)

The second sub-function, $s_2(x)$, is based on splitting the function in intervals to perform an interpolation. The procedure when developing the second sub-function is to perform a division of the original function, $f_{org}(x)$, with the first sub-function, $s_1(x)$. This division generates the help function, $f_{help}(x)$, as shown in (3-31).

$$f_{help}(x) = \frac{f_{org}(x)}{s_1(x)}$$

(3-31)

From the help function, $f_{help}(x)$, the second sub-function, $s_2(x)$, is developed as a Second-Degree Interpolation, as shown in (3-32) and Figure 3-6, where the number of intervals in the interpolation decides the order of the accuracy of the approximation.
To simplify the determination of the interval, \( i \), in hardware, the number of equal-range intervals in the second sub-function, \( I \), is chosen as 2 to the power of \( w \), where \( w \) is a natural number as shown in (3-33).

\[
I = 2^w
\]  

(3-33)

To simplify the normalization of the interval of \( x_w \), it is selected as an exponentiation by 2 of \( x \) where the integer part is removed. The normalization of \( x \) is therefore made by multiplying \( x \) with \( 2^w \), which in hardware is implemented as \( w \) left shifts. Furthermore, the integer part is dropped, which gives \( x_w \) as a fractional part of \( x \), as shown in (3-34).

\[
x_w = \frac{1}{2^w}x
\]  

(3-34)

This truncation performs normalization to the interval, as shown in Figure 3-6. The dropped integer part from the normalization is used to code the number of the interval in which the second sub-function is performed and is therefore synonymous with the index \( i \) in the sub-function, as shown in Figure 3-6. The index \( i \) is defined as the number of the interval starting with 0 and ending with \( I - 1 \). In (3-32), \( l_{2,i} \) is the starting point of an interval of the interpolation. This is computed by inserting the value of \( x \) for the starting point of the interval \( x_{start,i} \) of the help function, \( f_{help}(x) \), (3-31), as shown in (3-35) and Figure 3-6.

\[
l_{2,i} = f_{help}(x_{start,i})
\]  

(3-35)

Equation (3-35) does not apply to the start value of the first interval, which has to be calculated as the limit according to (3-36). Since the \( x^2 \) term in (3-36) goes faster towards 0 than the \( x \) term, it can be excluded when calculating the limit, as shown in (3-36).

\[
l_{2,0} = \lim_{x \to 0} \frac{f_{org}(x)}{x + c_1(x - x^2)} = \lim_{x \to 0} \frac{f_{org}(x)}{1 + c_1 x}
\]  

(3-36)

In (3-32), \( k_{2,i} \) is the gradient for an interpolation interval \( i \). The gradient \( k_{2,i} \) for an interval is computed as the end point value of the help function, \( f_{help}(x_{end,i}) \), subtracted with the start point value of the help function, \( f_{help}(x_{start,i}) \). Since the interval is normalized to 1 the denominator when computing the gradient, \( k_{2,i} \), is set to 1 and therefore no division is needed, as shown in (3-37) and Figure 3-6.

\[
k_{2,i}(x) = f_{help}(x_{end,i}) - f_{help}(x_{start,i})
\]  

(3-37)
In (3-32) the coefficient, $c_{2,i}$, in an interval $i$ of the second sub-function, $s_2(x)$, is pre-computed so that the second sub-function in an interval, $s_{2,i}(x_w)$, cuts the help function, $f_{\text{help}}(x)$, in the middle of the interval when $x_w = 0.5$. This satisfies the point $x_{\text{middle},i}$ for the help function, $f_{\text{help}}(x)$, as shown in (3-38) and Figure 3-6.

$$c_{2,i} = 4 \left( f_{\text{help}}(x_{\text{middle},i}) - l_{2,i} \right) - \frac{k_{2,i}}{2} \quad (3-38)$$

The sub-function in (3-31) can be simplified according to (3-39).

$$s_{2,i}(x) = l_{2,i} + c_{2,i} x_w - c_{2,i} x_w^2 \quad (3-39)$$

In (3-39), $j_{2,i}$ is pre-determined according to (3-40).

$$j_{2,i} = k_{2,i} + c_{2,i} \quad (3-40)$$

In equation (3-41), it is shown how the sub-function, $s_{2,i}(x)$, is divided into partial functions.

$$s_2(x) = \begin{cases} 
  s_{2,0}(x_n), & 0 \leq x < \frac{1}{2^w} \\
  s_{2,1}(x_n), & \frac{1}{2^w} \leq x < \frac{2}{2^w} \\
  \ldots & \\
  s_{2,I-1}(x_n), & \frac{I-1}{2^w} \leq x < 1 
\end{cases} \quad (3-41)$$

Note that, in (3-41), $x$ has been changed to $x_w$. The change is made because the intervals for the partial sub-functions, $s_{2,i}(x)$, in (3-41) have equal ranges.

### 3.3.2. Development of two sub-functions

When developing the approximation of the original function, $f_{\text{org}}(x)$, the first and second sub-functions are looked upon as one unity. While, in the Parabolic Synthesis methodology, the first sub-function, $s_1(x)$, was developed to have as good conformity as possible with $f_{\text{org}}(x)$, the objective of the Harmonized Parabolic Synthesis methodology is rather to develop the first sub-function in such a way that the product of the two sub-functions gives a good conformity to the original function. This includes that the characteristics and the distribution of the error are favorable and the hardware implementation as simple as possible. The development of the first sub-function cannot, as in the Parabolic Synthesis methodology, be based on independent analytical calculations since it is dependent on the performance of the second
sub-function, \( s_2(x) \). Therefore, the coefficient \( c_1 \) (3-30) in the first sub-function has to be determined by simulation of the absolute error, \( f_{error}(x) \), between the approximation and the original function according to (3-42).

\[
f_{error}(x) = |s_1(x) \cdot s_2(x) - f_{org}(x)|
\]

To perform this simulation of the absolute error, \( f_{error}(x) \), the second sub-function, \( s_2(x) \), has to be made dependent on coefficient \( c_1 \) in the first sub-function, \( s_1(x) \), as shown in (3-30) to (3-32) and (3-35) to (3-38). The simulation is interesting only as an indication of how the absolute error, \( f_{error}(x) \), depends on the coefficient \( c_1 \). Choosing the coefficient \( c_1 \) must be made with regard to both the characteristics of the error of the approximation and the efficiency of the hardware implementation. The number of intervals in the second sub-function, \( s_2(x) \), needs to be increased to achieve the intended accuracy; this must also be taken into account when performing the simulation of the error. Figure 3-7 shows the bit accuracy for the sine function, shown when using 1, 2, 4 and 8 intervals in the second sub-function, \( s_2(x) \), with different values of the coefficient, \( c_1 \).

![Figure 3-7. The bit accuracy depending on \( c_1 \) for 1, 2, 4 and 8 intervals in the second sub-function, \( s_2(x) \).](image)

From Figure 3-7, values of the coefficient \( c_1 \) are chosen so as to allow efficient hardware implementation. The behavior of the bit accuracy in Figure 3-7 results from the approximation using different values of the coefficient \( c_1 \). From this, the number of intervals and the value on the coefficient \( c_1 \) used in the design can be selected.
Implementations using Harmonized Parabolic Synthesis methodology are described in Paper H and Paper I. The function implemented in the first paper is the logarithm, and a comparison is made with an implementation using the Parabolic Synthesis methodology. In the second paper, the inverse square root function is implemented and a comparison is made with an implementation using the Newton-Raphson method with regard to chip area, critical path delay and power consumption per sample. The second paper also includes an analysis of the characteristics and distribution of the error in the approximations implemented.

3.4. Summary

The development of the Parabolic Synthesis methodologies has been made with a view toward small chip area, short computation time, low power consumption, simple implementation in hardware and the ability to tailor the characteristics and the distribution of the error. Several tools are used to accomplish this. A high degree of parallelism is used to minimize the length of the critical path. In order to simplify the addressing of the look-up tables, this is done directly from the operand. Besides this, only operations that are simple to implement in hardware, such as additions and multiplications, are used. To establish a high degree of parallelism and faster convergence, the algorithm is founded on a synthesis of factors where the factors are second-order parabolic functions. The synthesis of second-order parabolic functions’ factors also implies adjustability for the performance of the algorithm. The next chapter will show the architectures resulting from the use of the methodologies.
The algorithms used in the Parabolic Synthesis methodologies are developed to obtain small chip area, short computation time, low power consumption and enable tailoring of the characteristics and the distribution of the error. This is achieved by having architectures with a high degree of parallelism, high adaptability and low complexity operations that are simple to implement in hardware. The methodologies developed are the original Parabolic Synthesis methodology [Paper G] and the two improved methodologies, Parabolic Synthesis combined with Second-Degree Interpolation [Paper G] and Harmonized Parabolic Synthesis [Paper H]. All methodologies are founded on multiplications of two or more second-order parabolic functions called sub-functions, $s_n(x)$. Since these sub-functions are combined through multiplication, a highly parallel structure can be obtained by a tree structure. Figure 4-1 shows the architecture of the original Parabolic Synthesis methodology using four sub-functions.

Figure 4-1. Architecture of the Parabolic Synthesis methodology using four sub-functions.
The difference between the original methodology and the improved methodologies lies mainly in how the accuracy of the approximation is accomplished. In the Parabolic Synthesis methodology, the accuracy of the approximation is determined by the number of sub-functions, \( s_n(x) \), used in the implementation. In the improved methodologies, only two sub-functions are used, as shown in Figure 4-2.

![Figure 4-2. Architecture of the improved methodologies using two sub-functions.](image)

In the improved methodologies, the second sub-function, \( s_2(x) \), is a second-degree interpolation. The interpolation in the second sub-function is performed by dividing the range over which the approximation is made into a power of two number of intervals. In each of these intervals, the sub-function is then approximated with a second-order parabolic function. The accuracy of the approximation for these methodologies is therefore determined by the number of intervals used in the interpolation.

### 4.1. The architecture of the original Parabolic Synthesis methodology

The architecture of the original Parabolic Synthesis methodology is founded on multiplication of second-order parabolic functions called sub-functions, \( s_n(x) \), as shown in Figure 4-1. For the sub-functions with an index, \( n \), of 3 or higher, the sub-function contains sub-second-order parabolic functions called sub-sub-functions, \( s_{n,i}(x) \). The number of sub-sub-functions in a sub-function is determined by the index according to \( 2^{n-2} \). The number of intervals is therefore always a power of two. This combined with the fact that the range of the interval of a sub-function is restricted to \( 0 \leq x < 1 \) implies that the \( n-2 \) most significant bits in the input \( x \) are used to address the intervals for every sub-sub-function inside each sub-function. Figure 4-3 shows a detailed description of an implementation using the Parabolic Synthesis methodology and with four sub-functions. The implementation in Figure 4-3 has four sub-functions \( s_1(x) \), shown in (4-1), \( s_2(x) \), shown in (4-2), \( s_3(x) \), shown in (4-3), and \( s_4(x) \), shown in (4-4).

\[
\begin{align*}
    s_1(x) &= x + c_1(x - x^2) \\
    s_2(x) &= 1 + c_2(x - x^2)
\end{align*}
\]
\begin{align*}
    s_{3, i}(x_3) &= 1 + c_{3, i}(x_3 - x_3^2) \quad \text{(4-3)} \\
    s_{4, h}(x_4) &= 1 + c_{4, h}(x_4 - x_4^2) \quad \text{(4-4)}
\end{align*}

Figure 4-3. Architecture using the Parabolic Synthesis methodology and with four sub-functions.

The first and second sub-functions are not divided into intervals as the third and fourth sub-functions are. The fact that the sub-functions are not divided into intervals implies that the input operand is $x$ for both. The third and fourth sub-functions are divided into intervals, and thus the input operands are $x_3$ and $x_4$ respectively. Operand $x_3$ is the operand $x$ without the most significant bit $i$, which is instead used for addressing look-up table $c_{3, i}$. Look-up table $c_{3, i}$ contains the coefficients to be used in the two intervals of the third sub-function. Operand $x_4$ is the operand $x$ with-
out the two most significant bits, $h$, which instead are used for addressing look-up table $c_{4,h}$. Look-up table $c_{4,h}$ contains the coefficients to be used in the four intervals of the fourth sub-function. Note that, since for sub-functions with an index, $n$, of 3 or higher, a squaring must be performed on a partial product of the operand $x$, a special squarer architecture has been developed. The special squarer architecture [Paper A and C] computes the squaring of the operand $x$ and all its partial products, in this case $x_3^2$ and $x_4^2$. The squarer architecture is described in the following chapter. After the four sub-functions in Figure 4-3 have been computed, they are multiplied in a tree structure determining the output result, $y$.

4.2. The architecture of the improved Parabolic Synthesis methodologies

![Diagram of the improved Parabolic Synthesis methodologies](image)

**Figure 4-4.** Architecture using the improved Parabolic Synthesis methodologies.

The architecture of the improved Parabolic Synthesis methodologies shown in Paper G and H is founded on multiplications of a second-order parabolic function, $s_1(x)$, and a second-degree interpolation, $s_2(x)$, as shown in Figure 4-2. Since the second sub-function is an interpolation, the function is divided into intervals where the number of intervals is always a power of two. This, combined with the fact that the range of the interval of a sub-function is restricted to $0 \leq x < 1$, implies that the
most significant bits in the input operand \( x \) are used for addressing the intervals of the second sub-function.

Figure 4-4 is a detailed description of an implementation using the improved Parabolic Synthesis methodologies. The implementation in Figure 4-4 uses two sub-functions, \( s_1(x) \), shown in (4-1), and \( s_2(x) \), shown in (4-5).

\[
s_{2,i}(x) = l_{2,i} + j_{2,i}x_w - c_{2,i}x_w^2
\]  

(4-5)

The first is not broken into intervals, while the second is. Where the sub-function is not divided into intervals, the input operand is \( x \). The input operand \( x \) is separated into two parts, the one addressing the look-up tables, \( i \), and the normalized operand, \( x_w \), for the intervals. The number of bits in \( i \) depends on the number of intervals used. These bits \( i \) are used in the addressing of look-up tables \( c_{2,i} \), \( l_{2,i} \), and \( j_{2,i} \) that contain the coefficients for the intervals in the second sub-function. Also here, the squarer architecture, to be described in the next chapter, is used. After the computation of the two sub-functions in Figure 4-4, the sub-functions are multiplied, determining the output result \( y \).

4.3. Pipelining

In all the methodologies, it is feasible to introduce pipelining in the architectures. An example of where pipeline stages can be introduced in the improved Parabolic Synthesis methodologies architecture is shown in Figure 4-4. As seen, the introduction of pipeline stages is simple since it is easy to find natural cuts in the architecture. The introduction of data pipelining stages in the architectures will have a low impact on the size of the hardware, since the data paths are few.

4.4. Reuse

The same architecture can in principle perform approximations of all the functions that are possible for the methodology. Initially, the functions can be divided into two groups, those that start in 0 and end in 1 and those that start in 1 and end in 0. This can be managed through substituting the leftmost \( x \) in (4-1) with \( 1 - x \), when the function starts in 1.

When developing approximations, different functions need different numbers of sub-functions in the Parabolic Synthesis methodology or different numbers of intervals in the improved Parabolic Synthesis methodologies to obtain a certain accuracy. In the Parabolic Synthesis case the architecture is fixed to the number of sub-functions in the architecture. This has the effect that the resulting accuracy for different functions will differ. In the case of the improved Parabolic Synthesis, the function needing the most intervals gets to decide the number of intervals.
4.5. Architectures for other approximation methodologies

To give a perspective of the architectures of the Parabolic Synthesis methodologies, this section describes examples of architectures based on other methodologies, the Newton-Raphson method and the CORDIC algorithm. In hardware implementation, the Newton-Raphson method is commonly used only for a few operations, such as inverse, square root and inverse square root. The structure of its algorithm, shown in (2-2), implies that very hardware efficient implementations, using only operations such as additions and multiplications, can be made. The CORDIC algorithm is chosen because of its special status of being the standard method for implementing unary functions in hardware. Comparative studies of the Newton-Raphson method with Parabolic Synthesis are made in Paper I and with the CORDIC algorithm in Paper D and Paper G.

4.5.1. Architecture: Newton-Raphson

The Newton-Raphson method is an iterative method where the desired accuracy is achieved by the number of iterations performed. Since it is preferred that the number of iteration stages is kept low in an unrolled design, due to delay and chip area, a look-up table is introduced in the architecture. The look-up table provides the approximation algorithm with an initial value for the first iteration. The better the initial value, the fewer iterations are needed, but the larger the size of the look-up table. As an example, retrieved from Paper I, that presents an architecture using the Newton-Raphson method, the algorithm performing the inverse square root, shown in (4-6), is used.

\[ y_i = y_{i-1} \cdot \frac{1}{2} \left( 3 - v \cdot y_{i-1}^2 \right) \]  (4-6)

In Figure 4-5 the first iteration stage is shown for the inverse square root.

In the architecture shown in Figure 4-5, the look-up table (LUT) supplies the first iteration stage with an initial value, \( y_1 \). This value is dependent on the input \( v \).
algorithm described in (4-6) is performed in the iteration stage shown. If only one iteration stage is used, the result of the approximation is directly on the output \(y_2\). If more iteration stages are to be used, copies of the iteration stage in Figure 4-5 are added in the architecture.

### 4.5.2. Architecture: CORDIC

The CORDIC method is an iterative algorithm founded on vector rotation. The desired accuracy is achieved by the number of vector rotations performed. As an example of an architecture using the CORDIC algorithm, the \(\sin(\alpha)\) and \(\cos(\alpha)\) functions are used. The calculations that perform one iteration are shown in (4-7).

\[
\begin{align*}
    x_{i+1} &= x_i - \text{sign}_i \cdot y_i \cdot 2^{-i} \\
    y_{i+1} &= y_i + \text{sign}_i \cdot x_i \cdot 2^{-i}
\end{align*}
\]  

Figure 4-6 shows an unrolled 4-stage CORDIC architecture.

![Figure 4-6. An unrolled 4-stage CORDIC architecture.](image)

The upper part of Figure 4-6 is the control mechanism for the iteration stages shown in the lower part of the figure, where four iterations are performed. The results on the outputs are \(\sin(\alpha)\) and \(\cos(\alpha)\). Generally, it can be estimated that the accuracy of an approximation is given by the number of stages subtracted with 1 bit. In this case, with four stages, the accuracy can be estimated to 3 bits.
4.6. Comparing physical performance

This section generally describes performance parameters such as chip area, critical path delay and power consumption for the three Parabolic Synthesis methodologies, the Newton-Raphson method and the CORDIC algorithm. A discussion is given of the performance of the methodologies contra the accuracy of the approximations. For all methodologies, the width of the data path will grow with increasing accuracy. This will lead to increasing chip area, critical path delay and power consumption. The accuracy as a requirement parameter is not ideal since the error distributions show different characteristics for the methodologies. This survey should therefore be seen more as a discussion of the trends in the different methodologies.

4.6.1. Parabolic Synthesis

In the Parabolic Synthesis methodology, the number of sub-functions correlates to the accuracy. This means that more sub-functions in the architecture result in higher accuracy. An advantage of the methodology is that an addition of one sub-function gives a relative large increase in the accuracy. As shown in Figure 4-3, the architecture has a tree structure; an addition of one sub-function will therefore result in at least one additional multiplier in the architecture. Using four sub-functions generally results in a maximum accuracy of around 15 bits, depending on the function that the approximation is performed upon. Stepping up the number of sub-functions with one will generally increase the accuracy by between 3 and 4 bits; this is also dependent on the function. From this it can be concluded that, for a moderate accuracy of up to around 15 bits, the methodology can provide a very efficient implementation. For higher accuracies, the complexity the architecture will gradually increase faster and faster.

4.6.2. Improved Parabolic Synthesis methodologies

In the improved Parabolic Synthesis methodologies, the accuracy is determined by the number of intervals used in the second sub-function. The higher the number of intervals, the higher the accuracy that can be achieved. A feature in this methodology is that, with an increasing number of intervals in the second sub-function, no extra components are added. Beyond the width of the data path, it is only the size of the look-up tables in the second sub-function, shown in Figure 4-4, that will grow. Using four intervals in the second sub-function generally results in a maximum accuracy of around 15 bits, depending on which function the approximation is performed upon. Doubling the number of intervals will generally increase the accuracy by between 3 and 4 bits; this is also dependent on the function. These methodologies can therefore enable very high accuracies. The limitations lie rather in when the physical size of the look-up tables and the multipliers reaches an upper limit.
An interesting feature of the improved Parabolic Synthesis methodologies is that, for a certain accuracy, chip area, critical path delay and power consumption can be significantly decreased by increasing the number of intervals in the second sub-function. The reason for this is that the width of the data path delay will decrease in parts of the second sub-function. This will lead to a shrinking of the size of the multipliers, reducing chip area, critical path and power consumption. However, the look-up tables will grow with the number of intervals, which is why this works only up to a certain point at which these parameters start to increase.

4.6.3. Newton-Raphson method

The Newton-Raphson method is an iterative method that, by iterations, gradually finds better approximations of the roots of a unary function. To reduce the number of iterations, a look-up table (LUT) with suitable initial values to the iteration is introduced, as in the example shown in Figure 4-5. It is desirable to use only one iteration, since this will minimize the hardware usage. In that case, when increasing the accuracy, the look-up table will grow very fast; introducing more iteration stages is therefore the only solution to fall back on. Implementing it unrolled will drastically increase the chip area, critical path and power consumption.

4.6.4. CORDIC algorithm

The CORDIC algorithm is an iterative algorithm where the number of iteration stages decides the accuracy. As shown in the example in Figure 4-6, the architecture has the advantage of using only simple operations such as additions and shifts. Since the improvement of the accuracy for each iteration stage is limited, the number of iterations will grow very fast with an increasing accuracy of the approximation. Implementing it unrolled will very drastically increase chip area, critical path and power consumption.

4.6.5. Performed comparisons

Comparisons have been made of the Parabolic Synthesis methodologies with the CORDIC algorithm and the Newton-Raphson method in some of the publications presented. In Paper D, implementations using the Parabolic Synthesis methodology and the CORDIC algorithm implementing the logarithm and the exponential function are compared. This comparison is expanded in Paper G with a comparison of the logarithm implemented using Parabolic Synthesis combined with Second-Degree Interpolation. In Paper I, implementations of the inverse square root function have been implemented using the Harmonized Parabolic Synthesis methodology and the Newton-Raphson method. Two different implementations of each
method are described in the paper. The comparison in the paper deals with both the physical performance and the distribution of the error.
In the hardware architecture of the designs based on the Parabolic Synthesis methodologies, the squaring, $x^2$, and the partial squaring, $x_w^2$, are reoccurring operations. An example is shown in Figure 4-4, where the hardware architecture of the Parabolic Synthesis combined with Second-Degree Interpolation is shown. In the first sub-function, $s_1(x)$, the result of the squaring, $x^2$, is needed. Then, in the second sub-function, $s_2(x)$, the partial result of the squaring, $x_w^2$, is required. To reduce the hardware consumption, a special hardware architecture for computing the squaring result and its partial squaring results, has been developed [Papers A, B and C].

When analyzing the architecture of the developed squarer, a great resemblance to a bit-serial squarer [24] [25] is found. The partial results of $x_w^2$ are easily extracted by introducing registers in the design of the bit-serial squarer. The squaring algorithm can thus be simplified to one addition only when computing each partial product.

5.1. Squaring and the partial squaring

The special hardware architecture for the squaring operation, computing the squaring result as well as its partial squaring results, will have input and outputs according to Figure 5-1.

$$x \rightarrow \text{Squarer} \rightarrow p_{n-1} \rightarrow p_{n-2} \rightarrow \ldots \rightarrow p_1 \rightarrow p_0$$

Figure 5-1. The squaring operation computing the square and its partial squares.
In Figure 5-1 the input \( x \) is an \( n \)-bit unsigned binary integer \((x_{n-1}x_{n-2} \ldots x_1x_0)_{\text{two}}\). The result of the squaring of \( x \) is given as output \( p_0 \), and the results of the partial products are given as output \( p_{n-1} \). On \((x_{n-2}x_{n-3} \ldots x_1x_0)_{\text{two}}\), the partial product \( p_1 \) is computed, \( p_2 \) is computed on \((x_{n-3}x_{n-4} \ldots x_1x_0)_{\text{two}}\), and the following partial products are computed until only last bit \( x_0 \) remains to square in the input \( x \).

A 4-bit unsigned binary integer \((x_3x_2x_1x_0)_{\text{two}}\) is used to illustrate the method. Figure 5-2 shows the calculations of the squaring product combined with the squaring of the partial products of a 4-bit input.

As shown in Figure 5-2, the squaring of \( x \) and the partial squaring can both be computed in the same hardware. This is an important feature of the squarer since this will reduce the chip area compared to using separate multipliers for computing the squaring and partial squaring products.

In fact, the calculations can be considerably simplified, as indicated in Figure 5-2, resulting in the reduced calculations described in Figure 5-3. A term \( x_i x_j \) can be reduced to \( x_j \) and a pair of terms \( x_i x_j \) and \( x_j x_i \) in a given column can be replaced by \( x_j x_i \) in the next higher column. Examples of this are shown in the framed terms in Figure 5-2. Note also the terms \( p_{0,0} \), \( p_{1,0} \), \( p_{2,0} \) and \( p_{3,0} \) are all equal to \( x_0 \), as shown.
below in (5-1). Similarly, that the terms $p_{0,0}$, $p_{1,0}$, $p_{2,0}$, and $p_{3,0}$ all are equal to 0, as shown in (5-2).

Figure 5-3 shows the simplified squaring product calculations on a 4-bit input.

When simplifying the squaring algorithm in Figure 5-3, the result of the component $p_{3,0}$ in the vector $p_3$ is simplified according to (5-1).

$$p_{3,0} = x_0x_0 = x_0$$ (5-1)

The result of the component $p_{3,1}$ in the vector $p_3$ is equal to 0 since the result of $p_{3,0}$ can never contribute to $p_{3,1}$.

The result of the component $p_{2,1}$ in the vector $p_2$ is simplified according to (5-2).

$$p_{2,1} = p_{3,1} \cdot 2^1 + x_1x_0 \cdot 2^1 + x_0x_1 \cdot 2^1 = 0 \cdot 2^1 + x_0x_1 \cdot 2^1 = 0 \cdot 2^1$$ (5-2)

The result of the component $p_{2,2}$ in the vector $p_2$ is simplified according to (5-3).

$$p_{2,2} = x_1x_0 \cdot 2^2 + x_1x_1 \cdot 2^2 = x_1x_0 \cdot 2^2 + x_1 \cdot 2^2$$ (5-3)

The result of the component $p_{1,2}$ in the vector $p_1$ is simplified according to (5-4).

$$p_{1,2} = p_{2,2} \cdot 2^2 + x_2x_0 \cdot 2^2 + x_0x_2 \cdot 2^2 = p_{2,2} \cdot 2^2 + x_2x_0 \cdot 2^3 = p_{2,2} \cdot 2^2$$ (5-4)

The result of the component $p_{1,3}$ in the vector $p_1$ is simplified according to (5-5).

$$p_{1,3} = p_{2,3} \cdot 2^3 + x_2x_1 \cdot 2^3 + x_1x_2 \cdot 2^3 + x_2x_0 \cdot 2^3 =$$

$$p_{2,3} \cdot 2^3 + x_2x_1 \cdot 2^4 + x_2x_0 \cdot 2^3 = p_{2,3} \cdot 2^3 + x_2x_0 \cdot 2^3$$ (5-5)

The result of the component $p_{1,4}$ in the vector $p_1$ is simplified according to (5-6).
\[ p_{1,4} = x_2 x_1 \cdot 2^4 + x_2 x_2 \cdot 2^4 = x_2 \cdot 2^4 + x_2 x_1 \cdot 2^4 \]  
\hfill (5-6)

The result of the component \( p_{0,3} \) in the vector \( p_0 \) is simplified according to (5-7).

\[ p_{0,3} = p_{1,3} \cdot 2^3 + x_3 x_0 \cdot 2^3 + x_0 x_3 \cdot 2^3 = p_{1,3} \cdot 2^3 + x_3 x_0 \cdot 2^4 = p_{1,3} \cdot 2^3 \]  
\hfill (5-7)

The result of the component \( p_{0,4} \) in the vector \( p_0 \) is simplified according to (5-8).

\[ p_{0,4} = p_{1,4} \cdot 2^4 + x_3 x_0 \cdot 2^4 + x_3 x_1 \cdot 2^4 + x_1 x_3 \cdot 2^4 = \]  
\[ p_{1,4} \cdot 2^4 + x_3 x_0 \cdot 2^4 + x_3 x_1 \cdot 2^5 = p_{1,4} \cdot 2^4 + x_3 x_0 \cdot 2^4 \]  
\hfill (5-8)

The result of the component \( p_{0,5} \) in the vector \( p_0 \) is simplified according to (5-9).

\[ p_{0,5} = p_{1,5} \cdot 2^5 + x_3 x_1 \cdot 2^5 + x_3 x_2 \cdot 2^5 + x_2 x_3 \cdot 2^5 = \]  
\[ p_{1,5} \cdot 2^5 + x_3 x_1 \cdot 2^5 + x_3 x_2 \cdot 2^6 = p_{1,5} \cdot 2^5 + x_3 x_1 \cdot 2^5 \]  
\hfill (5-9)

The result of the component \( p_{0,6} \) in the vector \( p_0 \) is simplified according to (5-10).

\[ p_{0,6} = x_3 x_2 \cdot 2^6 + x_3 x_3 \cdot 2^6 = x_3 \cdot 2^6 + x_3 x_2 \cdot 2^6 \]  
\hfill (5-10)

### 5.2. Advantages

Since multipliers are very power hungry, it is essential to reduce the presence of these components. The hardware architecture that has been developed that performs the squaring combined with the partial squaring in a single hardware is therefore important. Other advantages that the squarer exhibits compared to an equivalent multiplier are that the chip area and critical path are halved. With the squarer developed, only one hardware squarer is needed for all squarer operations in the design. The halved chip area and also halved critical path delay will therefore considerably reduce the power consumption in the architectures based on the Parabolic Synthesis methodologies.
6 Error Analysis

Error analysis is an essential part of the process of developing approximations since the characteristics and the distribution of the error are important for the performance. A set of tools for analyzing the characteristics and the distribution of the error is therefore vital when developing an approximation algorithm. The analyzing tools should study which kind of and quantity of the error, or uncertainty, that may be present in the approximation. Actually, the desired characteristics and distribution of the error of the approximation are very dependent on the context in which they are. In many contexts, the asymptotic normal distributed error is preferred since the normal distribution has many attractive features. In probability theory, the Central Limit Theorem states that, under certain conditions, the arithmetic mean of a satisfactorily large number of repeated independent random variables has a well-defined anticipated value, a well-defined variance and an approximately normal distribution, regardless of the underlying distribution [27]. When applied to the distribution of the error of an approximation, this feature implies that the standard normal distribution is preferable. Due to the Central Limit Theorem, a standard normal distributed error is preferable when considering the following computation in an algorithm. When analyzing the word length needed in an implementation of an approximation, an observation has been that a standard normal distributed error to some extent also makes the implementation more efficient.

6.1. The characteristic and distribution of the error

Five metrics are used to characterize the error of an approximation [28]. These metrics are maximum absolute error, mean error, median error, standard deviation and root mean square error. With these metrics, the quantity of an error in an approximation can be analyzed when the algorithm is developed. These metrics are also tools that are employed when comparing different implementations using disparate implementation methodologies. Figure 6-1 shows a subtraction of a function with its approximation, also called the error.
6.1.1. Maximum absolute error

The absolute error $\Delta x_i$ is the absolute value of the difference between the approximation value $\hat{x}_i$ and its actual value $x_i$, as shown in (6-1).

$$\Delta x_i = |\hat{x}_i - x_i| \quad (6-1)$$

The maximum absolute error is the maximum value of the absolute error in the interval in which the investigation is performed. As an example, the marked maximum absolute error in Figure 6-1 is 0.0000159.

6.1.2. Mean error

The mean error $\bar{x}$ of $n$ separate values of an error in a sequence of errors, $\hat{x}_i - x_i$, is defined as in (6-2).

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} (\hat{x}_i - x_i) = \frac{(\hat{x}_1 - x_1) + (\hat{x}_2 - x_2) + \ldots + (\hat{x}_n - x_n)}{n} \quad (6-2)$$

(6-2) merely states that the average of a sequence of $n$ numbers is the sum of those numbers divided by $n$. As an example, the mean error in Figure 6-1 is 0.0000000195, which is very close to the desirable value, which is 0.

Figure 6-1. The distribution of the error of a function, with its maximum absolute error marked.
6.1.3. Median error

The median \( \tilde{x} \) is an order statistic that gives the "middle" value of a sample of errors. As an example, the median error in Figure 6-1 is -0.0000000250 which, by its small size, shows that the error is evenly distributed.

6.1.4. Standard deviation

The standard deviation \( \sigma \) is similar to the average deviation, except that averaging is made with power instead of amplitude. This is achieved by squaring each of the deviations before taking the average. Finally, the square root is taken to compensate for the initial squaring. The standard deviation is calculated as shown in (6-3).

\[
\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^{n} [\tilde{x}_i - \bar{x}]^2} 
\tag{6-3}
\]

As an example, the standard deviation in Figure 6-1 is 0.00000474, which indicates a fairly narrow distribution of the error.

6.1.5. Root mean square

The standard deviation is a measure of how far the signal fluctuates from the mean. The variance represents the power of this fluctuation. Another is the Root-Mean-Square (RMS) value, frequently used in electronics. By definition, the standard deviation only measures the AC portion of a signal, while the RMS value measures both the AC and DC components. If a signal has no DC component, its RMS value is identical to its standard deviation. \( x_{rms} \) is defined as shown in (6-4).

\[
x_{rms} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\tilde{x}_i - x_i)^2} 
\tag{6-4}
\]

As an example, the RMS value in Figure 6-1 is 0.00000474. Since the RMS corresponds with the standard deviation, this confirms an even distribution of the error.

6.1.6. Decibel

Using logarithms when presenting statistical measures and diagrams that show the accuracy of a function will provide greater resolution and simplify the understanding of the results.

A decibel (dB) is defined in two equivalent ways. When referring to measurements of power or intensity, it is as shown in (6-5):
But, when referring to measurements of amplitude, it is as shown in (6-6).

\[ x_{dB} = 20\log_{10}\left(\frac{x}{x_0}\right) \]  

(6-6)

In (6-5) and (6-6), \( x_0 \) is a specified reference with the same units as \( x \). The reference used depends on convention and context.

### 6.1.7. Decibel and binary numbers

Binary numbers and decibel (dB) work very well together since 1 bit in resolution is \( 20\log(2) = 20\cdot(0.301) \approx 6 \text{dB} \). Displaying the result in dB will therefore simplify the understanding of the result. For example, an error of 0.01 is equal to \( 20\log(0.01) = 20\cdot(-2) = -40 \text{dB} \). Transforming the error into accuracy in bits is then done by simply making the dB value into a positive number and dividing it by 6. As an example, -40dB is converted to 40 and then divided by 6, which gives 6.67, or, since this is larger than 6, the error is less than 6 bits. Figure 6-2 shows the error in dB of the error shown in Figure 6-1.

As an example, the largest error in Figure 6-2 is about -96dB, which is about 16 bits of accuracy.
6.1.8. Distribution of the error

When developing approximations, the challenge is to develop an efficient approximation that conforms to the function to be approximated in the desired interval. Since the approximations to be developed will be used in a series of computations, it is also important that the error of the approximation is not of unilateral polarity. It is therefore important to investigate the distribution of the error. A helpful tool in visualizing and interpreting the deviation of the error is a diagram of the distribution of the error. The diagram will simplify the understanding of the standard deviation and root-mean-square values. Figure 6-3 shows the distribution of the error in Figure 6-1.

![Distribution of Elements in %](image)

Figure 6-3. The distribution of the error in Figure 6-1.

The distribution shown in Figure 6-3 confirms the previous results of the mean error, median error, standard deviation and the RMS value.
For most unary functions, directly applying an approximation methodology in a straightforward way will not lead to an efficient implementation. Instead, a dedicated algorithm often has to be developed.

The functions’ roots, inverse and inverse roots are computed in large quantity in many complex matrix operations, such as QR decomposition, making these functions particularly interesting. In applications such as future MIMO communication systems [26], a massive computation of these functions is needed. Therefore, to obtain sufficiently high performance in such applications, efficient algorithms are very important. Since these algorithms are in hardware, it must also be ensured that the algorithms meet high requirements in terms of small chip area, low computation time and low power consumption. Low power consumption is especially important since many applications are battery powered.

7.1. Algorithms for roots, inverse and inverse roots

For the functions’ roots, inverse and inverse roots, algorithms [Paper I] have been developed founded on using floating-point numbers. Algorithms for these functions are commonly developed founded on the use of fixed-point numbers. Note that the algorithms are developed without regard to a specific approximation methodology. The format of the floating-point numbers used in the algorithms is simplified compared to the IEEE standard for floating-point arithmetic (IEEE 754). The floating-point number format is customized for hardware implementation. The floating-point number consists of a mantissa with a range from 1 to < 2 and an exponent where the exponent is a scaling of the mantissa. Using floating-point numbers as an internal representation, the computation can be divided into separate parts for the mantissa and the exponent. This will reduce the complexity since the approximation is performed on a mantissa of limited range, and the computation of the exponent is a very
simple operation. Table 7-1 shows a conversion of a fixed-point number into a floating-point number in base 2.

<table>
<thead>
<tr>
<th>Base 10</th>
<th>387</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point Base 2</td>
<td>000000110000011</td>
</tr>
<tr>
<td>Exponent Index</td>
<td>0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td>Floating-point Base 2</td>
<td>1.100001100000000 \cdot 2^8</td>
</tr>
</tbody>
</table>

Table 7-1. Conversion from fixed point to floating point in base 2.

As shown in Table 7-1, the index of the exponent is given by the most significant 1 in the fixed-point number, which is 8 in this case. The floating-point number is shown in the last line of Table 7-1, in which the mantissa is scaled by the exponent. When computing the binary function on the mantissa, this is performed as an approximation. As shown in Figure 7-1, the mantissa of the floating-point number is the input $v$ to the approximation, and $z$ is the output from the approximation.

![Figure 7-1: Block diagram of input and output of the approximation.](image)

7.1.1. Algorithms for computing roots

In addition to that the algorithms for computing roots being founded on floating-point numbers, they are also based on changing the number base of the exponent. The number base used when performing an approximation of a root depends on the order, $d$, of the root. The base used in the approximation is $2^d$. Although the number base is changed, the binary number base is retained in the representation of the mantissa. The algorithms computing roots can therefore only compute roots of order $d$, where $d$ is a natural number. The purpose of changing the number base is that, after computing the root, the base of the exponent will always be 2, as shown in (7-1).
\[ d \sqrt{2^d} = 2 \quad (7-1) \]

As shown in (7-2), the mantissa consists of \( d \) integer bits \( M \) and \( h \) fractional bits \( m \).

\[ M_{d-1}m_{d-2}...m_0 \quad (7-2) \]

The range of the mantissa will be \( 1 \leq \text{mantissa} < 2^d \).

### 7.1.1.1. Algorithm for computing the square root

Computing the square root \( d = 2 \) gives that the floating point uses an exponent with base 4. Table 7-2 shows a conversion from a fixed-point number in number base 2 into a floating-point number with an exponent base 4 and binary number representation of the mantissa.

<table>
<thead>
<tr>
<th>Base 10</th>
<th>387</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point Base 2</td>
<td>0000000110000011</td>
</tr>
<tr>
<td>Exponent</td>
<td>00 00 00 01 10 00 00 11</td>
</tr>
<tr>
<td>Index</td>
<td>7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>Floating-point Base 4</td>
<td>01.10000011000000·4^4</td>
</tr>
</tbody>
</table>

Table 7-2. Conversion from fixed point in base 2 into floating point in base 4.

As shown in Table 7-2, when deciding the index of the exponent, the fixed-point number in number base 2 is transformed into sequential pairings of number base 2 digits. Table 7-2 gives the most significant pair of digits for index 4. The pair of digits for index 4 is the integer bits in the mantissa, and the remaining pairs of digits are the fractional bits of the mantissa. The last line of Table 7-2 shows the floating-point number in number base 4.

The computation to be performed when computing the square root, \( z \), is shown in (7-3).

\[ z = \sqrt{M_1M_0.m_{-1}m_{-2}...m_{-h}} \cdot 4^{\text{index}} \quad (7-3) \]

In (7-4), (7-3) is simplified to get the exponent in number base 2.

\[ z = \sqrt{M_1M_0.m_{-1}m_{-2}...m_{-h}} \cdot 2^{\text{index}} \quad (7-4) \]

As shown in (7-4), computing the exponent is done through a simple change of number base.
When computing the approximation of the square root, this is performed only on the range $1 \leq v < 4$, as shown in Figure 7-2.

![Figure 7-2. The square root function.](image)

A summary of the algorithm for computing the square root function is shown in the block diagram in Figure 7-3.

![Figure 7-3. Block diagram of the square root algorithm.](image)

As shown in Figure 7-3, the starting point is a floating-point number in number base 4. The exponent and mantissa are computed separately. When computing the exponent, the number base is changed from 4 to 2 without changing the index. When computing the mantissa, it is assumed that the incoming mantissa is in the number
base 4. An approximation of the square root function is performed on the incoming mantissa. After the approximation, the result, $z$, is in the range $1 \leq z < 2$, which is the desired range of the mantissa with the number base 2.

### 7.1.2. Algorithms for computing the inverse

The algorithms for computing the inverse are also founded on using a floating-point number.

The computation to be performed to compute the inverse, $z$, is shown in (7-5).

$$z = \frac{1}{M_0.m_{-1}m_{-2}...m_{-h} \cdot 2^{index}}$$

(7-5)

In (7-6), (7-5) is simplified to get the exponent in the numerator, thus with a negative index.

$$z = \frac{1}{M_0.m_{-1}m_{-2}...m_{-h}} \cdot 2^{-index}$$

(7-6)

As shown in (7-6), when computing the exponent, only the sign of the index is changed.

The approximation of the inverse is performed in the range $1 \leq v < 2$, as shown in Figure 7-4.

![Figure 7-4. The inverse function.](image)

A summary of the algorithm for computing the inverse function is shown in the block diagram of Figure 7-5.
As shown in Figure 7-5, the starting point is a floating-point number. Also as shown in Figure 7-5, the exponent and mantissa are computed separately. When computing the mantissa, the range after approximation is $0.5 < z \leq 1$. A special case is when $z = 1$, since then the mantissa is in the appropriate format. In most cases, however, when $z < 1$, a multiplication by 2 must be performed to get the mantissa in the appropriate format. When computing the exponent, the sign of the index is initially changed. Depending on the result when computing the approximation of the mantissa, when $z = 1$, the exponent remains untouched, whereas, when $z < 1$, the exponent is subtracted with 1.

![Block diagram of the inverse algorithm.](image)

**Figure 7-5. Block diagram of the inverse algorithm.**

### 7.1.3. Algorithms for computing inverse roots

The algorithm for computing inverse roots combines the algorithms for roots and inverse described in Sections 7.1.1 and 7.1.2.

The computation used to perform to compute the inverse, $z$, is shown in (7-7).

$$ z = \frac{1}{d^{\frac{1}{d}}M_{d-1}M_{d-2}m_{-1}m_{-2}...m_{-h} \cdot 2^{d \cdot index}} \quad (7-7) $$

As shown in (7-7), the inverse root, $d$, is performed on a floating-point number with the number base $2^d$ and in binary number representation. As shown in (7-7), the mantissa consists of $d$ integer bits, $M$, and fractional bits, $m$, where $h$ is the number of bits used in the fractional part.

In (7-8), (7-7) is simplified to get the exponent in the numerator thus in base 2 with a negative *index*. 
As shown in (7-8), computing the exponent is made through a simple change of number base changed sign of index. Since the integer part of the mantissa must always be 1 or larger, the range of the mantissa will be $1 \leq \text{mantissa} < 2^d$ dependent on the root $d$ to be computed.

When computing the approximation of the square root, this is performed only on the range $1 \leq v < 4$, as shown in Figure 7-6.

![Figure 7-6. The inverse square root function.](image)

A summary of the algorithm for computing the inverse square root function is shown in the block diagram of Figure 7-7.

As shown in Figure 7-7, the starting point is a floating-point number. Also as shown in Figure 7-7, the exponent and mantissa are computed separately. When computing the mantissa, the range after approximation is $0.5 < z \leq 1$. A special case is when $z = 1$, since then the mantissa is in the appropriate format. In most cases, however, when $z < 1$, a multiplication by 2 must be performed to get the mantissa in the appropriate format. When computing the exponent, the sign of the index is initially changed. Depending on the result when computing the approximation of the mantissa, when $z = 1$, the exponent remains untouched, whereas, when $z < 1$, the exponent is subtracted with 1.
Figure 7-7. Block diagram of the inverse square root algorithm.
The problem that motivates this work is that the development of speed in semiconductor technology has stagnated while the demand for higher performance continues to increase. Future applications in telecommunications, image processing and other application areas will be increasingly more computation intensive; there are therefore growing needs for efficient algorithms. Efficiency is demanded, not only in terms of computational capacity but also in power consumption since many applications are battery powered. A performance limiting part of many algorithms is the approximation of unary functions, such as trigonometric functions, logarithm and square root, as well as binary functions, such as division. Qualities of the approximations in terms of characteristics and distribution of the error have a major impact on the performance of a hardware implemented algorithm. There is therefore a requirement that it is possible to make an adjustment of the approximation error characteristics. Furthermore, when implementing approximations of some functions, it is required that the functions are transformed. Developing effective transformations of these functions is especially important in the case that these functions are used frequently in the algorithm.

An approximation methodology of unary functions in hardware that is founded on a synthesis of parabolic functions by multiplication has been developed to address the demands for higher performance. The methodology is called Parabolic Synthesis. The accuracy of the approximation in this methodology is given by the number of parabolic functions used. Attractive features of the methodology are efficient hardware implementation, a high degree of parallelism, power efficient hardware architecture and adjustable error distribution.

It was further found that an improvement of the Parabolic Synthesis methodology was possible by combining it with second-degree interpolation. By doing this, the architecture is reduced to a synthesis of one parabolic function and one second-degree interpolation; the latter is divided into intervals. Here the parabolic function is a rough approximation of the desired function, whereas the second-degree interpolation enhances the approximation to an extent that is given by the number of
intervals used. The benefits of this methodology are reductions in hardware, critical path length and power consumption. Due to the greater possibility to adjust the approximation error, compared to Parabolic Synthesis, the number of functions to perform approximations on is extended.

It was found that there were further improvements that could be made on the Parabolic Synthesis Combined with Second-Degree Interpolation methodology. The improvement comes from the adoption of a design strategy that takes a holistic view of the design process. With this strategy, performance in terms of chip area, critical path delay and power consumption was further improved. Since the design strategy allows a higher degree of flexibility and adaptability in the design process, it also results in extending the range of unary functions that can be approximated even further.

When developing an approximation, it is necessary - to a varying degree - to develop an adaptation of the function to fit the Parabolic Synthesis methodologies. Most applications need highly efficient algorithms; in particular, algorithms for computing roots, the inverse and inverse roots are of interest since these are used in many computation intensive applications such as within telecommunication and image processing. Finding efficient algorithms for these functions is therefore of great interest. The new algorithms developed for these functions that are presented in this thesis are all founded on the use of floating-point number representation. This will scale down the problem so that the approximations need only be performed on a range limited mantissa of the floating-point number. For the roots and the inverse roots, changing the number base is used since this also simplifies the computation of the approximation. By using number bases that are a power of 2, roots and inverse roots of the order of natural numbers from 2 and up can be more efficiently computed. These algorithms are characterized by being simple to implement in hardware and by providing high performance.

When developing algorithms for different applications and using different approximation methods, it was found how important the characteristics of the error and its distribution are for the computing performance. It was observed that a given algorithm performs in very different ways depending on the distribution of the error of the approximation method used. When analyzing the distribution of the error of these methods, the methods in which the approximation errors are evenly distributed around zero performed significantly better than the methods that give an uneven distribution of the error. To deal with this, an initial set of tools has been developed, tools that characterize the error and analyze the distribution of the error. This tool set is therefore an essential asset in the development of an optimal algorithm for implementation in hardware. An optimal width of the data path in relation to characteristics and the distribution of the error can be found using this approach. A successful optimization of an implementation would result in improved performance in
terms of smaller chip area, shorter critical path delay, reduced power consumption and better distribution of the error.

An essential part of the development of the Parabolic Synthesis methodologies is the comparison with existing methods. A more thorough investigation has therefore been made of some selected methods, polynomial approximation, the Newton-Raphson method and CORDIC. The polynomial approximation methods were chosen since they are commonly used in hardware implementations. The Newton-Raphson method was selected since it shows excellent performance for a limited number of functions. The CORDIC algorithm can be regarded as the “state of the art” and therefore essential to include in the comparison. In these studies, the focus has not only been on comparison but also on improvement of the existing methods. Areas for improvement that have been investigated in the study are chip area, critical path delay, power consumption and distribution of the error. In all existing methods that have been studied, there have been possibilities for improvement in one or more of these aspects.

When comparing the existing methods with the Parabolic Synthesis methodologies, the existing methods that were selected proved to perform worse or, in some cases, on parity with the Parabolic Synthesis methodologies. It was found in the comparisons that there seems to be a breaking point at an accuracy of about 15 bit. For higher accuracies, the Parabolic Synthesis methodologies always prove to be preferable. This breaking point comes earlier for the polynomial approximation and CORDIC. The Newton-Raphson method performs best, but, because of its structure, only a limited number of functions are feasible for hardware implementation. It was found in the comparison that especially the two latest developed Parabolic Synthesis methodologies can be used to implement a large amount of unary functions. It was also found that these methodologies - especially when accuracy was increased - were outstanding when it comes to chip area, critical path delay and power consumption. The ability that these methodologies show with regard to managing the characteristics and the distribution of the error is also superior to the existing methods that have been studied.
9 Future Work

As the Parabolic Synthesis methodologies are capable of performing efficient implementations of approximations in hardware, there is a great development potential in adopting these methodologies in practice. Specifically, the development potential is based on the properties of:

- Approximations of a large range of unary functions.
- Efficient hardware implementation.
- High computation speed.
- Low power consumption.
- Wide range of hardware design optimization possibilities.
- Wide range of adjustable accuracy in the computation.
- Tailorable characteristics and distribution of the approximation error.

In relation to the work in this thesis, I have considered the following topics for future research:

- An essential part of the computation in telecommunications, image processing and other applications is computing matrix inversions, with a need for very high throughput. A challenge is therefore to develop a set of highly efficient algorithms for matrix inversion in these applications.

- The Fast Fourier transform (FFT) is one of the most important tools in digital signal processing and therefore used in numerous applications. It has been seen in previous work that Parabolic Synthesis methodologies can contribute very much to improving performance in these algorithms. It is therefore of interest to continue this research for further improvement of performance.

- An arithmetic-logic unit (ALU) that can handle floating-point arithmetics is often a neglected part of the smaller processor systems. Since many of the algorithms developed for computing approximations of unary functions are founded
on floating-point number representation, and, since the Parabolic Synthesis methodologies manage high accuracy, this is a very attractive task for future work. This work will also involve developing algorithms for more unary functions.

- The characteristics and the distribution of the error are strongly connected to the different parameters of a hardware design. It is thus essential to have control over the characteristics and the distribution of the error when developing a highly optimized algorithm for implementation in hardware. It is therefore very important to further develop the tools for analyzing the characteristics of the error of an algorithm, and this needs to be continued.

- How the chosen accuracy affects the size of the hardware for the different approximation methodologies is an important issue when choosing which methodology to use, for example to be included in a design tool.
References


Paper A
Abstract  This paper introduces a parabolic synthesis methodology for developing approximations of unary functions like trigonometric functions and logarithms which are specialized for efficient hardware mapped VLSI design. The advantages with the methodology are, short critical path, fast computation and high throughput enabled by a high degree of architectural parallelism. The feasibility of the methodology is shown by developing an approximation of the sine function for implementation in hardware.

Index Terms  Algorithms implemented in hardware, computer arithmetic, parabolic synthesis, parallel design style, VLSI.

I.  INTRODUCTION

Unary functions, such as trigonometric functions and logarithms, are extensively used in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics, etc. For these high-speed applications, software solutions are in many cases not sufficient and a hardware implementation is therefore needed. Implementing a numerical function \( f(x) \), by a single look-up table is simple and fast which is straightforward for low-precision computations of \( f(x) \), i.e., when \( x \) only has a few bits. However, when performing high-precision computations a single look-up table implementation is impractical due to the huge table size and the long execution time.

Approximations only using polynomials have the advantage of being ROM-less, but they can impose large computational complexities and delays [1]. By introducing table-based methods to the polynomials methods the computational complexity can be reduced and the delays can also be decreased to some extent [1].

The CORDIC (COordinate Rotation Digital Computer) algorithm [2] [3] has been used for these applications since it is faster than a software approach. CORDIC is an iterative method and therefore slow which makes the method insufficient for this kind of applications.

This paper proposes a parabolic synthesis methodology to develop functions that performs an approximation of original functions in hardware. The architecture of the processing part of the methodology is using parallelism to reduces the execution time. For the development of approximations of functions a parabolic synthesis methodology has been applied. Only low complexity operations that are simple to implement in hardware are used.

II.  METHODOLOGY

The methodology is developed for implementing approximations of unary functions in hardware. The approximation part is of course the important part of this work but there are sometimes two other steps that are necessary, a preprocessing normalization and postprocessing transformation as described by P.T.P. Tang [1]. The computation is therefore divided into three steps, normalizing, approximation and transforming.

A. Normalizing

The purpose with the normalization is to facilitate the hardware implementation by limiting the numerical range.

The normalization has to satisfy that the values are in the interval \( 0 \leq x < 1 \) on the \( x \)-axis and \( 0 \leq y < 1 \) on the \( y \)-axis. The coordinates of the starting point shall be \((0,0)\). Furthermore, the ending point shall have coordinates smaller than \((1,1)\) and the function must be strictly concave or convex through the interval. An example of such a function, called an original function \( f_{\text{org}}(x) \), is shown in Fig. 1.

![Fig. 1 Example of normalized function, in this case \( \sin(\pi x/2) \).](image-url)
B. Developing the Hardware Architecture

When developing a hardware architecture that approximate an original function, only low complexity operations are used. Operations such as shifts, additions and multiplications are efficient to implement in hardware and therefore searched for. The down scaling of the semiconductor technologies and the development of efficient multiplier architectures has made the multiplication operation efficient in both size and execution time when implemented in hardware. The multiplier is therefore commonly used in this methodology when developing the hardware.

As in Fourier analysis [4] the proposed methodology is based on decomposition of basis functions. The proposed methodology is not, as in Fourier analysis, a decomposition method in terms of sinusoidal functions but in second order parabolic functions. Second order parabolic function are used since they can be implemented using low complexity operations. The proposed methodology also differs from Fourier synthesis process since the proposed methodology are using multiplications in the recombination process and not additions as in the Fourier case.

The proposed methodology is founded on terms of second ordered parabolic functions called sub-functions \( s_n(x) \), that when recombined, as shown in (1), obtains to the original function \( f_{org}(x) \). When developing the approximative function the accuracy depends on the number of sub-functions used.

\[
f_{org}(x) = s_1(x) \cdot s_2(x) \ldots \cdot s_n(x) \quad (1)
\]

The procedure when developing sub-functions is to divide the original function \( f_{org}(x) \), with the first sub-function \( s_1(x) \). This division generates the first function \( f_1(x) \), as shown in (2).

\[
f_1(x) = \frac{f_{org}(x)}{s_1(x)} \quad (2)
\]

The first sub-function \( s_1(x) \), will be chosen to be feasible for hardware, according to the methodology described in (4). In the same manner the following functions \( f_n(x) \), are generated, as shown in (3).

\[
f_{n+1}(x) = \frac{f_n(x)}{s_{n+1}(x)} \quad (3)
\]

C. Methodology for developing sub-functions

The methodology for developing sub-functions is founded on decomposition of the original function \( f_{org}(x) \), in terms of second order parabolic functions for the interval \( 0 \leq x < 1.0 \) and the sub intervals within the interval. The second order parabolic function is chosen as decomposition function since the structure is reasonable simple to implement in hardware i.e. only low complexity operations such as additions and multiplications are used.

First sub-function

The first sub-function \( s_1(x) \), is developed by dividing the original function \( f_{org}(x) \), with \( x \) as an approximation.

\[
\text{As shown in Fig. 2 there are two possible results after dividing the original function with } x, \text{ one where } f(x) > 1 \text{ and one where } f(x) < 1.
\]

\[
\text{To approximate these functions } 1/(c_1 \cdot (1-x)) \text{ is used. The first sub-function } s_1(x), \text{ is given by a multiplication of } x \text{ and } 1/(c_1 \cdot (1-x)) \text{ which results is a second order parabolic function according to (4).}
\]

\[
s_1(x) = x \cdot (1 + (c_1 \cdot (1-x))) = x + \left( c_1 \cdot \left( x - x^2 \right) \right) \quad (4)
\]

In (4) the coefficient \( c_1 \) is determined as the limit from the division of the original function with \( x \) and subtracted with 1, according to (5).

\[
c_1 = \lim_{x \to 0} \frac{f_{org}(x)}{x} - 1 \quad (5)
\]

Second sub-function

The first function \( f_1(x) \), is calculated according to (2) and the result of this operation is a function which appearance is similar to a parabolic function, as shown in Fig. 3.

\[
\text{The second sub-function } s_2(x), \text{ is chosen according to the methodology as a second order parabolic function as shown in (6).}
\]

\[
s_2(x) = 1 + \left( c_2 \cdot \left( x - x^2 \right) \right) \quad (6)
\]

In (6) the coefficient \( c_2 \), is chosen to satisfy that the quo-

\[
\text{Fig. 2 Two possible results after dividing an original function with } x.
\]

\[
\text{Fig. 3 Example of the first function } f_1(x) \text{ compared with sub-function } s_2(x).
\]
tient between the function \( f_2(x) \), and the second sub-function \( s_2(x) \), is equal to 1 when \( x \) is equal to 0.5 as shown in (7).

\[
c_2 = 4 \left( f_1\left(\frac{1}{2}\right) - 1 \right)
\]

Thereby the second function \( f_2(x) \), will get a shape of a lying S, as shown in Fig. 4.

When developing the third sub-function \( s_3(x) \), the function is to be spitted into two parabolic functions where the first function is restricted by function \( f_2(x) \) to be in the interval \( 0 \leq x < 0.5 \) and the second function is thus restricted to the interval \( 0.5 \leq x < 1.0 \). By splitting the function we get strictly convex and concave functions in each interval. The intervals can be chosen differently but that will lead to a more complex hardware, as shown in section 3.

**Sub-functions when \( n > 2 \)**

For functions \( f_n(x) \) when \( n > 2 \), the function is characterized by the form of one or more S shaped functions. When developing the higher order sub-functions, each S shaped function is divided into two parabolic functions. For each sub interval, a parabolic sub-function is developed as an approximation of the interval of the function \( f_n(x) \) in the sub interval. To show which sub interval the partial functions is valid for, the subscript index is increased with the following appearance of the partial function \( f_{n,m}(x) \). In equation (8) it is shown how the function \( f_{n,m}(x) \), is divided into partial functions \( f_{n,m}(x) \), when \( n > 2 \).

\[
f_{n}(x) = \begin{cases} f_{n,0}(x), & 0 \leq x < \frac{1}{2^n - 1} \\ f_{n,1}(x), & \frac{1}{2^n - 1} \leq x < \frac{2}{2^n - 1} \\ \vdots & \\ f_{n,2^n - 1}(x), & \frac{2^n - 1}{2^n - 1} \leq x < 1 \\ \end{cases}
\]

As shown in (8), the number of partial functions is doubled for each order of \( n > 1 \) i.e. the number of partial functions is \( 2^{n-1} \). From these partial functions, the corresponding sub-functions are developed. Analogous to the function \( f_n(x) \), also the sub-function \( s_{n+1}(x) \), will have partial sub-functions \( s_{n+1,m}(x) \). In equation (9) it is shown how the function \( s_n(x) \), is divided into partial functions when \( n > 2 \).

\[
s_n(x) = \begin{cases} s_{n,0}(s_n), & 0 \leq x < \frac{1}{2^n - 2} \\ s_{n,1}(s_n), & \frac{1}{2^n - 2} \leq x < \frac{2}{2^n - 2} \\ \vdots & \\ s_{n,2^n - 2}(x), & \frac{2^n - 2}{2^n - 2} \leq x < 1 \\ \end{cases}
\]

Note that in (9), the partial functions to the sub-functions; \( x \) has been changed to \( x_n \). The change to \( x_n \) is normalization to the corresponding interval which simplifies the hardware implementation of the parabolic function.

To simplify the normalization of the interval of \( x_n \) it is selected as an exponentiation by 2 of \( x \) where the integer part is removed. The normalization of \( x \) is therefore done by multiplying \( x \) with \( 2^{n-2} \), which in hardware is \( n-2 \) left shifts and the integer part is dropped, which gives \( x_n \) as a fractional part of \( x \), as shown in (10).

\[
s_n = \text{frac}\left(2^{n-2} \cdot x\right)
\]

As in the second sub-function \( s_2(x) \), the second order parabolic function is used as an approximation of the interval of the function \( f_{n-1}(x) \), as shown in (11).

\[
s_{n,m}(x_n) = 1 + c_{n,m} \left( x_n - s_{n} \right)^2
\]

Where the coefficients \( c_{n,m} \) is computed according to (12).

\[
c_{n,m} = 4 \cdot f_{n-1,m}\left( 2 \cdot \frac{x_n}{2^n - 1} - 1 \right)
\]

After the approximation part the result is transformed into its desired form.

### III. Hardware Implementation

For the hardware implementation two s complement representation [5] is used. The implementation is divided into three hardware parts, preprocessing, processing, and post-processing as shown in Fig. 5 and as introduced by P.T.P. Tang [1].

![Fig. 5 The hardware architecture of the methodology.](image-url)
A. Preprocessing

In this part the incoming operand \( v \) is normalized to prepare the input to the processing part, according to section 2A.

If the approximation is implemented as a block in a system the preprocessing part can be taken into consideration in the previous blocks which implies that the preprocessing part can be excluded.

B. Processing

In the processing part the approximation of the original function is directly computed in either iterative or parallel hardware architecture.

The three equations (4), (6) and (11) has the same structure which gives that the approximation can be implemented as an iterative architecture as shown in Fig. 6.

![Fig. 6](image)

**The principle of an iterative hardware architecture.**

The benefit of the iterative architecture is the small chip area whereas the disadvantage is longer computation time.

The advantages with a parallel hardware architecture is that it gives a short critical path and fast computation to the prize of a larger chip area. The principle of the parallel hardware architecture for four sub-functions is shown in Fig. 7.

![Fig. 7](image)

**The architecture principle for four sub-functions.**

To increase the throughput even more, pipeline stages can be implemented in the parallel hardware architecture.

In the sub-functions (4), (6) and (11) \( x^2 \) and \( x_n^2 \) are reoccurring operations. Since the square operation \( x_n^2 \) in the parallel hardware architecture is a partial result of \( x^2 \) a unique squarer has been developed. In Fig. 8 the method that performs the squaring and delivers partial product of \( x_n^2 \) is described.

![Fig. 8](image)

**Squarer that delivers the partial products \( x_n^2 \).**

From (4), (6) and (11) it is found that only the coefficients values differentiate when implementing different unary functions. This implies that different unary functions can be realized in the same hardware in the processing part, just by using sets of coefficients.

When implementing the processing part a limited number of sub-functions are used to accomplish the desired precision of the approximation. If the order of the last used sub-function is \( n > 1 \), an improvement of the precision can be done by optimize coefficients \( c_2 \) in (7) or \( c_{n,m} \) in (12). The optimization of coefficients will minimize the error in the last used sub-function. Such coefficient optimization are performed numerically by computer simulations.

C. Postprocessing

The postprocessing part transforms the value to the output result \( z \).

If the approximation is implemented as a block in a system the postprocessing part can be taken into consideration in the following blocks which implies that the postprocessing part can be excluded.

VI. IMPLEMENTATION OF THE SINE FUNCTION

An implementation of \( \sin(v) \), using the proposed methodology is described in this section as an example.

A. Preprocessing

To satisfy that the values of the incoming operand \( x \) is in the interval \( 0 \leq x < 1 \) a \( \pi/2 \) is multiplied with the operand as shown in (13).

\[
\nu = \frac{\pi}{2} \cdot x \quad (13)
\]

To normalize the \( \sin(\nu) \) function \( \nu \) is substituted with \( x \) which gives the original function \( f_{\text{org}}(x) \) (14).

\[
f_{\text{org}}(x) = \sin\left(\frac{2\pi}{\pi/2} \cdot x\right) \quad (14)
\]
B. Processing

For the processing part, sub-functions are developed according to the proposed methodology. For the first sub-function \( s_1(x) \), the coefficient \( c_1 \) is defined according to (5). The determined value of the coefficient is shown in (15).

\[
s_1(x) = x + \left( \frac{5}{3} - 1 \right) \left( x^2 - x^3 \right)
\]  
(15)

The first function \( f_1(x) \), is computed as shown in (16).

\[
f_1(x) = \frac{f_{arg}(x)}{s_1(x)}
\]  
(16)

When developing the second sub-function \( s_2(x) \), the coefficient \( c_2 \) is defined according to (7). The determined value of the coefficient is shown in (17).

\[
s_2(x) = 1 + \left( 0.400857 \cdot \left( x^3 - x^2 \right) \right)
\]  
(17)

The second function \( f_2(x) \), is computed as shown in (18).

\[
f_2(x) = \frac{f_{arg}(x)}{s_2(x)}
\]  
(18)

To develop the third sub-functions \( s_3(x) \), the second function \( f_2(x) \), is divided into its two partial functions as shown in (8). The third order of sub-functions is thereby divided into two sub-functions, where \( s_{3,0}(x) \) is restricted to the interval \( 0 \leq x < 0.5 \) and \( s_{3,1}(x) \) is restricted to the interval \( 0.5 \leq x < 1.0 \) according to (9). A normalization of \( x \) to \( x_3 \) is done to simplify in the implementation in hardware, which is described in (10).

For each sub-function, the corresponding coefficients \( c_{3,0} \) and \( c_{3,1} \) is determined. These coefficients are determined according to (12) so that higher order of sub-functions can be developed. The determined values of the coefficients are shown in (19).

\[
s_{3,0}(x_3) = 1 + \left( 0.0122449 \cdot \left( x_3^3 - x_3^2 \right) \right), \quad 0 \leq x < 0.5
\]
(19)

\[
s_{3,1}(x_3) = 1 + \left( 0.0105948 \cdot \left( x_3^3 - x_3^2 \right) \right), \quad 0.5 \leq x < 1
\]

The third function \( f_3(x) \), is computed as shown in (20).

\[
f_3(x) = \frac{f_{arg}(x)}{s_3(x)}
\]  
(20)

To develop the fourth sub-functions \( s_4(x) \), the third function \( f_3(x) \), is divided into its four partial functions as shown in (8). The fourth order of sub-functions is thereby divided into four sub-functions, where \( s_{4,0}(x) \) is restricted to the interval \( 0 \leq x < 0.25 \), \( s_{4,1}(x) \) is restricted to the interval \( 0.25 \leq x < 0.5 \), \( s_{4,2}(x) \) is restricted to the interval \( 0.5 \leq x < 0.75 \) and \( s_{4,3}(x) \) is restricted to the interval \( 0.75 \leq x < 1.0 \) according to (9). A normalization of \( x \) to \( x_4 \) is done to simplify the hardware implementation, which is described in (10).

For each sub-function, the corresponding coefficients \( c_{4,0} \), \( c_{4,1} \), \( c_{4,2} \) and \( c_{4,3} \) is determined. These coefficients are determined according to (12) which accomplish that higher order of sub-functions can be developed. The determined values of the coefficients are shown in (21).

\[
\begin{align*}
s_{4,0}(x_4) &= 1 + \left( -0.0022398 \cdot \left( x_4^3 - x_4^2 \right) \right), \quad 0 \leq x < 0.25 \\
s_{4,1}(x_4) &= 1 + \left( 0.00192499 \cdot \left( x_4^3 - x_4^2 \right) \right), \quad 0.25 \leq x < 0.5 \\
s_{4,2}(x_4) &= 1 + \left( -0.00119209 \cdot \left( x_4^3 - x_4^2 \right) \right), \quad 0.5 \leq x < 0.75 \\
s_{4,3}(x_4) &= 1 + \left( 0.00126505 \cdot \left( x_4^3 - x_4^2 \right) \right), \quad 0.75 \leq x < 1
\end{align*}
\]  
(21)

No postprocessing is needed since the result out from the processing part has the right size.

Optimization

If no more sub-functions are to be developed the precision of the approximation can be further improved by optimization of coefficients \( c_{4,0} \), \( c_{4,1} \), \( c_{4,2} \) and \( c_{4,3} \). As shown in Fig. 10 sub-function \( s_{4,3}(x) \) in the interval \( 0.75 \leq x < 1.0 \) has the largest relative error. Since this error can only be slightly improved by optimization no optimization of coefficients is performed.

Architecture

In Fig. 9, architecture of the approximation of the sine function using the proposed methodology is shown.

The \( x^2 \) block in Fig. 9 is the special designed multiplier described in Fig. 6 which delivers the partial results \( q_2 \) and \( q_4 \) used in the following blocks. In the \( x-q \) block, \( x \) is subtracted with the partial result \( q_1 \), from the \( x^2 \) block. The result \( r \) from the \( x-q \) block is then used in the two following blocks as shown in Fig. 9. In the \( 1+c_1 \cdot r \) block is \( s_1(x) \) performed, in \( 1+c_2 \cdot r \) is \( s_2(x) \) performed, in \( 1+(c_3 \cdot (x-q_3)) \) is \( s_3(x) \) performed and in \( 1+(c_4 \cdot (x-q_4)) \) is \( s_4(x) \) performed. Note, that in the blocks for sub-function \( s_3(x) \) and \( s_4(x) \), the individual index \( m \) is addressing the MUX that selects the coefficients in the block.
**Precision**

In Fig. 10 the resulting precision when using one to four sub-functions is shown. Beicelbe scale is used to visualize the precision since the combination of binary numbers and dB works very well together. In dB scale 2 is equal to \(20 \log_{10}(2) = 6\) dB and since 6 dB corresponds to 1 bit, this will make it simpler to understand the result. As shown in Fig. 10 is the relative error decreases with the number of used sub-functions. With 4 sub-functions we can see that we have an accuracy better that 14 bits that will result in at least a latency of 14 adders in the CORDIC algorithm is used.

Fig. 10 Estimation of the relative error between the original function and different numbers of sub-functions.

As shown in Fig. 10 the relative error decreases with the number of sub-functions used. However increases the latency with the number of sub-function as shown in Table 1.

<table>
<thead>
<tr>
<th>Number of sub-functions</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 mult + 2 add</td>
</tr>
<tr>
<td>2</td>
<td>3 mult + 2 add</td>
</tr>
<tr>
<td>3 to 4</td>
<td>4 mult + 2 add</td>
</tr>
<tr>
<td>5 to 8</td>
<td>5 mult + 2 add</td>
</tr>
</tbody>
</table>

The latency of one multiplier is about one adder.

**VI. COMPARISON**

The most common methods used when implementing approximation of a unary functions in hardware is look-up tables, polynomials, table-based methods with polynomials and CORDIC.

Computation by table look-up is attractive since memory is much denser than random logic in VLSI realizations, but since the size of the look-up table grows exponentially with increasing word lengths, both the table size and execution time becomes totally intolerable.

Computation by polynomials is attractive since it is ROM-less. The disadvantages are that it can impose large computational complexities and delays.

Computation by table-based methods combined with polynomials is attractive since it reduces the computational complexity and decreases the delays. Since the size of the look-up tables grows with the accuracy the execution time also increases with the needed accuracy.

Computation by using CORDIC is attractive since it is using an angular rotation algorithm that can be implemented with small look-up tables and a hardware which is limited to simple shifts and additions. The CORDIC algorithm is an iterative method with high latency and long delays. This makes the method insufficient for applications where short execution time is essential.

In all methods including the proposed method, it is a trade-off between complexity and memory storage. By using parallelism in the computation and parabolic synthesis in the recamination process, the proposed methodology thereby gets a short critical path which assures fast computation.

**V. CONCLUSION**

A novel methodology for implementing approximations of unary functions such as trigonometric functions, logarithmic functions, etc., in hardware is introduced in this paper. The architecture of the processing part automatically gives a high degree of parallelism. The methodology to develop the approximation algorithm is founded on parabolic synthesis. This combined with that the methodology is founded on operations that are simple to implement in hardware such as addition, shifts, multiplication, contributes to that the implementation in hardware is simple to perform. By using the parallelism and parabolic synthesis one of the most important characteristics with the outcome hardware is the parallelism that gives a short critical path and fast computation. The structure of the methodology will also assure an area efficient hardware implementation. The methodology is also suitable for automatic synthesis.

**REFERENCES**

Paper B
Parabolic Synthesis Methodology Implemented on the Sine Function

Erik Hertz, Member, IEEE
Department of Electrical and Information Technology
Lund University, Lund, Sweden
e-mail: Erik.Hertz@eit.lth.se

Peter Nilsson, Member, IEEE
Department of Electrical and Information Technology
Lund University, Lund, Sweden
e-mail: Peter.Nilsson@eit.lth.se

Abstract—This paper introduces a parabolic synthesis methodology for implementation of approximations of unary functions like trigonometric functions and logarithms, which are specialized for efficient hardware mapped VLSI design. The advantages with the methodology are, short critical path, fast computation and high throughput enabled by a high degree of architectural parallelism. The feasibility of the methodology is shown by developing an approximation of the sine function for implementation in hardware.

I. INTRODUCTION

Unary functions, such as trigonometric functions and logarithms, are extensively used in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics, etc. For these high-speed applications, software solutions are in many cases not sufficient and a hardware implementation is therefore needed. Implementing a numerical function $f(x)$, by a single look-up table is simple and fast which is straightforward for low-precision computations of $f(x)$, i.e., when $x$ only has a few bits. However, when performing high-precision computations a single look-up table implementation is impractical due to the huge table size and the long execution time. Approximations only using polynomials have the advantage of being ROM-less, but they can impose large computational complexities and delays [1]. By introducing table-based methods to the polynomials methods the computational complexity can be reduced and the delays can also be decreased to some extent [1]. The CORDIC (COordinate Rotation DIgital Computer) algorithm [2] [3] has been used for these applications since it is faster than a software approach. CORDIC is an iterative method and therefore slow which makes the method insufficient for this kind of applications.

II. METHODOLOGY

The methodology is developed for implementing approximations of unary functions in hardware. The approximation part is of course the important part of this work but there are sometimes two other steps that are necessary, a preprocessing normalization and postprocessing transformation as described by P.T.P. Tang [1]. The computation is therefore divided into three steps, normalizing, approximation and transforming.

A. Normalizing

The purpose with the normalization is to facilitate the hardware implementation by limiting the numerical range.

The normalization has to satisfy that the values are in the interval $0 \leq x < 1$ on the $x$-axis and $0 \leq y < 1$ on the $y$-axis. The coordinates of the starting point shall be $(0,0)$. Furthermore, the ending point shall have coordinates smaller than $(1,1)$ and the function must be strictly concave or convex through the interval. An example of such a function, called an original function $f_{org}(x)$, is shown in Fig. 1.

Figure 1. Example of normalized function, in this case $\sin(\pi/2)$.

B. Developing the Hardware Architecture

When developing hardware architecture that approximates an original function, only low complexity operations are used. Operations such as shifts, additions and multiplications are efficient to implement in hardware and therefore searched for. The downscaling of the semiconductor technologies and the development of efficient multiplier architectures has made the multiplication operation efficient in both size and execution time when implemented in hardware. The multiplier is therefore commonly used in this methodology when developing the hardware.

As in Fourier analysis [4] the proposed methodology is based on decomposition of basic functions. The proposed methodology is not, as in Fourier analysis, a decomposition method in terms of sinusoidal functions but in second order parabolic functions. Second order parabolic functions are used since they can be implemented using low complexity operations. The proposed methodology also differs from the Fourier synthesis process since the proposed methodology is
using multiplications in the recombination process and not additions as in the Fourier case.

The proposed methodology is founded on terms of second order parabolic functions called sub-functions \( s_n(x) \), that when recombined, as shown in (1), obtains to the original function \( f_{\text{org}}(x) \). When developing the approximate function, the accuracy depends on the number of sub-functions used.

\[
f_{\text{org}}(x) = s_1(x) \cdot s_2(x) \cdots s_n(x)
\]  

(1)

The procedure when developing sub-functions is to divide the original function \( f_{\text{org}}(x) \), with the first sub-function \( s_1(x) \). This division generates the first function \( f_1(x) \), as shown in (2).

\[
f_1(x) = \frac{f_{\text{org}}(x)}{s_1(x)}
\]  

(2)

The first sub-function \( s_1(x) \) will be chosen to be feasible for hardware, according to the methodology described in (4). In the same manner the following functions \( f_n(x) \), are generated, as shown in (3).

\[
f_{\text{org}}(x) = s_1(x) \cdot s_2(x) \cdots s_n(x)
\]  

(3)

C. Methodology for developing sub-functions

The methodology for developing sub-functions is founded on decomposition of the original function \( f_{\text{org}}(x) \), in terms of second order parabolic functions for the interval \( 0 \leq x < 1.0 \) and the sub intervals within the interval. The second order parabolic function is chosen as decomposition function since the structure is reasonable simple to implement in hardware i.e. only low complexity operations such as additions and multiplications are used.

First sub-function

The first sub-function \( s_1(x) \) is according to (4).

\[
s_1(x) = x + (c_1 \cdot (x - x'))
\]  

(4)

In (4) the coefficient \( c_1 \) is determined as the limit from the division of the original function with \( x \) and subtracted with 1, according to (5).

\[
c_1 = \lim_{x \to 0} f_{\text{org}}(x) = 1
\]  

(5)

Second sub-function

The first function \( f_1(x) \), is calculated according to (2) and the result of this operation is a function which appearance is similar to a parabolic function.

The second sub-function \( s_2(x) \), is chosen according to the methodology as a second order parabolic function, see (6).

\[
s_2(x) = 1 + (c_2 \cdot (x - x'))
\]  

(6)

In (6) the coefficient \( c_2 \) is chosen to satisfy that the quotient between the function \( f_1(x) \), and the second sub-function \( s_2(x) \), is equal to 1 when \( x \) is equal to 0.5, see (7).

\[
c_2 = 4 \cdot \left[ f_1(\frac{1}{2}) - 1 \right]
\]  

(7)

Thereby the second function \( f_2(x) \), will get a shape of a lying \( S \). When developing the third sub-function \( s_3(x) \), the function is to be split into two parabolic functions where the first function is restricted by function \( f_2(x) \) to be in the interval \( 0 \leq x < 0.5 \) and the second function is thus restricted to the interval \( 0.5 \leq x < 1.0 \). By splitting the function we get strictly convex and concave functions in each interval. The intervals can be chosen differently but that will lead to a more complex hardware, as shown in section 3.

Sub-functions when \( n > 2 \)

For functions \( f_n(x) \) when \( n > 2 \), the function is characterized by the form of one or more \( S \) shaped functions. For developing the higher order sub-functions, each \( S \) shaped function is divided into two parabolic functions. For each sub interval, a parabolic sub-function is developed as an approximation of the function \( f_n(x) \) in the sub interval. To show which sub interval the partial functions is valid for, the subscript index is increased with the index \( m \), which gives the following appearance of the partial function of the function \( f_n(x) \). In equation (8) it is shown how the function \( f_n(x) \), is divided into partial functions \( f_{n+m}(x) \), when \( n > 2 \).

\[
f_n(x) = \begin{cases} 
  f_{n,0}(x), & 0 \leq x < \frac{1}{2^{n-2}} \\
  f_{n,1}(x), & \frac{1}{2^{n-2}} \leq x < \frac{2}{2^{n-2}} \\
  \vdots & \\
  f_{n,2^{n-2}-1}(x), & \frac{2^{n-1}-1}{2^{n-2}} \leq x < 1 
\end{cases}
\]  

(8)

As shown in (8), the number of partial functions is doubled for each order of \( n > 1 \) i.e. the number of partial functions is \( 2^n \). From these partial functions, the corresponding sub-functions are developed. Analogous to the function \( f_n(x) \), also the sub-function \( s_{n+m}(x) \), will have partial sub-functions \( s_{n+m,k}(x) \). In equation (9) it is shown how the sub-function \( s_n(x) \), is divided into partial functions when \( n > 2 \).

\[
s_n(x) = \begin{cases} 
  s_{n,0}(x), & 0 \leq x < \frac{1}{2^{n-2}} \\
  s_{n,1}(x), & \frac{1}{2^{n-2}} \leq x < \frac{2}{2^{n-2}} \\
  \vdots & \\
  s_{n,2^{n-2}-1}(x), & \frac{2^{n-1}-1}{2^{n-2}} \leq x < 1 
\end{cases}
\]  

(9)

Note that in (9), the partial functions to the sub-functions; \( x \) has been changed to \( x_n \). The change to \( x_n \) is normalization to the corresponding interval, which simplifies the hardware implementation of the parabolic function. To simplify the normalization of the interval of \( x_n \), it is selected as an exponentiation by 2 of \( x \) where the integer part is removed. The normalization of \( x \) is therefore done by multiplying \( x \) with \( 2^{n-2} \), which in hardware is \( n-2 \) left shifts and the integer part is dropped, which gives \( x_n \) as a fractional part (\( \text{frac}(\ )) \) of \( x \), as shown in (10).

\[
x_n = \text{frac}(2^{n-2} \cdot x)
\]  

(10)

As in the second sub-function \( s_2(x) \), the second order parabolic function is used as an approximation of the interval of the function \( f_{\text{org}}(x) \), as shown in (11).

\[
s_{n,m}(x) = 1 + (c_{n-m} \cdot (x - x'))
\]  

(11)
Where the coefficients $c_{m,n}$ is computed according to (12).

$$c_{m,n} = 4 \left( f_{m,n} \left( \frac{2 (m+1) - 1}{2^{2^n}} \right) \right)$$

(12)

After the approximation part the result is transformed into its desired form.

### III. HARDWARE IMPLEMENTATION

For the hardware implementation two’s complement representation [5] is used. The implementation is divided into three hardware parts, preprocessing, processing, and postprocessing as introduced by P.T.P. Tang [1].

#### A. Preprocessing

In this part the incoming operand $v$ is normalized to prepare the input to the processing part, according to section 2A.

If the approximation is implemented as a block in a system the preprocessing part can be taken into consideration in the previous blocks, which implies that the preprocessing part can be excluded.

#### B. Processing

In the processing part the approximation of the original function is directly computed in either iterative or parallel hardware architecture. The benefit of the iterative architecture is the small chip area whereas the disadvantage is longer computation time.

The advantages with parallel hardware architecture are that it gives a short critical path and fast computation to the price of a larger chip area. To increase the throughput even more, pipeline stages can be implemented in the parallel hardware architecture.

In the sub-functions (4), (6) and (11) $x^2$ and $x_{n-2}$ are reoccurring operations. Since the square operation $x_{n-2}$, in the parallel hardware architecture is a partial result of $x^2$ a unique squarer has been developed. In Fig. 2 the algorithm that performs the squaring and delivers partial product of $x_{n-2}$ is described.

$$\begin{array}{cccccccc}
   x_3 & x_2 & x_1 & x_0 \\
   \times & x_3 & x_2 & x_1 & x_0 \\
   \hline
   & x_{n-2} \\
   + & x_3 & x_2 & x_1 & x_0 \\
   + & q_3 & q_2 & q_1 & p_0 \\
   + & x_0 \times \quad x_1 \times \quad x_2 \times \quad x_3 \times \\
   + & r_3 & r_2 & r_1 & r_0 \\
   + & s_3 & s_2 & s_1 & s_0 \\
\end{array}$$

Figure 2. Squaring algorithm for the partial products $x_{n-2}$.

In Fig. 2 the squaring algorithm that performs the partial products $x_{n-2}$, shown. The first partial product $p$, is the squaring of the least significant bit in $x$. The second partial product $q$, is the squaring of the two least significant bits in $x$. The partial product $r$, is the result of the squaring of the three least significant bits in $x$ and $s$ is the result of the squaring of $x$.

The squaring operation is performed with unsigned numbers. When analyzing the squarer in Fig. 2, it was found that the resemblance to a bit-serial squarer [6] [7] is large. By introducing registers in the design of the bit-serial squarer the partial results of $x_{n-2}$ is easily extracted. The squaring algorithm can thus be simplified to one addition only when computing each partial product.

From (4), (6) and (11) it is found that only the coefficients values differentiate when implementing different unary functions. This implies that different unary functions can be realized in the same hardware in the processing part, just by using sets of coefficients.

Since the methodology is calculating an approximation of the original function the error between the functions can be both positive and negative. Especially if the value of the approximation is less than the original function some extra bits of word length compared with the desired precision is needed to accomplish the desired precision of the approximation. If the order of the last used sub-function is $n > 1$, an improvement of the precision can be done by optimizing one or more coefficients $c_1$ in (7) or $c_{m,n}$ in (12). The optimization of coefficients will minimize the error in the last used sub-function and thereby it can reduce the word length needed to accomplish the desired accuracy. Computer simulations perform such coefficient optimization numerically.

#### C. Postprocessing

The postprocessing part transforms the value to the output result $z$. If the approximation is implemented as a block in a system the postprocessing part can be taken into consideration in the following blocks, which implies that the postprocessing part can be excluded.

### IV. IMPLEMENTATION OF THE SINE FUNCTION

An implementation of $\sin(v)$, using the proposed methodology is described in this section as an example.

#### A. Preprocessing

To satisfy that the values of the incoming operand $x$ is in the interval $0 \leq x < 1$ a $\pi/2$ is multiplied with the operand as shown in (13).

$$v = \frac{\pi}{2} x$$

(13)

To normalize the $\sin(v)$ function $v$ is substituted with $x$ which gives the original function $f_{org}(x)$ (14).

$$f_{org}(x) = \sin \left( \frac{\pi}{2} x \right)$$

(14)

#### B. Processing

For the processing part, sub-functions are developed according to the proposed methodology. For the first sub-function $s_1(x)$, the coefficient $c_1$ is defined according to (5). The determined value of the coefficient are, $c_1 = 0.570796$.

The first function $f_1(x)$, is computed as shown in (15).

$$f_1(x) = \frac{f_{org}(x)}{s_1(x)}$$

(15)
When developing the second sub-function \( s_2(x) \), the coefficient \( c_2 \) is defined according to (7). The determined value of the coefficient are, \( c_2 = 0.400857 \).

The second function \( f_2(x) \), is computed as shown in (16).

\[
f(x) = f_2(x) = \frac{f(x)}{s_2(x)}
\]

(16)

To develop the third sub-functions \( s_3(x) \), the second function \( f_2(x) \), is divided into its two partial functions as shown in (8). The third order of sub-functions is thereby divided into two sub-functions, where \( s_{3,0}(x) \) is restricted to the interval \( 0 \leq x < 0.5 \) and \( s_{3,3}(x) \) is restricted to the interval \( 0.5 \leq x < 1.0 \) according to (9). A normalization of \( x \) to \( x_3 \) is done to simplify in the implementation in hardware, which is described in (10).

For each sub-function, the corresponding coefficients \( c_{3,0} \) and \( c_{3,3} \) is determined. These coefficients are determined according to (12) so that higher order of sub-functions can be developed. The determined values of the coefficients are, \( c_{3,0} = 0.0122449 \) and \( c_{3,3} = 0.0105948 \).

The third function \( f_3(x) \), is computed as shown in (17).

\[
f(x) = f_3(x) = \frac{f(x)}{s_3(x)}
\]

(17)

To develop the fourth sub-functions \( s_4(x) \), the third function \( f_3(x) \), is divided into its four partial functions as shown in (8). The fourth order of sub-functions is thereby divided into four sub-functions, where \( s_{4,0}(x) \) is restricted to the interval \( 0 \leq x < 0.25 \), \( s_{4,1}(x) \) is restricted to the interval \( 0.25 \leq x < 0.5 \), \( s_{4,2}(x) \) is restricted to the interval \( 0.5 \leq x < 0.75 \) and \( s_{4,3}(x) \) is restricted to the interval \( 0.75 \leq x < 1.0 \) according to (9). A normalization of \( x \) to \( x_4 \) is done to simplify the hardware implementation, which is described in (10).

For each sub-function, the corresponding coefficients \( c_{4,0}, c_{4,1}, c_{4,2}, c_{4,3} \) is determined. These coefficients are determined according to (12) which accomplish that higher order of sub-functions can be developed. The determined values of the coefficients are, \( c_{4,0} = -0.00223398 \), \( c_{4,1} = 0.001292499 \), \( c_{4,2} = -0.00119209 \) and \( c_{4,3} = 0.00126505 \).

No postprocessing is needed since the result out from the processing part has the correct size.

C. Optimization

Only sub-function \( s_{4,3}(x) \) in the interval \( 0.75 \leq x < 1.0 \) has to be improved by optimization. By optimizing the coefficient the necessary word length to achieve the desired precision could be reduced from 17 bits to 16 bits. The optimized value of the coefficient is, \( c_{4,3} = 0.0128228 \).

V. CONCLUSION

The most common methods used when implementing approximation of a unary functions in hardware is look-up tables, polynomials, table-based methods with polynomials and CORDIC. Computation by table look-up is attractive since memory is much denser than random logic in VLSI realizations, but since the size of the look-up table grows exponentially with increasing word lengths, both the table size and execution time becomes totally intolerable. Computation by polynomials is attractive since it is ROM-less. The disadvantages are that it can impose large computational complexities and delays. Computation by table-based methods combined with polynomials is attractive since it reduces the computational complexity and decreases the delays. Since the size of the look-up tables grows with the accuracy the execution time also increases with the needed accuracy. Computation by using CORDIC is attractive since it is using an angular rotation algorithm that can be implemented with small look-up tables and hardware, which is limited to simple shifts and additions. The CORDIC algorithm is an iterative method with high latency and long delays. This makes the method insufficient for applications where short execution time is essential.

In all methods including the proposed method, it is a trade-off between complexity and memory storage. By using parallelism in the computation and parabolic synthesis in the recombination process, the proposed methodology thereby gets a short critical path, which assures fast computation.

VI. CONCLUSIONS

A novel methodology for implementing approximations of unary functions such as trigonometric functions, logarithmic functions, etc. in hardware is introduced in this paper. The architecture of the processing part automatically gives a high degree of parallelism. The methodology to develop the approximation algorithm is founded on parabolic synthesis. This combined with that the methodology is founded on operations that are simple to implement in hardware such as addition, shifts, multiplication, contributes to that the implementation in hardware is simple to perform. By using the parallelism and parabolic synthesis one of the most important characteristics with the out coming hardware is the parallelism that gives a short critical path and fast computation. The structure of the methodology will also assure an area efficient hardware implementation. The methodology is also suitable for automatic synthesis.

REFERENCES

Paper C
A Methodology for Parabolic Synthesis

Erik Hertz and Peter Nilsson

Department of Electrical and Information Technology, Lund University
Sweden

1. Introduction

In relatively recent research of the history of science interpolation theory, in particular of mathematical astronomy, revealed rudimentary solutions of interpolation problems date back to early antiquity (Meijering, 2002). Examples of interpolation techniques originally conceived by ancient Babylonian as well as early-medieval Chinese, Indian, and Arabic astronomers and mathematicians can be linked to the classical interpolation techniques developed in Western countries from the 17th until the 19th century. The available historical material has not yet given a reason to suspect that the earliest known contributors to classical interpolation theory were influenced in any way by mentioned ancient and medieval Eastern works. For the classical interpolation theory it is justified to say that there is no single person who did so much for this field as Newton. Therefore, Newton deserves the credit for having put classical interpolation theory on a foundation. In the course of the 18th and 19th century Newton’s theories were further studied by many others, including Stirling, Gauss, Waring, Euler, Lagrange, Bessel, Laplace, and Everett. Whereas the developments until the end of 19th century had been impressive, the developments in the past century have been explosive. Another important development from the late 1800s is the rise of approximation theory. In 1885, Weierstrass justified the use of approximations by establishing the so-called approximation theorem, which states that every continuous function on a closed interval can be approximated uniformly to any prescribed accuracy by a polynomial. In the 20th century two major extensions of classical interpolation theory is introduced: firstly the concept of the cardinal function, mainly due to E. T. Whittaker, but also studied before him by Borel and others, and eventually leading to the sampling theorem for band limited functions as found in the works of J. M. Whittaker, Kotel’nikov, Shannon, and several others, and secondly the concept of oscillatory interpolation, researched by many and eventually resulting in Schoenberg’s theory of mathematical splines.

The parabolic synthesis methodology

Unary functions, such as trigonometric functions, logarithms as well as square root and division functions are extensively used in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics, etc. For these high-speed applications, software solutions are in many cases not sufficient and a hardware implementation is therefore needed. Implementing a numerical function \( f(x) \), by a single
look-up table (Tang, 1991) is simple and fast which is strait forward for low-precision computations of \( f(x) \), i.e., when \( x \) only has a few bits. However, when performing high-precision computations a single look-up table implementation is impractical due to the huge table size and the long execution time.

Approximations only using polynomials have the advantage of being ROM-less, but they can impose large computational complexities and delays (Muller, 2006). By introducing table based methods to the polynomials methods the computational complexity can be reduced and the delays can also be decreased to some extent (Muller, 2006).

The CORDIC (COordinate Rotation Digital Computer) algorithm (Volder, 1959) (Andrata, 1998) has been used for these applications since it is faster than a software approach. CORDIC is an iterative method and therefore slow which makes the method insufficient for this kind of applications.

The proposed methodology of parabolic synthesis (Hertz & Nilsson, 2008) develops functions that perform an approximation of original functions in hardware. The architecture of the processing part of the methodology is using parallelism to reduce the execution time. For the development of approximations of functions a parabolic synthesis methodology has been applied. Only low complexity operations that are simple to implement in hardware are used.

2. Methodology

![Graph of normalized function](image)

Fig. 1. Example of normalized function, in this case \( \sin \left( \frac{\pi \cdot x}{2} \right) \).
The methodology is developed for implementing approximations of unary functions in hardware. The approximation part is of course the important part of this work but there are sometimes two other steps that are necessary, a preprocessing normalization and postprocessing transformation as described by (P.T.P. Tang, 1991) (Muller, 2006). The computation is therefore divided into three steps, normalizing, approximation and transforming.

2.1 Normalizing
The purpose with the normalization is to facilitate the hardware implementation by limiting the numerical range. The normalization has to satisfy that the values are in the interval $0 \leq x < 1$ on the $x$-axis and $0 \leq y < 1$ on the $y$-axis. The coordinates of the starting point shall be $(0,0)$. Furthermore, the ending point shall have coordinates smaller than $(1,1)$ and the function must be strictly concave or strictly convex through the interval. An example of such a function, called an original function $f_{\text{org}}(x)$, is shown in Fig. 1.

2.2 Developing the Hardware Architecture
When developing a hardware architecture that approximates an original function, only low complexity operations are used. Operations such as shifts, additions and multiplications are efficient to implement in hardware and therefore searched for. The downscaling of the semiconductor technologies and the development of efficient multiplier architectures has made the multiplication operation efficient in both size and execution, time when implemented in hardware. The multiplier is therefore commonly used in this methodology when developing the hardware.

As in Fourier analysis (Fourier, 1822) the proposed methodology is based on decomposition of basic functions. The proposed methodology is not, as in Fourier analysis, a decomposition method in terms of sinusoidal functions but in second order parabolic functions. Second order parabolic functions are used since they can be implemented using low complexity operations. The proposed methodology also differs from the Fourier synthesis process since the proposed methodology is using multiplications in the recombination process and not additions as in the Fourier case.

The proposed methodology is founded on terms of second ordered parabolic functions called sub-functions $s_n(x)$, that when recombined, as shown in (1), obtains to the original function $f_{\text{org}}(x)$. When developing the approximate function, the accuracy depends on the number of sub-functions used.

$$f_{\text{org}}(x) = s_1(x) \cdot s_2(x) \cdots s_n(x)$$  \hspace{1cm} (1)

The procedure when developing sub-functions is to divide the original function $f_{\text{org}}(x)$, with the first sub-function $s_1(x)$. This division generates the first function $f_1(x)$, as shown in (2).

$$f_1(x) = \frac{f_{\text{org}}(x)}{s_1(x)}$$  \hspace{1cm} (2)
The first sub-function $s_1(x)$, will be chosen to be feasible for hardware, according to the methodology described in (4). In the same manner the following functions $f_n(x)$, are generated, as shown in (3).

$$f_{s_1}(x) = \frac{f(x)}{s_{s_1}(x)}$$

(3)

The purpose with the normalization is to facilitate the hardware implementation by limiting the numerical range.

2.3 Methodology for developing sub-functions

The methodology for developing sub-functions is founded on decomposition of the original function $f_{org}(x)$, in terms of second order parabolic functions for the interval $0 \leq x < 1.0$ and the sub intervals within the interval. The second order parabolic function is chosen as decomposition function since the structure is reasonable simple to implement in hardware i.e. only low complexity operations such as additions and multiplications are used.

First sub-function

The first sub-function $s_1(x)$, is developed by dividing the original function $f_{org}(x)$, with $x$ as an approximation.

As shown in Fig. 2 there are two possible results after dividing the original function with $x$, one where $f(x)>1$ and one where $f(x)<1$. 
Fig. 2. Two possible results after dividing an original function with \( x \).

The first sub-function \( s_1(x) \), is according to (4). To approximate these functions \( 1+(c_1 (1-x)) \) is used. The first sub-function \( s_1(x) \), is given by a multiplication of \( x \) and \( 1+(c_1 (1-x)) \) which results in a second order parabolic function according to (4).

\[
s_1(x) = x \cdot (1 + (c_1 \cdot (1-x))) = x + (c_1 \cdot (x - x^2))
\]  

(4)

In (4) the coefficient \( c_1 \) is determined as the limit from the division of the original function with \( x \) and subtracted with 1, according to (5).

\[
c_i = \lim_{x \to 0} \frac{f(x)}{x} - 1
\]  

(5)

**Second sub-function**

The first function \( f_1(x) \), is calculated according to (2) and the result of this operation is a function which appearance is similar to a parabolic function, as shown in Fig. 3.

![Diagram](image_url)

Fig. 3. Example of the first function \( f_1(x) \) compared with sub-function \( s_2(x) \).

The second sub-function \( s_2(x) \), is chosen according to the methodology as a second order parabolic function, see (6).

\[
s_2(x) = 1 + (c_2 \cdot (x - x^2))
\]  

(6)
In (6) the coefficient \( c_2 \) is chosen to satisfy that the quotient between the function \( f_1(x) \), and the second sub-function \( s_2(x) \), is equal to 1 when \( x \) is equal to 0.5, see (7).

\[
c_2 = 4 \cdot f_1 \left( \frac{1}{2} \right) - 1
\]

(7)

Thereby the second function \( f_2(x) \), will get a shape of a lying S, as shown in Fig. 4.

Fig. 4. Example of the second function \( f_2(x) \), shaped like a lying S.

When developing the third sub-function \( s_3(x) \), the function is to be split into two parabolic functions where the first function is restricted by function \( f_2(x) \) to be in the interval \( 0 \leq x < 0.5 \) and the second function is thus restricted to the interval \( 0.5 \leq x < 1.0 \). By splitting the function we get strictly convex and concave functions in each interval. The intervals can be chosen differently but that will lead to a more complex hardware, as shown in section 3.

Sub-functions when \( n > 2 \)

For functions \( f_n(x) \) when \( n > 2 \), the function is characterized by the form of one or more S shaped functions. When developing the higher order sub-functions, each S shaped function is divided into two parabolic functions. For each sub interval, a parabolic sub-function is developed as an approximation of the function \( f_n(x) \) in the sub interval. To show which sub
interval the partial functions is valid for, the subscript index is increased with the index \( m \), which gives the following appearance of the partial function \( f_{n,m}(x) \).

\[
f_n(x) = \begin{cases} 
  f_{n,0}(x), & 0 \leq x < \frac{1}{2^{n-1}} \\
  f_{n,1}(x), & \frac{1}{2^{n-1}} \leq x < \frac{2}{2^{n-1}} \\
  \vdots \\
  f_{n,2^{n-1}-1}(x), & \frac{2^{n-1}-1}{2^{n-1}} \leq x < 1
\end{cases}
\]  

In equation (8) it is shown how the function \( f_n(x) \), is divided into partial functions \( f_{n,m}(x) \), when \( n > 2 \).

As shown in (8), the number of partial functions is doubled for each order of \( n > 1 \) i.e. the number of partial functions is \( 2^{n-1} \). From these partial functions, the corresponding sub-functions are developed. Analogous to the function \( f_n(x) \), also the sub-function \( s_{n+1}(x) \), will have partial sub-functions \( s_{n+1,m}(x) \). In equation (9) it is shown how the sub-function \( s_n(x) \), is divided into partial functions when \( n > 2 \).

\[
s_n(x) = \begin{cases} 
  s_{n,0}(x), & 0 \leq x < \frac{1}{2^{n-2}} \\
  s_{n,1}(x), & \frac{1}{2^{n-2}} \leq x < \frac{2}{2^{n-2}} \\
  \vdots \\
  s_{n,2^{n-2}-1}(x), & \frac{2^{n-2}-1}{2^{n-2}} \leq x < 1
\end{cases}
\]  

Note that in (9), the partial functions to the sub-functions; \( x \) has been changed to \( x_n \). The change to \( x_n \) is normalization to the corresponding interval, which simplifies the hardware implementation of the parabolic function. To simplify the normalization of the interval of \( x_n \) it is selected as an exponentiation by 2 of \( x \) where the integer part is removed. The normalization of \( x \) is therefore done by multiplying \( x \) with \( 2^{n-2} \), which in hardware is \( n-2 \) left shifts and the integer part is dropped, which gives \( x_n \) as a fractional part (\( \text{frac}(\ )) \) of \( x \), as shown in (10).

\[
x_n = \text{frac}(2^{n-2} \cdot x)
\]  

As in the second sub-function \( s_2(x) \), the second order parabolic function is used as an approximation of the interval of the function \( f_{n-1}(x) \), as shown in (11).

\[
s_{n,m}(x_n) = 1 + \left( c_{n,m} \left( x_n^2 - x_n^2 \right) \right)
\]  

Where the coefficients \( c_{n,m} \) is computed according to (12).
\[ c_{n,m} = 4 \cdot \left( 2^{(m+1)} \left( \frac{2^{n+1}-1}{2^{n-1}} \right) - 1 \right) \]  

(12)

After the approximation part the result is transformed into its desired form.

3. Hardware Implementation

For the hardware implementation two’s complement representation (Parhami, 2000) is used. The implementation is divided into three hardware parts, preprocessing, processing, and postprocessing as shown in Fig. 5, which was introduced by (P.T.P. Tang, 1991), (Muller, 2006).

![Hardware Architecture Diagram]

Fig. 5. The hardware architecture of the methodology.

3.1 Preprocessing

In this part the incoming operand \( v \) is normalized to prepare the input to the processing part, according to section 2.1.

If the approximation is implemented as a block in a system the preprocessing part can be taken into consideration in the previous blocks, which implies that the preprocessing part can be excluded.

3.2 Processing

In the processing part the approximation of the original function is directly computed in either iterative or parallel hardware architecture.

The three equations (4), (6) and (11) has the same structure which gives that the approximation can be implemented as an iterative architecture as shown in Fig. 6.
The benefit of the iterative architecture is the small chip area whereas the disadvantage is longer computation time.

The advantages with parallel hardware architecture are that it gives a short critical path and fast computation to the prize of a larger chip area. The principle of the parallel hardware architecture for four sub-functions is shown in Fig. 7.

Fig. 6. The principle of an iterative hardware architecture.

Fig. 7. The architecture principle for four sub-functions.
Fig. 8. Squaring algorithm for the partial products \( x_0^2 \).

To increase the throughput even more, pipeline stages can be implemented in the parallel hardware architecture.
In the sub-functions (4), (6) and (11) \( x^2 \) and \( x_0^2 \) are reoccurring operations. Since the square operation \( x_0^2 \), in the parallel hardware architecture is a partial result of \( x^2 \) a unique squarer has been developed. In Fig. 8 the algorithm that performs the squaring and delivers partial product of \( x_0^2 \) is described.
The squaring algorithm for the partial products \( x_0^2 \) can be simplified as shown in Fig. 9.
In Fig. 8 and Fig. 9, the squaring algorithm that performs the partial products $x_n^2$, shown. The first partial product $p$, is the squaring of the least significant bit in $x$. The second partial product $q$, is the squaring of the two least significant bits in $x$. The partial product $r$, is the result of the squaring of the three least significant bits in $x$ and $s$ is the result of the squaring of $x$. The squaring operation is performed with unsigned numbers. When analyzing the squarer in Fig. 8 and Fig. 9, it was found that the resemblance to a bit-serial squarer (Ienne & Viredaz, 1994) (Pekmestz et al., 2001) is large. By introducing registers in the design of the bit-serial squarer the partial results of $x_n^2$ is easily extracted. The squaring algorithm can thus be simplified to one addition only when computing each partial product.

From (4), (6) and (11) it is found that only the coefficients values differentiate when implementing different unary functions. This implies that different unary functions can be realized in the same hardware in the processing part, just by using different sets of coefficients.

Since the methodology is calculating an approximation of the original function the error to the desired precision can be both positive and negative. Especially, if the value of the approximation is less than the desired precision, the word length can have to be increased compared with the word length needed to accomplish the desired precision. If the order of the last used sub-function is $n > 1$, an improvement of the precision can be done by optimizing one or more coefficients $c_2$ in (7) or $c_{n,m}$ in (12). The optimization of the coefficients will minimize the error in the last used sub-function and thereby it can reduce the word length needed to accomplish the desired accuracy. Computer simulations perform such coefficient optimization numerically.

### 3.3 Postprocessing

The postprocessing part transforms the value to the output result $z$. If the approximation is implemented as a block in a system the postprocessing part can be taken into consideration in the following blocks, which implies that the postprocessing part can be excluded.
4. Implementation of the sine function

An implementation of the function \( \sin(v) \), using the proposed methodology (Hertz & Nilsson, 2009) is described in this section as an example.

4.1 Preprocessing

![Graph showing \( f(v) \) before normalization and the original function \( f_{\text{org}}(x) \).]

Fig. 10. The function \( f(v) \) before normalization and the original function \( f_{\text{org}}(x) \).

To satisfy that the values of the incoming operand \( x \) is in the interval \( 0 \leq x < 1 \) a \( \pi/2 \) is multiplied with the operand as shown in (13).

\[
v = \frac{\pi}{2} x \quad \quad (13)
\]

To normalize the \( f(v) = \sin(v) \) function \( v \) is substituted with \( x \) which gives the original function \( f_{\text{org}}(x) \) (14).

\[
f_{\text{org}}(x) = \sin\left(\frac{\pi}{2} x\right) \quad \quad (14)
\]

In Fig. 10 the \( f(v) \) function is shown together with the original function \( f_{\text{org}}(x) \).
4.2 Processing

For the processing part, sub-functions are developed according to the proposed methodology. For the first sub-function $s_1(x)$, the coefficient $c_1$ is defined according to (5). The determined value of the coefficient is shown in (15).

$$s_1(x) = x + \left( \frac{\pi}{2} - 1 \right) \left( x - x^3 \right)$$  \hspace{1cm} (15)

The first function $f_1(x)$, is computed as shown in (16).

$$f_1(x) = \frac{f_{se}(x)}{s_1(x)}$$  \hspace{1cm} (16)

To develop the second sub-function $s_2(x)$, the coefficient $c_2$ is defined according to (7). The determined value of the coefficient is shown in (17).

$$s_2(x) = 1 + \left( 0.400858 \right) \left( x - x^3 \right)$$  \hspace{1cm} (17)

The second function $f_2(x)$, is computed as shown in (18).

$$f_2(x) = \frac{f_2(x)}{s_2(x)}$$  \hspace{1cm} (18)

To develop the third sub-functions $s_3(x)$, the second function $f_2(x)$, is divided into its two partial functions as shown in (8). The third order of sub-functions is thereby divided into two sub-functions, where $s_{3,0}(x)$ is restricted to the interval $0 \leq x < 0.5$ and $s_{3,1}(x)$ is restricted to the interval $0.5 \leq x < 1.0$ according to (9). A normalization of $x$ to $x_3$ is done to simplify in the implementation in hardware, which is described in (10).

For each sub-function, the corresponding coefficients $c_{3,0}$ and $c_{3,1}$ is determined. These coefficients are determined according to (12) where higher order sub-functions can be developed. The determined values of the coefficients are shown in (19).

$$s_{3,0}(x) = 1 + \left( -0.0122452 \right) \left( x_3 - x_3^3 \right) \hspace{1cm} 0 \leq x < 0.5$$  \hspace{1cm} (19)

$$s_{3,1}(x) = 1 + \left( 0.0105947 \right) \left( x_3 - x_3^3 \right) \hspace{1cm} 0.5 \leq x < 1$$

The third function $f_3(x)$, is computed as shown in (20).

$$f_3(x) = \frac{f_3(x)}{s_3(x)}$$  \hspace{1cm} (20)

To develop the fourth sub-functions $s_4(x)$, the third function $f_3(x)$, is divided into its four partial functions as shown in (8). The fourth order of sub-functions is thereby divided into four sub-functions, where $s_{4,0}(x_4)$ is restricted to the interval $0 \leq x < 0.25$, $s_{4,1}(x_4)$ is restricted
to the interval $0.25 \leq x < 0.5$, $s_{4,2}(x_d)$ is restricted to the interval $0.5 \leq x < 0.75$ and $s_{4,3}(x_d)$ is restricted to the interval $0.75 \leq x < 1.0$ according to (9). A normalization of $x$ to $x_d$ is done to simplify the hardware implementation, which is described in (10).

For each sub-function, the corresponding coefficients $c_{4,0}$, $c_{4,1}$, $c_{4,2}$ and $c_{4,3}$ is determined. These coefficients are determined according to (12) which accomplish that higher order of sub-functions can be developed. The determined values of the coefficients are shown in (21).

\[
\begin{align*}
    s_{4,0}(x_d) &= 1 + \left( -0.00222363 \left( x_d - x_4^2 \right) \right) \quad 0 \leq x < 0.25 \\
    s_{4,1}(x_d) &= 1 + \left( 0.00192558 \left( x_d - x_4^2 \right) \right) \quad 0.25 \leq x < 0.5 \\
    s_{4,2}(x_d) &= 1 + \left( -0.00119216 \left( x_d - x_4^2 \right) \right) \quad 0.5 \leq x < 0.75 \\
    s_{4,3}(x_d) &= 1 + \left( 0.00126487 \left( x_d - x_4^2 \right) \right) \quad 0.75 \leq x < 1
\end{align*}
\]  

\hspace{1cm} (21)

No postprocessing is needed since the result out from the processing part has the right size.

4.3 Optimization

If no more sub-functions are to be developed the precision of the approximation can be further improved by optimization of coefficients $c_{4,0}$, $c_{4,1}$, $c_{4,2}$ and $c_{4,3}$. As shown in Fig. 12 sub-function $s_{4,3}(x)$ in the interval $0.75 \leq x < 1.0$ has the largest relative error. When performing an optimization of sub-function $s_{4,3}(x)$ in the interval $0.75 \leq x < 1.0$ it was found that the word length in the computations could be reduced from 17 bits to 16 bits.

4.4 Architecture

In Fig. 11, architecture of the approximation of the sine function using the proposed methodology is shown.

The $x^2$ block in Fig. 11 is the special designed multiplier described in Fig. 8 and Fig. 9 that delivers the partial results $q_1$, $q_2$ and $q_4$ used in the following blocks. In the $x$-$q$ block, $x$ is subtracted with the partial result $q_1$ from the $x^2$ block. The result $r$ from the $x$-$q$ block is then used in the two following blocks as shown in Fig. 11. In the $x+(c_1:r)$ block is $s_1(x)$ performed, in $1+\left(c_2:r\right)$ is $s_2(x)$ performed, in $1+\left(c_3:\left(x_3-q_3\right)\right)$ is $s_3(x)$ performed and in $1+(c_4:\left(x_4-q_4\right))$ is $s_4(x)$ performed. Note, that in the blocks for sub-function $s_3(x)$ and $s_4(x)$, the individual index $m$ is addressing the MUX that selects the coefficients in the block.

4.5 Optimization of Word Length

As shown in (19) and (21) the absolute value of the coefficients decreases in size with increasing index number of the coefficient. In similarity to the word length of the coefficients the word length of the $(x_n-q_n)$ part, shown in Fig. 11, will decrease in size with increasing index number. The decreased word length will cause that the size of the multiplier used in a sub-function to decreases accordingly to the highest value bit in the coefficients and of the $(x_n-q_n)$ part. In resemblance to above the size of the multipliers computing the multiplication of the sub-functions can be analyzed. This analysis will also result in that some of the following multipliers accordingly can be decrease in size.
Fig. 11. The architecture of the implementation of the sine function.

4.6 Precision
In Fig. 12 the resulting precision when using one to four sub-functions is shown. Decibel scale is used to visualize the precision since the combination of binary numbers and dB works very well together. In dB scale 2 is equal to $20\log_{10}(2) = 20 \cdot (0.3) = 6$ dB and since 6 dB corresponds to 1 bit, this will make it simpler to understand the result. As shown in Fig. 12, the relative error decreases with the number of used sub-functions. With 4 sub-functions we can see that we have accuracy better that 14 bits that will result in at least a latency of 14 adders in the CORDIC algorithm is used.
Fig. 12. Estimation of the relative error between the original function and different numbers of sub-functions.

As shown in Fig. 12, the relative error decreases with the number of sub-functions used. However, increases the delay with the number of sub-function as shown in Table 1.

<table>
<thead>
<tr>
<th>Number of sub-functions</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2 mult + 2 add</td>
</tr>
<tr>
<td>2</td>
<td>3 mult + 2 add</td>
</tr>
<tr>
<td>3 to 4</td>
<td>4 mult + 2 add</td>
</tr>
<tr>
<td>5 to 8</td>
<td>5 mult + 2 add</td>
</tr>
</tbody>
</table>

Table 1. Delay relative the number of sub-functions.

The delay of one multiplier is about two adders.

5. Comparison

The most common methods used when implementing approximation of a unary functions in hardware are look-up tables, polynomials, table-based methods with polynomials and CORDIC. Computation by table look-up is attractive since memory is much denser than random logic in VLSI realizations. However, since the size of the look-up table grows exponentially with increasing word lengths, both the table size and execution time becomes totally intolerable. Computation by polynomials is attractive since it is ROM-less. The disadvantages are that it can impose large computational complexities and delays.
Computation by table-based methods combined with polynomials is attractive since it reduces the computational complexity and decreases the delays. Since the size of the look-up tables grows with the accuracy the execution time also increases with the needed accuracy. Computation by using CORDIC is attractive since it is using an angular rotation algorithm that can be implemented with small look-up tables and hardware, which is limited to simple shifts and additions. The CORDIC algorithm is an iterative method with high latency and long delays. This makes the method insufficient for applications where short execution time is essential.

In all methods including the proposed method, it is a trade-off between complexity and memory storage. By using parallelism in the computation and parabolic synthesis in the recombination process, the proposed methodology thereby gets a short critical path, which assures fast computation.

6. Using the Methodology

It has been shown that the methodology of parabolic synthesis can directly compute the sine function but the methodology is also able to compute other trigonometric functions, logarithms as well as square root and division. In the following parts algorithms for elementary functions will be shown.

When describing the implementation of each function the different parts are shown in a table. The first row in the table shows the function to be implemented and in which interval the function is implemented. In the second row it is described how to perform the normalization of the function. The third row shows the original function to be used when developing the approximation. The last row describes how to perform transformation of the approximation into desired interval.

6.1 The Sine Function

When developing the algorithm that performs the approximation of the sine function, the normalization in the preprocessing part is performed as a substitution according to Table 2. Since the outcome of the approximation has the desired form no postprocessing is needed.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>( f(\nu) = \sin(\nu) )</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>( x = \frac{2}{\pi} \cdot \nu )</td>
</tr>
<tr>
<td>Processing</td>
<td>( y = \sin\left(\frac{\pi}{2} \cdot x\right) )</td>
</tr>
<tr>
<td>Postprocessing</td>
<td>( z = y )</td>
</tr>
</tbody>
</table>

Table 2. The algorithms for the sine function.

6.2 The Cosine Function

The algorithm that performs the approximation of the cosine function is founded on the algorithm that performs the approximation of the sine function. To perform the
approximation of the cosine function $x$ is substituted with $1-x$ in the preprocessing part of the approximation for the sine function.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>$f(v) = \cos(v)$</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>$x = 1 - \frac{2}{\pi} \cdot v$</td>
</tr>
<tr>
<td>Processing</td>
<td>$y = \sin\left(\frac{\pi}{2} \cdot x\right)$</td>
</tr>
<tr>
<td>Postprocessing</td>
<td>$z = y$</td>
</tr>
</tbody>
</table>

Table 3. The algorithm for the cosine function.

### 6.3 The Arcsine Function

When developing the algorithm that performs the approximation of the arcsine function, the methodology has problems to perform an approximation for angles larger than $\pi/4$. Therefore, the range of the approximation has been limited according to the range of the function in Table 4. To satisfy the requirements of the methodology in the preprocessing part a substitution according to Table 4 has to be performed. To get the desired outcome the approximation is multiplied with a factor according to Table 4.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>$f(v) = \arcsin(v)$</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>$x = \sqrt{2} \cdot v$</td>
</tr>
<tr>
<td>Processing</td>
<td>$y = \arcsin\left(\frac{x}{\sqrt{2}}\right)$</td>
</tr>
<tr>
<td>Postprocessing</td>
<td>$z = \frac{\pi}{4} \cdot y$</td>
</tr>
</tbody>
</table>

Table 4. The algorithm for the arcsine function.

### 6.4 The Arccosine Function

The algorithm that performs the approximation of the arccosine function is founded on the algorithm performing the approximation of the arcsine function. The difference between the two approximations is in the transformation in the postprocessing part, as shown in Table 5.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>$f(v) = \arccos(v)$</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>$x = \sqrt{2} \cdot v$</td>
</tr>
<tr>
<td>Processing</td>
<td>$y = \arccos\left(\frac{x}{\sqrt{2}}\right)$</td>
</tr>
<tr>
<td>Postprocessing</td>
<td>$z = \frac{\pi}{2} - y$</td>
</tr>
</tbody>
</table>

Table 5. The algorithm for the arccosine function.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(v) = \arccos(v)$</td>
<td>$0 \leq v &lt; \frac{1}{\sqrt{2}}$</td>
</tr>
<tr>
<td>$x = \sqrt{2} \cdot v$</td>
<td>$0 \leq x &lt; 1$</td>
</tr>
<tr>
<td>$y = \arcsin\left(\frac{x}{\sqrt{2}}\right)$</td>
<td>$0 \leq y &lt; 1$</td>
</tr>
<tr>
<td>$z = \pi \cdot \frac{\pi}{4} \cdot (1 - y)$</td>
<td>$\frac{\pi}{4} &lt; z \leq \frac{\pi}{2}$</td>
</tr>
</tbody>
</table>

Table 5. The algorithm for the arccosine function.

### 6.5 The Tangent Function

When developing the algorithm that performs the approximation of the tangent function the angle range is from 0 to $\pi/4$, since the tangent function is not strictly concave or convex for higher angles. To perform the normalization the preprocessing part is performed as a substitution according to Table 6. Since the outcome of the approximation has the desired form no postprocessing is needed.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(v) = \tan(v)$</td>
<td>$0 \leq v &lt; \frac{\pi}{4}$</td>
</tr>
<tr>
<td>$x = \frac{4 \cdot v}{\pi}$</td>
<td>$0 \leq x &lt; 1$</td>
</tr>
<tr>
<td>$y = \tan\left(\frac{\pi}{4} \cdot x\right)$</td>
<td>$0 \leq y &lt; 1$</td>
</tr>
<tr>
<td>$z = y$</td>
<td>$0 \leq z &lt; 1$</td>
</tr>
</tbody>
</table>

Table 6. The algorithm for the tangent function.

### 6.6 The Arctangent Function

When developing the algorithm that performs the approximation of the arctangent function it can only be performed in the range from 0 to 1 where the function is strictly concave or convex. To get the desired outcome the approximation is in the postprocessing part multiplied with a factor according to Table 7.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(v) = \arctan(v)$</td>
<td>$0 \leq v &lt; 1$</td>
</tr>
<tr>
<td>$x = v$</td>
<td>$0 \leq x &lt; 1$</td>
</tr>
<tr>
<td>$y = \arctan(x) \cdot \frac{4}{\pi}$</td>
<td>$0 \leq y &lt; 1$</td>
</tr>
<tr>
<td>$z = \frac{\pi}{4} \cdot y$</td>
<td>$0 \leq z &lt; \frac{\pi}{4}$</td>
</tr>
</tbody>
</table>

Table 7. The algorithm for the arctangent function.
6.7 The Logarithmic Function
When developing the algorithm that performs the approximation of the logarithm function with the base two, it is only perform on the mantissa of the floating-point number, since the exponent part is scaling the mantissa. For the preprocessing part a substitution according to Table 8 has to be performed to satisfy the normalization criteria’s for the methodology. Since the outcome of the approximation has the desired form no postprocessing is needed.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>$f(v) = \log_{2}(v)$</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>$x = v - 1$</td>
</tr>
<tr>
<td>Processing</td>
<td>$y = \log_{2}(1 + x)$</td>
</tr>
<tr>
<td>Postprocessing</td>
<td>$z = y$</td>
</tr>
</tbody>
</table>

Table 8. The algorithm for the logarithm function.

6.8 The Exponential Function
When developing the algorithm that performs the approximation of the exponential function with the base two, it is only performed on the fractional part of the logarithm since the integer part is scaling the fractional part of the logarithm. As shown in Table 9 only a one needs to be added in the postprocessing part to get the desired outcome.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>$f(v) = 2^v$</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>$x = v$</td>
</tr>
<tr>
<td>Processing</td>
<td>$y = 2^x - 1$</td>
</tr>
<tr>
<td>Postprocessing</td>
<td>$z = 1 + y$</td>
</tr>
</tbody>
</table>

Table 9. The algorithm for the exponential function.

6.9 The Division Function
When developing the algorithm that performs the approximation of the division it is limited to the range according to Table 10, since the division is not strictly concave or convex outside this range. The pre- and post-processing part both needs computation when performing the approximation of the division.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>$f(v) = \frac{1}{1 + v}$</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>$x = 2 \cdot (1 - v)$</td>
</tr>
<tr>
<td>Processing</td>
<td>$y = \frac{6}{1 + \left(1 - \frac{x}{2}\right)} - 3$</td>
</tr>
<tr>
<td>Postprocessing</td>
<td>$z = \frac{3 + y}{6}$</td>
</tr>
</tbody>
</table>

Table 10. The algorithm for the division function.
6.10 The Square Root Function
When developing the algorithm that performs the approximation of the square root function the range is limited according to Table 11. The pre- and post-processing part both needs computation when performing the approximation of the square root function.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function</td>
<td>$f(v) = \sqrt{1 + v}$</td>
</tr>
<tr>
<td>Preprocessing</td>
<td>$x = y - 1$</td>
</tr>
<tr>
<td>Processing</td>
<td>$y = \frac{\sqrt{2 + x} - \sqrt{2}}{\sqrt{3} - \sqrt{2}}$</td>
</tr>
<tr>
<td>Postprocessing</td>
<td>$z = \sqrt{2 + y} \left( \sqrt{3} - \sqrt{2} \right)$</td>
</tr>
</tbody>
</table>

Table 11. The algorithm for the square root function.

7. Conclusions
A novel methodology for implementing approximations of unary functions such as trigonometric functions, logarithmic functions, as well as square root and division functions etc. in hardware is introduced. The architecture of the processing part automatically gives a high degree of parallelism. The methodology to develop the approximation algorithm is founded on parabolic synthesis. This combined with that the methodology is founded on operations that are simple to implement in hardware such as addition, shifts, multiplication, contributes to that the implementation in hardware is simple to perform. By using the parallelism and parabolic synthesis, one of the most important characteristics with the upcoming hardware is the parallelism that gives a short critical path and fast computation. The structure of the methodology will also assure an area efficient hardware implementation. The methodology is also suitable for automatic synthesis.

8. References
J. Fourier (1822), *Théorie Analytique de la Chaleur*, Paris, France.


The process of Integrated Circuits (IC) started its era of VLSI (Very Large Scale Integration) in 1970’s when thousands of transistors were integrated into one single chip. Nowadays we are able to integrate more than a billion transistors on a single chip. However, the term “VLSI” is still being used, though there was some effort to coin a new term ULSI (Ultra-Large Scale Integration) for fine distinctions many years ago. VLSI technology has brought tremendous benefits to our everyday life since its occurrence. VLSI circuits are used everywhere, real applications include microprocessors in a personal computer or workstation, chips in a graphic card, digital camera or camcorder, chips in a cell phone or a portable computing device, and embedded processors in an automobile, et al. VLSI covers many phases of design and fabrication of integrated circuits. For a commercial chip design, it involves system definition, VLSI architecture design and optimization, RTL (register transfer language) coding, (pre- and post-synthesis) simulation and verification, synthesis, place and route, timing analyses and timing closure, and multi-step semiconductor device fabrication including wafer processing, die preparation, IC packaging and testing, et al. As the process technology scales down, hundreds or even thousands of millions of transistors are integrated into one single chip. Hence, more and more complicated systems can be integrated into a single chip, the so-called System-on-chip (SoC), which brings to VLSI engineers ever increasingly challenges to master techniques in various phases of VLSI design. For modern SoC design, practical applications are usually speed hungry. For instance, Ethernet standard has evolved from 10Mbps to 10Gbps. Now the specification for 100Mbps Ethernet is on the way. On the other hand, with the popularity of wireless and portable computing devices, low power consumption has become extremely critical. To meet these contradicting requirements, VLSI designers have to perform optimizations at all levels of design. This book is intended to cover a wide range of VLSI design topics. The book can be roughly partitioned into four parts. Part I is mainly focused on algorithmic level and architectural level VLSI design and optimization for image and video signal processing systems. Part II addresses VLSI design optimizations for cryptography and error correction coding. Part III discusses general SoC design techniques as well as other application-specific VLSI design optimizations. The last part will cover generic nano-scale circuit-level design techniques.

How to reference
In order to correctly reference this scholarly work, feel free to copy and paste the following:


www.intechopen.com
Paper D
A VLSI Implementation of Logarithmic and Exponential Functions Using a Novel Parabolic Synthesis Methodology Compared to the CORDIC Algorithm

Peyman Pouyan, Erik Hertz, and Peter Nilsson, Member, IEEE
Department of Electrical and Information Technology,
Faculty of Engineering, Lund University, Lund, Sweden,
sx07pp6@student.lth.se, Erik.Hertz@eit.lth.se, and Peter.Nilsson@eit.lth.se

Abstract— High performance implementations of unary functions are important in many applications e.g. in the wireless communication area. This paper shows the development and VLSI implementation of unary functions like the logarithmic and exponential function, by using a novel approximation methodology based on parabolic synthesis, which is compared to the well known CORDIC algorithm. Both designs are synthesized and implemented on an FPGA and as an ASIC. The results of such implementations are compared with metrics such as performance and area. The performance in the parabolic architecture is shown to exceed the CORDIC architecture by a factor 4.2, in a 65 nm Standard-F̃ ASIC implementation.

Index Terms: Logarithmic, Exponential, CORDIC, Parabolic Synthesis, VLSI, FPGA.

I. INTRODUCTION

There is a high demand for accelerating the computation time of unary functions like the logarithmic and exponential functions. These functions are frequently used in applications like wireless systems, computer graphics, digital signal processing, etc. In order to decrease the computation time of such functions a hardware implementation is needed in many cases.

The easiest method of hardware implementation of such functions is by using a single look-up-table [1][2], which is appropriate and easy for low precision applications. However, as the need for accuracy increases this method becomes inappropriate due to big table sizes and long execution times.

Another method is by using the CORDIC (Coordinate Rotation Digital Computer) [3] algorithm, which is an iterative method for hardware computations of unary functions. This algorithm is based on simple shift-and-add operations and it does not use any multipliers.

The CORDIC algorithm generally produces one additional bit of accuracy per iteration. Therefore, the iterative property of CORDIC is often too slow for high speed applications.

Recently a novel methodology, named parabolic synthesis is proposed by Erik Hertz and Peter Nilsson [4][5]. This method is based on developing functions that performs approximations of original unary functions in hardware. The parallel architecture based on this methodology increases the performance compared to the CORDIC algorithm among others. The parabolic synthesized implementations are also fairly easy to realize in hardware.

Characteristic for the parabolic architecture is that the same hardware can be used for a large number of different unary functions. They only differ by the use of different coefficient sets. The hardware, for the processing part, is thus very flexible. Since the architecture is fixed, for all the unary functions, the design work is only needed to be done once. The design can thus be directly reused for many different applications, without any changes.

In this paper, the logarithmic and exponential functions are implemented and compared by using both methods, i.e. the parabolic synthesis methodology and the CORDIC algorithm.

II. THE PARABOLIC SYNTHESIS METHODOLOGY

The parabolic synthesis methodology is based on second order parabolic functions, here called sub-functions \( s_i(x) \). The original function \( f_{org}(x) \), e.g. \( \log_2(x) \) and \( 2^x \), is approximated by multiplying these sub-functions together, as shown in (1), where the accuracy and precision depends on the number of sub-functions. It should be noted that (1) is presented in “serial” form, which would imply a long critical path. However, the algorithm is highly parallelizable, which leads to high performance.

\[
\begin{align*}
   f_{org}(x) &\approx s_1(x) \times s_2(x) \times ... \times s_n(x) \\
   \text{(1)}
\end{align*}
\]

The complete methodology for developing the sub-functions is described in [4][5] but without any hardware realizations and without comparisons to other methods to realize unary functions. In this paper the methodology is used to compute approximations of logarithmic and exponential functions. The parabolic approximation has been studied up to fourth order sub-functions, i.e. up to \( s_4(x) \) (1).

In Table I, a summary of the pre-defined sub-functions and their corresponding coefficients are shown. A complete procedure for development of these sub-functions can be found in [4][5].

The main part of the parabolic synthesis hardware is the processing part, based on (1). However, the parabolic synthesis is only defined in the range \( 0 \leq x < 1 \), which means that normalization and post-processing is sometimes needed, which also is the case for the CORDIC algorithm. The logarithmic function, \( \log_2(x) \), needs to be normalized to \( y = \log_2(1 + x) \) while the exponential \( 2^x \) function yields, \( y = 2^{x-1} \), which needs to be post-processed by \( z = 1 + y \).

Table I, shows that for the sub-functions \( s_1(x) \) and \( s_2(x) \), only one function is needed. However, for \( s_3(x) \) we need two sub-functions, \( s_{12}(x) \) and \( s_{13}(x) \), which are identical but operating with two different coefficients, \( c_{12} \) and \( c_{13} \). The reason is that the sub-functions must be strictly convex or concave, which they are not after the second sub-function. The interval \( 0 \leq x < 1 \), for \( s_3(x) \), must thus be split into two convex and/or concave fractions, \( 0 \leq x < 0.5 \) and \( 0.5 \leq x < 1 \) to get convex or concave functions. These functions can be seen as piecewise parabolic approximations. The same is valid for \( s_4(x) \), where the interval is split into four parts.

Note that the factors in (1) are all pre-defined, which is similar to e.g. the terms in a Taylor expansion. The difference is in the
The coefficient values for the logarithmic and exponential functions are pre-calculated and shown in Table II. Their values are determined by simulation, which can be compared to the methodologies to find the coefficients in a digital filter. The coefficients can thus be stored in a look-up-table if the flexibility is needed or be hardwired if the design is fixed.

TABLE II. COEFFICIENTS VALUES FOR THE LOGARITHMIC AND EXPONENTIAL FUNCTIONS

<table>
<thead>
<tr>
<th>Sub-function</th>
<th>Corresponding coefficient</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_i(x) = x + (c_i \times (x - x^2)) )</td>
<td>( c_i = \lim_{x \to 0} \frac{f_{s_i}(x)}{x} - 1 )</td>
</tr>
<tr>
<td>( s_i^2(x) = 1 + (c_i \times (x - x^2)) )</td>
<td>( c_i = 4 \times (f_j(0.5) - 1) )</td>
</tr>
</tbody>
</table>

The coefficient values for the logarithmic and exponential functions are pre-calculated and shown in Table II. Their values are determined by simulation, which can be compared to the methodologies to find the coefficients in a digital filter. The coefficients can thus be stored in a look-up-table if the flexibility is needed or be hardwired if the design is fixed.

TABLE III. CORDIC OUTPUT IN A HYPERBOLIC COORDINATE SYSTEM

<table>
<thead>
<tr>
<th>Rotation mode ( z_i \to 0 )</th>
<th>Vectoring mode ( y_i \to 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i = A_i \sqrt{1 - y_i^2} )</td>
<td>( y_i = A_i \sqrt{1 - x_i^2} )</td>
</tr>
</tbody>
</table>
| \( z_i = \tanh^{-1} \left( \frac{y_i}{x_i} \right) \) | \( A_i = \prod_{n=0}^\infty \sqrt{1 - 2^{-2n}} \approx 0.80 \)

The logarithmic function in base 2 can be achieved by (5), which is the result of the CORDIC vectoring mode in a hyperbolic coordinate system multiplied by a constant,

\[
\log_2(w) = \frac{2 \times \tanh^{-1} \left( \frac{y_i}{x_i} \right)}{\log(2)}
\]

For convergence reasons, a hyperbolic coordinate system has a certain number of iterations (4, 7, 10 .... 3k+1) that must be repeated [6]. In Table III, the functions that can be computed by a CORDIC in a hyperbolic coordinate system for both vectoring and rotation modes are given.

Note that the coefficient precision will gradually be more relaxed in higher level sub-functions, which saves area.

III. THE CORDIC ALGORITHM

It is shown by Walther [6] that the CORDIC algorithm can be used to compute hyperbolic functions. The logarithmic and the exponential functions are derived from the hyperbolic computations. The CORDIC equations for hyperbolic rotations in the rotation and vectoring mode are shown in (2).

\[
\begin{align*}
X_{n+1} &= X_n + y_n \times \mu_n \times 2^{-r} \\
y_{n+1} &= y_n + x_n \times \mu_n \times 2^{-r} \\
z_{n+1} &= z_n - \mu_n \times 2^{-r}
\end{align*}
\]

where \( \alpha_i = \arctanh(2^{-i}) \) are micro rotations in radians. In rotation mode, \( \mu_i \) depends on the sign of \( z_i \), which value is calculated as shown in (3).

In vectoring mode \( \mu_i \) depends on the sign of \( y_i \), and its value is calculated as shown in (4)

\[
\mu_i = -1 \text{ if } z_i < 0 \text{ and } \mu_i = +1 \text{ otherwise. (3)}
\]

\[
\mu_i = +1 \text{ if } y_i < 0 \text{ and } \mu_i = -1 \text{ otherwise. (4)}
\]

Furthermore, the exponential function can be derived by (6), which is the result of the CORDIC rotation mode in a hyperbolic coordinate system. By initializing \( x_0 \) and \( y_0 \) to 1/A0, the equation in the CORDIC output will change to (7) [7].

IV. HARDWARE IMPLEMENTATIONS

A. The Parabolic Architecture

A hardware mapped architecture is studied due to its high throughput and fast computation but to the cost of a larger chip area. In Fig. 1, the architecture for computing approximations of logarithmic and exponential functions using the parabolic synthesis methodology is shown, compare with Table I. The four sub-functions \( s_i(x) \), from (1), is multiplied together in a tree multiplier structure. Squaring and coefficient selection is done in the stage before.

In the architecture, the terms \( x_i^2 \) and \( x_i^2 \) are partial products of \( x^2 \). Only one squarer is thus needed. This is further described in [4][5]. In the higher level sub-functions, in Fig. 1, reduced versions of \( x \) are used, which implies a lower required input precision and lower hardware cost. The \( x_3 \) and \( x_4 \) signals are given by the fractional part of \( 2x \) and \( 4x \) respectively. This is achieved by simple hardwired operations based on shifts, with no extra hardware cost, see also [4][5]. To control the MUXes, in Fig. 1, one or two MSB
bits of $x$ are used, which sets the operating interval, see column 2 in Table I.

![Diagram of the global architecture for approximation of unary functions using the parabolic synthesis methodology.](image1)

An important observation, in contrast to (1), is that the architecture is highly parallel. High performance operations can thus be expected.

### B. The CORDIC architecture

The CORDIC algorithm can be implemented by using a bit-serial or a bit-parallel architecture. The architecture can also be iterative or unfolded. Here, a bit-parallel unfolded, hardware mapped, architecture is chosen. This architecture will be compared with the parabolic architecture, which also is bit-parallel and hardware mapped. The bit-parallel unfolded architecture contains $n$ stages of CORDIC units, where each stage is a basic CORDIC engine.

This architecture does not need a look-up-table since the values for the angle accumulator are fixed and hardwired as constants to each adder and subtractor in the angle accumulator chain. Also the barrel shifters can be eliminated, since non-combinatorial hardwired shifts can be applied to the adder and subtractor in each stage [8][9].

![Diagram of a bit-parallel unrolled CORDIC architecture.](image2)

The unfolded CORDIC architecture will thus only consist of combinational units. There is no need for a controller in the design either. This architecture can reach a higher throughput by being pipelined between the stages, by putting registers between the adders and subtractors [10]. However, the increase in hardware size leads to more area and power consumption in an ASIC. Fig. 2 shows a CORDIC architecture in rotation mode.

### V. RESULTS AND COMPARISON

#### A. The Parabolic architecture

For the hardware implementation, the hardware description language VHDL is used for simulation. FPGA and ASIC synthesis is used for the implementation. The digital computations are based on two’s complement representation.

![Graph showing the absolute error between the original function and the parabolic synthesis hardware output after using four sub-functions.](image3)

In Fig. 3, regarding the logarithmic approximation, the absolute error from the difference between the original function and the hardware result for the output when using four sub-functions, is shown. Fig. 3 shows a −92 dB absolute error. In dB scale, one hardware bit corresponds to $20 \log_{10}(2) \approx 6.02 \text{ dB}$. The accuracy, in Fig. 3, will thus correspond to $92/6 > 15$ bits of accuracy. If only the first two sub functions, $s_1$ and $s_2$, are used, the parabolic methodology shows an absolute error at −64 dB, which corresponds to an accuracy of 10 bits.

#### B. The CORDIC architecture

![Graph showing the absolute error between the original function and the hardware output from the CORDIC architecture.](image4)
In Fig. 4, the absolute error from the difference between the original function and the hardware result using the CORDIC architecture is shown, for the logarithmic approximation.

The CORDIC method requires \( n + 1 \) iterations to give an \( n \) bit precision at the output. In order to have the same number of bits of accuracy as for the four sub-function parabolic synthesis methodology, there is a need of \( n+1+4 = 20 \) iterations, where the iterations \( 4, 7, 10, \) and \( 13 \) are repeated.

C. Comparison

The parabolic synthesis methodology shows 15 bits of accuracy with a 13 bit input wordlength and an 18 bit output word length, while the CORDIC method needs a word length of \( n+2+\log_2(n) \) for the \( x \) and \( y \) data path and \( n+\log_2(n) \) bits for the computation of the angle \( z \) in fixed point arithmetic [11]. The \( x \), \( y \), and \( z \) input wordlengths used in the CORDIC are thus 21, 21, and 19 bits respectively.

The two designs have been imported and synthesized to one of the low cost Altera FPGAs, the Cyclone II 2C20 with the synthesis results, shown in Table IV. The FPGA synthesis result show that the non-pipelined parabolic synthesis methodology is efficient to implement on an FPGA, with hardware multipliers mapped inside. The critical path delay of the parabolic synthesis design is 2 times lower than for the non-pipelined CORDIC design.

<table>
<thead>
<tr>
<th>Implementation Method</th>
<th>Total Logic Elements</th>
<th>Embedded 9-bit Multipliers</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORDIC Algorithm</td>
<td>1873/18752</td>
<td>0/52</td>
<td>140.74 ns</td>
</tr>
<tr>
<td>Parabolic Synthesis</td>
<td>481/18752</td>
<td>31/52</td>
<td>70.41 ns</td>
</tr>
</tbody>
</table>

Furthermore, the designs have been synthesized and mapped to an ASIC with the synthesis result using a 65 nm Standard-\( V_T \) technology library, shown in Table V. The ASIC implementation shows even better results. The synthesized non-pipelined parabolic architecture is running faster than the CORDIC non-pipelined implementation, as well. The maximum frequency for the parabolic architecture is \( 48.0/11.5 = 4.2 \) times higher in the ASIC implementation.

<table>
<thead>
<tr>
<th>Implementation Methodology</th>
<th>Area mm(^2)</th>
<th>Max Frequency without pipelining</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORDIC Algorithm</td>
<td>0.01915</td>
<td>86.0 ns</td>
</tr>
<tr>
<td>Parabolic Synthesis</td>
<td>0.02640</td>
<td>20.8 ns</td>
</tr>
</tbody>
</table>

Both designs can run faster by being pipelined with registers between the stages. However, the CORDIC needs a large number of registers to come close to the speed of the parabolic architecture. Fully pipelined, the CORDIC architecture will require in the order of \( 3\times(n+4) = 57 \) \( n \)-bit registers. The CORDIC will in that case occupy a larger area than the parabolic design.

The parabolic synthesis methodology can be used to implement many unary functions, not only logarithmic and exponential functions. By using exactly the same architecture for the processing part, as in Fig. 1, unary functions such as sine, cosine, arcsine, arccosine, tangent, arctangent, division, and square root functions can easily be implemented, just by using a look-up-table containing different sets of coefficients. The parabolic architecture is thus very flexible. Using different sets of coefficients in a CORDIC require a large number of multiplexers to switch the CORDIC between rotation and vectoring mode, which will increase the area and cost.

Since the parabolic architecture is fixed, for all the unary functions, the design work is only needed to be done once. The design can thus be directly reused in many different applications, without any changes.

VI. Conclusion

The hardware development and implementation of the logarithmic and exponential functions using the CORDIC and parabolic synthesis methodology is studied in this paper. Both designs benefit from being simple to implement and from using basic arithmetic operations, only.

The CORDIC algorithm has major drawbacks in delay of the computations because there is not efficient architecture for parallel implementation. On the contrary, the parabolic synthesis methodology, the use of parallelism in the computations leads to a shorter critical path and fast computation.

The performance in the parabolic architecture is shown to exceed the CORDIC architecture by a factor 4.2, in a 65 nm Standard-\( V_T \) ASIC implementation.

REFERENCES


Paper E
Hardware Implementation of the Exponential Function Using Taylor Series

Peter Nilsson Senior Member IEEE, Ateeq Ur Rahman Shaik, Rakesh Gangarajaiah Student Member IEEE, Erik Hertz

Department of Electrical and Information Technology, Faculty of Engineering, LTH, Lund University, Lund, Sweden
Centre for Research on Embedded Systems, Halmstad University, Halmstad, Sweden

Abstract—This paper presents hardware implementations of Taylor series. The focus will be on the exponential function but the methodology is applicable on any unary function. Two different architectures are investigated, one, original, straight forward and one modified structure. The outcomes are higher performance, lower area, and lower power consumption for the modified architecture compared to the original.

Keywords—Taylor expansion, Exponential function, Dynamic Power, Static Power, Low Leakage, Integrated Circuit, IC, ASIC, CMOS.

I. INTRODUCTION

The exponential function, together with other unary functions such as logarithm, sine, and cosine functions, is often used in in signal processing and baseband wireless communication. ASIC solutions are often used if there are special requirements like high performance, small form factor, and low power consumption. There are different algorithms to solve the tasks such as look-up tables [1], CORDIC [2], interpolation, parabolic synthesis [3][4] etc. Another method is the well-known Taylor expansion [5], which is the focus for this paper. Two different Taylor architectures are compared and evaluated.

II. BACKGROUND

A Taylor series is a series expansion of a function around a point. With an increasing number of terms in the expansion, the approximation accuracy will increase. A one-dimensional Taylor series is defined as in (1), where the function \( f(x) \) is expanded around the point \( x = a \).

\[
f(x) = f(a) + f'(x-a) + \frac{f''(x-a)^2}{2!} + \frac{f'''(x-a)^3}{3!} + \ldots + \frac{f^{(n)}(x-a)^n}{n!}
\]

If \( a = 0 \), the expansion is known as a Maclaurin series. Taylor series of some common functions, e.g. for the exponential and logarithmic functions as well as for sine and cosine functions, are shown in (2). This paper will concentrate on the exponential function but the method is applicable to other functions as well.

\[
e^x = e^a \left[ 1 + (x-a) + \frac{(x-a)^2}{2} + \frac{(x-a)^3}{6} + \ldots \right]
\]

\[
\ln(x) = \ln(a) + \frac{(x-a)}{a} - \frac{(x-a)^2}{2a^2} + \frac{(x-a)^3}{3a^3} + \ldots
\]

\[
\cos(x) = \cos(a) - \sin(a)(x-a) - \frac{\cos(a)(x-a)^2}{2} + \frac{\sin(a)(x-a)^3}{6} + \ldots
\]

\[
\sin(x) = \sin(a) + \cos(a)(x-a) - \frac{\sin(a)(x-a)^2}{2} - \frac{\cos(a)(x-a)^3}{6} + \ldots
\]

A. Approximation of the exponential function

If \( a = 0 \), i.e. the Maclaurin series, we get a good approximation close to \( x = 0 \). However, it is better to do the approximation around a value \( a \), as illustrated in Fig. 1. The figure shows the 1st and 2nd order exponential approximation as well as the original function. The value \( a = 0.5 \), \( 0.12 \) is chosen in the figure. It can be seen that the approximations are better in the middle. In this case, \( a \) is chosen to give an error that is about equal in both ends.

Fig. 1. Taylor expansion around \( a = 0.5 \).

III. ARITHMETIC

A. The exponential function for the original architecture

\[
e^x = e^a \left[ 1 + q + \frac{q^2}{2} + \frac{q^3}{6} + \frac{q^4}{24} + \frac{q^5}{120} + \frac{q^6}{720} \right]
\]

\[
e^x = e^a \left[ 1 + q + \left( \frac{q}{2} \right)^2 + \frac{q^3}{24} + \frac{q^4}{120} + \frac{q^5}{720} \right]
\]

978-1-4799-5442-1/14/$31.00 ©2014 IEEE
A sixth order expansion, corresponding to 15 bits, of the exponential approximation is shown in (3), where \( q = x - a \). The original architecture is based on the lower part of (3), which is a well-known method to reduce the number of multiplications in Tailor series.

There are six factors, \( q = x - a \), in (3). The input value \( x \) is a 15-bit positive fractional word and the coefficient \( a \) only have 1 bit. However, \( a \) appears negative but since \( a \) is a fixed constant, it can be converted to its two’s-complement number by taking the complement and adding an LSB one as illustrated in (4). The result from \( x - a \) need two integer bits, where one is a sign bit, if \( x \) is allowed to be in the span \( [0 \leq x \leq 1] \). However, if \( x \) is not allowed to reach the number 1, i.e. the number interval is \([0 \leq x < 1]\), only one integer bit is needed.

\[
\begin{align*}
0.10000 & \quad a = 0.5 \\
1.01111 & \quad \text{one's complement} \\
1.10000 & \quad \text{two's complement of} \ a \\
\end{align*}
\]

The subtraction, \( q = x - a \), can be done fairly easy. Let say that a five bit word, \( x \), as the word in (5), is to be added. The subtraction \( x - a \) will thus be an addition \( q = x + (-a) \) in the hardware as shown in Table I.

\[
x = \{x_0, x_{-1}, x_{-2}, x_{-3}, x_{-4}\}
\]

\[
\begin{array}{cccccc}
x & x_0 & x_{-1} & x_{-2} & x_{-3} & x_{-4} \\
-1/2 & 1 & 1 & 0 & 0 & 0 \\
-1 & 1+x_0+x_{-1} & 1+x_{-1} & x_{-2} & x_{-3} & x_{-4} \\
\end{array}
\]

However, since \( x \) is a positive number \( x_0 = 0 \), the result from the addition of the MSB bits will be \( 1+x_{-1} \), i.e. a copy of the MSB, bit is used, as shown in Fig. 2b. The subtraction is thus done by using two half adders only

\[
\begin{array}{c}
- a \\
\hline
x - a \\
\hline
\end{array}
\]

\[
\begin{array}{cccccc}
\text{a) HA} & \text{b) HA} & \text{c) HA} \\
\end{array}
\]

Fig. 2. The adder in a), at bit level in b), and an example in c).

Using (3), the original architecture for the exponential approximation is shown, in Fig. 3. Unfortunately, it is not possible to do a parallel architecture of (3) and (5). It has to be serial architectures.

\[
\begin{array}{cccccccc}
c_6 & c_5 & c_4 & c_3 & c_2 & c_1 & c_0 & e^a \\
\hline
\end{array}
\]

Fig. 3. The original architecture for the exponential function

Fig. 3 shows such an architecture. The adder from Fig. 2a can be seen as \( q = x - a \) to the left. All multipliers must be able to handle two’s-complement numbers. However, the adders always give a positive output.

Table II shows the fixed coefficient values used in Fig. 3. It can be noted that the values get smaller and smaller, which might save some area. Adding a one or a half, for \( c_1, c_2, \) and \( c_3 \), can also lead to simplifications similar to (4).

\[
\begin{array}{|c|c|}
\hline
\text{Coefficient} & \text{Numerical value} \\
\hline
C_0 & 1 \quad 1.000000000000000 \\
C_1 & 1 \quad 1.000000000000000 \\
C_2 & 1/2 \quad 0.500000000000000 \\
C_3 & 1/6 \quad 0.166666666666667 \\
C_4 & 1/24 \quad 0.041666666666667 \\
C_5 & 1/120 \quad 0.008333333333333 \\
C_6 & 1/720 \quad 0.001388888888889 \\
\hline
\end{array}
\]

**TABLE II. The coefficients used in Fig. 3**

**B. Architecture at bit level**

Both the coefficients and the data signals in Fig. 3 are also shown in Fig. 4, for the first four stages. In the figure, the active bits in both the coefficients and the data signal are shown. It can be noted that the numbers often have zeroes both at the end and in the beginning, which are not shown in the figure. There is therefore no use to process those, which gives reduced sizes in the adders and multipliers. It can be noted that the left adder is very small, only with 2x1 input bits. How to design the adder is shown in Fig. 2.

**Fig. 4.** The first four stages in the architecture shown in Fig. 3

**C. The exponential function for the modified architecture**

To reach the modified architecture (3) can be expressed as shown in (6), where \( d_i \) are coefficient substitutes that can be hardwired.

\[
e^x = e^a (d_0 + x(d_1 + x(d_2 + x(d_3 + x(d_4 + d_5 x))))))
\]

In Fig. 5, the modified architecture for the exponential approximation is shown.

\[
\begin{array}{cccccccc}
d_5 & d_4 & d_3 & d_2 & d_1 & d_0 & e^a \\
\hline
\end{array}
\]

Fig. 5. The modified structure for the exponential approximation
The left adder used in Fig. 3, is not needed. The architecture has one multiplier and one adder less than the architecture in Fig. 3. It can also be noted that all coefficients and signals are positive. There is thus no need for two’s-complement numbers, which reduces the complexity.

**TABLE III. THE COEFFICIENTS USED IN (6)**

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Numerical value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(d_0) (\left(1 - a + \frac{a^2}{2} - \frac{a^3}{6} + \frac{a^4}{24} - \frac{a^5}{120} + \frac{a^6}{720}\right))</td>
<td>0.606532118055556</td>
</tr>
<tr>
<td>(d_1) (\frac{3a^2}{2} - \frac{3a^3}{6} + \frac{a^4}{24} + \frac{3a^5}{720} + \frac{a^6}{720})</td>
<td>0.606597222222222</td>
</tr>
<tr>
<td>(d_2) (\frac{1}{2} - \frac{3a}{2} + \frac{a^2}{6} - \frac{6a^3}{24} + \frac{6a^4}{720} - \frac{14a^5}{720})</td>
<td>0.302604166666667</td>
</tr>
<tr>
<td>(d_3) (\frac{1}{6} - \frac{4a}{24} + \frac{6a^2}{720} - \frac{16a^3}{720})</td>
<td>0.103472222222222</td>
</tr>
<tr>
<td>(d_4) (\frac{1}{24} \frac{34a + 9a^2}{720} )</td>
<td>0.021180555555556</td>
</tr>
<tr>
<td>(d_5) (\frac{7 - 2a}{720})</td>
<td>0.008333333333333</td>
</tr>
</tbody>
</table>

**IV. APPROXIMATION ERROR**

The target is 15 bits at the output. Fig. 6 shows the approximation error for structures using 5 and 6 Taylor stages. The figure also shows the 15 bit limit as the -90.3 dB line. It can be seen that five stages is not enough to reach the 15 bits at the output. Six stages will therefore be used.

![Fig. 6. The approximation error for 5 and 6 Taylor stages](image)

If 17 bits are used for the \(e^a\) coefficient, 17.6 output bits are given, when six Taylor stages are used. This gives some room to play around in the architecture. The graph is plotted with the coefficient \(a = 0.510\) corresponding to 0.1.

To conclude, we have one trivial one-bit \(a\) coefficient and one huge 17 bit coefficient \(e^a\). The latter coefficient leads to a large multiplier, which fortunately only appears once. The figures, in dB, can be transferred to bits by using (7).

From the figures, it can be noted that the gain for adding a new Taylor term is around 3 bits.

**V. RESULTS**

The synthesis is carried out by using Synopsys Design Compiler, for the area and performance. The power consumption is investigated by using Synopsys Design Vision and the Synopsys Prime Time PX tool together with Modelsim. A 65 nm CMOS technology including cell libraries is provided by STMicroelectronics. A Low Power High Threshold Voltage (LPHVT) cell library is used and the supply voltage is 1.2V

**A. Area**

The area has been estimated for the two architectures using the “minimum area constraint” in Design Vision.

**TABLE IV. THE AREA USING THE LPHVT LIBRARY**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Area</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>22400</td>
<td>(\mu m^2)</td>
</tr>
<tr>
<td>Modified</td>
<td>20700</td>
<td>(\mu m^2)</td>
</tr>
</tbody>
</table>

Table IV shows the simulated area for the original and modified architecture. The modified architecture occupies about 92% of the area compared to the original architecture, which could be expected since it has fewer arithmetic units.

**B. Performance**

The propagation delay, \(t_p\), using the “maximum speed constraint” is shown in Table V.

**TABLE V. THE CRITICAL PATH USING THE LPHVT LIBRARY**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Propagation delay</th>
<th>Max frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>49.8ns</td>
<td>20.1MHz</td>
</tr>
<tr>
<td>Modified</td>
<td>40.3ns</td>
<td>24.8MHz</td>
</tr>
</tbody>
</table>

It is shown that the minimum critical path for the modified architecture is reduced to 81% compared to the original architecture. In other words, the maximum frequency is 23% larger. The reduction of the critical path is expected since the path in the modified architecture has one adder and one multiplier less

**C. Power consumption**

Low power consumption is an important design criterion in a hardware design. The power results, using Synopsys Design Vision is presented in Table VI.

**TABLE VI. POWER CONSUMPTION USING THE LPHVT LIBRARY**

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Dynamic power at 10 MHz</th>
<th>Static power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>0.060 mW</td>
<td>75.4 nW</td>
</tr>
<tr>
<td>Modified</td>
<td>0.058 mW</td>
<td>66.4 nW</td>
</tr>
</tbody>
</table>
The dynamic power consumption for the modified architecture is reduced to 97% compared to the original architecture. The static power consumption is reduced to 88% when the modified architecture is used. According to these results, the static power consumption can be neglected at the frequency 10 MHz. However, the static power consumption is rather constant when the frequency is varied, i.e. at lower frequencies, the dynamic power consumption becomes lesser and lesser dominant.

To see the behavior towards the frequency, the power consumption for the modified architecture is plotted, as illustrated in Fig. 7. Synopsys PrimeTime has been used for the simulations.

Fig. 7 shows the power consumption in log-log scale. The graph shows the total power consumption for frequencies from 1Hz to 100MHz.

At 1 kHz the static and dynamic power consumption is about equal and at 10kHz, the static power consumption is about 7% of the dynamic. The static power consumption is simulated to 76.9nW and the dynamic to 0.959mW at 10 MHz. If the results from the two simulators are compared, it can be seen that the static power consumption is about the same for both simulators but the dynamic power consumption shows a 17 times higher value for the PrimeTime simulator. PrimeTime is known to be more accurate for dynamic power simulations, i.e. Design Vision gives a too optimistic result.

VI. CONCLUSIONS

The paper shows some improvements in hardware implementation when using different architectures for Taylor expansion. An original straight forward structure is compared to a modified architecture. The modified architecture shows better results regarding performance, area and power consumption. The paper also demonstrates different frequency regions where dynamic and static power consumption dominates.

ACKNOWLEDGEMENT

This work is a part of the DARE project and the authors would like to thank Lund University and the Swedish Foundation for Strategic Research.

REFERENCES


[5] Brook Taylor (1685–1731), a British matematician, which was refered using “Taylor series” in a paper by the Swiss matematician Simon Antoine Jean Lhuilier, 1786.


Paper F
Abstract—This paper shows a novel methodology to improve unrolled CORDIC architectures. The methodology is based on removing adder stages starting from the first stage. As an example, a 19-stage CORDIC is used but the methodology is applicable on CORDICs with an arbitrary number of stages. The CORDIC is implemented, simulated, and synthesized into hardware. In the paper, the performance is shown to be increased by 23% and that the dynamic power can be reduced by 27%.

I. INTRODUCTION

To compute non-linear functions is in most cases a demanding task to do in hardware. Look-up tables are useful if the precision is low but the size of the table grows exponentially with increasing accuracy, which make them unfeasible for high-precision hardware. Taylor expansion is another methodology for unary functions [1]. A novel methodology is to use parabolic functions to compute unary functions [2].

The focus of this paper is on the CORDIC algorithm [3]. Traditionally, iterative CORDICs [4] have been used due to their small area, which give low silicon cost. In today’s downscaled technologies e.g. below 40nm, the static power consumption tends to override the dynamic power consumption, in low-speed architectures. Iterative small-area CORDICs is therefore still important.

For high-speed designs, other parameters such as dynamic power consumption and performance are important. A relaxed focus on silicon area will thus make it feasible to use fast unrolled CORDICs. This paper shows a novel methodology to reduce the power consumption in unrolled CORDIC architectures. However, the methodology is, to a large extent, applicable to iterative CORDICs as well.

II. THE CORDIC ALGORITHM

The CORDIC algorithm is based on vector rotations to find an approximation of a non-linear function. The main advantage with the algorithm is that it is multiplier less. It uses additions and subtractions only. The CORDIC is built up with a number of stages. By increasing the number of stages the accuracy is improved. Fig. 1 shows an example with four vector rotations, corresponding to a four-stage CORDIC. The (blue) dashed vector is the vector with an angle of which, in this case, the approximate cosine and sine values are searched for.

There are many ways to choose where to start the rotations. In this paper, the starting point is chosen to be at the coordinates $(x = 1/R, y = 0)$, where $R$ is the last vector length if the starting vector has the length $R = 1$ i.e. a vector between origin and the unit circle. A starting point at $1/R$ will thus outcome in a result that is as close to the unit circle as possible when the last stage is computed. $R$ is a fixed coefficient for the chosen number of CORDIC stages. It can thus be hardware wired into the implementation. When the last rotation is done, the coordinates give the approximate cosine and sine values for the input angle, i.e. $(x_4, y_4)$ in the figure.

![Fig. 1. Four CORDIC rotations.](image)

The figure shows four rotations. The first is positive and detected to be too large. The second does a negative rotation, the third a positive and finally the fourth will be negative again. The angles of the rotations, $c_0$ in Fig. 1, are fixed coefficients, which are done in the order $c_0 = 45$, $c_1 = 27$, $c_2 = 14$, $c_3 = 7$ degrees etc. The reason is that $\arctan(1) = 45$, $\arctan(1/2) = 27$, $\arctan(1/4) = 14$, and $\arctan(1/8) = 7$. That is, they correspond to binary shifts.

III. THE ORIGINAL UNROLLED CORDIC

A 19 stage CORDIC is chosen to demonstrate the methodology. However, the methodology is valid for any size of the CORDIC. Fig. 2a shows the architecture of the first five stages of the 19-stage CORDIC. In the four adders at the top, the remaining angle is computed for each stage. The input signal $\alpha$ is the angle for the cosine and sine value that is searched for. At the top, the fixed coefficient angle values $c_0$ for each rotation are delivered. These are added or subtracted from $\alpha$ in each stage. That is, the remaining angle $v_0 = \alpha \pm c_0$, $v_1 = \alpha \pm c_1$, etc.

The coefficients are, in this architecture, chosen to be 23 bits long, which after synthesis becomes much shorter due to many zeros. They are fixed coefficients that can be hard wired or stored in a ROM. In the middle and the lower adder rows, the approximation of the $x_3 \approx \cos(\alpha)$ and $y_3 \approx \sin(\alpha)$ is computed, which is provided to the right.
In each vector rotation stage there is a crosswise vector that approximates the coordinates of the angle converging towards the vector that approximates the intermediate vectors are determined, in order to coordinates is delivered to the left. In each stage, new

\[ x_i = x_{i-1} \pm \frac{y_{i-1}}{2^{i-1}} \]

\[ y_i = y_{i-1} \mp \frac{x_{i-1}}{2^{i-1}} \]

Equation (1)

There are also divisions of the vector coordinates by a factor corresponding to $2^j$ where $j \in [1, 2, 3, \ldots]$. That is, division by 1, 2, 4, 8 ..., which are the right shifts, discussed in section II that are done by shifting the busses between the stages. There is thus no extra cost in hardware for the divisions.

IV. THE IMPROVED CORDIC ARCHITECTURES

To simplify the architecture in Fig. 2a, the architecture in Fig. 2b can be used. Two 19-bit adders at the left can be removed, since the $x_0$ and $y_0$ values are fixed. The methodolgy is valid for any chosen input vector. Here in this example, the coordinates $x_0 = 1/R$ and $y_0 = 0$ is chosen. That means that the output from the first stage is $y_1 = x_0$ and $x_1 = x_0$ in the respective lower adder row, as shown in (1).

To eliminate the secondadder stage, the architecture in Fig. 2c can be used. The removal of the two 19-bit adders require two 19-bit MUXes, where the input signals is pre-calculated as fixed coefficient values as shown in (1). The gain will then be lower power consumption and a shorter critical path since there is no ripple in the MUXes.

In Fig. 2d, another adder stage is removed, the third stage in the middle and lower adder row. Now, six MUXes are needed to replace the adders and all of the input signals are fixed as before, see (1). Note that the two last indices of the input coefficients stand for the sign-bit values $d_i$ and $d_j$.

The adder stage reduction from Fig. 2d can be expanded to a reduction in the fourth stage, fifth stage, and so on. The removal of the fourth stage will require $8+4+2 = 14$ MUXes with a wordlength of 19 bits and the elimination of five adder stages will require 30 MUXes. However, now the power consumption and silicon area begin to increase but the delay, the critical path, will be shorter. This paper stops with three eliminated adder stages in the middle and lower adder rows, but there is potential for more reductions.

Until now, the upper adder row has been left intact. However, it is possible to eliminate adder stages in the upper row, as well. In (2), the sign-bit $d_i$ can be detected by the logic function $d_i \land d_j$, which is also used in the architecture in Fig. 2e. The sign-bit detection can for instance be realized with two inverter, three 2-input NANDs and one 3-input NAND. One adder stage is removed by that operation. An extra MUX is also needed to select between coefficient angle values $c_0+c_1$.

Fig. 2. The first 5 stages of the 19-bit CORDIC.

The initial vector value with x-axis and y-axis coordinates is delivered to the left. In each stage, new intermediate vectors are determined, in order to converge towards the vector that approximates the vector that approximates the coordinates of the angle $\alpha$. In each vector rotation stage there is a crosswise addition or subtraction of the vector coordinates, as shown in (1).
and $c_0 - c_1$, that is, if the vector rotation should be positive or negative for the $c_1$ rotation.

$$d_{\text{catch}} = \alpha_{\text{no19}} + \alpha_{\text{no17}} \alpha_{\text{no15}} + \alpha_{\text{no15}} \alpha_{\text{no11}} \alpha_{\text{no15}} \alpha_{\text{no14}}$$

This operation can be extended to the sign-bit detection of the function $d_{\text{catch}}$, $d_{\text{catch}}$, and so on. However, the design will be more complex but still there will be a gain in shorter delay. Here, the investigation stopped with the first sign-bit detection.

V. RESULTS

A low power high $V_T$ (LPHVT) 65nm technology is used for the simulations. The designs are implemented in VHDL and STMicroelectronics has provided the cell library. Design Compiler and PrimeTime are the used tools. For the simulations, a supply voltage at 1.2V at a 25 degree temperature is chosen.

Table I shows the silicon area for two different cases, when the tools are set with a maximum speed constraint and when they are set with a minimum area constraint.

Maximum speed is the frequency when there is a zero slack in the critical path, e.g. at 76.7MHz for the original design and 99.6MHz for the final. To find the minimum area, it is appropriate to find a suitable frequency where the area is not decreasing any more. Table II shows the area for the architectures, synthesized for different frequencies.

Table I shows the silicon area for two different cases, when the tools are set with a maximum speed constraint and when they are set with a minimum area constraint.

Table II shows the area for the architectures, synthesized for different frequencies.

The power consumption, when the maximum speed constraint is set, is shown in Table IV. In Table IV it can be seen that the dynamic power is lowered by 28%.

![Fig. 3. Area vs. the frequency](image-url)
and that the static power is increased by 8%. The increase in static power is expected since the area is increased.

However, it is not so interesting to compare the power consumption when the frequencies are different, since the power changes linearly with the frequency. In Table V, the power consumption at 10MHz is shown. The table shows that the dynamic power is decreased by 27% and the static power is 6% lower. The decrease in static power is expected since the area is decreased, due to the replacement of adder cells with MUXes.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Frequency (MHz)</th>
<th>Dynamic Power (mW)</th>
<th>Static Power (nW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 2a</td>
<td>10</td>
<td>0.2399 (100%)</td>
<td>131.5 (100%)</td>
</tr>
<tr>
<td>Fig. 2b</td>
<td>10</td>
<td>0.2263 (94%)</td>
<td>131.5 (100%)</td>
</tr>
<tr>
<td>Fig. 2c</td>
<td>10</td>
<td>0.2124 (89%)</td>
<td>126.2 (96%)</td>
</tr>
<tr>
<td>Fig. 2d</td>
<td>10</td>
<td>0.1864 (77%)</td>
<td>123.5 (94%)</td>
</tr>
<tr>
<td>Fig. 2e</td>
<td>10</td>
<td>0.1748 (73%)</td>
<td>123.5 (94%)</td>
</tr>
</tbody>
</table>

In Fig. 4, the power consumption $P$ vs. the frequency $f$ is illustrated for the final architecture, i.e. the architecture displayed in Fig. 2e.

![Fig. 4. The power consumption in the final architecture.](image)

The dynamic power consists of switching power and internal power. Citing the tool vendor, “The switching power is determined by the capacitive load and the frequency of the logic transitions on a cell output.” [5]. Furthermore, “The internal power is caused by the charging of internal loads as well as by the short-circuit current between the N and P transistors of a gate when both are on” [5]. The sum of them is thus needed to determine the total dynamic power consumption. It can be noted that the switching power is higher than the internal power but they are, however, basically at the same size. The switching power is shown by the long-dashed (blue) line and the internal power by the short-dashed (red) line in Fig. 4. The static power is shown by the lower solid (blue) line and the total power consumption is shown by the upper solid (red) line.

It can be noted that when the frequency comes close to the limit, the switching power is not linearly increasing with respect to the frequency, i.e. it is increasing more than expected. The same phenomena can be seen for the static power, which is not constant any more.

When the three power components are added, the total power, the upper solid (red) line in Fig. 4, will be the result. Two asymptotic regions turn up, where the static power dominates to the left and the dynamic to the right. It can be seen that the switching and internal power is increasing linearly towards the frequency and that the static power is unaffected regarding the frequency. To the left in the diagram, the dynamic power can be ignored compared to the static power and to the right the static power can be neglected when relating to the dynamic. Somewhere around 10 kHz, the dynamic power becomes larger than the static power.

VI. ACKNOWLEDGEMENT

We would like to thank STMicroelectronics, SSF, and Vinnova for their support.

VII. CONCLUSIONS

A methodology to improve unrolled CORDIC architectures is presented. The methodology is based on removing stages to the cost of a number of MUXes. It is shown that the performance can be increased by 23% and that the dynamic power consumption can be reduced by 27%, by interchanging adders by MUXes.

REFERENCES


Paper G
Combining the parabolic synthesis methodology with second-degree interpolation

Erik Hertz\(^1\), Bertil Svensson\(^1\), Peter Nilsson\(^2\)

\(^{1}\)Centre for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
\(^{2}\)Electrical and Information Technology Department, Lund University, Lund, Sweden

A R T I C L E   I N F O

Article history:
Received 26 June 2015
Accepted 26 January 2016
Available online 26 February 2016

Keywords:
Approximation
Parabolic synthesis
Unary functions
Elementary functions
Second-degree interpolation
Arithmetic computation
CORDIC
VLSI
Look-up table

A B S T R A C T

The Parabolic Synthesis methodology is an approximation methodology for implementing unary functions, such as trigonometric functions, logarithms and square root, as well as binary functions, such as division, in hardware. Unary functions are extensively used in baseband for wireless/wireline communication, computer graphics, digital signal processing, robotics, astrophysics, fluid physics, games and many other areas. For high-speed applications, as well as in low-power systems, software solutions are not sufficient and a hardware implementation is therefore needed. The Parabolic Synthesis methodology is a way to implement functions in hardware based on low complexity operations that are simple to implement in hardware. A difference in the Parabolic Synthesis methodology compared to many other approximation methodologies is that it is a multiplicative, in contrast to additive, methodology. To further improve the performance of Parabolic Synthesis based designs, the methodology is combined with Second-Degree Interpolation. The paper shows that the methodology provides a significant reduction in chip area, computation delay and power consumption with preserved characteristics of the error. To evaluate this, the logarithmic function was implemented, as an example, using the Parabolic Synthesis methodology in comparison to the Parabolic Synthesis methodology combined with Second-Degree Interpolation. To further demonstrate the feasibility of both methodologies, they have been compared with the CORDIC methodology. The comparison is made on the implementation of the fractional part of the logarithmic function with a 15-bit resolution. The designs implemented using the Parabolic Synthesis methodology – with and without the Second-Degree Interpolation – perform 4x and 8x better, respectively, than the CORDIC implementation in terms of throughput. In terms of energy consumption, the CORDIC implementation consumes 140% and 800% more energy, respectively. The chip area is also smaller in the case when the Parabolic Synthesis methodology combined with Second-Degree Interpolation is used.

\(©\) 2016 Elsevier B.V. All rights reserved.

1. Introduction

Computation of elementary functions is legion in a multitude of applications, such as digital signal processing (DSP), control applications, 2D and 3D computer graphics, computer aided design (CAD), virtual reality and physical simulation. The accuracy of the functions is of course important. Software routines can be designed to provide extremely accurate results, but software is often too slow for numerically intensive and real-time applications. There is therefore a significant interest in hardware implementations of function generators.

Hardware computation of elementary functions can be performed by employing many different algorithms \([1,2]\), such as table-based methods, polynomial and rational approximation methods, and functional iteration methods.

Table-based methods remain manageable for low precision computation when the input operand is up to 12–16 bits, corresponding to table sizes of 4–64 K words. The size of the table grows exponentially with the addressing length and becomes unacceptably large when operating with higher precision. An alternative way of making approximations is based on polynomials. Since polynomials involve only additions, subtractions and multiplications, using them is a natural way to approximate elementary functions. A number of schemes are available for polynomial approximations, such as Taylor, Maclaurin, Legendre, Chebyshev, Jacobi and Laguerre \([1]\). For a given precision, the chosen polynomial scheme affects the number of terms included, and thus the computational complexity. Two development strategies are available in...
developing an approximation, one to minimize the average error, called least squares approximation, and one to minimize the worst case error, called least maximum approximation [1]. An example of when least squares approximation is favorable is when the approximation is used in a series of computations. On the other hand, least maximum approximation is favorable when it is important that the maximum error to the function to be approximated is kept small. An example of when least maximum approximation is favorable is when the error from the approximation has to be within a limit from the true function value. An advantage of polynomials is that they are table-less, but their drawback is that they impose large computational complexities and delays [1]. A reduction in computational complexity can be accomplished by combining table-based methods with polynomial based methods, and the delays can also, to some extent, be decreased [1].

For implementation of elementary functions in hardware, the sum of bit-products methodology [3] can be beneficial, since it can give an area efficient implementation with a high throughput at a reasonable accuracy.

The commonly used Coordinate Rotation Digital Computer (CORDIC) algorithm [4,5] is an iterative algorithm. The benefit of the algorithm is that the hardware for the basic elementary functions requires only a small look-up table, simple shifts and additions. The CORDIC algorithm is used in applications where aspects such as high speed, low power and low area have to be considered. However, since it is an iterative method, it is inherently somewhat slow and therefore often insufficient for very high performance applications. The Parabolic Synthesis methodology [6–9] has a parallel architecture, which significantly reduces the propagation delay and thus provides an advantage over the serial CORDIC algorithm. The reduction in propagation delay also allows a lower clock frequency and thereby a significant reduction in power consumption.

To further reduce chip area, critical path delay and power consumption, an extension of the Parabolic Synthesis methodology has been developed, which is described in this paper. The extension is achieved by combining Parabolic Synthesis with Second-Degree Interpolation [2,10,11]. In contrast to the Parabolic Synthesis methodology, which is a synthesis of second-order functions and thus provides an accuracy that depends on the number of second-order functions, the accuracy of the combined methodology depends on the number of intervals in the Second-Degree Interpolation part. The architecture of the Parabolic Synthesis methodology is characterized by a high degree of parallelism, which ensures low execution times. Like the Parabolic Synthesis methodology, the Parabolic Synthesis methodology combined with Second-Degree Interpolation utilizes only low complexity operations, again ensuring simple implementation in hardware. The extension admits that the characteristics can to a great extent be tailored. This enables making a rough internal error compensation of the approximation, which improves the distribution of the error.

The feasibility of the Parabolic Synthesis methodology has been verified for implementation on a vast range of unary functions, as shown in [8]. The extended methodology described in this paper has the same broad applicability.

The remaining part of this paper is organized as follows: Section 2 describes the Parabolic Synthesis methodology; Section 3 describes the Parabolic Synthesis Combined with Second-Degree Interpolation; Section 4 describes the general structure of the hardware architecture of both methodologies; Section 5 proposes a general optimization strategy for both methodologies, using the sine function as an illustrative example; Section 6 describes tools for characterization of the error when approximations are made; Section 7 presents the implementation of the logarithm function in both Parabolic Synthesis and Parabolic Synthesis Combined with Second-Degree Interpolation; Section 8 gives a comparison of implementations performed in the different approximation methodologies, CORDIC, Parabolic Synthesis and Parabolic Synthesis Combined with Second-Degree Interpolation, where the comparison is made with respect to chip area, critical path delay and power consumption; and Section 9 closes the paper with conclusions.

2. Parabolic synthesis

The Parabolic Synthesis methodology [6–9] is an approximation methodology for implementing unary functions, such as trigonometric functions, logarithms and the square root, as well as binary functions, such as division, in hardware. The methodology is mainly intended for use in computation intensive applications such as computer graphics, digital signal processing, communication systems, robotics, astrophysics and fluid physics.

The methodology is founded on multiplications of sub-functions, $s_{n}(x)$, $n = 1,2,\ldots$ each sub-function being a second-order parabolic function. When multiplying these sub-functions, as shown in (1), the original function, $f_{org}(x)$, is obtained. Note that, in the Parabolic Synthesis methodology, the function is based on multiplication of factors, unlike most other methodologies that are based on summation of terms. This is the key to the possibility to parallelize the architecture. Note that $f_{org}(x)$ is almost always a transformation of the function to be approximated, to satisfy the Parabolic Synthesis methodology. The accuracy of an approximation with this methodology depends on the number of sub-functions used.

$$f_{org}(x) = s_{1}(x) \cdot s_{2}(x) \cdots \cdot s_{\infty}(x)$$

Using infinitely many factors in (1) will recreate $f_{org}(x)$. The procedure for obtaining sub-functions is to develop help functions from which sub-functions are developed. To compute the first help function, $f_{1}(x)$, the ratio function between the original function, $f_{org}(x)$, and the first sub-function, $s_{1}(x)$ is computed. This division generates the first help function, $f_{1}(x)$, as shown in (2).

$$f_{1}(x) = \frac{f_{org}(x)}{s_{1}(x)} = s_{2}(x) \cdot s_{3}(x) \cdots \cdot s_{\infty}(x)$$

The following help functions, $f_{n}(x)$, are generated in the same manner, as shown in (3).

$$f_{n}(x) = \frac{f_{n-1}(x)}{s_{n}(x)} = s_{n+1}(x) \cdot s_{n+2}(x) \cdots \cdot s_{\infty}(x)$$

2.1. First sub-function

To facilitate the development of the first sub-function, $s_{1}(x)$, the original function, $f_{org}(x)$, must cut the function $x$ in ($0,0$) and ($1,1$) as shown in Fig. 1.

Thus, the function to be approximated has to be normalized and must satisfy the requirement that the values are in the interval $0 \leq x < 1$ on the x-axis and $0 \leq y < 1$ on the y-axis and have the starting point in ($0,0$). The normalization of the function to be approximated creates the original function, $f_{org}(x)$.

To satisfy the demands of the methodology, the original function, $f_{org}(x)$, must fulfill three additional criteria.

1. It must be strictly concave or convex through the interval in which it is approximated. The original function, $f_{org}(x)$, has to be strictly concave or convex through the interval since the approximation used is a second-order parabolic function.

2. The original function, $f_{org}(x)$, after it is divided by the first sub-function, $s_{1}(x)$, must have a limit value when $x$ goes towards 0. If the function has no limit value, it implies that a help function, $f_{1}(x)$, is not defined when $x = 0$. 
3. The limit value according to criterion 2, subtracted with 1, must be smaller than or equal to 1 or larger than or equal to –1.

If the limit value in criterion 2 is outside this interval, the gradient of the first sub-function, \( s_1(x) \), will not be positive through the entire interval and therefore not be deployable as an approximation of the original function. Fortunately, for almost all interesting functions, criterion 3 is fulfilled. Examples of original functions that do not fulfill criterion 3 are higher order roots than cube roots.

2.1.1. Developing the first sub-function

The first sub-function, \( s_1(x) \), is a second-order parabolic function, as shown in (4).

\[
s_1(x) = l_1 + k_1 \cdot x + c_1 \cdot (x - x^2)
\]

(4)

It is then found from Fig. 1 that the starting point for the first sub-function, \( s_1(x) \), is in \((0,0)\). This gives the start value, \( l_1 \), to be 0. Furthermore, the gradient, \( k_1 \), is 1 since the gradient starts in \((0,0)\) and ends in \((1,1)\) and the range of \( x \) is therefore 1. The first sub-function, \( s_1(x) \), can therefore be rewritten according to (5).

\[
s_1(x) = x + c_1 \cdot (x - x^2)
\]

(5)

The second sub-function, \( s_2(x) \), shall start in \((0,1)\) and end in \((1,1)\). Based on the start criterion, the coefficient \( c_1 \) in (4) can be computed from the limit in (6).

\[
1 = \lim_{x \to 0} \frac{f_{\text{org}}(x)}{x + c_1 \cdot (x - x^2)}
\]

(6)

The limit in (6) is rewritten according to (7) since the \( x^2 \) term in (6) goes faster towards 0 than \( x \) and the \( x^2 \) term can therefore be excluded as shown in (7).

\[
1 = \lim_{x \to 0} \frac{f_{\text{org}}(x)}{x \cdot (1 + c_1)}
\]

(7)

The limit in (7) is rewritten according to (8) and the coefficient \( c_1 \) can be pre-calculated from this.

\[
c_1 = \lim_{x \to 0} \frac{f_{\text{org}}(x)}{x} - 1
\]

(8)

2.2. Second sub-function

The second sub-function, \( s_2(x) \), is developed as an approximation of the first help function, \( f_1(x) \). The computing of \( f_1(x) \) is performed according to (2), and the result of this operation is a concave or convex function. The appearance of this function is similar to a parabolic function, as illustrated in Fig. 2.

As an approximation of the first help function, \( f_1(x) \), the second sub-function, \( s_2(x) \), is chosen as a second-order function [2] for which the start value, \( l_{2,\text{start}} \), is 1 and the end value, \( l_{2,\text{end}} \), is 1, as shown in Fig. 2. Since the interval on the x-axis is normalized to 1, as shown in Fig. 2, the denominator when computing the gradient, \( k_2 \), is 1. The gradient \( k_2 \) is computed as \( k_2 = l_{2,\text{end}} - l_{2,\text{start}} \). It is found from Fig. 2 that both \( l_{2,\text{end}} \) and \( l_{2,\text{start}} \) are 1, which leads to the gradient \( k_2 \) being equal to 0. This implies that the second-order function can be reduced according to (9).

\[
s_2(x) = l_{2,\text{start}} + k_2 \cdot x + c_2 \cdot (x - x^2) = 1 + c_2 \cdot (x - x^2)
\]

(9)

In (9) the coefficient \( c_2 \) is chosen so that the quotient between the first help function, \( f_1(x) \), and the second sub-function, \( s_2(x) \), is equal to 1 for \( f_1(0.5) \), according to (10), which is also shown in Fig. 2.

\[
c_2 = 4 \cdot (f_1(0.5) - 1)
\]

(10)

2.3. Sub-functions, \( s_n(x) \), when \( n > 2 \)

When calculating the second help function, \( f_2(x) \), according to (3), the result is a pair of opposite, strictly concave or convex functions. The first function is in the interval \( 0 \leq x < 0.5 \) and the second function is in the interval \( 0.5 \leq x < 1.0 \), as shown in Fig. 3.

The third sub-function, \( s_3(x) \), is an approximation of the second help function, \( f_2(x) \). As shown in Fig. 3, the help function is divided into two opposite, strictly concave or convex functions. The first function is restricted to the interval \( 0 \leq x < 0.5 \) and the second function is thus restricted to the interval \( 0.5 \leq x < 1.0 \). When the third sub-function, \( s_3(x) \), is developed, it is divided into two partial functions, one for each interval.
A general approach when developing help functions, \( f_n(x) \), when \( n > 1 \), is to develop in each interval a strictly concave or strictly convex function as a partial help function. When developing the higher order sub-functions, \( s_n(x) \), when \( n > 2 \), each partial strictly concave or convex function is divided into two separate parabolic functions. For each sub-interval, a parabolic sub-function is developed as an approximation of the help function, \( f_n(x) \), in the sub-interval. To show the sub-interval for which the partial functions are valid, the subscript index is extended with the index \( m \), which gives the partial help function, \( f_{n,m}(x) \), according to (11). It is shown in Eq. (11) how the help function, \( f_n(x) \), is divided into partial help functions, \( f_{n,m}(x) \), when \( n > 1 \).

\[
\begin{align*}
    f_n(x) &= \begin{cases} 
        f_{n,0}(x), & 0 \leq x < \frac{1}{2^{n-1}} \\[1em]
        f_{n,1}(x), & \frac{1}{2^{n-1}} \leq x < \frac{2}{2^{n-1}} \\[1em]
        \vdots \\
        f_{n,2^{n-1}-1}(x), & \frac{2^{n-2}-1}{2^{n-1}} \leq x < 1
    \end{cases} \\
\end{align*}
\]  

(11)

As shown in (11), the number of partial help functions is doubled for each order of \( n > 2 \), i.e., the number of partial help functions is \( 2^n - 1 \).

The corresponding sub-functions are developed from these parabolic help functions. Analogous to the help function, \( f_n(x) \), the sub-function, \( s_{n,1,m}(x) \), will have partial sub-functions, \( s_{n,2^{n-1}}(x) \). It is shown in Eq. (12) how the sub-function, \( s_n(x) \), is divided into partial functions when \( n > 2 \).

\[
\begin{align*}
    s_n(x) &= \begin{cases} 
        s_{n,0}(x_n), & 0 \leq x < \frac{1}{2^{n-2}} \\[1em]
        s_{n,1}(x_n), & \frac{1}{2^{n-2}} \leq x < \frac{2}{2^{n-2}} \\[1em]
        \vdots \\
        s_{n,2^{n-2}-1}(x_n), & \frac{2^{n-3}-1}{2^{n-2}} \leq x < 1
    \end{cases} \\
\end{align*}
\]  

(12)

Note that, in (12), the input variable \( x \) has been changed to \( x_n \). The change to \( x_n \) is a normalization to the corresponding interval, which simplifies the hardware implementation of the parabolic function.

To simplify the normalization of the interval of \( x_n \), it is chosen as an exponentiation by 2 of \( x \) where the integer part is removed. The normalization of \( x \) is therefore done by multiplying \( x \) with \( 2^{n-2} \), which is hardware in \( n-2 \) left shifts. The integer part is thus dropped, which gives \( x_n \) as a fractional part of \( x \), as shown in (13).

\[
x_n = \text{frac}(2^{n-2} \cdot x)
\]  

(13)

The dropped integer part from the normalization is used as an identifier for the interval in which the sub-function is performed, which is equal to the index \( m \) in the sub-function.

As in the second sub-function, shown in (9), the second-order function is used as an approximation in each interval of a partial help function, \( f_{n-1,m}(x) \). The start value for each partial help function is 1, which implies that the constant term, \( l_{n,m} \), in the sub-function (14) is 1. The end value in each partial help function’s interval is also 1, which implies that the gradient, \( k_{n,m} \), becomes 0 for each interval in the sub-function, \( s_{n,m}(x) \). (14). The range of \( x_n \) in each interval of the partial help function, \( f_{n-1,m}(x) \), is 1, which allows the sub-functions, \( s_{n,m}(x) \), to be reduced according to (14).

\[
s_{n,m}(x) = l_{n,m} + k_{n,m} \cdot x_n + c_{n,m} \cdot (X_n - x_n^2) = 1 + c_{n,m} \cdot (X_n - x_n^2)
\]  

(14)

In (14) the coefficients, \( c_{n,m} \), are chosen so that the quotients between the help functions, \( f_{n-1,m}(x) \), and the sub-sub-functions, \( s_{n,m}(x) \), are equal to 1 when \( x_n \) is equal to 0.5. This is given by (15).

\[
c_{n,m} = 4 \cdot \left( f_{n-1,m} \left( \frac{2 \cdot (m + 1) - 1}{2^n - 1} \right) - 1 \right)
\]  

(15)

3. Parabolic synthesis combined with second-degree interpolation

A side effect in the Parabolic Synthesis methodology is that, to increase the accuracy of the approximation, the number of sub-functions has to be increased, which will increase the size of the hardware. By combining Parabolic Synthesis and Second-Degree Interpolation, the structure is similar to the Parabolic Synthesis methodology in that a synthesis of sub-functions is done (1). However, never more than two sub-functions are required (16). The sub-function developed with Parabolic Synthesis gives a rough approximation, while the approximation is enhanced by increasing the number of intervals in the sub-function developed with Second-Degree Interpolation.

\[
f_{\text{arg}}(x) = s_{1}(x) \cdot s_{2}(x)
\]  

(16)

Other benefits are:

1. The Second-Degree Interpolation part relaxes criteria 1 and 3 described in Section 2.1. These are demands on the original function, \( f_{\text{arg}}(x) \), that must be fulfilled to enable that an approximation can be performed. However, it is an advantage if the criteria are fulfilled since this will imply an efficient implementation.

2. The reduction to two sub-functions enables a reduction in hardware and, with that, a reduction of the power consumption.

3. With the combination of the methodologies, a shortening of the critical path can be predicted since the number of sub-function values to be multiplied is reduced.

4. An advantage of using the Second-Degree Interpolation is that a rough internal error compensation to improve the distribution of the error can be accomplished.

3.1. Second sub-function

As in the Parabolic Synthesis methodology, the second sub-function, \( s_2(x) \), is developed as Second-Degree Interpolations, as shown in (17) and Fig. 4.

\[
s_2(x) = l_{2,1} \cdot x + k_{2,1} \cdot xw + c_{2,1} \cdot (xw - x_n^2)
\]  

(17)
To simplify the decoding of the intervals, \( i \), in hardware, the number of equal range intervals in the second sub-function, \( I \), is chosen as 2 to the power of \( w \), where \( w \) is a natural number \( (18) \).

\[
I = 2^w \tag{18}
\]

To simplify the normalization of the interval of \( x_w \), the interval is selected as an exponentiation by 2 of \( x \) where the integer part is removed. The normalization of \( x \) is therefore done by multiplying \( x \) with \( 2^w \), which in hardware is \( w \) left shifts. Thereafter, the integer part is dropped, which gives \( x_w \) as a fractional part of \( x \), as shown in \((19)\).

\[
x_w = \text{frac}(2^w \cdot x) \tag{19}
\]

This truncation performs normalization to the interval, as shown in Fig. 4. The dropped integer part from the normalization is used as an identifier for the interval in which the second sub-function is performed, which is therefore synonymous with the index \( i \) in the sub-function, as shown in Fig. 4.

The index \( i \) is defined as the number of the interval, starting with 0 and ending with \( I - 1 \). In \((17)\), \( l_{2,i} \) is the starting point of an interval of the interpolation. This is computed by inserting the starting point value of the interval, \( x_{\text{start},i} \), in the help function, \( f_1(x) \), as shown in \((20)\) and Fig. 4.

\[
l_{2,i} = f_1(x_{\text{start},i}) \tag{20}
\]

In \((17)\), \( k_{2,i} \) is the gradient for an interpolation interval. The gradient \( k_{2,i} \) for an interval is computed as the end point value of the help function, \( f_1(x_{\text{end},i}) \), subtracted with the start point value of the help function, \( f_1(x_{\text{start},i}) \). Since the interval is normalized to 1, this excludes the denominator, as shown in \((21)\) and Fig. 4.

\[
k_{2,i} = f_1(x_{\text{end},i}) - f_1(x_{\text{start},i}) \tag{21}
\]

In \((17)\), \( c_{2,i} \) is pre-computed so that the second sub-function for an interval, \( s_2(x_{\text{w},i}) \), cuts the help function, \( f_1(x) \), in the middle of interval \( i \) when \( x_{\text{w}} = 0.5 \), which satisfies the point \( x_{\text{middle},i} \) for the help function, \( f_1(x) \), as shown in \((22)\) and Fig. 4.

\[
c_{2,i} = 4 \cdot (f_1(x_{\text{middle},i}) - l_{2,i} - k_{2,i} \cdot 0.5) \tag{22}
\]

The sub-function in \((17)\) can be simplified according to \((23)\).

\[
s_2(x) = l_{2,i} + j_{2,i} \cdot x_{\text{w}} - c_{2,i} \cdot x_{\text{w}}^2 \tag{23}
\]

In \((23)\), \( j_{2,i} \) is predetermined according to \((24)\).

\[
j_{2,i} = k_{2,i} \tag{24}
\]

It is shown in Eq. \((25)\) how the sub-function, \( s_2(x) \), is divided into partial functions.

\[
s_2(x) =\begin{cases} s_{2,0}(x_{\text{w}}), & 0 \leq x < \frac{1}{2w} \\ s_{2,1}(x_{\text{w}}), & \frac{1}{2^w} \leq x < \frac{2}{2^w} \\ \vdots & \vdots \\ s_{2,l-1}(x_{\text{w}}), & \frac{l-1}{2^w} \leq x < 1 \end{cases} \tag{25}
\]

Note that, in \((25)\), \( x \) has been changed to \( x_{\text{w}} \). The change to \( x_{\text{w}} \) is made because the intervals for the sub-sub-functions, \( s_{2,i}(x) \), in \((25)\) have equal ranges.

4. Hardware architecture

The hardware architecture resulting from the methodologies can be divided into three parts as shown in Fig. 5, and as introduced by Tang \([12]\). In the preprocessing part, the incoming operand is transformed to fit the processing part, where the approximation is performed. The output from the processing part is the incoming operand to the postprocessing part, where it is transformed to the desired output format.

![Fig. 5. The three parts of the architecture.](image1)

4.1. Preprocessing

In the preprocessing part, the incoming operand, \( v \), is transformed in order to prepare the input to the processing part. In most cases this means a normalization of the operand, \( v \), but the transformation of the operand can also be more comprehensive, such as converting a fixed-point number into a floating-point number. If the approximation is implemented as a block in a larger system, the preprocessing part can be integrated in the previous blocks, which implies that the preprocessing part can be reduced or even be excluded.

4.2. Processing

The approximation of the original function, \( f_{\text{org}}(x) \), is computed in the processing part. Two approaches for the processing part will be described: first the processing part according to Parabolic Synthesis and then according to the Parabolic Synthesis Combined with Second-Degree Interpolation.

4.2.1. Parabolic synthesis

The computation of the original function, \( f_{\text{org}}(x) \), can be performed in either an iterative or a loop unrolled hardware architecture. According to the three equations \((5)\), \((9)\) and \((14)\), the Parabolic Synthesis methodology has the same structure in each step, which makes it possible for the approximation to be implemented as an iterative architecture, as shown in Fig. 6.

The benefit of the iterative architecture is the small chip area, whereas the disadvantage is a longer computation time. The advantages of loop unrolled hardware architectures are that they give faster computation, albeit at the price of a larger chip area. The principle of the loop unrolled hardware architecture for four sub-functions is shown in Fig. 7.

Fig. 8 shows a more detailed description of the loop unrolled architecture of four sub-functions, which are implemented by using the Parabolic Synthesis methodology.
As seen in (5) and (9), the \((x-x^2)\) parts are common for the first and second sub-functions, which is taken into account in the hardware architecture shown in Fig. 8. The result of the \((x-x^2)\) part is then multiplied with \(c_1\) in the first sub-function and with \(c_2\) in the second sub-function. After the multiplication with \(c_1\) in the first sub-function, \(x\) is added, whereas a 1 is added in the second sub-function after the multiplication with \(c_2\). In (14), for the third and fourth sub-functions, the partial products of \(x_3^2\) and \(x_4^2\) must be computed. A special squaring unit, termed \(x^2\), has been developed to carry out this computation, which is described in [6,8,9]. An advantage of the squaring unit is that its chip area, as well as the latency, is halved compared to a corresponding multiplier. In the third sub-function, the result of \((x_3-x_3^2)\) is multiplied with the coefficient \(c_{3,i}\). The index \(i\) is the most significant bit in \(x\) and indicates in which of the two intervals \(x\) is, i.e. if \(i=0\), \(x\) is in the first interval and, if \(i=1\), \(x\) is in the second interval. The index \(i\) also gives the address in the look-up table for \(c_{3,i}\). In the fourth sub-function, the result of \((x_4-x_4^2)\) is multiplied with the coefficient \(c_{4,h}\). The index \(h\) is defined by the two most significant bits in \(x\) and gives in which of the four intervals \(x\) is. If \(h=00\), \(x\) is in the first interval, if \(h=01\), \(x\) is in the second interval, and so forth. The index \(h\) also gives the address in the look-up table for \(c_{4,h}\). When the sub-functions are computed, the first and second sub-functions are multiplied in parallel with the third and fourth sub-functions. The results of these two multiplications are in turn multiplied with each other.

4.2.2. Parabolic synthesis combined with second-degree interpolation

The architecture of the Parabolic Synthesis Combined with Second-Degree Interpolation contains two sub-functions implemented in parallel. The first sub-function is implemented using the Parabolic Synthesis methodology and the second sub-function is implemented using the Second-Degree Interpolation methodology. Fig. 9 shows the complete architecture.

In the first sub-function, the result of the \((x-x^2)\) part is multiplied with \(c_1\) and, after that, \(x\) is added. The second sub-function is implemented as a Second-Degree Interpolation, consisting of a look-up table containing the coefficients \(j_{2,i}\) from (24) for each interval \(i\). The coefficient \(j_{2,i}\) is multiplied with \(x_{w,i}\) which is the remaining part of \(x\) when the index part \(i\) is removed. The value of \(x_{w,i}\) is also the normalized value of \(x\) for the interval. After the multiplication, the start value of the interval \(j_{2,i}\) is added. The third branch of the second sub-function consists of a look-up table containing the coefficients \(c_{2,i}\) from (22) for each interval \(i\). The coefficient \(c_{2,i}\) for the interval \(i\) is multiplied with \(x_{w,i}^2\), which is the partial product of \(x^2\) for the interval \(i\). The result of the multiplication of \(c_{2,i}\) with \(x_{w,i}^2\) is then subtracted from the result of the previous addition, as shown in Fig. 9. When the first and second sub-functions have been computed, they are multiplied with each other.
4.3. Postprocessing

In the postprocessing part, the incoming operand, y, is transformed to the desired output format. In many cases, no or very little transformation is needed, since the output from the processing part is in the right format. In some cases, the postprocessing can be more comprehensive, such as in converting a floating-point number into a fixed-point number.

4.4. Pipelining

It is feasible for both methodologies to introduce pipelining in the architectures. An example of where pipeline stages can be introduced in the Parabolic Synthesis Combined with the Second-Degree Interpolation architecture is shown in Fig. 9. As seen, the introduction of pipeline stages is simple since it is easy to find natural cuts in the architecture. The introduction of data pipelining stages in the architectures will have a low impact on the size of the hardware, since the data paths are few.

4.5. Reuse

As shown in [8], many implementations of approximations use the same architecture of the processing part. A result of this is that this part of the architecture can be reused by only replacing the set of coefficients.

5. Optimizing

To increase the accuracy of the approximation, an optimization can be performed of the coefficients in the last sub-function. This optimization is carried out in a trial and error manner and should be performed in parallel with the evaluation of the word length, because the truncation error effects influence the performance of the calculations in the design. The strategy is to systematically vary coefficients and word lengths in the design for best accuracy and distribution of the error. To simplify the understanding of the optimization strategy, the effects of the truncation error will not be taken into consideration in this section. In the text below, the optimization strategy will be illustrated using the sine function since this function is commonly used and has a simple implementation for both methodologies. The target accuracy for the implementations is chosen to be around 15 bit.

For the optimization, a bit-accurate C model has been developed for evaluations of the approximations, and MatLab has been used to analyze the performance of the approximations.

5.1. Parabolic synthesis

The optimization strategy for the Parabolic Synthesis methodology is demonstrated on a design with four sub-functions. Four sub-functions are chosen since this will give an accuracy of around 15 bit for the sine function. When optimizing a design, the optimization is only performed on the sub-function with the highest index. The sub-function with the highest index in this case is the fourth sub-function, shown in (26).

\[ s_{4,m}(x_{4}) = 1 + c_{4,m} \cdot (x_{4} - x_{4}^2) \]  

(26)

The initial values of the coefficients are calculated using (8) and (15). The optimization is then performed by varying the four coefficients \( c_{4,0} \) through \( c_{4,3} \) for a minimum error in each interval. To minimize the error, the coefficients are chosen in such a way that the error is leveled out between the intervals, as illustrated in Fig. 10.

In Fig. 10, the largest error in the interval \( 0 \leq x < 0.25 \) is reduced by nearly one bit in resolution after the optimization. In terms of the largest error of the approximation, the result of the optimization is negligible.

5.2. Parabolic synthesis combined with second-degree interpolation

The optimization strategy for the Parabolic Synthesis methodology when combined with Second-Degree Interpolation is to perform the optimization on the second sub-function, \( s_{2}(x) \). Four intervals in the second sub-function, \( s_{2}(x) \), are chosen for the sine function since this will give an accuracy of around 15 bit in resolution. The second sub-function, \( s_{2}(x) \), is shown in (27).

\[ s_{2}(x) = l_{2,1} + j_{2,1} \cdot x_{w} - c_{2,1} \cdot x_{w}^2 \]  

(27)

The initial values of the coefficients are calculated by using (8), (20), (21), (22) and (24). The optimization is performed by systematically varying the 12 coefficients, \( l_{2,0} \) through \( l_{2,3} \), \( j_{2,0} \) through \( j_{2,3} \), and \( c_{2,0} \) through \( c_{2,3} \). Note the dependency of \( j_{2,1} \) according to (24). To facilitate the optimization of the second sub-function, it should be performed on the non-simplified second sub-function shown in (28).

\[ s_{2}(x) = l_{2,1} + j_{2,1} \cdot x_{w} + c_{2,1} \cdot (x_{w} - x_{w}^2) \]  

(28)

The optimization is mainly performed by varying the four coefficients \( c_{2,0} \) through \( c_{2,3} \) since these coefficients adjust the height of the parabolic part of the second sub-function. To minimize the error, these coefficients are chosen in such a manner that the largest error in the lower and upper range of the interval is leveled out as shown in Fig. 11.

The largest error in the interval \( 0 \leq x < 0.25 \) is reduced in Fig. 11 by nearly half a bit in resolution after the optimization. In terms of the largest error of the approximation, the result of the optimization is negligible.

5.3. Optimization strategy

The optimization is performed as a bit-accurate model simulation. Since the effects of truncations in the design are difficult to anticipate, the optimization procedure is to be considered as a trial and error strategy. In this strategy, optimizing the coefficients and the word lengths is done in parallel, since the truncation error effects influence the performance of calculations in the design. To obtain the desired performance, the strategy is to adjust the word lengths and the coefficients in a bit-accurate model. This process is
iterative and is ongoing until the desired performance is obtained in terms of the accuracy and distribution of the error.

6. Characterization and distribution of the error

The characteristics of the error of an approximation are important for the performance of the algorithm in which the approximation is included. It is therefore important to have metrics by which the error and its distribution can be characterized. Five metrics are used to characterize the error. These are the maximum absolute error, the mean error, the median error, the standard deviation and the Root Mean Square (RMS) error [13].

- The maximum absolute error gives the largest error possible in the approximation.
- The mean error gives the average error over the approximation’s sample space.
- The median error gives the skewness in the error distribution, if there is any.
- The standard deviation is a measure of the variation from the mean error.
- The RMS error is a measure of the magnitude of a varying quantity of the error.

In addition, the evenness of the error distribution is obtained by comparing the standard deviation with the RMS error. If the standard deviation and the RMS are equal, this indicates that the error distribution is even around zero.

An advantage of the two methodologies is that the probability of obtaining an evenly distributed error around zero is very high.

The sine function is chosen to illustrate the error distribution of the methodologies since it is commonly used and is interesting for direct digital synthesis (DDS). Another reason why the sine function is chosen is that it illustrates well the error characteristics of a function implemented using the two methodologies.

6.1. Parabolic synthesis

To illustrate the distribution of the error, an implementation is carried out according to the Parabolic Synthesis methodology using four sub-functions. The error, computed as the approximation subtracted with the original function, is shown in Fig. 12.

As shown in Fig. 12, the approximate value oscillates around the original function in a desirable manner, and the error is evenly distributed around zero, which is confirmed by the distribution of the error shown in Fig. 13.

6.2. Parabolic synthesis combined with second-degree interpolation

The appearance of the distribution of the error of Parabolic Synthesis combined with Second-Degree Interpolation is nearly a copy of the distribution of the error of the Parabolic Synthesis methodology shown in Figs. 12 and 13. An advantage of Parabolic Synthesis combined with Second-Degree Interpolation is that the second sub-function, \( s_2(x) \), offers greater opportunities to tailor the error of an approximation, since the second sub-function, \( s_2(x) \), has more adjustability than the sub-function with the highest order in the Parabolic Synthesis methodology. When truncation effects and error tailoring are taken into account, the more extensive opportunities to optimize the result are a major advantage of Parabolic Synthesis combined with Second-Degree Interpolation.

7. Implementation of the logarithm

The implementation of the algorithm that will now be demonstrated performs an approximation of the logarithm function on a binary floating-point number. The approximation is performed on the mantissa only, since the exponential part is just a scaling of the mantissa [14,15]. Approximation of the binary logarithm in the interval from 1.0 to 2.0 will be implemented using the two methodologies, Parabolic Synthesis and Parabolic Synthesis.
Combined with Second-Degree Interpolation. The target accuracy of the implementations is around 15 bit.

7.1. Parabolic synthesis

The approximation of the logarithm using the Parabolic Synthesis methodology will need four sub-functions in order to achieve an accuracy of around 15 bit.

7.1.1. Preprocessing

As described in Section 2.1, to facilitate the hardware implementation of the approximation, normalization must be performed to satisfy the requirement that the values are in the interval \( 0 \leq x < 1 \) on the \( x \)-axis and \( 0 \leq y < 1 \) on the \( y \)-axis. The result of the binary logarithm satisfies that the values are in the interval \( 0 \leq y < 1 \). To ensure that the incoming operand, \( x \), is within the interval \( 0 \leq x < 1 \), a 1 must be added to the operand, as shown in (29).

\[
v = 1 + x
\]  

To normalize the \( f(v) = \log_2(v) \) function, \( v \) is substituted according to (29), which gives the original function \( f_{\text{org}}(x) \) shown in (30).

\[
f_{\text{org}}(x) = \log_2(1 + x)
\]  

Fig. 14 shows the function, \( f(v) \), together with the normalized function, \( f_{\text{org}}(x) \).

7.1.2. Processing

For the processing part, the sub-functions are developed according to the Parabolic Synthesis methodology. In the first sub-function, \( s_1(x) \), coefficient \( c_1 \) is defined according to (6). The determined value of the coefficient is shown in (31).

\[
s_1(x) = x + c_1 \cdot (x - x^2) = x + \left( \frac{1}{\ln(2)} - 1 \right) \cdot (x - x^2)
\]  

The first help function, \( f_1(x) \), is computed as shown in (32).

\[
f_1(x) = \frac{f_{\text{org}}(x)}{s_1(x)} = \frac{\log_2(1 + x)}{x + \left( \frac{1}{\ln(2)} - 1 \right) \cdot (x - x^2)}
\]  

Fig. 15 shows the first help function, \( f_1(x) \).

To develop the second sub-function, \( s_2(x) \), coefficient \( c_2 \) is defined according to (10). The determined value of the coefficient \( c_2 \) in (9) is shown in Table 1.

The second help function, \( f_2(x) \), is computed as shown in (33).

\[
f_2(x) = \frac{f_1(x)}{s_2(x)} = \frac{f_1(x)}{1 + c_2 \cdot (x - x^2)}
\]  

Fig. 16 shows the second help function, \( f_2(x) \).

To develop the third sub-function, \( s_3(x) \), the second help function, \( f_2(x) \), is divided into its two partial functions, as shown in (11). The third sub-function is thereby split into two sub-functions, of which \( s_{3,0}(x_3) \) is restricted to the interval \( 0 \leq x < 0.5 \) and \( s_{3,1}(x_3) \) is restricted to the interval \( 0.5 \leq x < 1 \), according to (12). A normalization of \( x \) to \( x_3 \) is carried out to simplify the implementation in hardware, which is described in (13).

For each sub-function, the corresponding coefficients, \( c_{1,0} \) and \( c_{1,1} \), are determined according to (15). The partial sub-functions of the third sub-function are shown in (34) and the coefficients in Table 1.

\[
s_{3,0}(x_3) = 1 + c_{3,0} \cdot (x_3 - x_3^2), \quad 0 \leq x < 0.5
\]

\[
s_{3,1}(x_3) = 1 + c_{3,1} \cdot (x_3 - x_3^2), \quad 0.5 \leq x < 1
\]  

Table 1

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c_1 )</td>
<td>0.4429053040888963</td>
</tr>
<tr>
<td>( c_2 )</td>
<td>-0.68341247025477</td>
</tr>
<tr>
<td>( c_{1,0} )</td>
<td>-0.00066873472054</td>
</tr>
<tr>
<td>( c_{1,1} )</td>
<td>0.003241263125913</td>
</tr>
<tr>
<td>( c_{3,0} )</td>
<td>-0.00165040000000</td>
</tr>
<tr>
<td>( c_{3,1} )</td>
<td>0.00825040000000</td>
</tr>
<tr>
<td>( c_{3,2} )</td>
<td>0.00012300000000</td>
</tr>
<tr>
<td>( c_{3,3} )</td>
<td>-0.00326500000000</td>
</tr>
</tbody>
</table>

Fig. 16 shows the third help function, \( f_3(x) \).

To develop the fourth sub-function, \( s_4(x) \), the third help function, \( f_3(x) \), is divided into its four partial functions as shown in (11). The fourth sub-function is thereby obtained as the four
sub-functions specified in (35). To simplify the implementation in hardware, a normalization of x to x_4 is carried out, which is described in (13).

For each sub-function, the corresponding coefficients, c_{4,0} through c_{4,3}, are determined according to (15). They are shown in Table 1.

\[
s_{4,0}(x) = 1 + c_{4,0} \cdot (x_4 - x_4^2), \quad 0 \leq x < 0.25 \\
s_{4,1}(x) = 1 + c_{4,1} \cdot (x_4 - x_4^2), \quad 0.25 \leq x < 0.5 \\
s_{4,2}(x) = 1 + c_{4,2} \cdot (x_4 - x_4^2), \quad 0.5 \leq x < 0.75 \\
s_{4,3}(x) = 1 + c_{4,3} \cdot (x_4 - x_4^2), \quad 0.75 \leq x < 1
\]

After the coefficients, c_{4,0} through c_{4,3}, are determined, an optimization, as described in Section 5, Section 5.1, is performed to minimize the error of the approximation. The values of the coefficients are shown in Table 1.

No postprocessing is needed since the result from the processing part has the right output format.

7.2. Characterization and distribution of the error

Analysis of the error for the implementation using the Parabolic Synthesis methodology shows that the difference between the approximation and the original function oscillates around zero, see Fig. 17.

As shown in Fig. 18, the accuracy of the approximation is nearly 16 bit.

The error distribution of the approximation is, as shown in Fig. 19, even around zero. This is a preferred distribution of the error in most algorithms since it normally reduces the size of accumulated errors in the result.

Table 2 shows statistics of the implementation.

The conclusion from Table 2 is that the maximum absolute accuracy is near 16 bit, the mean error is very small, and the median confirms that the error distribution is not skewed. Comparison of the standard deviation value with the root mean square value shows that they are nearly identical, which confirms that the error of the approximation is evenly distributed around zero.

7.3. Parabolic synthesis combined with second-degree interpolation

When the logarithm is implemented using the Parabolic Synthesis Combined with the Second-Degree Interpolation methodology, the first sub-function is implemented using the Parabolic Synthesis methodology and the second sub-function as a Second-Degree Interpolation. The coefficient c_1 is retrieved from the implementation done exclusively with Parabolic Synthesis, as shown earlier in this section. The second sub-function is implemented as a Second-Degree Interpolation of the first help function, f_1(x), shown in Fig. 15.

7.3.1. Developing the second sub-function

When investigating the number of intervals needed in the second sub-function, simulations were made of the error using different numbers of intervals. The simulations showed that, with two intervals, the accuracy was better than 12 bit; with four intervals it was better than 15 bit; and, with eight intervals, the accuracy was better than 18 bit. Since the target accuracy is around 15 bit four intervals were chosen. The sub-functions for the Second-Degree Interpolation with four intervals are shown in (36).

\[
s_{2,0}(x) = l_{2,0} + j_{2,0} \cdot x_2 - c_{2,0} \cdot x_2^2, \quad 0 \leq x < 0.25 \\
s_{2,1}(x) = l_{2,1} + j_{2,1} \cdot x_2 - c_{2,1} \cdot x_2^2, \quad 0.25 \leq x < 0.5 \\
s_{2,2}(x) = l_{2,2} + j_{2,2} \cdot x_2 - c_{2,2} \cdot x_2^2, \quad 0.5 \leq x < 0.75 \\
s_{2,3}(x) = l_{2,3} + j_{2,3} \cdot x_2 - c_{2,3} \cdot x_2^2, \quad 0.75 \leq x < 1
\]
The coefficients, $b_{2,0}$ through $b_{2,3}$, are computed according to (20), the coefficients, $k_{2,0}$ through $k_{2,3}$, are computed according to (21), and the coefficients, $c_{2,0}$ through $c_{2,3}$, are computed according to (22). This gives the coefficients, $b_{2,0}$ through $b_{2,3}$, according to (24). After the coefficients, $c_{2,0}$ through $c_{2,3}$, are determined, an optimization is performed to minimize the error of the approximation. The optimization is performed according to Section 5.2, and the values of the computed coefficients are shown in Tables 3–6. In Table 5 and Table 6, if the negative signs of the values of the coefficients are ignored, the most significant bits in the coefficients are zeros. This implies that a significant reduction of the word length of the coefficients can be carried out, which reduces the hardware significantly.

No postprocessing is needed since the result from the processing part has the right format.

### Table 3
Coefficients $b_{2,0}$ through $b_{2,3}$.

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$b_{2,0}$</td>
<td>1.000000000000</td>
</tr>
<tr>
<td>$b_{2,1}$</td>
<td>0.966735509669</td>
</tr>
<tr>
<td>$b_{2,2}$</td>
<td>0.957896898187</td>
</tr>
<tr>
<td>$b_{2,3}$</td>
<td>0.969207401815</td>
</tr>
</tbody>
</table>

### Table 4
Coefficients $k_{2,0}$ through $k_{2,3}$.

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_{2,0}$</td>
<td>-0.033264409330</td>
</tr>
<tr>
<td>$k_{2,1}$</td>
<td>-0.00813602482</td>
</tr>
<tr>
<td>$k_{2,2}$</td>
<td>0.01130503628</td>
</tr>
<tr>
<td>$k_{2,3}$</td>
<td>0.03072959184</td>
</tr>
</tbody>
</table>

### Table 5
Coefficients $c_{2,0}$ through $c_{2,3}$.

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c_{2,0}$</td>
<td>-0.013880000000</td>
</tr>
<tr>
<td>$c_{2,1}$</td>
<td>-0.010700000000</td>
</tr>
<tr>
<td>$c_{2,2}$</td>
<td>-0.009640000000</td>
</tr>
<tr>
<td>$c_{2,3}$</td>
<td>-0.010027000000</td>
</tr>
</tbody>
</table>

### Table 6
Coefficients $j_{2,0}$ through $j_{2,3}$.

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j_{2,0}$</td>
<td>-0.047144409330</td>
</tr>
<tr>
<td>$j_{2,1}$</td>
<td>-0.019548892482</td>
</tr>
<tr>
<td>$j_{2,2}$</td>
<td>0.00166503628</td>
</tr>
<tr>
<td>$j_{2,3}$</td>
<td>0.02076489184</td>
</tr>
</tbody>
</table>

7.3.2. Characterization and distribution of the error

Fig. 20 shows the error of the implementation.

As shown in Fig. 21, the accuracy of the approximation is nearly 16 bit.

The error of the approximation is, as shown in Fig. 22, evenly distributed around zero, which, as mentioned, is a preferred distribution of the error in most algorithms.

Table 7 shows statistics for the implementation.

The conclusion from Table 7 is that, also in this case, the accuracy of the approximation is nearly 16 bit, the mean error is small and the error distribution is even. Again, the standard deviation value is nearly identical to the root mean square value.

### Table 7
Error statistics.

<table>
<thead>
<tr>
<th>Value</th>
<th>Expressed in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max absolute error</td>
<td>0.00000015895240</td>
</tr>
<tr>
<td>Mean error</td>
<td>0.000000253845</td>
</tr>
<tr>
<td>Median error</td>
<td>0.000000183136</td>
</tr>
<tr>
<td>Standard deviation</td>
<td>0.00000090003823</td>
</tr>
<tr>
<td>RMS error</td>
<td>0.0000009003005</td>
</tr>
</tbody>
</table>
8. Comparing methodologies

The performance of different approximation methodologies can be compared using the metrics described in Section 6. However, since it is difficult to predict the effects of these metrics in a particular context, a comparison of different approximation methodologies is preferably made in the same context. The reason for this is that, in a comparison of different approximation methodologies in the same context, the influence of the individual characteristics of the error can result in a redesign of the approximation or the context. These redesigns can often have a huge influence on parameters such as chip area, critical path delay and power consumption. Comparing different approximation methodologies, each with its unique error characteristics, without a given context, can therefore be misleading. Despite this, the two Parabolic Synthesis methodologies (which have equivalent error characteristics) will, without a given context, be compared with the CORDIC methodology. The error can to a much greater extent be tailored to fit the context with the two Parabolic Synthesis methodologies than with the CORDIC methodology. This advantage for the two Parabolic Synthesis methodologies will however not be taken into account in the comparison with the CORDIC methodology. This comparison of the methodologies is thus only done to get an indication of the feasibility of the new methodologies. The CORDIC methodology is chosen since it is frequently used in applications for which the two Parabolic Synthesis methodologies are suitable [16].

8.1. Characterization and distribution of the error

The CORDIC methodology has the disadvantage that the error is not evenly distributed around zero; rather, it is distributed on one side of zero. This is because, when computing the approximation, the result is computed bit by bit in decreasing order from the most significant bit to the least significant bit. This kind of unilateral distribution of the error can lead to severe problems in the subsequent calculations, in the form of accumulated errors. The way to reduce the impact of this problem is to increase the accuracy, which leads to an increase in chip area, critical path delay and power consumption.

For the two Parabolic Synthesis methodologies, the error is evenly distributed around zero. The distribution of the error can also be customized to some extent in the Parabolic Synthesis methodology and, to a broad extent, in the Parabolic Synthesis combined with Second-Degree Interpolation. This gives the two Parabolic Synthesis methodologies a great advantage over CORDIC.

8.2. Resource need and performance

As a comparative case study regarding resource needs and performance, an implementation was carried out of the fractional part of the 2-logarithm in the range from 1.0 through 2.0. The required accuracy for the implementation was set to 15 bit. The methodologies were implemented in their basic versions with no features such as pipelining. The implementations were performed as an ASIC with a 65 nm Standard-Vt (1.2 V) technology. Synopsis Design Compiler [17], Synopsis Primetime [18] and Mentor Graphics Modelsim [19] were used for all implementations.

8.2.1. CORDIC implementation

The CORDIC implementation is an unrolled implementation using 20 iterations (15+1 for the accuracy and repeating iterations 4, 7, 10 and 13 to ensure convergence) to obtain the desired accuracy of 15 bit. The implementation is described in [9]. Since the designs in [9] were implemented with bonding pads, a redesign without bonding pads has been performed. The new design yields the results shown in Table 8. The results for the CORDIC implementation are used as a baseline and we therefore refer to them with the label “100%”.

8.2.2. Parabolic synthesis implementation

Four sub-functions are needed (see Section 7.1) in the implementation using the Parabolic Synthesis methodology, in order to reach 15-bit precision. Unfortunately, in the design in [9], no optimization of the word lengths was performed; therefore longer word lengths than needed were used in the design (the word lengths could have been nearly halved in some parts of the design). This yields a large impact on the chip area and the critical path delay of the design. Actually, the implementation made with CORDIC can also be improved – however not as much as the implementation using the Parabolic Synthesis. It is therefore true to say that the implementation with the Parabolic Synthesis methodology has not been given advantages in the comparison. Again, since the designs in [9] was implemented with bonding pads, this was also redesigned without bonding pads, yielding the results shown in Table 8.

8.2.3. Parabolic synthesis combined with second-degree interpolation implementation

To reach 15 bit precision, four intervals are used in the second sub-function (see Section 7.3) for the implementation based on the new methodology described in this paper, Parabolic Synthesis Combined with Second-Degree Interpolation. In this methodology, many implementations of approximations use the same architecture, as mentioned in Section 4.5. An example of identical architectures is the logarithm and the sine function. Accordingly, this part of the architecture can be reused only by replacing the set of coefficients. This equivalence, of course, allows drawing conclusions regarding the chip area, critical path delay and energy consumption for an implementation of the 2-logarithm from an implementation of the sine function.

An implementation of the sine and cosine functions using Parabolic Synthesis Combined with Second Degree Interpolation is...
described in [20], and we can thus use the results of this to estimate chip area and critical path delay of the logarithm implementation with the same number of intervals (in this case, four). The chip area can be estimated by extracting from [20] one first sub-function, 2000 μm², one second sub-function, 5000 μm², and the multiplier that performs the multiplication of the result of the two sub-functions, 2000 μm². By summing the areas of these three parts, the chip area of the logarithm implementation is calculated to 9000 μm². The delay of the most critical path of the logarithm implementation cannot be extracted from [20] since the implementation in [20] has extra parts in the critical path compared to the logarithm implementation. The additional parts in the implementation of the sine or cosine functions are a preprocessing and a postprocessing part, the lengths of which cannot be determined from the critical path. The critical path delay of the logarithm implementation is therefore conservatively estimated to the entire critical path delay of the design in [20], which is 10 ns.

The estimation of the energy consumption is based on the implementation of the logarithm performed with Parabolic Synthesis, because the resemblance to the hardware architecture of Parabolic Synthesis Combined with Second-Degree Interpolation is close. The energy consumption per sample can roughly be estimated to correlate linearly with the area and linearly with the delay of the critical path. As shown in Table 8, compared to the Parabolic Synthesis, the area is reduced by 45% (from 16,258 to 9000 μm²) and the delay of the critical path is reduced by 52% (from 21 to 10 ns). A rough estimate of energy consumption is therefore 0.55 × 0.48 × 0.065 nW = 0.017 nW.

Table 8 summarizes the comparison between the three methodologies regarding the chip area, critical path delay and energy consumption per sample, all implemented with an accuracy of 15 bits. The indication from Table 8 is that the Parabolic Synthesis Combined with Second-Degree Interpolation is clearly superior to the others. For example, its energy efficiency is more than six times better than CORDIC and more than two times better than Parabolic Synthesis.

9. Conclusion

This paper describes two methodologies, Parabolic Synthesis and Parabolic Synthesis Combined with Second-Degree Interpolation, for developing approximations of unary functions in hardware, such as trigonometric functions, logarithmic functions, exponential functions and the square root. The methodologies are mainly intended for computation intensive applications in, e.g., computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics and many other application areas.

When going from Parabolic Synthesis to Parabolic Synthesis Combined with Second-Degree Interpolation, the major attractiveness lies in how increasing the accuracy is done. In Parabolic Synthesis, the way to increase the accuracy of the approximation is to introduce more sub-functions. In Parabolic Synthesis Combined with Second-Degree Interpolation, which uses only two sub-functions, it is instead done by increasing the number of intervals in the second sub-function. The effect of this is that, with increasing accuracy, the resulting chip area, critical path delay and energy consumption will show a much slower increase than in Parabolic Synthesis. The Parabolic Synthesis Combined with Second-Degree Interpolation also has more extensive opportunities to handle truncation effects and to achieve a desired error distribution than the Parabolic Synthesis has. The use of second-degree interpolation in the second sub-function also has the positive effect that the range of functions that can be approximated is significantly expanded compared to when only Parabolic Synthesis is used. The architecture of Parabolic Synthesis combined with Second-Degree Interpolation is, like Parabolic Synthesis, very suitable for pipelining. Further, the architecture of both methodologies can easily be reused for approximations of other unary functions by simply changing the set of coefficients.

It was found when analyzing the hardware architecture for implementing the logarithm that, by combining the Parabolic Synthesis methodology with Second-Degree Interpolation, the complexity of the architecture could be significantly reduced. From the implementation of the logarithm with the two methodologies, it can also be concluded that, to achieve higher accuracy in the Parabolic Synthesis Combined with Second-Degree Interpolation, only the word length in the computations and the number of coefficients need to be increased. However, for implementations carried out with Parabolic Synthesis only, increasing the accuracy also requires the introduction of an additional number of sub-functions. When analyzing the distribution of the error for the implementation of the logarithm, it was found that the Parabolic Synthesis Combined with Second-Degree Interpolation has a mean error that is near zero and a median that is zero or near zero. This implies that only a minimum of skewness is present in the error distribution and that the difference between the standard deviation and the root mean square value is very small.

To get an indication of the implementation performance of the two methods, implementations based on these methodologies were compared with an implementation carried out with the CORDIC methodology. Such a comparison - even though it is done on implementations with the same accuracies - actually has multiple dimensions since the error distribution for CORDIC differs significantly from the others. The comparison shows that, with the Parabolic Synthesis Combined with Second-Degree Interpolation, the chip area will be smaller than that of CORDIC. The throughput is also better for both implementations carried out with the methodologies containing Parabolic Synthesis. In addition, the energy consumption per sample is less than with CORDIC.

References

Erik Hertz received his MS degree in electrical engineering in 1983 and his Licentiate degree in 2011, both at the Faculty of Engineering, Lund University, Sweden. Since 1989 he has done research at the Swedish Defense Research Agency, Linköping, Sweden, Ericsson Mobile Platforms AB, Lund, Sweden, and Cargine AB, Angelholm, Sweden, connected to Koenisegg Automotive AB. His research has been in a wide range of areas such as radar, radio, control engineering, sensors, microwave, ASIC and solid state physics. His main interest is in algorithm development for hardware implementation. In 2012 he joined the Centre for Research on Embedded Systems (CERES), Halmstad University, Halmstad, Sweden, where he is currently pursuing his doctoral degree.

Bertil Svensson received his MS degree in electrical engineering in 1970 and PhD degree in computer engineering in 1983, both from Lund University, Sweden. He was a member of the faculty of Halmstad University, Sweden, from 1983, served as acting professor at Luleå University of Technology, Sweden, from 1989 and was professor of computer systems engineering at Chalmers University of Technology, Sweden, from 1991. Since 1998, he has been professor of computer systems engineering at Halmstad University, where he has been the dean of the school of information science, computer and electrical engineering and the initiator and director of several research centers. He is currently the director of CERES, the center for research on embedded systems, and of Research for Innovation, a strategic, university over-arching research program. His research interests are parallel computer architecture, embedded parallel computing and cooperating embedded systems, including wireless and intelligent systems. He is a member of the IEEE and the ACM.

Peter Nilsson (’590-M’96-M’11) has been, since July 2008, a full professor in electronic design. In February 1988, Peter reached his Master of Science degree in Electrical Engineering and in May 1996, his degree Doctor of Philosophy in Engineering was awarded, both at the Faculty of Engineering, Lund University. Since 1996 Peter has supervised seven PhD students to the completion of their doctoral degree, and now has two on the way. He has supervised and/or examined 156 students at the Master’s thesis level, both nationally and internationally. Peter has authored or co-authored 126 peer-reviewed journal and conference papers. He was the program manager as well as the person authorized to sign for the “Socware Research & Education” program, a five-year national 15 million USD program. He was an Associate Editor for IEEE Transactions on Circuits and Systems I, 2004–05, and is a member of the VLSI Systems and Applications Technical Committee in IEEE Circuits and Systems Society. He has been awarded the IEEE Senior Member grade. He is also a Technical Program Committee (TPC) member of the IEEE International Solid State Circuits Conference (ISSCC). Peter’s main interest is implementation of digital ASICs, often with a focus on wireless communication.
Paper H
The harmonized parabolic synthesis methodology for function generation in hardware

Erik Hertz, Jingou Lai, Bertil Svensson, and Peter Nilsson

Erik Hertz
Centre for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
Phone: +46 702890215
FAX: +46 35120348
E-mail: erik.hertz@hh.se

Jingou Lai
Department of Electrical and Information Technology, Lund University, Lund, Sweden
Phone: +86 13162983380
FAX: +46 46129948
E-mail: ljobro@gmail.com

Bertil Svensson
Centre for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
Phone: +46 706480843
FAX: +46 35120348
E-mail: bertil.svensson@hh.se

Peter Nilsson
Department of Electrical and Information Technology, Lund University, Lund, Sweden
Phone: +46 702890215
FAX: +46 35120348
E-mail: erik.hertz@hh.se

Abstract
The Harmonized Parabolic Synthesis methodology is a further development of the Parabolic Synthesis methodology for approximation of unary functions such as trigonometric functions, logarithms and the square root, as well as binary functions such as division, in hardware. These functions are extensively used in computer graphics, digital signal processing, communication systems, robotics, astrophysics, fluid physics and many other application areas. For these high-speed applications, software solutions are in many cases not sufficient and a hardware implementation is therefore needed. The Harmonized Parabolic Synthesis methodology has two outstanding advantages: it is parallel, thus reducing the execution time, and it is based on low
complexity operations, thus is simple to implement in hardware. A notable difference in the Harmonized Parabolic Synthesis methodology compared to many other approximation methodologies is that it is a multiplicative and not an additive methodology. Without harming the favorable distribution of the approximation error presented in earlier described Parabolic Synthesis methodologies it is possible to significantly enhances the performance of the Harmonized Parabolic Synthesis methodology, in terms of reducing chip area, computation delay and power consumption. Furthermore it increases the possibility to tailor the characteristics of the error, which improves the conditions for subsequent calculations. It also extends the set of unary functions that approximations can be performed upon since the possibilities to elaborate with the characteristics and distribution of the error increases. To evaluate the proposed methodology, the fractional part of the logarithm has been implemented and its performance is compared to the Parabolic Synthesis methodology. The comparison is made with 15-bit resolution. The design implemented using the Harmonized Parabolic Synthesis methodology performs 3x better than the Parabolic Synthesis implementation in terms of throughput. In terms of energy consumption, the new methodology consumes 90% less. The chip area is 70% smaller than for the Parabolic Synthesis methodology. In summary, the new technology presented in this paper further increases the advantages of Parabolic Synthesis.

Keywords
Approximation, parabolic synthesis, unary functions, elementary functions, second-degree interpolation, arithmetic computation, look-up table, VLSI.

1. Introduction

Computation of elementary functions, such as trigonometric functions, logarithms and the square root, as well as binary functions, such as division, are numerous in a multitude of applications. A trend in wireless communication systems is handheld applications with very high data rates. With such applications follows also requirements on small chip area and low power consumption. Such an emerging application is wireless communication systems with multiple antennas on the transmitter and receiver, known as Multiple-Input Multiple-Output (MIMO). The great interest in MIMO falls back on its ability to cost-effectively improve transmission performance. To accomplish the high data rates these systems are performing an extensive amount of computation. An essential part of the computation is spent on matrix inversions, which are often executed as QR decompositions in which very high throughput is needed. An example of the required levels of performance for a QR decomposition is described in [1] for an Ordered Successive Interference Cancellation (OSIC) detector. The computation needed in the OSIC detector is 1.56 million inversions of complex-valued 4×4 channel matrices per second. Since each inversion uses 12 computations of trigonometric functions this implies that 12×1560000=18.72 million computations per second of the trigonometric functions sine and cosine have to be executed. For these numerically intensive real-time applications, software routines - although they capable of providing extremely accurate results - are too slow. Therefore, for the future of high-speed
wireless communication, there is a significant interest for new development regarding hardware implementations of function generators.

Hardware computation of elementary functions can be performed by employing many different algorithms [2,3], such as table-based methods, polynomial and rational approximations and functional iteration methods. Table-based methods remain manageable for low precision computations as long as the input operand is up to 12-16 bits, corresponding to table sizes of 4K-64K words. The size of the table grows exponentially with the word length and becomes unacceptably large when operating with higher precision. An alternative way of making approximations is based on polynomials. Since polynomials only involve additions, subtractions and multiplications, using them is a natural way to approximate elementary functions. There are a number of polynomial schemes available for polynomial approximations, such as Taylor, Maclaurin, Legendre, Chebyshev, Jacobi, and Laguerre. For a given precision, the chosen polynomial scheme affects the number of terms included and thus the computational complexity. Two development strategies are available in developing an approximation, one to minimize the average error, called least squares approximation, and one to minimize the worst case error, called least maximum approximation [2]. An example of when least squares approximation is favorable is when the approximation is used in a series of computations. On the other hand, least maximum approximation is favorable when it is important that the maximum error to the function to be approximated is kept small. An example of when least maximum approximation is favorable is when the error from the approximation has to be within a limit from the true function value. An advantage of polynomials is that they are table-less, but their drawback is that they impose large computational complexities and delays [2]. A reduction in computational complexity can be accomplished by combining table-based methods with polynomial based methods and the delays can also, to some extent, be decreased [2].

For implementation of elementary functions in hardware, the approximation methodology of sum of bit-products [4] can be beneficial since it can give an area efficient implementation with a high throughput at a reasonable accuracy.

The commonly used COordinate Rotation DIgital Computer (CORDIC) algorithm [5,6] is an iterative algorithm. The benefit of the algorithm is that when implementing approximations of basic elementary functions, these only require a small look-up table, simple shifts and addition operations. However, since it is an iterative method, it is inherently slow and therefore insufficient for very high performance applications.

The Parabolic Synthesis methodology [7-10] is founded on a multiplicative synthesis of factors of second-order functions. In fact, the most fundamental difference in the Parabolic Synthesis methodology compared to many other approximation methodologies, like polynomial and CORDIC, is that it is multiplicative and not an additive methodology. With the introduction of the Parabolic Synthesis methodology, the following improvements were accomplished compared to CORDIC. First, due to a highly parallel architecture, a significant reduction of the propagation delay was achieved, which also led to a significant reduction of the power consumption. Second, a further improvement of the Parabolic Synthesis methodology was developed by combining it with second-degree interpolation [3,11,12] specifically shown in [13].
When developing a multiplicative approximation based on such a combination of the methodologies the first factor is computed using the Parabolic Synthesis methodology. It is a rough approximation of the target function. The second factor is computed using a Second-Degree Interpolation methodology where the number of intervals in the interpolation decides the accuracy of the approximation. An attractive feature with the Parabolic Synthesis methodology combined with Second-Degree Interpolation [13] is that it enables a rough internal error compensation as part of the approximation. The error compensation can be performed in order to improve the distribution of the error to suit the properties required of the approximation.

Compared to the Parabolic Synthesis methodology the Parabolic Synthesis methodology Combined with Second-Degree Interpolation accomplishes the following [13]: A reduction in chip area because the number of second-order functions is reduced to two. A reduction of the critical path delay because the number of second-order functions is reduced to two. A reduction in power consumption because of reduced chip area and critical path delay. And, last, improved possibilities to tailor the distribution of the error.

This paper proposes an extension of the Parabolic Synthesis methodology Combined with Second-Degree Interpolation. This extension is named the Harmonized Parabolic Synthesis methodology because the two parts in this new methodology are developed as a unity in order to improve and optimize performance such as chip area, critical path delay and power consumption. The feasibility of Parabolic Synthesis has been verified by implementing a vast range of unary functions, as shown in [9]. With the Harmonized Parabolic Synthesis the range of unary functions is extended even further since the boundary conditions for the functions to be approximated are relaxed.

The remaining part of this paper is organized as follows: Section 2 describes the Harmonized Parabolic Synthesis methodology; Section 3 describes the general structure of the hardware architecture resulting from the methodology; Section 4 proposes a general strategy for truncation as well as optimization, which, if combined, can have a positive impact on the characteristic of the error; Section 5 presents the implementation of the logarithm in Harmonized Parabolic Synthesis and the distribution of its error; Section 6 gives a comparison of implementations performed in the methodologies Parabolic Synthesis and Harmonized Parabolic Synthesis. The comparison is made with respect to chip area, critical path delay and power consumption; and Section 7 closes the paper with conclusions.

2. Harmonized parabolic synthesis

The Harmonized Parabolic Synthesis methodology is founded on two factors, both in the form of second-order parabolic functions, called the first sub-function, \( s_1(x) \), and the second sub-function, \( s_2(x) \). When recombined, as shown in (1), they equal the original function \( f_{org}(x) \).

\[
f_{org}(x) = s_1(x) \cdot s_2(x)
\]  

In (1) the first sub-function, \( s_1(x) \), is a second-order parabolic function, which in conjunction with the second sub-function, \( s_2(x) \), develops an approximation of the original function, \( f_{org}(x) \). The second sub-function is a second-degree interpolation [3,11,12] specifically shown in [13], where
the number of intervals in the interpolation decides the final accuracy of the approximation and allows the distribution of the error to be tailored. When developing an approximation with the proposed methodology, an empirical design methodology is the only feasible approach since the sub-functions are designed as a complete unity to fulfill the design criteria.

2.1. Requirements on the original function

To facilitate the development it is required that the function to be approximated is being normalized. When normalized the function has to satisfy the requirement that the values are in the interval $0 \leq x < 1.0$ on the $x$-axis and $0 \leq y < 1.0$ on the $y$-axis as well as have the starting point at $(0,0)$, as illustrated in Fig. 1. The normalization of the function to be approximated creates the original function, $f_{org}(x)$.

In addition, since the approximation used through the interval is a second-order parabolic function, it is advantageous for an efficient implementation if the original function is strictly concave or convex through the interval in which it is approximated. If the original function is not strictly concave or convex through the interval it is still possible to perform an implementation, albeit with a limited efficiency. The limited efficiency comes from that the complexity of the implementation will increase.

Finally, the original function, $f_{org}(x)$, divided by the first sub-function, $s_1(x)$, must have a limit value when $x$ goes towards 0. If the function has no limit value it implies that the help function, $f_1(x)$, to be, described in Section 2.3, is not defined when $x = 0$.

Compared to Parabolic Synthesis, the number of criteria that a function to perform an approximation on has decreased from three to one and where two of the criteria are transformed into recommendations. Thereby, the number of functions that can be approximated with the methodology is increased. The third criterion which limits the absolute gradient of $f_{org}(x)$, divided by $x$ when $x$ goes towards 0 to 1 is eliminated because the Harmonized Parabolic Synthesis methodology can handle cases where the absolute gradient is higher.
2.2. The first sub-function

The first sub-function, \( s_1(x) \), is a second-order parabolic function as shown in (2).

\[
s_1(x) = l_1 + k_1 \cdot x + c_1 \cdot (x-x_0^2)
\]  

(2)

It can be seen that the two terms on the left form a linear function with a constant \( l_1 \) and a gradient \( k_1 \), while the rightmost term is the nonlinear part. As described above, the starting point for the first sub-function, \( s_1(x) \), is in \((0,0)\). This gives the start value, \( l_1 \), to be 0. Furthermore, the gradient, \( k_1 \), is 1 since the function starts in \((0,0)\) and ends in \((1,1)\). The first sub-function, \( s_1(x) \), can therefore be rewritten according to (3).

\[
s_1(x) = x + c_1 \cdot (x-x_0^2)
\]  

(3)

2.3. The second sub-function

The second sub-function, \( s_2(x) \), is based on splitting the function in intervals to perform an interpolation in each of them. The procedure when developing the second sub-function is to perform a division of the original function, \( f_{\text{org}}(x) \), with the first sub-function, \( s_1(x) \). This division generates the help function, \( f_{\text{help}}(x) \), as shown in (4).

\[
f_{\text{help}}(x) = \frac{f_{\text{org}}(x)}{s_1(x)}
\]  

(4)

From the help function, \( f_{\text{help}}(x) \), the second sub-function, \( s_2(x) \), is developed as a second-degree interpolation, as shown in (5) and Fig. 2, where the number of intervals in the interpolation decides the order of the accuracy of the approximation.

\[
s_2(x) = l_2,i + k_2,i \cdot x + c_2,i \cdot (x-x_0^2)
\]  

(5)

To simplify the determination of the interval, \( i \), in hardware, the number of equal-range intervals in the second sub-function, \( I \), is chosen as 2 to the power of \( w \), where \( w \) is a natural number, as shown in (6).

\[
I = 2^w
\]  

(6)

To simplify the normalization of the interval of \( x_w \), it is selected as an exponentiation by 2 of \( x \) where the integer part is removed. The normalization of \( x \) is therefore made by multiplying \( x \) with \( 2^w \), which in hardware is implemented as \( w \) left shifts. Furthermore, the integer part is dropped, which gives \( x_w \) as a fractional part of \( x \), as shown in (7).

\[
x_w = \text{frac}(2^w \cdot x)
\]  

(7)
This truncation performs normalization to the interval, as shown in Fig. 2. The dropped integer part from the normalization is used to decode the interval in which the second sub-function is performed and is therefore synonymous with the index $i$ in the sub-function, as shown in Fig. 2.

Fig. 2. Description of the development of the second-degree interpolation

The index $i$ is defined as the number of the interval, starting with 0 and ending with $I-1$. In (5), $l_{2,i}$ is the starting point of an interval of the interpolation. This is computed by inserting the value of $x$ for the starting point of the interval, $x_{\text{start},i}$, of the help function, $f_{\text{help}}(x)$, (4) as shown in (8) and Fig. 2.

$$l_{2,i} = f_{\text{help}}(x_{\text{start},i}) \quad (8)$$

Eq. (8) does not apply on the start value of the first interval, which has to be calculated as the limit according to (9). Since the $x^2$ term in (9) goes faster towards 0 than the $x$ term, it can be excluded when calculating the limit, as shown in (9).

$$l_{2,0} = \lim_{x \to 0} \frac{f_{\text{org}}(x)}{x + c_1 \cdot (x - x^2)} = \lim_{x \to 0} \frac{f_{\text{org}}(x)}{(1 + c_1) \cdot x} \quad (9)$$

In (5), $k_{2,i}$ is the gradient for an interpolation interval $i$. The gradient $k_{2,i}$ for an interval is computed as the end point value of the help function, $f_{\text{help}}(x_{\text{end},i})$, subtracted with the start point value of the help function, $f_{\text{help}}(x_{\text{start},i})$. Since the interval is normalized to 1 the denominator when computing the gradient, $k_{2,i}$, is set to 1, and therefore no division is needed, as shown in (10) and Fig. 2.

$$k_{2,i} = f_{\text{help}}(x_{\text{end},i}) - f_{\text{help}}(x_{\text{start},i}) \quad (10)$$

In (5) the coefficient, $c_{2,i}$, in an interval, $i$, of the second sub-function, $s_2(x)$, is pre-computed so that the second sub-function in an interval, $s_{2,i}(x_w)$, cuts the help function, $f_{\text{help}}(x)$, in the middle.
of the interval when \( x_w = 0.5 \). This satisfies the point \( x_{\text{middle},i} \) for the help function, \( f_{\text{help}}(x) \), as shown in (11) and Fig. 2.

\[
e_{2,i} = 4 \cdot (f_{\text{help}}(x_{\text{middle},i}) - l_{2,i} - c_{2,i} \cdot 0.5)
\]  

(11)

The sub-function in (5) can be simplified according to (12).

\[
s_{2,i}(x) = l_{2,i} + j_{2,i} \cdot x_w - c_{2,i} \cdot x_w^2
\]

(12)

In (12), \( j_{2,i} \) is pre-determined according to (13).

\[
j_{2,i} = k_{2,i} + c_{2,i}
\]

(13)

In Eq. (14), it is shown how the sub-function, \( s_{2,i}(x) \), is divided into partial functions.

\[
s_{2,i}(x) = \begin{cases} 
  s_{2,0}(x_w), & 0 \leq x < \frac{1}{2^w} \\
  s_{2,1}(x_w), & \frac{1}{2^w} \leq x < \frac{2}{2^w} \\
  \vdots \\
  s_{2,l-1}(x_w), & \frac{l-1}{2^w} \leq x < 1 
\end{cases}
\]

(14)

Note that, in (14), \( x \) has been changed to \( x_w \). The change is because the intervals for the partial sub-functions, \( s_{2,i}(x) \), in (14) have equal ranges.

2.4. Simultaneous development of the two sub-functions

The foundation of the harmonized parabolic synthesis methodology is to approach the development with a more holistic view. This is expressed in that the development of the two sub-functions is made simultaneously. When developing an approximation of an original function, \( f_{\text{org}}(x) \), the first and second sub-function are looked upon as one device. While, in the parabolic synthesis methodology, the first sub-function, \( s_1(x) \), was developed to have as good conformity as possible with \( f_{\text{org}}(x) \), the objective of the harmonized parabolic synthesis methodology is rather to develop the first sub-function in such a way that the product of the two sub-functions gives a good conformity to the original function. This includes that the distribution of the error is to be favorable and the hardware implementation as simple as possible. The approach when developing the first sub-function, \( s_1(x) \), can, in contrast to the parabolic synthesis methodology, not be based on independent analytical calculations since it is dependent on the performance of the second sub-function, \( s_2(x) \). Therefore, the coefficient \( c_1 \) in the first sub-function, \( s_1(x) \), has to be determined by, for different values of the coefficients, calculating the maximum absolute error, \( f_{\text{error},i}(x) \), between the approximation and the original function according to (15).
To perform the calculation of the absolute error, $f_{\text{error}}(x)$, the second sub-function, $s_2(x)$, has to be made dependent on the coefficient $c_1$ in the first sub-function, $s_1(x)$, as shown in (3) to (5) and (8) to (11). The calculation is interesting only as an indication of how the absolute error, $f_{\text{error}}(x)$, depends on the coefficient $c_1$. When choosing the coefficient $c_1$ it has to be made with regard to both the behavior of the error of the approximation and the efficiency of the hardware implementation. The number of intervals in the second sub-function, $s_2(x)$, needs to be increased to achieve the intended accuracy; this has also to be taken into account when performing the calculation of the error. As an example, Fig. 3 shows the bit accuracy for the sine function, shown when using 1, 2, 4 and 8 intervals in the second sub-function, $s_2(x)$ with different values of the coefficient, $c_1$. In our example implementation of the logarithm later in this paper, further details of how these calculations are done are given.

Based on Fig. 3, values of the coefficient $c_1$ are chosen to allow an efficient implementation of the hardware. As shown in (3), $c_1$ is feed into a multiplier why choosing a value that is a power of two is desirable. As an example shown in Fig. 3, the desired accuracy is 15 bit. To accomplish this, least four intervals are necessary. As shown in Fig. 3, only $c_1 = 1.0$ for four intervals is interesting since it is a power of two. If increasing the number of intervals to eight, the coefficient $c_1$ can be chosen to 0.0. This is interesting since this will exclude the multiplication in (3). The behavior of the bit accuracy in Fig. 3 results from the approximation using different values of the coefficient $c_1$. From this the number of intervals and the value on the coefficient $c_1$ used in the design can be selected.

3. Hardware architecture

The hardware architecture resulting from the methodology can be divided into three parts as shown in Fig. 4, following a principle described by Tang [14]. In the preprocessing part the incoming operand is transformed to fit the processing part, in which the approximation is performed. In the postprocessing part the result is transformed to the desired output format.

$$f_{\text{error}}(x) = |s_1(x) \cdot s_2(x) - f_{\text{arg}}(x)|$$
In most cases preprocessing of the operand means a normalization, but the transformation of the operand can also be more comprehensive such as converting a fixed-point number into a floating-point number. If the approximation is implemented as a block in a larger system, the preprocessing part can be integrated in the previous blocks, in which case the preprocessing part can be reduced or even be excluded. For the postprocessing, similar conditions apply.

In the processing part, the approximation of the original function, \( f_{org}(x) \), is directly computed in the way now described.

The architecture of Harmonized Parabolic Synthesis implements two sub-functions computed in parallel. The first sub-function is implemented as a second-order parabolic function and the second sub-function is implemented using the Second-Degree Interpolation methodology. Fig. 5 shows the architecture, where the two sub-functions are implemented by the upper and lower half, respectively, and then combined via a multiplication.

It is worth noting that this, in fact, is a generic architecture, which can be used for approximating several different functions. The set of parameters defines the function.
In the first sub-function, the result of the \((x-x^2)\) part is multiplied with \(c_1\) and then added to \(x\). In the proposed methodology, the coefficient \(c_1\) is chosen as described in conjunction with Fig. 3, to reduce the hardware consumption. This implies that the algorithm of the first sub-function, \(s_1(x)\), can be simplified, which also reduces the complexity of the implementation. The second sub-function is implemented as a second-degree interpolation, consisting of a look-up table containing, for each interval \(i\), the coefficients \(j_{2,i}\) from (13). The coefficient \(j_{2,i}\) is multiplied with \(x_{w,i}\), which is the remaining part of \(x\) when the index part \(i\) is removed. The value of \(x_{w}\) is also the normalized value of \(x\) for the interval. After the multiplication, the start value of the interval \(l_{2,i}\) is added. The third branch of the second sub-function consists of a look-up table containing the coefficients \(c_{2,i}\) from (11) for each interval \(i\). The coefficient \(c_{2,i}\) for interval \(i\) is multiplied with \(x_{w,2}\), which is the partial products of \(x^2\) for interval \(i\). To simplify this computation, a special squaring unit to compute \(x^2\) has been developed; this unit is described in [7,9]. The benefit of the squarer is that it computes all the partial products to \(x^2\) in the same hardware and that it halves the chip area and computation time compared to a multiplier. The result of the multiplication between \(c_{2,i}\) and \(x_{w,2}\) is subtracted from the result of the previous addition. After the values of the first and second sub-functions are computed, they are multiplied with each other.

The architecture is suitable for pipelining in order to increase the throughput. An example of where pipeline stages can be introduced is shown in Fig. 5. Introduction of pipeline stages is simple since it is easy to find natural cuts in the architecture. The introduction of data pipelining stages in the architectures will only have little effect on the size of the hardware, since the data paths are few. However, registers consume power, thus power consumption will increase.
4. Optimizing

Finding the optimal set of coefficients can be done as an iterative procedure starting from an initial set. After deciding the initial coefficients in the first and second sub-function the next step is to start the optimization of the architecture. The method for the optimization is to decide the word lengths used in the architecture and then optimize the coefficients. This optimization is performed in an iterative trial and error manner and the evaluation of different coefficient values should be performed in parallel with the evaluation of the word length, since the truncation error effects influence the performance of calculations in the design. The strategy is to adjust coefficients and word lengths in the design for best accuracy and distribution of the error. In the text below the optimization strategy will be illustrated using the sine function since this function is commonly used and has a simple implementation, thus making the steps of the optimization easy to follow. The target accuracy for the implementations is chosen to be around 15 bit.

In practice, the simulation of the approximation is performed with a bit-accurate C model and the performance of the approximation is analyzed in MatLab.

4.1. Truncation and optimization

Truncation always results in a negative offset of the error compared to the non-truncated value, as illustrated in Fig. 6. In the figure, the gray curve shows the error before the truncation and the black is the error after truncation.

Fig. 6. The error before and after truncation of the approximated value

Fig. 6 shows the errors in both curves winding through the eight intervals. The winding of the curves is caused by the rightmost term in (5), which is the nonlinear part. This term in each interval in the second sub-function, \( s_2(x) \), causes one winding of the curve. To counteract the negative offset of the error caused by truncation, the coefficients in the second sub-function, \( s_2(x) \), can be adjusted. A favorable way to do this is to seek a solution in which the error has a distribution which is asymptotically normal. An advantage of an asymptotically normal error distribution is that it is centered around zero and the major part of the errors also are around zero. This distribution implies that the number range is optimally utilized which has positive effects on the architecture, in reduced size of hardware and in the subsequent calculations to reduce their error. In Fig. 7 the gray curve shows the error before the truncation of the word lengths and
optimization of the coefficients, and the black is the error after truncation and optimization of the coefficients.

![Graph showing error before and after truncation and optimization](image)

**Fig. 7.** The error before and after truncation and optimization

When comparing the two figures it can be seen that the error after truncation and optimization, i.e., the one in Fig. 7, is more evenly distributed around zero than the one without optimization. This shows that it is mainly by adjusting the coefficients that the distribution of the error is determined.

4.2. The characteristic metrics

The error resulting from an approximation can be characterized in several ways [15], the most important being the following:
- The maximum absolute error denotes the largest error possible using the approximation.
- The mean error denotes the average error over the approximations sample spaces.
- The median error denotes the skewness in the error distribution, if there is any.
- The standard deviation is a measure of the variation from the mean error.
- The Root Mean Square (RMS) error is a measure of the magnitude of a continuously varying quantity of the error.

Furthermore, the evenness of the error distribution is determined by comparing the standard deviation with the RMS error. If the standard deviation and the RMS error are equal, it indicates that the error distribution is symmetric around zero. This is a preferred error distribution. An advantage of the methodology presented in this paper is that the possibility for obtaining a symmetrically distributed error around zero is very good. To illustrate the distribution of the error a histogram is used. Fig. 8 shows the histogram of the distribution of the error that was shown in Fig. 7. As shown, the error distribution is asymptotically normal and with a mean value near zero.
5. Implementation of the logarithm

As an illustration of the Harmonized Parabolic Synthesis methodology, an implementation is presented of an algorithm that performs an approximation of the logarithm function. It is performed on a floating-point number where the mantissa is in the range from 1.0 to 2.0. The approximation is only performed on the mantissa, since the exponential part of the input value is directly used as the integer part of the result and scaling the mantissa, [16,17].

The implementation is in hardware using a 14 bit input. The target accuracy is 15 bits and the target distribution of the error is the normal distribution.

The evaluation of the approximation is analyzed as described in Section 4.

5.1. Preprocessing

As described in Section 2.1, to facilitate the hardware implementation of the approximation, a normalization to satisfy that the values are in the interval 0 ≤ x < 1.0 on the x-axis and 0 ≤ y < 1.0 on the y-axis has to be performed. The result of the binary logarithm satisfies that the values are in the interval 0 ≤ y < 1.0. To satisfy that the incoming operand x is within the interval 0 ≤ x < 1, a 1 has to be added to the operand as shown in (16).

\[ v = 1 + x \]  

To normalize the \( f(v) = \log_2(v) \) function, \( v \) is substituted according to (17), which gives the original function, \( f_{org}(x) \), shown in (17).

\[ f_{org}(x) = \log_2(1 + x) \]  

Fig. 9 shows the function, \( f(v) \), together with the normalized function, \( f_{org}(x) \).
5.2. Processing

For the processing part, the sub-functions are developed according to the description in Section 2.4. Developing the coefficient $c_1$ is made with two aspects in mind: the error distribution of the approximation, and the simplicity of the hardware implementation. The number of intervals that the second sub-function, $s_2(x)$, needs to be divided into, to achieve the intended accuracy, has also to be taken into account when performing the calculation of the error. Fig. 10 shows the resulting minimum number of bits of accuracy for the logarithm function when using 1, 2, 4 and 8 intervals in the second sub-function, $s_2(x)$.

The needed accuracy for the implementation was set to 15 bits and the distribution of the error shall be similar to a normal distribution. Fig. 10 shows that using 4 or 8 intervals will fulfill the accuracy demand with some specific ranges of coefficient, $c_1$. The requirement to have a distribution of the error that is similar to the normal distribution implies that using 8 intervals would be advantageous since this will give greater margin when developing the second sub-
function, \( s_2(x) \). Fig. 10 shows that using 8 intervals in the second sub-function will allow the coefficient \( c_1 \) to be 0, which in turn will imply that the hardware in the first sub-function, \( s_1(x) \), is reduced, as shown in (18).

\[
s_1(x) = x + c_1 \cdot (x - x^2) = x + 0 \cdot (x - x^2) = x
\]  

(18)

The help function, \( f_{\text{help}}(x) \), is computed as shown in (19).

\[
f_{\text{help}}(x) = \frac{f_{\text{aux}}(x)}{s_1(x)} = \frac{\log_2(1 + x)}{x}
\]  

(19)

Fig. 11 shows the help function, \( f_{\text{help}}(x) \).

![Graph showing the help function, \( f_{\text{help}}(x) \), for the approximation of the logarithm.](image)

Fig. 11. The help function, \( f_{\text{help}}(x) \), for the approximation of the logarithm

From the help function, \( f_{\text{help}}(x) \), an initial second sub-function, \( s_2(x) \), is developed according to the description in Section 2.3 using 8 intervals.

When the initial second sub-function, \( s_2(x) \), is developed, the next task is to achieve a distribution of the error similar to the normal distribution, which is performed as described in Section 4.

Table 1 to Table 3 show the developed coefficients in (12) for 8 intervals.

Table 1 shows the coefficients for the initial value, \( l_{2,i} \), for each interval, \( i \), in (12).
Table 2 shows the coefficients $l_{2,i}$, for each interval, $i$, in (12).

<table>
<thead>
<tr>
<th>Coefficient</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_{2,0}$</td>
<td>$1.44268798828125000_{\text{dec}}$\newline$1.011100010101000_{\text{bin}}$</td>
</tr>
<tr>
<td>$l_{2,1}$</td>
<td>$1.35939788818359375_{\text{dec}}$\newline$1.01011100000000011_{\text{bin}}$</td>
</tr>
<tr>
<td>$l_{2,2}$</td>
<td>$1.28771209716796875_{\text{dec}}$\newline$1.0100100110100111_{\text{bin}}$</td>
</tr>
<tr>
<td>$l_{2,3}$</td>
<td>$1.22514343261718750_{\text{dec}}$\newline$1.00111001101000110_{\text{bin}}$</td>
</tr>
<tr>
<td>$l_{2,4}$</td>
<td>$1.16991424560546875_{\text{dec}}$\newline$1.001010111111111_{\text{bin}}$</td>
</tr>
<tr>
<td>$l_{2,5}$</td>
<td>$1.12069702148437500_{\text{dec}}$\newline$1.000111101100110_{\text{bin}}$</td>
</tr>
<tr>
<td>$l_{2,6}$</td>
<td>$1.07646942138671875_{\text{dec}}$\newline$1.000100111001100_{\text{bin}}$</td>
</tr>
<tr>
<td>$l_{2,7}$</td>
<td>$1.0364561767578125_{\text{dec}}$\newline$1.0000100101010001_{\text{bin}}$</td>
</tr>
</tbody>
</table>
It can be seen from Table 2 that the three most significant bits after the binary point in the binary representation of the coefficients are zeros. As a result of this, the word length of the coefficients can be reduced from 15 bits to 12 bits.

Table 3 shows the coefficients for the parabolic part, $c_{2,i}$, for each interval, $i$, in (12).

<table>
<thead>
<tr>
<th>Coefficients</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j_{2,0}$</td>
<td>-0.089294433593750&lt;sub&gt;dec&lt;/sub&gt; -0.000101101101110&lt;sub&gt;bin&lt;/sub&gt;</td>
</tr>
<tr>
<td>$j_{2,1}$</td>
<td>-0.076629638671875&lt;sub&gt;dec&lt;/sub&gt; -0.000100111001111&lt;sub&gt;bin&lt;/sub&gt;</td>
</tr>
<tr>
<td>$j_{2,2}$</td>
<td>-0.066589355468750&lt;sub&gt;dec&lt;/sub&gt; -0.000100010001110&lt;sub&gt;bin&lt;/sub&gt;</td>
</tr>
<tr>
<td>$j_{2,3}$</td>
<td>-0.058471679687500&lt;sub&gt;dec&lt;/sub&gt; -0.000011101111100&lt;sub&gt;bin&lt;/sub&gt;</td>
</tr>
<tr>
<td>$j_{2,4}$</td>
<td>-0.051849365234375&lt;sub&gt;dec&lt;/sub&gt; -0.000011010100011&lt;sub&gt;bin&lt;/sub&gt;</td>
</tr>
<tr>
<td>$j_{2,5}$</td>
<td>-0.046447759062500&lt;sub&gt;dec&lt;/sub&gt; -0.000010101000110&lt;sub&gt;bin&lt;/sub&gt;</td>
</tr>
<tr>
<td>$j_{2,6}$</td>
<td>-0.041900634765625&lt;sub&gt;dec&lt;/sub&gt; -0.000010101011101&lt;sub&gt;bin&lt;/sub&gt;</td>
</tr>
<tr>
<td>$j_{2,7}$</td>
<td>-0.038085937500000&lt;sub&gt;dec&lt;/sub&gt; -0.000010011110000&lt;sub&gt;bin&lt;/sub&gt;</td>
</tr>
</tbody>
</table>
It can be seen from Table 3 that the seven most significant bits after the binary point in the binary representation of the coefficients are zeros. As a result of this, the word length of the coefficients can be reduced from 16 bits to 9 bits.

When developing the algorithm for the approximation, it is an interaction between the word lengths and the values of the coefficients in the architecture. The word lengths of the coefficients and the data paths in the design are shown in Fig. 12.
In the process of truncating words in the design and the optimizing coefficients, the size and distribution of the error is analyzed and adjusted. Fig. 13 shows the error of the final implementation compared to the error before the truncation of the word lengths in the design and optimization of the coefficients.

It can be seen that the error is fairly equal over the interval.

Fig. 14 shows the absolute error of the implementation. It shows that the implemented approximation well meets the requirement of an accuracy of about 15 bits.
Fig. 14. The absolute error in bits of the approximation of the logarithm

When optimizing the coefficients, the intention has been to achieve a normal distribution of the error. Such a distribution is beneficial in the calculations in most algorithms. Fig. 15 shows the distribution of the error in the final, optimized implementation.

Fig. 15 shows a very high resemblance with the normal distribution.

Table 4 shows the error statistics of the implementation.
The conclusion from Table 4 is that the maximum absolute error corresponds to an accuracy of more than 15 bits. The mean error is very small and the median confirms that the error distribution is not skewed. When comparing the standard deviation value with the root mean square value, it can be seen that the values are nearly identical. The resemblance confirms that the error of the approximation is symmetrically distributed.

5.3. Postprocessing

No postprocessing is needed since the result from the processing part is in the right format.

6. Comparing methodologies

This section compares implementations of the logarithm using two different methodologies; on the one hand the Parabolic Synthesis methodology, where the accuracy is decided with the number of sub-functions, and on the other hand, the Harmonized Parabolic Synthesis methodology, where the accuracy is decided with the number of intervals. The comparison will be performed in terms of chip area, critical path delay and power consumption. Both implementations are performed with the same bit accuracy and so that the error is asymptotically normally distributed.

The methodologies are implemented in their basic versions, without features like pipelining. The implementations are realized as an ASIC with a 65nm Standard-$V_T$ (1.2V) technology. Synopsys Design Compiler [18], Synopsys Primetime [19] and Mentor Graphics Modelsim [20] are used for all implementations.

The comparison is performed for an approximation of the $\log_2$ of a mantissa in the range from 1 through 2. The required accuracy for the implementation is set to 15 bits.

The results of the implementation carried out with the Parabolic Synthesis are based on [10] after a redesign without bonding pads. The results for chip area, path delay, and energy consumption per sample are shown in the upper entry of Table 5. In [10] four sub-functions were used in the architecture. Unfortunately, in the implementation performed with Parabolic Synthesis in [10], no optimization of the word lengths was performed; therefore longer word lengths than needed were used in the design (in some parts of the design the word lengths could have been nearly halved).

<table>
<thead>
<tr>
<th>Value</th>
<th>Expressed in bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max absolute error</td>
<td>&gt;15</td>
</tr>
<tr>
<td>Mean error</td>
<td>&gt;25</td>
</tr>
<tr>
<td>Median error</td>
<td></td>
</tr>
<tr>
<td>Standard deviation of error</td>
<td></td>
</tr>
<tr>
<td>Root Mean Square error</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 4: Error statistics
This yields an impact on the chip area and, to some extent, on the critical path delay of the design. Hence the parameters such as chip area, critical path delay and power consumption are somewhat larger than they need to be. For the implementation using Harmonized Parabolic Synthesis [21] eight intervals are used in the second sub-function. In this case, optimization of the word lengths in the design was fully performed. The results of the implementation are shown in the lower entry of Table 5.

**TABLE 5: Comparison of chip area, critical path delay and energy consumption per sample.**

<table>
<thead>
<tr>
<th>Methodology</th>
<th>Implemented Function</th>
<th>Chip Area</th>
<th>Critical Path Delay</th>
<th>Energy Consumption per Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parabolic Synthesis</td>
<td>log(x)</td>
<td>100% 16258 μm²</td>
<td>100% 21 ns</td>
<td>100% 0.065 nW</td>
</tr>
<tr>
<td>Harmonized Parabolic Synthesis</td>
<td>log(x)</td>
<td>30% 4865 μm²</td>
<td>33% 7 ns</td>
<td>9% 0.0061 nW</td>
</tr>
</tbody>
</table>

The conclusion from Table 5 is that the Harmonized Parabolic Synthesis is clearly superior to Parabolic Synthesis. The chip area is reduced with 70%, mainly because Parabolic Synthesis requires four sub-functions whereas Harmonized Parabolic Synthesis only needs two. To some extent also the not fully performed optimization of the Parabolic Synthesis implementation affects its performance. With full optimization the chip area for the Parabolic Synthesis is estimated to be reduced with roughly 10%. In computation speed the Harmonized Parabolic Synthesis is 3 times faster. This is primarily due to that the implementation performed with Parabolic Synthesis in the critical path has more and larger multipliers. The smaller chip area and the shorter critical path of the Harmonized Parabolic Synthesis methodology results in an 11 times lower energy consumption for the Harmonized Parabolic Synthesis.

**7. Conclusion**

This paper has described the Harmonized Parabolic Synthesis methodology for developing approximations in hardware of unary functions, such as trigonometric functions, logarithm functions, exponential functions and square root. The methodology is mainly intended for computation intensive applications in, e.g., computer graphics, digital signal processing, communication systems, robotics, astrophysics and fluid physics, as well as in many other application areas. An emerging computation intensive application that serves as a strong motivating example is wireless communications systems with multiple antennas on the transmitter and receiver, known as Multiple-Input Multiple-Output (MIMO). An essential part of the computation in these systems is performed when computing matrix inversions, which are often executed as QR decompositions in which very high throughput is needed. The QR decomposition algorithm mainly requires approximations of trigonometric functions, roots and inverses. For these computation demands the Harmonized Parabolic Synthesis with its benefits in terms of small chip area, short critical path and low energy consumption is most favorable.
When going from Parabolic Synthesis to Harmonized Parabolic Synthesis, the major attractiveness lies in how increasing the accuracy is done. In Parabolic Synthesis the way to increase accuracy of the approximation is to introduce more sub-functions. In Harmonized Parabolic Synthesis, which only uses two sub-functions, it is instead done by increasing the number of intervals in the second sub-function. The effect of this is that, with increasing accuracy, the resulting chip area, critical path delay, and energy consumption will show a much slower increase than for Parabolic Synthesis. The Harmonized Parabolic Synthesis also has more extensive opportunities to handle truncation effects and to achieve a desired distribution of the error. The use of second-degree interpolation in the second sub-function also has the positive effect that the range of functions that can be approximated is significantly expanded compared to when only using Parabolic Synthesis.

The architecture based on Harmonized Parabolic Synthesis is, like the one based on Parabolic Synthesis, very suitable for pipelining. Further, both architectures can easily be reused for approximations of other unary functions by simply changing the set of coefficients. When analyzing the hardware architecture for implementation of the logarithm, it was found that Harmonized Parabolic Synthesis reduced the complexity of the architecture significantly, over Parabolic Synthesis.

When analyzing the characteristics of the error for the implementation of the logarithm, it was found that the Harmonized Parabolic Synthesis has a mean error that is near zero and a median that is zero or near zero. This implies that only a minimum of skewness is present in the error distribution and that the difference between the standard deviation and the root mean square value is very small, which in turn guarantees that the error is evenly distributed.

To get an indication of the implementation performance of the Harmonized Parabolic Synthesis, it has been compared to an implementation carried out with the Parabolic Synthesis. The comparison shows that approximations with the Harmonized Parabolic Synthesis methodology results in a chip area that is smaller than with Parabolic Synthesis. The throughput is also higher, and the energy consumption per sample is almost ten times lower.

References


Erik Hertz received his MS degree in electrical engineering in 1983 and his Licentiate degree in 2011, both at the Faculty of Engineering, Lund University, Sweden. Since 1989 he has done research at the Swedish Defense Research Agency, Linköping, Sweden, Ericsson Mobile Platforms AB, Lund, Sweden and Cargine AB, Ängelholm, Sweden, connected to Koenigsegg Automotive AB. His research has been in a wide range of areas such as radar, radio, control engineering, sensors, microwave, ASIC and solid state physics. His main interest is in algorithm development for hardware implementation. In 2012 he joined the Centre for Research on Embedded Systems (CERES), Halmstad University, Halmstad, Sweden where he is currently pursuing his doctoral degree.
Jingou Lai received his MSc degree on System on Chip Design at Lund University, Sweden in 2013. His master thesis was on the Implementation of Logarithm using Harmonized Parabolic Synthesis. He received the BSc degree in Electrical Engineering from University of Gävle, Sweden, in 2011. His bachelor thesis was on Wireless Communication and Digital Signal Processing. In summer 2012, he worked as an intern on MCU Logic Design & Verification in Appotech, China. In 2013, he worked as a Product Validation Engineer on imagine sensor in Galaxy core, China. Since April 2014, he has been working as an R&D in Cadence Design System, China.

Bertil Svensson received his MS degree in electrical engineering in 1970 and PhD degree in computer engineering in 1983, both from Lund University, Sweden. He was a member of the faculty of Halmstad University, Sweden, from 1983, served as acting professor at Luleå University of Technology, Sweden, from 1989 and was professor of computer systems engineering at Chalmers University of Technology, Sweden, from 1991. Since 1998, he has been professor of computer systems engineering at Halmstad University, where he has been the dean of the school of information science, computer and electrical engineering and the initiator and director of several research centres. Until recently he was the director of CERES, the centre for research on embedded systems and of Research for Innovation, a strategic, university over-arching research program. His research interests are parallel computer architecture, embedded parallel computing and cooperating embedded systems, including wireless and intelligent systems. He is a member of the IEEE and the ACM.

Peter Nilsson passed away in February 2016. Peter was since July 2008, a full professor in electronic design. In February 1988, Peter reached his Master of Science degree in Electrical Engineering and in May 1996, his degree Doctor of Philosophy in Engineering was awarded, both at Faculty of Engineering, Lund University. Since 1996 Peter has supervised seven PhD students to the completion of their doctoral degree, and had two on the way. He has supervised and/or examined 156 students at the Master’s thesis level, both nationally and internationally. Peter authored or co-authored 126 peer-reviewed journal and conference papers. He was the program manager as well as the person authorized to sign for the “Socware Research & Education” program, a five-year national 15 million USD
program. He was an Associate Editor for IEEE Transactions on Circuits and Systems I, 2004-05, and was a member of the VLSI Systems and Applications Technical Committee in IEEE Circuits and Systems Society. He was awarded the IEEE Senior Member grade. He was also a Technical Program Committee (TPC) member of the IEEE International Solid State Circuits Conference (ISSCC). Peter’s main interest was implementation of digital ASICs, and then often with a focus on wireless communication.
Paper I
Algorithms for implementing roots, inverse and inverse roots in hardware

Erik Hertz, Niclas Thuning, Leo Bärring, Bertil Svensson, and Peter Nilsson

Erik Hertz
Centre for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
Phone: +46 702890215
FAX: +46 35120348
E-mail: erik.hertz@hh.se

Niclas Thuning
Department of Electrical and Information Technology, Lund University, Lund, Sweden
Phone: +46 767709145
FAX: +46 35120348
E-mail: niclas.thuning@gmail.com

Leo Bärring
Department of Electrical and Information Technology, Lund University, Lund, Sweden
Phone: +46 0705635685
FAX: +46 35120348
E-mail: leo.barring@gmail.com

Bertil Svensson
Centre for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
Phone: +46 706480843
FAX: +46 35120348
E-mail: bertil.svensson@hh.se

Peter Nilsson
Department of Electrical and Information Technology, Lund University, Lund, Sweden
Phone: +46 702890215
FAX: +46 35120348
E-mail: erik.hertz@hh.se

Abstract
In applications as in future MIMO communication systems a massive computation of complex matrix operations, such as QR decomposition, is performed. In these matrix operations, the functions roots, inverse and inverse roots are computed in large quantities. Therefore, to obtain high enough performance in such applications, efficient algorithms are highly important. Since these algorithms need to be realized in hardware it must also be ensured that they meet high requirements in terms of small chip area, low computation time and low power consumption. Power consumption is particularly important since many applications are battery powered.
For most unary functions, directly applying an approximation methodology in a straightforward way will not lead to an efficient implementation. Instead, a dedicated algorithm often has to be developed. The functions roots, inverse and inverse roots are in this category. The developed approaches are founded on working in a floating-point format. For the roots functions also a change of number base is used. These procedures not only enable simpler solutions but also increased accuracy, since the approximation algorithm is performed on a mantissa of limited range.

As a summarizing example the inverse square root is chosen. For comparison, the inverse square root is implemented using two methodologies: Harmonized Parabolic Synthesis and Newton-Raphson method. The novel methodology, Harmonized Parabolic Synthesis (HPS), is chosen since it has been demonstrated to provide very efficient approximations. The Newton-Raphson (NR) method is chosen since it is known for providing a very efficient implementation of the inverse square root. It is also commonly used in signal processing applications for computing approximations on fixed-point numbers of a limited range. Four implementations are made; HPS with 32 and 512 interpolation intervals and NR with 1 and 2 iterations. Summarizing the comparisons of the hardware performance, the implementations HPS 32, HPS 512 and NR 1 are comparable when it comes to hardware performance, while NR 2 is much worse. However, HPS 32 stands out in terms of better performance when it comes to the distribution of the error.

**Index Terms**

Approximation, unary functions, elementary functions, arithmetic computation, root, inverse, inverse roots, harmonized parabolic synthesis, Newton-Raphson method.

**1. Introduction**

For most unary functions, directly applying an approximation methodology in a straightforward way will not lead to an efficient implementation. Instead, a dedicated algorithm often has to be developed.

The functions roots, inverse and inverse roots are computed in large quantity in many complex matrix operations, such as QR decomposition, why these functions are particularly interesting. In applications such as future MIMO communication systems [1] a massive computation of these functions are needed. Therefore, to obtain high enough performance in such applications, efficient algorithms are highly important. Since these algorithms need to be implemented in hardware it must also be ensured that they meet high requirements in terms of small chip area, low computation time and low power consumption. Among these, low power consumption is particularly important, since many applications are battery powered.

The remaining part of this paper is organized as follows: Section 2 describes the algorithms for the functions roots, inverse, and inverse roots; Section 3 presents the implementation of the inverse square root function using two approximation methods, the Harmonized Parabolic Synthesis method and the Newton-Raphson method; Section 4 presents the results of the implementations in terms of both physical characteristics and error characteristics, and Section 5 contains a comparative discussion of the implementations. The comparison is made with respect.
to chip area, critical path delay, power consumption and error distribution; Section 6 closes the paper with conclusions.

2. Algorithms for roots, inverse, and inverse roots

For the functions roots, inverse and inverse roots, algorithms have been developed founded on using floating-point numbers. Commonly, algorithms for these functions are developed founded on using fixed-point numbers. Note that the algorithms are developed without regard to a specific approximation methodology. The format of the floating-point numbers used in the algorithms is simplified compared to the IEEE standard for floating-point arithmetic (IEEE 754). The floating-point number format is customized for hardware implementation. The floating-point number consists of a mantissa with the range from 1 to < 2 and an exponent where the exponent is a scaling of the mantissa. By using floating-point numbers as an internal representation the computation can be divided into separate parts for the mantissa and the exponent. This will reduce the complexity since the approximation is performed on a mantissa of limited range, and the computation of the exponent is a very simple operation. In Table 1 a conversion of a fixed-point number into a floating-point number in base 2 is shown.

<table>
<thead>
<tr>
<th>Base 10</th>
<th>387</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fixed-point Base 2</td>
<td>0000000110000011</td>
</tr>
<tr>
<td>Exponent Index</td>
<td>0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1</td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>Floating-point Base 2</td>
<td>1.100000110000000·2^8</td>
</tr>
</tbody>
</table>

As shown in Table 1, the index of the exponent is given by the most significant 1 in the fixed-point number, which is 8 in this case. In the last line of Table 1 the floating-point number is shown, in which the mantissa is scaled by the exponent.

When computing the function on the mantissa this is performed as an approximation. As shown in Fig. 1 the mantissa of the floating-point number is the input $v$ to the approximation and $z$ is the output from the approximation.

![Fig. 1. Block diagram of input and output of the approximation.](image-url)
2.1. Algorithms for computing roots

Besides that the algorithms for computing roots is founded on floating-point numbers it is also based on changing the number base of the exponent. The number base used when performing an approximation of a root depends on the order, \( d \), of the root. The base used in the approximation is \( 2^d \). Although the number base is changed, the binary number base is retained in the representation of the mantissa. The algorithms computing roots can therefore only compute roots of order \( d \), where \( d \) is a natural number. The purpose of changing number base is that after computing the root the base of the exponent always will be 2, as shown in (1).

\[
\frac{\sqrt[2]{x^d}}{2} = 2
\]

(1)

As shown in (2) the mantissa consists of \( d \) integer bits \( M \) and \( h \) fractional bits \( m \).

\[
M_{d-1} \cdots M_0, m_{-1} m_{-2} \cdots m_{-h}
\]

(2)

The range of the mantissa will be \( 1 \leq \text{mantissa} < 2^d \).

2.1.1. Algorithm for computing the square root

When computing the square root \( d = 2 \), which gives that the floating-point uses an exponent with base 4. Table 2 shows a conversion from a fixed-point number in number base 2 into a floating-point number with exponent base 4 and binary number representation of the mantissa.

<table>
<thead>
<tr>
<th>Base 10</th>
<th>Fixed-point Base 2</th>
<th>Exponent Index</th>
<th>Floating-point Base 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>387</td>
<td>0000000110000011</td>
<td>00 00 00 01 10 00 00 11 7 6 5 4 3 2 1 0</td>
<td>01.10000011000000 4^4</td>
</tr>
</tbody>
</table>

As shown in Table 2 when deciding the index of the exponent the fixed-point number in number base 2 is transformed into sequential pairings of number base 2 digits. In Table 2 the most significant pair of digits is found for index 4. The pair of digits for index 4 is the integer bits in the mantissa and the remaining pairs of digits are the fractional bits of the mantissa. In the last line of Table 2 the floating-point number in number base 4 is shown.

The computation to perform when computing the square root, \( z \), is shown in (3).

\[
z = \sqrt[M_{d-1} \cdots M_0, m_{-1} m_{-2} \cdots m_{-h}]{4^{\text{index}}}
\]

(3)

In (4), (3) is simplified to get the exponent in number base 2.
As shown in (4), computing the exponent is made through a simple change of number base.

When computing the approximation of the square root, this is performed only on the range $1 \leq \nu < 4$, as shown in Fig. 2.

![Fig. 2. The square root function.](image)

A summary of the algorithm for computing the square root function is shown in the block diagram in Fig. 3.

![Fig. 3. Block diagram of the square root algorithm.](image)

As shown in Fig. 3 the starting point is a floating-point number in number base 4. The exponent and mantissa are computed separately. When computing the exponent the number base is changed from 4 to 2 without changing the index. When computing the mantissa it is assumed that the incoming mantissa is in the number base 4. On the incoming mantissa an approximation of the square root function is performed. After the approximation the result $z$ is in the range $1 \leq z < 2$ which is the desired range of the mantissa with the number base 2.
2.2. Algorithms for computing the inverse

The algorithms for computing the inverse is also founded on using floating-point number.

The computation to perform to compute the inverse, \( z \), is shown in (5).

\[
z = \frac{1}{M_{0, m_1 m_2 \cdots m_h} \cdot 2^{\text{index}}}
\]  

(5)

In (6), (5) is simplified to get the exponent in the numerator, thus with negative index.

\[
z = \frac{1}{M_{0, m_1 m_2 \cdots m_h}} \cdot 2^{-\text{index}}
\]  

(6)

As shown in (6), when computing the exponent only the sign of index is changed.

The approximation of the inverse is performed in the range \( 1 \leq \nu < 2 \), as shown in Fig. 4.

![Fig. 4. The inverse function.](image)

A summary of the algorithm for computing the inverse function is shown in the block diagram of Fig. 5.
As shown in Fig. 5 the starting point is a floating-point number. The exponent and mantissa are computed separately. When computing the exponent the sign of the index is changed and subtracted with 1 depending on the result of the approximation. When computing the mantissa the range after approximation is $0.5 < z \leq 1$. This implies that when $z < 1$ a multiplication with 2 has to be performed on the mantissa.

2.3. Algorithms for computing inverse roots

The algorithm for computing inverse roots is combining the algorithms for roots and inverse described in Section 2.1 and 2.2.

The computation to perform to compute the inverse, $z$, is shown in (7).

$$z = \frac{1}{d|M_{d-1}...M_0| m_{-1} m_{-2} ... m_{-h} \cdot 2^{d \cdot \text{index}}}$$  \hspace{1cm} (7)

As shown in (7) the inverse root $d$ is performed on a floating-point number with the number base $2^d$ and in binary number representation. As shown in (7) the mantissa consist of $d$ integer bits $M$ and fractional bits $m$ where $h$ is the number of bits used in the fractional part.

In (8), (7) is simplified to get the exponent in the numerator, thus in base 2 with negative index.

$$z = \frac{1}{d|M_{d-1}...M_0| m_{-1} m_{-2} ... m_{-h} \cdot 2^{d \cdot \text{index}}} \cdot 2^{-d \cdot \text{index}}$$ \hspace{1cm} (8)

As shown in (8), when computing the exponent only the sign of index is changed. Since the integer part of the mantissa always has to be 1 or larger the range of the mantissa will be $1 \leq \text{mantissa} < 2^d$. 

---

As shown in the block diagram, the starting point is a floating-point number. The exponent and mantissa are computed separately. When computing the exponent, the sign of the index is changed and subtracted with 1 depending on the result of the approximation. When computing the mantissa, the range after approximation is $0.5 < z \leq 1$. This implies that when $z < 1$, a multiplication with 2 has to be performed on the mantissa.

2.3. Algorithms for computing inverse roots

The algorithm for computing inverse roots is combining the algorithms for roots and inverse described in Section 2.1 and 2.2.

The computation to perform to compute the inverse, $z$, is shown in (7).

$$z = \frac{1}{d|M_{d-1}...M_0| m_{-1} m_{-2} ... m_{-h} \cdot 2^{d \cdot \text{index}}}$$ \hspace{1cm} (7)

As shown in (7), the inverse root $d$ is performed on a floating-point number with the number base $2^d$ and in binary number representation. As shown in (7), the mantissa consists of $d$ integer bits $M$ and fractional bits $m$ where $h$ is the number of bits used in the fractional part.

In (8), (7) is simplified to get the exponent in the numerator, thus in base 2 with negative index.

$$z = \frac{1}{d|M_{d-1}...M_0| m_{-1} m_{-2} ... m_{-h} \cdot 2^{d \cdot \text{index}}} \cdot 2^{-d \cdot \text{index}}$$ \hspace{1cm} (8)

As shown in (8), when computing the exponent only the sign of index is changed. Since the integer part of the mantissa always has to be 1 or larger, the range of the mantissa will be $1 \leq \text{mantissa} < 2^d$. 

---

Fig. 5. Block diagram of the inverse algorithm.
When computing the approximation of the inverse square root, this is performed only on the range $1 \leq v < 4$, as shown in Fig. 6.

![Inverse Square Root Function](image)

Fig. 6. The inverse square root function.

A summary of the algorithm for computing the inverse square root function is shown in the block diagram of Fig. 7.

![Block Diagram of Inverse Square Root Algorithm](image)

Fig. 7. Block diagram of the inverse square root algorithm.

As shown in Fig. 7 the starting point is a floating-point number. Also as shown in Fig. 7 the exponent and mantissa are computed separately. When computing the mantissa the range after approximation is $0.5 < z \leq 1$. A special case is when $z = 1$, since then the mantissa in the appropriate format. In most cases, however, when $z < 1$ a multiplication with 2 has to be performed to get the mantissa in the appropriate format. When computing the exponent, initially the sign of the index is changed. Depending on the result when computing the approximation of the mantissa, when $z = 1$ the exponent remains untouched whereas when $z < 1$ the exponent is subtracted with 1.
3. Implementations of the inverse square root

As a summarizing example of the three algorithms described the implementation of the inverse square root is chosen. It is chosen since it includes both the square root and the inverse described in this paper. For comparison the inverse square root is implemented using two methodologies: Harmonized Parabolic Synthesis [2] and Newton-Raphson method [3] [4]. The novel methodology, Harmonized Parabolic Synthesis is chosen since it has been demonstrated to provide very efficient approximations. The Newton-Raphson method is chosen since it is known for providing a very efficient implementation of the inverse square root. It is also commonly used in signal processing applications for computing approximations on fixed-point numbers of a limited range. To facilitate the correct evaluation of the two methodologies the distribution of the error for both is set to be a normal distribution.

When implementing the inverse square root algorithm, only an approximation on the mantissa of the floating-point number is implemented. This is namely the only part that is different for the two implementations. The approximation is performed in the interval from 1.0 to nearly 4.0. The input to the approximation is a 15 bit fixed-point number and the output is also a fixed-point number with a target accuracy of 15 bits and with a normal distributed error. A bit-accurate MatLab model has been developed for simulations of the approximations and when for analysis of the performance of the approximations.

The implementations are performed as an ASIC with a ST Microelectronics 65nm General Purpose Standard-VT (1.1V) technology. For all implementations Synopsys Design Compiler [5], Synopsys Primetime [6] and Mentor Graphics Modelsim [7] are used.

3.1. Implementation using the harmonized parabolic synthesis methodology

The Harmonized Parabolic Synthesis methodology is founded on multiplication of two factors, both in the form of second-order parabolic functions, called the first sub-function, \( s_1(x) \), and the second sub-function, \( s_2(x) \). When recombined, as shown in (9), they approximate the original function \( f_{org}(x) \).

\[
f_{org}(x) = s_1(x) \cdot s_2(x) \quad (9)
\]

The first sub-function, \( s_1(x) \), is shown in (10).

\[
s_1(x) = x + c_1 \cdot (x-x^2) \quad (10)
\]

In (10), the coefficient \( c_1 \) determines the amplitude of the non-linear part. The second sub-function, \( s_2(x) \), is developed as a second-degree interpolation, as shown in (11), where the number of intervals in the interpolation decides the order of the accuracy of the approximation.

\[
s_2(x) = l_{2,1} + j_{2,1} \cdot x_w - c_{2,1} \cdot x_w^2 \quad (11)
\]
In (11), the coefficient $l_{2,i}$ is the starting point of an interval, coefficient $j_{2,i}$ is the sum of the coefficients of the linear and non-linear parts and $c_{2,i}$ is the coefficient of the non-linear part. To simplify the coding of the interval number, $i$, in hardware, the number of equal-range intervals in the second sub-function are chosen as 2 to the power of $w$, where $w$ is a natural number.

To facilitate the hardware implementation of the inverse square root, a normalization to satisfy the criterion that the values are in the interval $0 \leq x < 1.0$ on the $x$-axis and $0 \leq y < 1.0$ on the $y$-axis has to be performed. As shown in Fig. 8 the inverse square root function, $f(v)$, is in the interval $1.0 \leq v < 4.0$ on the $v$-axis and $1 \leq y < 0.5$ on the $f(v)$-axis. This means that a transformation on the function $f(v)$ has to be performed to generate the normalized function, $f_{org}(x)$, as shown in Fig. 8.

![Fig. 8. The function $f(v)$ before normalization and the normalized function, $f_{org}(x)$.](image)

To move the function $f(v)$ which is in the interval $1.0 \leq v < 4.0$ into the normalized interval on the $x$-axis $0 \leq x < 1.0$ is carried out by the substitution of $v = 3x + 1$. The developed expression of the preprocessing function is shown in (12).

$$ x = \frac{v - 1}{3} \quad \text{(12)} $$

After the substitution the function $f(v)$ also has to be moved into the normalized interval $0 \leq y < 1.0$ on the $y$-axis. As shown in (13) initially the function has to be multiplied with 2 and then subtracted with 1 achieve the normalized function, $f_{org}(x)$.

$$ f_{org}(x) = \frac{1}{\sqrt{3x + 1}} \cdot 2 - 1 = \frac{2}{\sqrt{3x + 1}} - 1 \quad \text{(13)} $$

The result $y$ from the processing part is in the interval $0 \leq y < 1.0$ and the desired output $z$ from the approximation shall be in the interval $0.5 < z \leq 1.0$. A transformation of $y$ is therefore needed. As shown in (14), initially 1 is added to $y$ which gives an offset of 1. To get the result into the desired interval is has to be divided with 2 as shown in (14). The developed expression of the post-processing function is shown in (14).
As shown in Fig. 8 the original function, \( f_{\text{org}}(x) \), starts in (0.0,1.0) and ends in (1.0,0.0). \( x \) has therefore to be substituted with 1-x as shown in (15). The result after the substitution is also shown in (15).

\[
s_1(x) = (1-x) + c_1 \cdot ((1-x) - (1-x)^3) = 1 - x + c_1 \cdot (1 - x - (1 - 2x + x^2)) = 1 - x + c_1 \cdot (x - x^3)
\]  

The only modification needed of (10) is to replace \( x \) with 1 - \( x \). As shown in (15) will this only affect the linear part, while the non-linear part remains unchanged.

When proceeding in the development of the algorithm of the processing stage the coefficients in (15) and (11) have to be decided but before that the number of intervals needed in (11) also have to be decided. The foundation of the Harmonized Parabolic Synthesis methodology is to approach the development with a holistic view. This is done by developing the two sub-functions simultaneously. When developing an approximation of an original function the first and second sub-functions are looked upon as one device. The methodology is to develop the first sub-function in such a way that the product of the two sub-functions gives a good conformity to the original function. The approach when developing the first sub-function, \( s_1(x) \), is not based on independent analytical calculations since it is dependent on the performance of the second sub-function, \( s_2(x) \). Therefore, the coefficient \( c_1 \) in the first sub-function has to be determined by, for different values of the coefficients, calculating the maximum absolute error, \( f_{\text{error}}(x) \), between the approximation and the original function according to (16).

\[
f_{\text{error}}(x) = |s_1(x) \cdot s_2(x) - f_{\text{org}}(x)|
\]

To perform the calculation of the absolute error, \( f_{\text{error}}(x) \), the second sub-function, \( s_2(x) \), has to be made dependent on the coefficient \( c_1 \) in the first sub-function, \( s_1(x) \) as shown in [2]. The calculation is interesting only as an indication of how the absolute error, \( f_{\text{error}}(x) \), depends on the coefficient \( c_1 \). When choosing the coefficient \( c_1 \) it has to be made with regard to both the characteristics of the error and the efficiency of the hardware implementation. The number of intervals in the second sub-function, \( s_2(x) \), needs to be increased to achieve the intended accuracy; this has also to be taken into account when performing the calculation of the error. In Fig. 9 the bit accuracy for the inverse square root function is shown when using 2, 4, 8, 16 and 32 intervals in the second sub-function, \( s_2(x) \) with different values of the coefficient, \( c_1 \).
Fig. 9. The bit accuracy depending on $c_1$ for 2, 4, 8, 16 and 32 intervals in the second sub-function, $s_2(x)$.

Values of the accuracy in bits for $c_1$ values less than -0.5 are not shown since the results are not defined because of divisions by numbers near 0 when performing the calculations. Based on Fig. 9, values of the coefficient $c_1$ are chosen to allow an efficient implementation of the hardware. As shown in (15), $c_1$ is fed into a multiplier; therefore choosing a value that is a power of two is desirable. In Fig. 9, the desired accuracy at 15 bit as indicated. To accomplish this, at least 32 intervals are necessary. As shown in Fig. 9, choosing $c_1 = 0.0$ for 32 intervals is interesting since this will exclude the multiplication in (15) and thereby reducing the hardware. In the implementation study presented later in this paper, we have also included a design with more than 32 intervals. With higher number of intervals the internal word lengths in the architecture can be shortened. Thereby it is possible to improve chip area and speed in a design. Table 3 shows preliminary study of chip area and critical path delay dependent of the number of intervals.

<table>
<thead>
<tr>
<th>Number of Intervals</th>
<th>Chip Area</th>
<th>Critical Path Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>16300 $\mu$m²</td>
<td>4.3 ns</td>
</tr>
<tr>
<td>64</td>
<td>15700 $\mu$m²</td>
<td>4.2 ns</td>
</tr>
<tr>
<td>128</td>
<td>15900 $\mu$m²</td>
<td>4.3 ns</td>
</tr>
<tr>
<td>256</td>
<td>17600 $\mu$m²</td>
<td>4.1 ns</td>
</tr>
<tr>
<td>512</td>
<td>19500 $\mu$m²</td>
<td>3.9 ns</td>
</tr>
</tbody>
</table>

Since the implementation using 512 intervals in Table 3 is the fastest and that the increase in chip area is relatively small, it was used in this survey.
3.1.1. Architecture
As a comparison, two implementations have been made, one with 32 intervals and one with 512 intervals.

Finding the optimal set of coefficients can be done as an iterative procedure starting from an initial set. After deciding the initial coefficients in the first and second sub-function the next step is to optimize the architecture. The method for the optimization is to decide the word lengths used in the architecture and to optimize the coefficients. The optimization is performed in an iterative trial and error manner. The evaluation of different coefficient values should be performed in parallel with the evaluation of the word length, since the truncation error effects have impact on the performance of calculations in the design. The strategy is to adjust coefficients and word lengths in the design for best accuracy and distribution of the error. The values of the coefficients are specified in [8].

In practice, the simulation of the approximation is performed with a bit-accurate C model and the performance of the approximation is analyzed in MatLab.

The architecture consists of the three parts, preprocessing, processing, and post-processing. The preprocessing and the post-processing are implementations of (12) and (14) whereas the processing is implemented as an approximation of (13). Pre- and postprocessing are not influenced by the number of intervals used in the processing part.

Preprocessing
The preprocessing part is implemented according to (12), as shown in Fig. 10. In Fig. 10 $c = 1/3$.

![Fig. 10. The preprocessing stage with word lengths annotated with names.](image)

After optimization, the word lengths of the data paths in the preprocessing are as shown in Table 4.

<table>
<thead>
<tr>
<th>Data path</th>
<th>Word length</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>15</td>
</tr>
<tr>
<td>v2</td>
<td>15</td>
</tr>
<tr>
<td>c1</td>
<td>14</td>
</tr>
<tr>
<td>x1</td>
<td>16</td>
</tr>
</tbody>
</table>

TABLE 4: The word lengths in the preprocessing stage
**Processing**

The implementation of the processing part is made as a product of the first sub-function (15) and second the sub-function (11). Since the coefficient $c_1 = 0.0$ the first sub-function is reduced to $1 - x$ as shown in Fig. 11.

After optimization in the way described above, the word lengths of the data paths of the implementations with 32 and 512 intervals, described in Fig. 11, are as shown in Table 5.

**TABLE 5: The word lengths in the processing stage for both 32 and 512 intervals**

<table>
<thead>
<tr>
<th>Data path</th>
<th>Word length 32 intervals</th>
<th>Word length 512 intervals</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>xs</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>xw</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>x2</td>
<td>12</td>
<td>5</td>
</tr>
<tr>
<td>i1</td>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>c2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>cx</td>
<td>11</td>
<td>6</td>
</tr>
<tr>
<td>l2</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>j2</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td>jx</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>jl</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>jlc</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>y1</td>
<td>16</td>
<td>16</td>
</tr>
</tbody>
</table>
Postprocessing

The postprocessing part is implemented according to (14), as shown in Fig. 12. Note that the division with 2 in (14) was replaced with one right shift. The word lengths of the data paths in Fig. 12 are as shown in Table 6.

![Diagram](image)

Fig. 12. The postprocessing stage with word lengths annotated with names.

<table>
<thead>
<tr>
<th>Data path</th>
<th>Word length</th>
</tr>
</thead>
<tbody>
<tr>
<td>y1</td>
<td>16</td>
</tr>
<tr>
<td>z1</td>
<td>17</td>
</tr>
<tr>
<td>z2</td>
<td>18</td>
</tr>
</tbody>
</table>

TABLE 6: The word lengths in the postprocessing stage for both 32 and 512 intervals

3.2. Implementation using the Newton-Raphson method

The Newton-Raphson method is a method to find a root of an arbitrary function, \( f(y) = 0 \), numerically. Successively, through iterations, the method finds an increasingly accurate approximation. The general iteration formula for the Newton-Raphson method is shown in (17).

\[
y_i = y_{i-1} - \frac{f(y_{i-1})}{f'(y_{i-1})}
\]  

(17)

In (17), \( i \) is the index for the iteration and the iteration is initiated by a presupplied guess \( y_0 \). To perform iteration the derivative in (17) must meet the condition that \( f'(y) \neq 0 \).

Initially the function to perform the approximation of the inverse square root of \( v \) is developed. In (18) the expression to the function in (19) is developed.

\[
y_{i-1} = \frac{1}{\sqrt{v}} \Leftrightarrow \sqrt{v} = \frac{1}{y_{i-1}} \Leftrightarrow v = \left( \frac{1}{y_{i-1}} \right)^2 \Leftrightarrow \frac{1}{y_{i-1}^2} - v = 0
\]  

(18)

\[
f\left( \frac{1}{\sqrt{v}} \right) = \frac{1}{y_{i-1}^2} - v
\]  

(19)

In (20) is the derivative (19) computed.
With (19) and (20) inserted in (17) the iteration formula is developed in (23).

\[ y_i = \frac{3 - v \cdot y_{i-1}^2}{2} \]  

The iterations in the Newton-Raphson method are initiated by a presupplied guess of \( y_0 \). To improve the convergence, thus reducing the number of iterations, the guess should be as close as possible to the actual result. To accomplish this, a look-up table storing initial values is used. Where the addressing is done through using the operand \( v \).

The strategy used for determining the initial values stored in the look-up table is illustrated in Fig. 13.

![Fig. 13. The target function together with the strategy for the initial values.](image)

The strategy is trying to solve several problems. As shown in Fig. 13, the range width within which an initial value is valid depends on the slope of the function on which the approximation is performed. The steeper the slope the shorter the range within which an initial value is valid. This is to get a uniform convergence of the different intervals. Besides this, the initial values in the look-up table are situated both above and below the function. This accomplishes both that the convergence is improved and that the error is distributed around zero.

A drawback with this strategy is that when the input \( v = 1 \) the output \( y \neq 1 \). To solve this the initial value \( y = 1 \) is stored in the look-up table when \( v = 1 \).

3.2.1. Architecture
To investigate the influence of the number of iterations on an implementation, two implementations have been made, one using one iteration and one using two. Also for these implementations the desired accuracy was 15 bit. After optimization in the way described above,
it was found that the implementation using one iteration needs 95 entries and the implementation using two iterations needs 14 entries.

One iteration

The algorithm to implement is shown in (21), giving the architecture shown in Fig. 14. The size of the look-up table (LUT) is 95 initial values.

![Diagram](image1)

Fig. 14. The processing stage with word lengths annotated with names.

After optimization, the word lengths of the data paths marked in Fig. 14 are as shown in Table 7.

<table>
<thead>
<tr>
<th>Data path</th>
<th>Word length</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>15</td>
</tr>
<tr>
<td>l1</td>
<td>15</td>
</tr>
<tr>
<td>l2</td>
<td>16</td>
</tr>
<tr>
<td>lv</td>
<td>19</td>
</tr>
<tr>
<td>s3</td>
<td>19</td>
</tr>
<tr>
<td>d2</td>
<td>19</td>
</tr>
<tr>
<td>z1</td>
<td>18</td>
</tr>
</tbody>
</table>

Two iterations

The second implementation, uses two iterations of (21) and has a look-up table (LUT) with 14 initial values, as shown in Fig. 14.

![Diagram](image2)

Fig. 15. The processing stage with word lengths annotated with names.
After optimization, the word lengths of the data paths marked in Fig. 15 are as shown in Table 8.

<table>
<thead>
<tr>
<th>Data path</th>
<th>Word length</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>15</td>
</tr>
<tr>
<td>l1</td>
<td>16</td>
</tr>
<tr>
<td>l2</td>
<td>16</td>
</tr>
<tr>
<td>lv</td>
<td>16</td>
</tr>
<tr>
<td>s3</td>
<td>16</td>
</tr>
<tr>
<td>d2</td>
<td>16</td>
</tr>
<tr>
<td>i1</td>
<td>16</td>
</tr>
<tr>
<td>i2</td>
<td>17</td>
</tr>
<tr>
<td>iv</td>
<td>17</td>
</tr>
<tr>
<td>m3</td>
<td>18</td>
</tr>
<tr>
<td>q2</td>
<td>18</td>
</tr>
<tr>
<td>z2</td>
<td>17</td>
</tr>
</tbody>
</table>

4. Results

In this section the results of the implementations of the inverse square root using the Harmonized Parabolic Synthesis methodology and the Newton-Raphson method are described. Four implementations are made, two following the Harmonized Parabolic Synthesis methodology using 32 and 512 intervals, respectively, and two following the Newton-Raphson method with one and two iterations, respectively. We consider only the results of the approximation of the mantissa. The characteristics and distribution of the error are compared with respect to maximum error, symmetric distribution around zero and narrow gravity center around zero of the distribution. The implementation aspects are compared with respect to chip area, critical path delay and power consumption.

4.1. Characteristics and distribution of the error

The characteristics and the distribution of the error are studied in the interval from 1.0 to nearly 4.0.

4.1.1. The harmonized parabolic synthesis methodology using 32 intervals

Fig. 16 shows the distribution of the error of the implementation using the Harmonized Parabolic Synthesis methodology with 32 intervals in the second sub-function.
Fig. 16. Distribution of the error of the design based on the Harmonized Parabolic Synthesis methodology with 32 intervals.

As shown in Fig. 16, the error is evenly distributed around zero, which is confirmed by the related histogram of the distribution of the error in Fig. 17. The distribution is well centered around zero, which is an advantageous characteristic for the following calculations in an algorithm. The advantageous characteristics of the error are confirmed in Table 9 later in this section by the fact that the standard deviation and the root mean square value are equal.

Fig. 17. Histogram of the distribution of the error of the design using the Harmonized Parabolic Synthesis methodology with 32 intervals.

4.1.2. The harmonized parabolic synthesis methodology using 512 intervals

Fig. 18 shows the distribution of the error of the implementation using the Harmonized Parabolic Synthesis methodology with 512 intervals in the second sub-function.
Again, as shown in Fig. 18 the error is evenly distributed around zero. This is also confirmed by the related histogram in Fig. 19.

Fig. 19 shows a triangle-like shape of the distribution of the error around zero. This indicates a synthesis of two rectangular error distributions, probably from the two look-up tables \( l_{2,i} \) and \( j_{2,i} \) in Fig. 20. The error is distributed around zero, which is an advantageous characteristic for the following calculations in an algorithm. The advantageous characteristics of the error are confirmed in Table 9 by the fact that the standard deviation and the root mean square value are equal.

4.1.3. The Newton-Raphson method using one iteration

Fig. 20 shows the distribution of the error of the implementation using the Newton-Raphson method with one iteration and with a look-up table with 95 initial values.
Fig. 20. Distribution of the error of the design using the Newton-Raphson method with one iteration and with a look-up table with 95 initial values.

As shown in Fig. 20 the error tends not to be evenly distributed around zero which is confirmed by the related histogram in Fig. 21.

Fig. 21 shows that the distribution of the error has a displacement towards the right side of the histogram. The error is not sufficiently evenly distributed around zero for it not to be a disadvantage in the following calculations. This distortion is confirmed by the difference between the standard deviation and the root mean square values, given in Table 9.

4.1.4. The Newton-Raphson method using two iterations

Fig. 22 shows the distribution of the error of the implementation using the Newton-Raphson method with two iterations and with a look-up table with 14 initial values.
As shown in Fig. 22 there is a very small tendency to the error not to be evenly distributed around zero. In the related histogram in Fig. 23 however, the tendency is hard to see.

Fig. 23 shows not clearly that the distribution of the error has a displacement towards the right side of the histogram. This small distortion is however confirmed by the difference between the standard deviation and the root mean square values in Table 9. But since the maximum error is so small this will not have any influence on the following calculations.

4.1.5. Error characteristics
The error characteristics of the four designs are summarized in Table 9.
Table 9 shows some general differences between the two implementation methodologies. The mean error and median error are much smaller for the implementation using the Harmonized Parabolic Synthesis methodology. Similar there are no differences in the standard deviation and the root mean square values. The better error characteristics of the designs using the Harmonized Parabolic Synthesis methodology depends on that this methodology has greater potential in the optimization of the design depending on its structure. The Newton-Raphson method is lacking these opportunities to control the error characteristics.

<table>
<thead>
<tr>
<th>Implementation Methodology</th>
<th>Maximum error</th>
<th>Mean error</th>
<th>Median error</th>
<th>Standard Deviation</th>
<th>Root Mean Square</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harmonized Parabolic Synthesis 32 intervals</td>
<td>2.9032e-05</td>
<td>1.2857e-08</td>
<td>2.9354e-08</td>
<td>6.9841e-06</td>
<td>6.9841e-06</td>
</tr>
<tr>
<td>Harmonized Parabolic Synthesis 512 intervals</td>
<td>3.0508e-05</td>
<td>1.5962e-08</td>
<td>9.2308e-08</td>
<td>1.2863e-05</td>
<td>1.2863e-05</td>
</tr>
<tr>
<td>Newton-Raphson method one iteration</td>
<td>2.9733e-05</td>
<td>1.9028e-06</td>
<td>1.6665e-06</td>
<td>8.9800e-06</td>
<td>9.1794e-06</td>
</tr>
<tr>
<td>Newton-Raphson method two iterations</td>
<td>1.8403e-05</td>
<td>3.8414e-07</td>
<td>3.6850e-07</td>
<td>5.8901e-06</td>
<td>5.9026e-06</td>
</tr>
</tbody>
</table>
4.2. Hardware performance

The implementations are made as an ASIC with a ST Microelectronics 65nm General Purpose Standard-$V_T$ (1.1V) technology. Their power consumption at 10 MHz is shown in Table 10 together with chip area and critical path delay.

<table>
<thead>
<tr>
<th>Implementation Methodology</th>
<th>Chip Area</th>
<th>Critical Path Delay</th>
<th>Energy Consumption per Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>Harmonized Parabolic Synthesis 32 intervals</td>
<td>13900 μm²</td>
<td>4.3 ns</td>
<td>0.002073 nW 129%</td>
</tr>
<tr>
<td>Harmonized Parabolic Synthesis 512 intervals</td>
<td>17100 μm²</td>
<td>3.9 ns</td>
<td>0.001895 nW 118%</td>
</tr>
<tr>
<td>Newton-Raphson method 1 iteration</td>
<td>15600 μm²</td>
<td>4.6 ns</td>
<td>0.001610 nW 100%</td>
</tr>
<tr>
<td>Newton-Raphson method 2 iterations</td>
<td>22800 μm²</td>
<td>6.7 ns</td>
<td>0.006378 nW 396%</td>
</tr>
</tbody>
</table>

As shown in Table 10, the Harmonized Parabolic Synthesis methodology with 32 intervals has the smallest chip area. The Harmonized Parabolic Synthesis methodology with 512 intervals has the shortest critical path delay and the energy consumption per sample is smallest for the Newton-Raphson method with one iteration. The Newton-Raphson method with two iterations is outperformed by the others in all aspects.

5. Comparing implementations

This section summarizes the comparison of the four implementations of the inverse square root function done. Two of them used the Harmonized Parabolic Synthesis methodology, and two of them used the Newton-Raphson method. With the Harmonized Parabolic Synthesis methodology the implementations use 32 (HPS 32) and 512 (HPS 512) intervals in the second sub-function. In the implementations use the Newton-Raphson method, one (NR 1) and two (NR 2) iteration stages are used.

Quantitative comparisons of different approximation methodologies should always be done in the context in which the approximations are going to be used. In the absence of this, the comparison of the implementations will therefore be made in the form of reasoning.

**Chip area**

As shown in Table 10 HPS 32 has the smallest chip area, while it is slightly larger for NR 1 and HPS 512 and additionally larger for NR 2.
Critical path delay
Table 10 also shows that HPS 512 has the shortest critical path delay while it is slightly larger for HPS 32 and NR 1 and, again, additionally larger for NR 2.

Energy consumption per sample
When it comes to energy consumption per sample, Table 10 shows that it is best for NR 1 while it is slightly higher for HPS 512 and HPS 32 and much higher for NR 2.

The overall comparison of the hardware gives that NR 2 is worst-performing in all categories. For HPS 32, HPS 512 and NR 1 each of them are best in one of the categories. The difference in performance between these three implementations is generally small; it is therefore difficult to appoint a winner.

Characteristics and the distribution of the error
In the Sections 4.1.1, 4.1.2 and 4.1.5 it was shown that both HPS 32 and HPS 512 have a distribution of the error that is centered around zero. It was also shown that the distribution of the error for HPS 32 is better than for HPS 512 since the standard deviation is smaller. A comparatively smaller standard deviation implies that the distribution of the error as more narrow around zero.

The Sections 4.1.3, 4.1.4 and 4.1.5 show that both NR1 and NR 2 have a distribution of the error that is not well centered around zero. This may in the following calculations result in an accumulated error which can ruin the computation. Since it is hard to center the distribution of the error around zero when designing using the Newton-Raphson method. It is common to increase the accuracy in order to reduce the effects of the accumulated error.

The small maximum error for NR 2 (according to Table 8) comes with the prize of larger chip area, longer critical path delay and higher power consumption per sample (according to Table 9).

Summary of comparisons
When making a summary of the comparisons of the hardware performance and the distribution of the error the three implementations HPS 32, HPS 512 and NR 1 are comparable when it comes to hardware performance. However, HPS 32 distinguishes itself by better performance when it comes to the distribution of the error. As mentioned initially in this section, the context in which the implementation is going to be used is decisive for selection. The different performance parameters need therefore to be weighed against each other.

6. Conclusion
In many future applications, such as in the MIMO communication systems, a massive computation of complex matrix operations needs to be performed. In these matrix operations, such as QR decomposition, the functions roots, inverse and inverse roots are computed in large quantities. For these functions an implementation in a straightforward way applying an approximation methodology will not lead to an efficient implementation. This paper introduces novel dedicated algorithm for these functions founded on the use of floating-point format and, for the roots functions, also a change of number base. Using the floating-point format enables simpler
solutions but also increased accuracy, since the approximation algorithm is performed on a mantissa of limited range. Also the change of number base when computing the roots enables simpler solutions when computing the approximation. The most important characteristics of the developed dedicated algorithms are fast computation, low power, and simple and efficient hardware implementation.

In this paper, two implementations of each of two methodologies, the Harmonized Parabolic Synthesis methodology and the Newton-Raphson methodology, are made. In the Harmonized Parabolic Synthesis methodology the accuracy is scaling with the number of intervals used in the interpolation part. In the Newton-Raphson method the accuracy is scaling with the number of guesses stored in the look-up table and the number of iterations. For each methodology, one of the two implementations was made allowing higher accuracy than the required. A deduction that could be made from this was that, with increased accuracy, the size of the hardware is growing much slower when using the Harmonized Parabolic Synthesis methodology than when using the Newton-Raphson methodology.

References
