RATIONAL ARITHMETIC UNITS IN COMPUTER SYSTEMS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Oskar Mencer

February 2000

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Michael J. Flynn
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Martin Morf

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Gio Wiederhold

Approved for the University Committee on Graduate Studies:

Abstract

Computer arithmetic remains important as we move to systems-on-a-chip that dedicate large areas for special-purpose arithmetic units. System application areas such as signal processing, multimedia, and mobile computing require the evaluation of functions as fast as possible with as little power as possible. It is well known that rational approximations and continued fractions offer fast approximations and efficient algorithms to compute rational functions. Due to the difficulty of converting between continued fractions and binary numbers continued fractions have been impractical for the design of computer systems.

Rational arithmetic is about division of two numbers, division of polynomials and/or division of digits. Continued fractions(CFs) enable the development of divide-add structures for digits of rational numbers. Continued fraction arithmetic deals with computing rational functions where input and output values are represented as simple continued fractions. The basic remaining problems of previous work are threefold: choosing the optimal CF-digit representation, converting between simple continued fractions and binary numbers, and error control.

The M-log-Fraction Transform(MFT), introduced in this work, solves all three problems. Instant conversion is shown to be related to the distance between the '1's of the binary number. Applying M-log-Fractions to continued fraction arithmetic algorithms reduces the complexity of the implementation of the CF algorithm to shift-and-add structures, or more specifically, digit-serial arithmetic algorithms for

computing rational functions. A multiplication-based scheme can be used to evaluate higher-degree rational approximations.

This thesis demonstrates two applications of the MFT:

(1) a rational arithmetic unit computing functions such as (ax+b)/(cx+d) in a shift-and-add-based structure.

(2) the evaluation of rational approximations (or continued fraction approximations) in a multiplication-based structure.

The MFT bridges the gap between continued fractions and the binary number representation, enabling the design of a new class of efficient rational arithmetic units and the efficient evaluation of rational approximations.

Acknowledgments

I am at a loss of words to express my gratitude to my parents Nada and Lazar for showing me the path that ultimately led to this thesis. For the recent years, I want to thank my wife, Kati, for all the patience and support that made it easier to cope with the workload of a graduate program at Stanford.

I am very fortunate to have had Prof. Flynn and Prof. Morf as my advisors in academic and non-academic issues over the past few years. This thesis would not have been possible without their support, constant encouragement, and patience of listening to my ideas. At the Technion, I am indebted to Prof. Birk and Dr. Mendlson (now at Intel) for teaching me about computer systems, supporting my applications to graduate school, and their guidance even years after I left Israel.

Here at Stanford I am grateful to all Stanford faculty who have taught me, and my fellow students from whom I learned so much. I want to especially mention Prof. Rosenblum (for giving me the opportunity to see what programming is really about), Prof. DeMicheli, Prof. Cheriton, Prof. Horowitz, Fred Gibbons, and my colleagues Gerald Engel, Pablo Molinero-Fernandez, Marco Platzner, Luc Séméria, and Krishna Nayak. I want to thank past and present students of the Computer Architecture and Arithmetic Group which I enjoyed working with over the past four years, especially, Stuart Oberman, Kevin Rudd, Alice Yu, and Albert Liddicoat.

For discussions about my reasearch and especially this thesis I want to thank Prof. Bump (Math/Stanford), Dr. Luk (Imperial College), Prof. Kahan (Berkeley), Prof.

Ercegovac (UCLA), and Prof. Delosme(Université d'Evry).

For her patience and technical support I thank Susan Gere and for system administrational emergency help I thank Charlie Orgish. Finally, this research is supported by DARPA and a grant by Compaq Systems Research Center.

Contents

List of Figures

Chapter 1

Introduction

The following table summarizes the notation that will be used throughout the thesis.

Table 1.1: Symbols and Definitions

$\stackrel{\triangle}{=}$ definition	$\equiv$ equivalence	$\simeq$ approximately equal	
$\overline{p_i} \stackrel{\triangle}{=} \frac{1}{p_i}$	$\overline{s_i} \stackrel{\triangle}{=} \frac{1}{s_i}$	$\overline{M_i} \stackrel{\triangle}{=} \sum_{j=1}^{i} M_j$	$0.\overline{3} \stackrel{\triangle}{=} 0.3333\ldots$
$\underline{A}$ Matrix	$\lvert\underline{A}\rvert$ Determinant	$\lfloor x \rfloor$ largest integer smaller than x	

$\{a, b, c, \ldots\}$	a set of values
$\mathcal{N}$	set of natural numbers $\{1, 2, 3, \ldots\}$
$\mathcal{R}$	set of rational numbers $r = \frac{x}{y}$ where $x, y \in \mathcal{N}$
$[a_0; a_1, \ldots a_n]$	$a_0 + \frac{1}{\lvert a_1} + \frac{1}{\lvert a_2} + \cdots + \frac{1}{a_n} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \cdots \frac{1}{a_n}}}$
$[x]$	citation, reference number x
$< M_1, M_2, M_3, M_4, M_5, \ldots, M_n >$	$[0; 2^{M_1}, -(2^{-M_1} + 2^{M_2}), (2^{-M_2} + 2^{M_3}), -(2^{-M_3} + 2^{M_4}), (2^{-M_4} + 2^{M_5}) \ldots \pm (2^{-M_{n-1}} + 2^{M_n})]$

Although the following mathematical terms are well known, they are defined here for completeness.

Theorem: A formula, proposition, or statement in mathematics or logic deduced or to be deduced from other formulas or propositions (Webster [72]).

Lemma: An auxiliary proposition used in the demonstration of another proposition (Webster [72]).

Corollary: A proposition inferred immediately from a proved proposition with little or no additional proof (Webster [72]).

Equivalence: In this thesis, 'equivalence' refers to non-trivial equalities that are only valid within a specific set of conditions.

1.1 Motivation for Computer Arithmetic

Computer arithmetic[2][4][5][6] deals with the computation and implementation of mathematical operations and functions in computer systems. In general, any operation can be implemented in hardware or – by using the arithmetic units of a processor – in software. Software implementations use the arithmetic operations that are available in a general purpose microprocessor to approximate more complex operations or functions. Hardware implementations make use of specialized arithmetic circuits.

Implementations of arithmetic circuits in hardware are based on mapping the desired function to an Application Specific Integrated Circuit (ASIC) or to a Field-Programmable Gate Array(FPGA). With shrinking feature sizes, advanced arithmetic units also get incorporated into general purpose microprocessors. For example, modern microprocessors extend the division unit to a divide/square-root[48][50]. Multimedia extensions such as Intel's MMX enable parallel 8-bit computations within larger arithmetic units. Microprocessors for signal processing typically include an integrated multiply-add unit that joins a multiplication and an addition unit to speedup the evaluation of, for example, polynomial approximations.

The main focus of this thesis is on efficient general-purpose hardware (VLSI) implementation of rational arithmetic for computer systems, i.e. rational arithmetic units. Rational arithmetic units compute functions that include division. An example

for a rational arithmetic unit is the bilinear function $y = \frac{ax+b}{cx+d}$. Just as the multiply-add unit is a natural extension to multiplication units, rational arithmetic units are candidate-extensions of current floating-point divide units.

Which applications are likely to benefit from a rational arithmetic unit? The datapath of a computer system consists of arithmetic units and memory (storage). The performance of a computer system for a given application is limited by the "bottleneck"[1][10] of the computation. The bottleneck is created by either the arithmetic units or the memory hierarchy. The arithmetic units implemented in this work are most beneficial for applications that are limited by arithmetic resources. Applications that are limited by arithmetic resources are within application areas that tend to exhibit large amounts of parallelism at the arithmetic level.

Data-intensive applications such as signal processing, multimedia, graphics, and geometrical computations with regular data access require fast approximation of specific analytic functions. In general, such functions are approximated either with polynomials or rational functions (e.g. the ratio of two polynomials). However, for most functions rational approximations converge faster than polynomial approximations.

Rational approximation offers efficient evaluation of analytic functions represented by the ratio of two polynomials.

$$f(x) \sim \frac{((a_n x + a_{n-1})x + \cdots a_1)x + a_0}{((b_m x + b_{m-1})x + \cdots b_1)x + b_0} = \frac{c_n x^n + c_{n-1}x^{n-1} \cdots c_1 x + c_0}{d_m x^m + d_{m-1}x^{m-1} \cdots d_1 x + d_0} \qquad (1.1)$$

Koren[34] evaluates rational approximations with a latency of *max(m,n)* multiply-add (MA) operations and a final division. Typically, division is implemented either with multiplicative[1], or digit-serial (iterative) methods, also called digit-recurrence methods[50]. The algorithms presented in this thesis fall into the class of digit-serial methods. The main objective is to compute rational functions such as $\frac{ax+b}{cx+d}$ producing

the result one digit at a time. The proposed algorithms combine a novel number representation or encoding with state-of-the-art rational algorithms.

1.2 Representing Numbers in Computer Systems

Computer arithmetic is based on partitioning integer or rational numbers into representable digits. The size of the digits is called radix r. Using the *weighted positional number system* numbers are represented by digits s_i and powers of radices r as follows:

$$x = s_0 r^0 + s_1 r^1 + s_2 r^2 + s_3 r^3 + \cdots \tag{1.2}$$

A binary number consists of binary digits $s_i = \{0, 1\}$ with radix $r = 2$. However, alternative encodings of binary digits are sometimes advantageous. We distinguish redundant and non-redundant encodings of numbers. Non-redundant encodings guarantee that for every number there is one and only one series of digits that represents the number.

Redundant encodings are defined by Omondi[6] page 456, as: "A radix-R redundant signed-digit number system is one that is based on a digit set $S \triangleq \{-N, -(N-1), \ldots, -1, 0, 1, \ldots, M-1, M\}$, where $1 \leq N \leq R-1$, $1 \leq M \leq R-1$ and $|S| > R$. The last condition allows each digit to assume more than R values and gives rise to redundancy."

Non-redundant encodings use a minimal amount of memory to store numbers, while redundant encodings enable high-speed arithmetic algorithms for division (e.g. see [41]). For example, digits $s_i = \{1, -1\}$ create a non-redundant representation of numbers called *signed digits*[53]. An example of redundant digits for radix $r = 2$ is $s_i = \{1, 0, -1\}$ e.g.

$$(1) \cdot 2^2 + (0) \cdot 2^1 + (-1) \cdot 2^0 = (0) \cdot 2^2 + (1) \cdot 2^1 + (1) \cdot 2^0 = 3 \tag{1.3}$$

Iterative division uses redundant digit-sets $s_i = \{-a, \ldots, +a\}$[41]. What is the advantage of the redundant representation for division? At each iteration the basic division algorithm has to compute the correct next output digit based on the state within the arithmetic unit. In the case of non-redundant digits, there is only one correct next output digit. In the case of redundant digit sets there are multiple correct candidates for the next output digit. Imagine a table that maps the internal state of the divider and the current input digit to the next correct output digit. For redundant digits, there are multiple choices possible, thus the table consists of many "don't care" entries. These "don't care" entries translate directly to smaller and faster logic to compute the next digit[41].

Digit-serial arithmetic units operate on streams of input digits producing a stream of output digits. The digits can enter and leave the arithmetic unit in a least-significant digit (LSD) first, or most-significant digit (MSD) first order. MSD first is also called *Online Arithmetic*[19][20].

Digit-serial design uses iterative arithmetic schemes. In general, a digit-serial arithmetic unit consumes one input digit and produces one output digit in one iteration. The size of the digit (or radix) and the overall precision (bits per datum) determine the number of required iterations. For example, computing with an overall precision of 16-bit numbers with radix-2 requires 16 iterations, while radix-4 reduces the latency to 8 iterations. Increasing the radix also typically increases the delay per iteration, reducing the advantage of increasing the radix.

IEEE Floating-point[9] introduces standardized exact rounding modes for individual operations. How do we control the accumulation of roundoff error during the evaluation of expressions? The order in which the expressions are evaluated has an effect on overall precision. Details on how to deal with expressions at the compiler-level are shown in [49]. In order to avoid resulting numerical instabilities of computational algorithms it is sometimes necessary to increase the precision of the computation,

e.g. when dividing by very small numbers, or when dividing by the difference of two very large numbers. An optimal solution is to use *exact arithmetic*: In exact arithmetic, rational numbers are represented as fractions and irrational numbers are represented as symbolic expressions such as square-root[26], e.g. $\sqrt{2}$. A proposed method for exact arithmetic hardware is to store each value as a ratio of two numbers separated by a "slash": the *slash* number system[47], e.g. represent $0.\overline{3}$ by (1)/(3). Besides being redundant, the slash numbers are not distributed uniformly, making error estimation very difficult. Another disadvantage of this method is that the two numbers (especially the denominator) grow very quickly. Expanding the value of a fractional number into continued fraction digits gives us an alternative approach to exact arithmetic which is proposed in [45].

An alternative approach to improve the accumulation of roundoff error is to build compound arithmetic units. Compound arithmetic units join arithmetic operations together evaluating entire expressions. At each iteration the state of the arithmetic unit encodes the remaining error. Exact arithmetic is achieved by terminating the computation as soon as the error is sufficiently small. In our case continued fractions will lead to compound arithmetic units for rational functions.

1.3 Introduction to Continued Fractions

Continued fractions can be used as a rational number representation. This section provides the basic definitions and terminology that will be used throughout the thesis.

Let us start by defining some continued fraction(CF) forms. (1) *Finite continued fractions* represent rational numbers that are constructed as follows: for $A_i, B_i, a_i, b_i \in \mathcal{R}$

$$\frac{A_n}{B_n} = a_0 + \frac{b_1}{a_1 + \frac{b_2}{a_2 + \cdots \frac{b_n}{a_n}}} = a_0 + \frac{b_1|}{|a_1} + \frac{b_2|}{|a_2} + \cdots + \frac{b_n}{a_n} \tag{1.4}$$

(2) *Simple continued fractions* form a special case of a finite continued fraction where all partial quotients $b_i = 1$. The following vector notation denotes *simple continued fractions.*

$$[a_0; a_1, \ldots a_n] \triangleq a_0 + \frac{1|}{|a_1} + \frac{1|}{|a_2} + \cdots + \frac{1}{a_n} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \cdots \frac{1}{a_n}}} \tag{1.5}$$

(3) *Regular continued fractions* are simple continued fractions with all $a_i \in \mathcal{N}^+$, except $a_0 \in \mathcal{N}$.

Regular continued fractions are alternative representations of rational numbers[62]. In addition, CFs are at the basis of rational approximation theory. Therefore, the representation of rational numbers by CFs connects number representation to the rational approximation of transcendental functions. Peter Henrici already mentions that "we are still missing a general theory explaining the connection between transcendental functions and continued fractions"(the introduction in [17]). In other words, there currently is no theory explaining the simple continued fraction approximations below. Still, the already known continued fraction expansions cover most useful functions[35]. Examples are:

$$\frac{e}{e-1} = [0; 1, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, \ldots] \tag{1.6}$$

$$\tan(x) = \left[0; \frac{1}{x}, -\frac{3}{x}, \frac{5}{x}, -\frac{7}{x}, \frac{9}{x}, -\frac{11}{x}, \ldots\right] \tag{1.7}$$

$$arctan(x) = \left[0; \frac{1}{x}, \frac{3}{x}, \frac{5}{x}\left(\frac{1}{2}\right)^2, \frac{7}{x}\left(\frac{2}{3}\right)^2, \frac{9}{x}\left(\frac{3}{4}\right)^2, \ldots\right] \tag{1.8}$$

$$\frac{arcsin(x)}{\sqrt{1-x^2}} = \left[0; \frac{1}{x}, -\frac{3}{1\cdot 2}\cdot\frac{1}{x}, \frac{5}{1\cdot 2}\cdot\frac{1}{x}, \frac{7}{3\cdot 4}\cdot\frac{1}{x}\ldots\right] \tag{1.9}$$

or the incomplete Γ-function:

$$\Gamma(0.5, x) = e^{-x}\cdot x^{0.5}\cdot\left[0; x, (0.5)^{-1}, x, (1.5)^{-1}, x, (2.5)^{-1}, x, (3.5)^{-1}\ldots\right] \tag{1.10}$$

Given the above CF approximations, let us take a closer look at a more general way of transforming CFs to rational numbers. The following non-obvious two equations are at the heart of continued fraction theory showing the link between rational numbers and continued fractions. The step from rational numbers to continued fractions is one step of the journey from a rational binary number to continued fractions, which is required to build rational arithmetic units presented in this thesis. A finite continued fraction with n partial quotients can always be transformed into a ratio $\frac{A_i}{B_i}$ using the iterative equations:

$$A_n = a_n A_{n-1} + b_n A_{n-2} \tag{1.11}$$

$$B_n = a_n B_{n-1} + b_n B_{n-2} \tag{1.12}$$

where $\frac{A_{n-1}}{B_{n-1}}$ corresponds to the value of the same continued fraction without the i^{th} partial quotient. Initial conditions are $A_0 = a_0$, $B_0 = 1$, $A_{-1} = 1$, and $B_{-1} = 0$ (see for example [12],[63]). The iterative equations can be used to convert between continued fractions and rational values. Evaluating continued fractions reduces to multiply-add operations with a final division $\frac{A}{B}$.

Another way of expressing the connection between rational approximations and

continued fractions is given by Wall.

Equivalence 1 *(Wall[63]) Given the rational function $H(z)$ written as the ratio of two polynomials, it is possible to find r_i's and s_i's such that:*

$$H(z) = \frac{a_{00}z^n + a_{01}z^{n-1} + \cdots + a_{0n}}{a_{11}z^{n-1} + a_{12}z^{n-2} + \cdots + a_{1n}} \equiv [r_1 z + s_1, r_2 z + s_2, \cdots, r_n z + s_n] \quad (1.13)$$

with all $a_{ij} \neq 0$, and $r_i \neq 0$.

Equivalence 1 shows the main advantage of continued fractions. The powers of z in the polynomial representation are reduced to linear terms for each continued fraction digit. It seems that if we could store and manipulate continued fractions within a computer system, we could more efficiently compute rational approximations. Ideally, the evaluation of the rational approximation on the left is transformed in a set of linear approximations which can be evaluated in parallel. This suggests that for rational approximations it is possible to make use of symmetries in the continued fraction space. In general, continued fraction algorithms take a continued fraction input and compute a function storing the result again as a continued fraction. Before we take a look at continued fraction algorithms it is necessary to investigate how to represent continued fractions in computer systems.

Converting between binary numbers and continued fractions is non-trivial. In the worst case, conversion requires $O(N)$ divisions where N is the number of continued fraction digits. Using the above iteration equations 1.11 and 1.12 it is possible to reduce the conversion to $O(N)$ multiply-adds followed by one division. Clearly, this overhead for converting in and out of a number representation is to large.

In order to make continued fractions attractive for practical arithmetic units three fundamental problems need to be solved:

1. What is the optimal digit representation for simple continued fractions?

In other words, what is a digit of a continued fraction. How does it look like, and how many bits should we use to represent it? Is there an upper bound on the value of a digit? Mathematical literature focuses on regular continued fractions with integer digits. However, the value of a regular continued fraction digit has no upper bound, i.e. the value of the digit is between one and infinity. Clearly we need to restrict the maximal value of a digit. In fact, we need to restrict the range of a digit to a small set of values compared to the number of values of the number that we want to represent with the digits. The upper bound on a regular continued fraction digit has the side-effect that we can not represent *all* rational numbers, even if we use an infinite number of digits. However, all we want to do in computer arithmetic is to represent a finite number of values within one stored number. The problem is to choose the optimal set of values for the CF digits in order to represent the finite number of values of the stored binary number.

2. How to convert between continued fractions and binary numbers?

   This problem is related to the first one. Once we choose a digit representation, how do we convert binary digits to CF digits and vice versa without unreasonable effort?

3. How to control the errors during the computation with continued fractions?

   Specifically, given a continued fraction with n digits, can we give a bound on the error of the stored number compared to the actual value? For example, $0.\overline{3} = [0;3]$ is exact. $0.\overline{6} = [0;1,2]$ is also exact. But how close is the substring $[0;1,2]$ to 0.7, i.e. what is the error of the continued fraction after the first two digits. What happens if the digits are not integers?

This thesis solves all three problems of continued fraction arithmetic and shows

how to build practical and efficient rational arithmetic units for extending floating point division to rational approximation.

For more information on continued fractions see the CF literature, e.g. [12][17][63][40]. For a historical discussion on continued fractions and their development towards a practical tool for computer arithmetic units see Appendix B.

1.4 Organization of this Thesis

The following chapters are organized as follows. Chapter 2 investigates the possibilities for representing digits within a continued fraction environment. Chapter 3 introduces the M-log-Fraction Transform (MFT) discussing the optimal digit representation for continued fractions. The MFT enables efficient rational arithmetic shown in chapters 4 and 5.

The appendices present some related issues. Appendix A details work on the precision of regular continued fraction arithmetic which which is the initial work that led to the MFT and the results of this thesis. Appendix B summarizes some of the relevant historical background of continued fractions and, especially, continued fraction arithmetic.

Chapter 2

Integer versus Logarithmic Digits for Simple Continued Fractions

In this chapter we investigate the tradeoffs in using integer digits versus logarithmic digits for simple continued fractions. Integer digits are similar to digits of regular continued fractions. However, regular continued fractions defined for mathematical purposes assume digits $a_i \in \mathcal{N}^+$. In computer systems we generally have to choose a fixed number of bits to represent a digit. In case of digit overflow it is possible to concatenate digits to form larger digits by inserting a zero in-between:

$$[\ldots, 20, \ldots] = [\ldots, 10, 0, 10, \ldots] \tag{2.1}$$

How many bits should we choose for representing integer digits? Given regular continued fractions as a representation for fractional numbers, what is the average information content of a regular continued fraction digit, or stated differently, how many regular continued fraction digits are necessary *on average* to represent a decimal number with n decimal digits ? The theorem of Lochs gives information relating decimal numbers to regular CFs.

Theorem 1 *(Lochs [66]) For almost all irrational numbers x and their approximation in the form of a regular continued fraction:*

$$\lim_{n\to\infty} \frac{k_n(x)}{n} \simeq 0.9702.$$

with $k_n(x)$, the number of partial quotients, and n, the number of approximated decimal digits.

In other words, the precision of a regular continued fraction with n partial quotients corresponds roughly to the precision of a decimal number with n decimal digits. Thus, the information content of one partial quotient is on average constant and comparable to one decimal digit, or $3-4$ binary digits.

Given a fixed number of bits per digit, *logarithmic digits* enable the representation of larger values for digits, at the expense of coverage: Given stored digits $a_0, a_1, a_2, a_3, \ldots$, in the logarithmic representation the simple continued fraction is

$$[a_0; 2^{a_1}, 2^{a_2}, 2^{a_3}, \ldots] \tag{2.2}$$

In order to understand the tradeoff between logarithmic and integer digits, we need to understand the interaction between number of bits of the binary number, the number of bits per CF digit, and the maximal number of CF digits required to represent all values of the binary number. These three parameters create a 3-dimensional design space. Figure 2.1 shows two sample cuts through this design space.

The graphs in figure 2.1 compare two alternatives for representing the digits of a simple continued fraction. *INT* denotes representing each digit (partial quotient) by an integer value (a_i) with a given number of bits, while *LOG* denotes the graph for logarithmic CF digits, i.e. digit $= 2^{a_i}$. The two representations result in simple

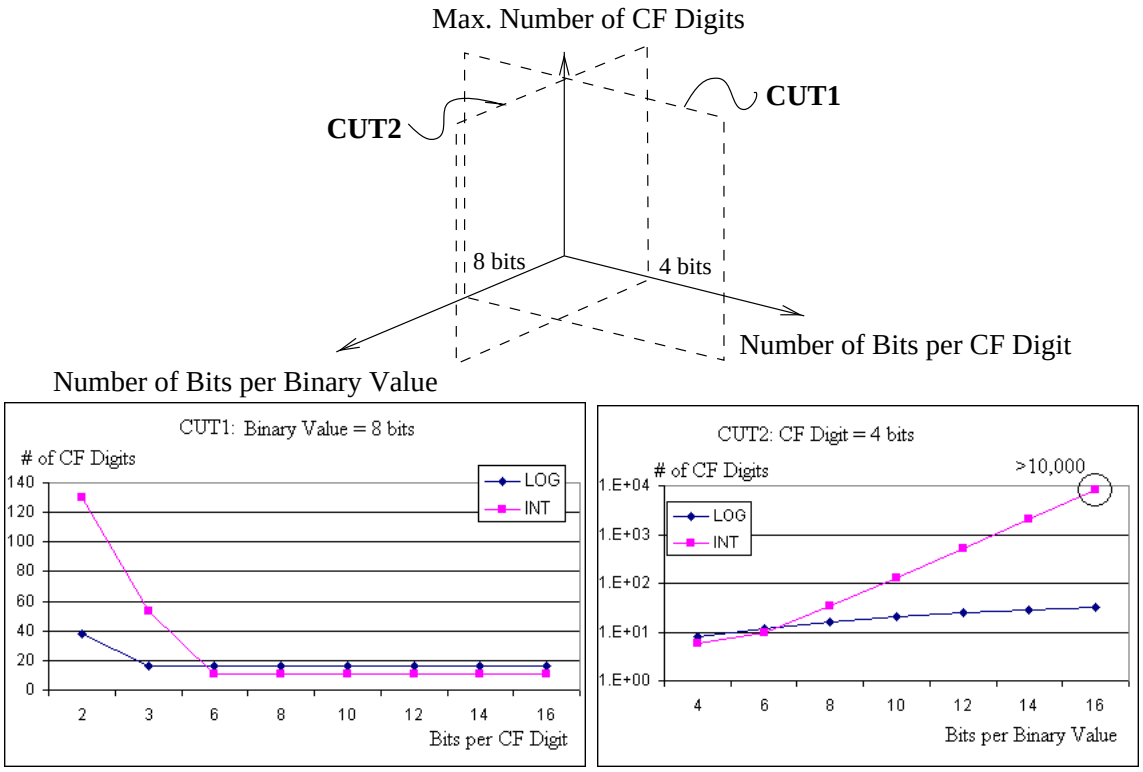


Figure 2.1: *The top figure shows the 3-dimensional space of bits per binary value, bits per CF digit and maximal number of digits required to represent the binary value. INT stands for integer representation of CF digits and LOG stands for logarithmic representation of CF digits. The bottom two graphs show two cuts through the 3-dimensional space above.*

continued fractions $[a_0; a_1, a_2, a_3, \ldots]$ and $[a_0; 2^{a_1}, 2^{a_2}, 2^{a_3}, \ldots]$ respectively.

CUT1 keeps the overall precision (bits per binary number) constant at 8 bits. For CF digit sizes close to or larger than the overall precision, integer CF digits result in fewer digits. However, it does not make sense to use a CF digit representation where the CF digits are close in size to the actual value that we want to represent. For CF digits significantly smaller than 8 bits, the logarithmic representation results in less digits.

CUT2 shows that for integer digits the number of digits in the worst case grows

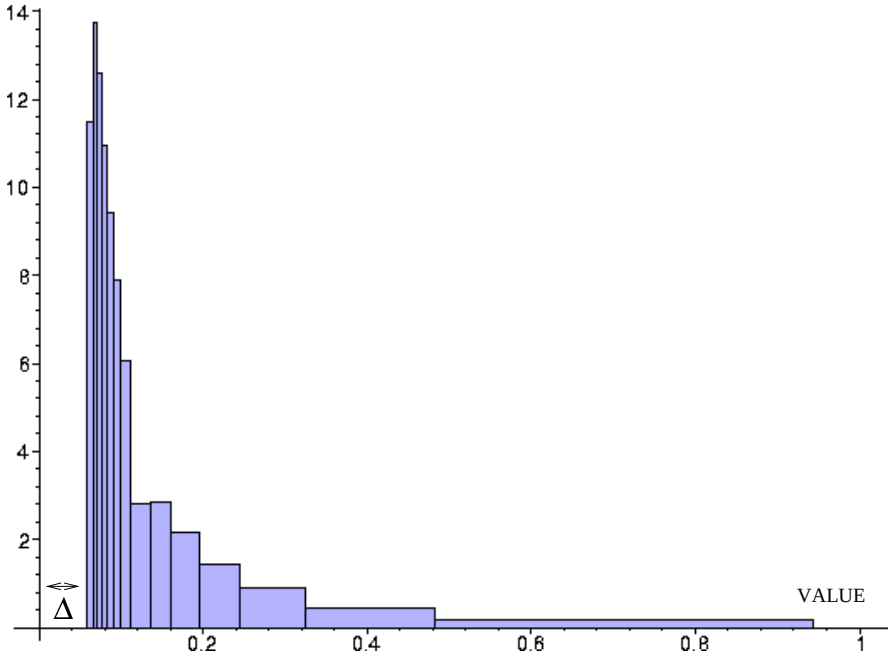


Figure 2.2: *The histogram shows the non-uniform distribution of the values of continued fractions with integer CF digits.*

exponentially so that for 16-bit binary numbers we need in the worst case more than 10,000! CF digits with 4 bits each. For the same 16-bit values less than 100 logarithmic digits are necessary. The reason for the enormous number of integer CF digits lies in the worst case. Let's take a look at the smallest 16-bit value larger than zero: $2^{-16} = [0; 2^{16}]$. In the case of integer CF digits, the largest digit value is 15 and $2^{-16} = [0; 2^{16}] = [0; 2^{15}, 0, 2^{15}] = [0; 15, 0, 15, 0, 15, \ldots]$ resulting in about 10,000 digits. In the case of logarithmic digits it is enough to simply represent the power of two, i.e. 16.

Another problem with continued fractions is error control. For conventional binary numbers the error is bounded by the position p of the least-significant bit, i.e. for a fractional number, $error < 2^{-p}$. The reason why this simple error bound works is the uniform distribution of values of conventional binary numbers. If we look at the

values of binary numbers, two neighboring values always have the same distance 2^{-p}. In general, and especially for continued fractions, this is not true. If the distribution is non-uniform, such as in the case of integer CF digits (INT), the error bound does not only depend on the least-significant digit, but also on the value itself. Therefore, the error of a computation does not only depend on the number of computed digits, but also on the value itself.

Rational representations such as continued fractions sometimes lead to a non-uniform distribution of values. In our case, continued fractions representing numbers between 0.0 and 1.0 with integer CF digits (INT) result in many values close to $0+\Delta$ and very few values close to 1.0. The non-uniform distribution of values representable by continued fractions with integer digits is shown in figure 2.2. Logarithmic CF-digits also result in a very similar non-uniform distribution. However, the M-log-Fraction proposed in the next chapter is a variation of the logarithmic CF-digits with a uniform distribution of values. The M-log-Fraction combines the advantages of logarithmic CF digits with the uniform distribution of values needed to control the error during computation, and thus the M-log-Fraction solves all three problems previously associated with the continued fraction representation.

Chapter 3

The M-log-Fraction Transformation (MFT)

Practical arithmetic units based on continued fractions rest on the resolution of three fundamental questions:

1. What is the optimal digit representation for simple continued fractions?

2. How to convert between continued fractions and binary numbers?

3. How to control the error during the computation with continued fractions?

Ideally we are looking for the following three features of the digit-representation: First, we need a reasonably compact digit representation with a small number of bits per digit compared to the number of bits for the binary number. In addition, a few such CF digits should suffice to represent the binary number. Second, conversion between binary numbers and continued fractions has to be very simple and efficient. Third, the values represented by the continued fractions have to be distributed uniformly to enable us to bound and thus control the error during computation,

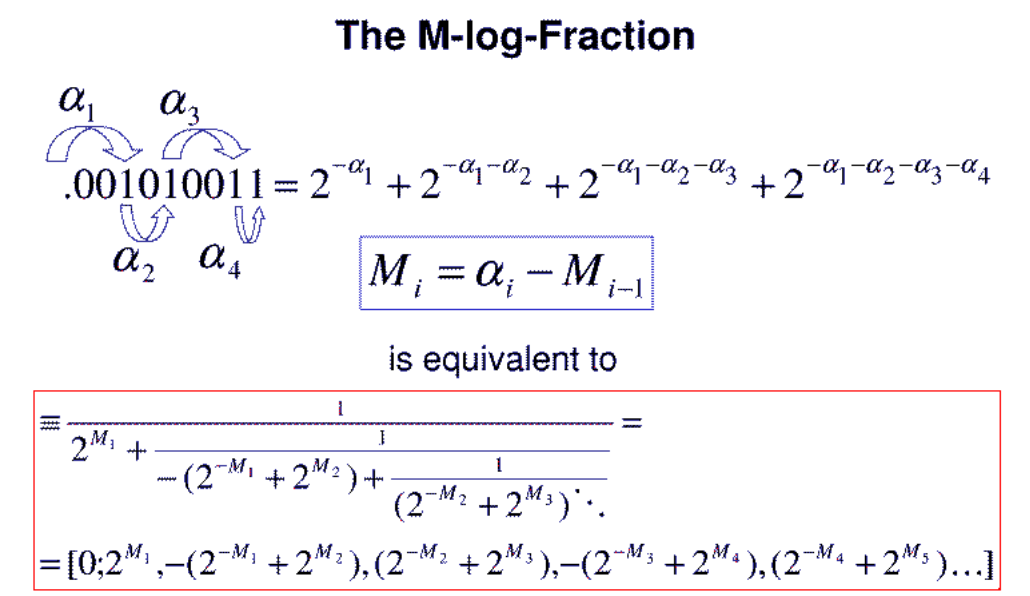


Figure 3.1: *The figure shows the connection between the distances between the '1's (α's) and the M-log-Fraction.* $M_1 = \alpha_1$.

Logarithmic CF digits defined in the previous chapter solve the first problem. However, conversion and error control are still significant hurdles even with logarithmic CF digits. This chapter provides the theoretical treatment of the M-log-Fraction Transform(MFT) for rational arithmetic units, and shows how the MFT implicitly combines all three necessary features outlined above.

Theorem 2 *(MFT Theorem) A binary number B with p binary digits, containing n '1's is equivalent to a simple continued fraction with n partial quotients: the M-log-Fraction $< M_1, M_2, M_3, \ldots >$ where if*

$$\begin{aligned} B &= b_1 b_2 b_3 \cdots b_p = 2^{\beta_1} + 2^{\beta_2} + 2^{\beta_3} + 2^{\beta_4} + 2^{\beta_5} + \ldots + 2^{\beta_n} \qquad \texttt{then} \\ B &\equiv [0; 2^{M_1}, -(2^{-M_1} + 2^{M_2}), (2^{-M_2} + 2^{M_3}), -(2^{-M_3} + 2^{M_4}), (2^{-M_4} + 2^{M_5}) \\ &\quad \ldots \pm (2^{-M_{n-1}} + 2^{M_n})] \\ &\equiv < M_1, M_2, M_3, M_4, M_5, \ldots, M_n > \end{aligned} \tag{3.1}$$

where M_i are related to α_i, the distances between the '1's of the binary number B, by the recursion:

$$M_1 = \alpha_1 = -\beta_1, \quad M_2 = \alpha_2 - M_1, \quad M_i = \alpha_i - M_{i-1} \tag{3.2}$$

The α's, β's, and M's are maximally N-bit integers where N is the number of bits in the binary number B. As shown, α_i is the sum of M_i and M_{i-1}. α's, the distances between the '1's, can also be seen as a 0-runlength encoding of the binary number B; i.e. α_i is the number of '0's between the i^{th} and $(i+1)^{th}$ '1'. Theorem 2 in one sentence is: "An M-log-Fraction of length n is equivalent to n powers of 2 for any $n \in \mathcal{N}$."

Figure 3.1 shows the connection between binary numbers and the M-log-Fraction. Before the proof, let us look at some implications of the MFT. The *M-log-Fraction* is a special case of a logarithmic simple continued fraction. First, given an N bit binary number, the MFT representation requires at most N digits with $\log(N)$ bits per CF digit. Thus, $N\log(N)$ bits are used to store the binary number in continued fraction (MFT) form. Second, the MFT enables instant conversion between simple continued fractions and binary numbers. All that is required to convert between binary numbers and an M-log-Fraction is a modified "Leading One Detect" circuit similar to the one used in floating point units to normalize the mantissa to the required format: $1.xxxx\ldots$. Conversion, i.e. a leading one detect circuit, takes one to two clock cycles. Being the equivalent of a binary number, error bounds for the M-log-Fraction are equal to error bounds of binary numbers. Thus, an N digit M-log-Fraction has the same error bound as an N digit binary number.

Figure 3.1 shows the correspondence of the distances between '1's (0-runlength encoding) called α's, and the digits of the M-log-Fraction. 0-runlength encoding refers to counting the number of zeros between the ones of a binary numbers, i.e. the runlength of the strings of zeros. Each α of the binary number is the sum of two consecutive M_i's of the M-log-Fraction.

The full proof of Theorem 2 (MFT Theorem) by induction follows.

3.1 Proof of Theorem 2 (MFT Theorem)

Please note the following *notation*, making the following proofs more readable.

$$\overline{M_i} = \sum_{j=1}^{i} M_j \tag{3.3}$$

$$< M_1, M_2, M_3, \ldots, M_i > \text{ denotes the M-log-Fraction with parameters } \{M_j\}. \tag{3.4}$$

Before starting the proof of Theorem 2 it is necessary to review a few fundamental equations from continued fraction theory. We know from the above introduction to continued fraction theory that a simple continued fraction can be written as a ratio $[x_0, x_1, x_2, x_3, \ldots] = \frac{A_i}{B_i}$. Simplifying the general equations 1.11,1.12 we obtain the following iteration equations for simple continued fractions:

$$A_i = x_i A_{i-1} + A_{i-2} \tag{3.5}$$

$$B_i = x_i B_{i-1} + B_{i-2} \tag{3.6}$$

Initial conditions are $A_0 = a_0$, $B_0 = 1$, $A_{-1} = 1$, and $B_{-1} = 0$. In our case, the digits of the simple continued fraction x_i are defined by the binary M-log-Fraction (equation 3.1).

$$x_i = (-1)^i (2^{M_{i+1}} + 2^{-M_i}) \tag{3.7}$$

The short version of Theorem 2 is: "An M-log-Fraction of length n is equivalent to n powers of 2 for any $n \in \mathcal{N}$." We prove "is equivalent to" with one induction for each direction. First, given an M-log-Fraction with N MFT-digits we prove by induction the correspondence to a binary number with N powers of 2 (one's). Second, in direction 2 of the proof we start with a binary number with N powers of 2 and show by induction the correspondence to an M-log-Fraction with N digits.

Direction 1: Binary M-log-Fraction of length $n \rightarrow n$ powers of 2.

The proof uses induction on the length of the M-log-Fraction, so for any $n \in \mathcal{N}$, we must show that (remember the definition of $\overline{M_i}$ above)

$$<M_1, M_2, M_3, M_4, \ldots M_n> \rightarrow 2^{-M_1} + 2^{-2M_1 - M_2} + 2^{-2M_1 - 2M_2 - M_3} + \cdots + 2^{-2\overline{M_{n-1}} - M_n}$$

$$\rightarrow 2^{\beta_1} + 2^{\beta_2} + 2^{\beta_3} + \cdots + 2^{\beta_n} \tag{3.8}$$

First, M-log-Fractions of length $n = 1$ and $n = 2$ reduce to a binary number:

$$< M_1 >= 2^{-M_1} \qquad < M_1, M_2 >= 2^{-M_1} + 2^{-2M_1 - M_2} \tag{3.9}$$

Assuming that direction 1 of Theorem 2 holds for M-log-Fractions of length n:

$$< M_1, M_2, \ldots, M_n >= 2^{-M_1} + 2^{-2M_1 - M_2} + \cdots + 2^{-2\overline{M_{n-1}} - M_n} \tag{3.10}$$

We want to show that Theorem 2 holds for $n + 1$ '1's. This means we have to show that for M-log-Fractions of length $n + 1$:

$$< M_1, M_2, \ldots, M_{n+1} >=< M_1, M_2, \ldots, M_n > +2^{-2\overline{M_n} - M_{n+1}} \tag{3.11}$$

To simplify the previous equation we use an equation given by Thron ([17], equation 2.1.9):

$$A_{i+1}B_i - A_iB_{i+1} = (-1)^i \tag{3.12}$$

Using this (3.12) and rewriting equations 3.4,3.5 and 3.6 we obtain,

$$< M_1, M_2, \ldots, M_{n+1} > - < M_1, M_2, \ldots, M_n > \; = \; \frac{A_{n+1}}{B_{n+1}} - \frac{A_n}{B_n} = \tag{3.13}$$

$$= (-1)^n \frac{1}{B_{n+1}B_n} \tag{3.14}$$

The last step of direction 1 of the proof is to apply the following lemma:

Lemma 1 *The following equation holds for binary M-log-Fractions:*

$$\boxed{B_i = (-1)^{\lfloor 0.5 \cdot i \rfloor} \cdot 2^{\overline{M_i}}} \tag{3.15}$$

Lemma 1 is almost a direct consequence of equation 3.6 and the digits of the M-log-Fraction. In fact, lemma 1 shows how the digits of the M-log-Fraction cancel out and lead to a single power of two for B. Let us assume the correctness of lemma 1 and delay its proof until after the current proof of the MFT theorem. Applying lemma 1 to equation 3.14 we obtain

$$(-1)^n \frac{1}{B_{n+1} B_n} = 2^{\overline{M_n}} \cdot 2^{\overline{M_{n+1}}} = 2^{-2\overline{M_n} - M_{n+1}} \tag{3.16}$$

concluding direction 1 of the proof.

Direction 2: n powers of $2 \rightarrow$ Binary M-log-Fraction of length n, for any $n \in \mathcal{N}$, we must show that

$$2^{\beta_1} + 2^{\beta_2} + 2^{\beta_3} + \cdots + 2^{\beta_n} \rightarrow 2^{-M_1} + 2^{-2M_1 - M_2} + 2^{-2M_1 - 2M_2 - M_3} + \cdots + 2^{-2\overline{M_{n-1}} - M_n}$$
$$\rightarrow < M_1, M_2, M_3, \ldots M_n > \tag{3.17}$$

First, for length $n = 1$ and $n = 2$, we obtain by inspection:

$$2^{\beta_1} = < -\beta_1 > \qquad 2^{\beta_1} + 2^{\beta_2} = < -\beta_1, 2\beta_1 - \beta_2 > \tag{3.18}$$

Assuming that direction 2 of Theorem 2 holds for binary numbers with n powers of 2:

$$2^{\beta_1} + 2^{\beta_2} + \cdots + 2^{\beta_n} = < M_1, M_2, \ldots, M_n > \tag{3.19}$$

we want to show that

$$2^{\beta_1} + 2^{\beta_2} + \cdots + 2^{\beta_{n+1}} = < M_1, M_2, \ldots, M_{n+1} > \tag{3.20}$$

Combining lemma 1 with equations 3.12 from direction 1 ,3.19 above, and $\beta_{n+1} = -2\overline{M_n} - M_{n+1}$, we can conclude that

$$< M_1, M_2, \ldots, M_{n+1} > \; - \; < M_1, M_2, \ldots, M_n > = \frac{A_{n+1}}{B_{n+1}} - \frac{A_n}{B_n} = \tag{3.21}$$

$$= \; 2^{-2\overline{M_n} - M_{n+1}} = 2^{\beta_{n+1}} \tag{3.22}$$

concluding direction 2 of the proof.

Thus, an M-log-Fraction of length n is equivalent to n powers of 2, with the relation between M's and the powers of 2 as shown in equation 3.8. *q.e.d.*

In order to complete the proof we now need to prove lemma 1.

Lemma 1 The following equation holds for binary M-log-Fractions:

$$B_i = (-1)^{\lfloor 0.5 \cdot i \rfloor} \cdot 2^{\overline{M_i}} \tag{3.23}$$

Proof of lemma 1 by induction

For $i = 1$ and $i = 2$ lemma 1 follows by inspection using the initial conditions for iteration equation 3.6:

$$B_1 = x_1 = 2^{M_1} \qquad B_2 = B_1 x_2 + B_0 = -2^{M_1 + M_2} \tag{3.24}$$

The equation for $i+1$ follows from equation 3.6:

$$B_{i+1} = (-1)^i(2^{M_{i+1}} + 2^{-M_i})B_i + B_{i-1} \tag{3.25}$$

Now let us assume that lemma 1 holds for i and $i-1$. Applying lemma 1 for B_i and B_{i-1} results in the equation for B_{i+1}:

$$B_{i+1} = (-1)^i\left((-1)^{\lfloor 0.5\cdot i\rfloor}2^{\overline{M_{i+1}}} + (-1)^{\lfloor 0.5\cdot i\rfloor}2^{\overline{M_{i-1}}}\right) + (-1)^{\lfloor 0.5(i-1)\rfloor}2^{\overline{M_{i-1}}} = \tag{3.26}$$

$$= (-1)^{\lfloor 0.5(i+1)\rfloor}\cdot 2^{\overline{M_{i+1}}} \tag{3.27}$$

q.e.d.

This concludes the proof of the equivalence of M-log Fractions and binary numbers (Theorem 2).

Another way of understanding the principles behind the MFT is to look at the following equivalence of series and CFs.

Equivalence 2 *of series and continued fractions: for $c_i \neq 0$,(Euler[60])*

$$c_0 + c_1 + c_2 + \cdots \equiv c_0 + \frac{c_1|}{|1} - \frac{\frac{c_2}{c_1}\,|}{|1+\frac{c_2}{c_1}} - \frac{\frac{c_3}{c_2}\,|}{|1+\frac{c_3}{c_2}} - \cdots \tag{3.28}$$

Each partial quotient of the CF on the right depends on two neighboring terms of the series on the left. Thus, the precision of an n element CF is equivalent to the precision of a finite series of length n. Looking at a binary number as a sum of weighted digits we obtain the equivalence of binary numbers and continued fractions.

The basic *M-log-Fraction Transform (MFT)* enables the encoding of binary numbers with a sequence of M_i's as shown above. In fact, the first digit M_1 corresponds to the integer part of the logarithmic representation of B, i.e. $M_1 = \lfloor \log(B) \rfloor$. Thus,

theoretically the MFT is a variant of the logarithmic number system[44] with the advantage that conversion just requires counting distances between the '1's.

Although the M-log-Fraction shown above solves the three problems associated with continued fractions, there is a small drawback left. A continued fraction digit corresponds roughly to one '1' in the binary number. Given a target precision of 16 bits it is not clear a priori how many iteration are required to compute a satisfactory result. In the worst case with "all '1's" 16 iterations are required. In order to enable a regular hardware structure for the rational arithmetic unit, we introduce the *signed-digit M-log-Fraction.*

3.2 The Signed-Digit M-log-Fraction

This section shows a modified version of the MFT, using signed-digit binary numbers. Our objective is to is to use the digits of a binary number $B = s_1 2^{-1} + s_2 2^{-2} + s_3 2^{-3} + s_4 2^{-4} + \ldots + s_n 2^{-n}$ directly to convert B to the M-log-Fraction. In the binary case $s_i = \{0, 1\}$. The main problem to direct conversion stems from the restriction on Euler's equivalence 3.28, in our case $s_i \neq 0$. In order to circumvent this restriction, we use signed-digit binary numbers. In case of radix-2 signed-digits are in $\{+1, -1\}$.

A drawback of the general M-log-Fraction is that a CF digit corresponds to a '1' in the binary number. Therefore the result does not converge uniformly and requires a variable number of iterations per result. In fact, in the worst case (all ones) the algorithm is limited to retiring one bit at a time. Signed-digit binary numbers eliminate the non-uniform convergence and enable us to improve the rate of convergence in the worst case. Applying the signed digits to the MFT leads to the Signed-Digit MFT.

The *signed-digit binary M-log-Fraction* shown below connects signed-digit binary numbers (see also the introduction on representing numbers in computer systems in

section 1.2 of this thesis) with the MFT. In the weighted positional number system with numbers $B = s_1 2^{\beta_1} + s_2 2^{\beta_2} + s_3 2^{\beta_3} + \ldots + s_n 2^{\beta_n}$, β's are fixed to $\beta_i = -i$:

$$\{\beta_1, \beta_2, \beta_3, \ldots, \beta_p\} = \{-1, -2, -3, \ldots, \beta_p\} \tag{3.29}$$

Fixing the β's enables us to fix the shifts M_i according to Theorem 2. From the β's above we know that $\alpha_i = \beta_{i-1} - \beta_i = 1 = M_i + M_{i-1}$. The most efficient (yielding the smallest hardware implementation) set of M_i's with this restriction is $M_i = i \bmod 2$:

$$\{M_1, M_2, M_3, \ldots, M_n\} = \{1, 0, 1, 0, 1, \ldots, M_n\} \tag{3.30}$$

The resulting equation below is a corollary on Theorem 2:

Corollary 3 *Signed-Digit Binary M-log-Fraction: A signed-digit binary number B_R with n '1's (n, even), and $s_i \in \{+1, -1\}$ is equivalent to a simple continued fraction with n partial quotients as follows:*

$$\begin{aligned} B_R &= s_1 2^{-1} + s_2 2^{-2} + s_3 2^{-3} + s_4 2^{-4} + \ldots + s_n 2^{-n} \\ \equiv &[0; s_1 2^1, -(s_1 2^{-1} + s_2 2^0), (s_2 2^0 + s_3 2^1), -(s_3 2^{-1} + s_4 2^0), \ldots - (s_{n-1} 2^{-1} + s_n 2^0)] \end{aligned}$$

In order to convert binary numbers to signed-digit binary M-log-Fractions requires two steps. The first step is to convert the binary number to a signed-digit binary number. This can be done in about the time required for an addition. For details on converting between signed-digits and binary digits see [6]. Second, the signed-digit numbers are converted to the signed-digit M-log-Fraction as shown in Corollary 3 above. Converting between signed binary digits and continued fraction digits consists of combining two neighboring digits as shown in Corrolary 3. The choice of $M_i =$

$i \bmod 2$ keeps the powers of two in the set $\{2^{-1}, 2^0, 2^1\}$–basically a left shift, a right shift, or no shift. The actual implementation of rational arithmetic units shown in the next chapter absorbs this conversion operation into the iteration equations of the algebraic algorithm–thus eliminating conversion (MFT) overhead.

The next chapters show the implications of the MFT to continued fraction algorithms and the implementation of the first practical rational arithmetic units for general purpose processors based on continued fraction algorithms and the MFT.

Chapter 4

Algorithm Class 1:
A Rational Arithmetic Unit

The previous chapters showed how to represent digits of a continued fraction, eliminating the conversion overhead between binary numbers and continued fractions, and keeping the same level of error control as for binary numbers. Why do we want to use continued fractions in the first place? One of the main motivations for using continued fractions are the continued fraction algorithms shown below. Continued fractions are a natural representation for rational functions. However, until now continued fraction were impractical due to the large overhead of converting in and out of the representation and the non-uniform distribution of representable values. The MFT introduced in the previous chapter opens up all continued fraction algorithms for practical implementations in computer systems based on binary numbers.

The conventional method to compute rational approximations uses multiply-add structures to evaluate the numerator and denominator, followed by a final division. Each multiply-add step introduces roundoff error, accumulating error for the final result. The following rational arithmetic units compute entire rational functions in one arithmetic unit. Such arithmetic units are also called compound arithmetic units.

For example, instead of computing $\frac{ax+b}{cx+d}$ with 2 multiplies, 2 adds and a final divide, the algorithm proposed below starts with loading a, b, c, d into internal state registers, takes the first input digit of x and produces the first output digit of the final result after the first iteration. An application example for such extended division units is signal processing, where the bilinear function could be used to compute rational approximations, replacing the multiply-add units that are currently used to evaluate polynomial approximations in Digital Signal Processors (DSPs).

The goal of this chapter is to show the details of a rational arithmetic unit. The previous chapter showed us how to convert between binary numbers and continued fractions. The following continued fraction algorithms show the advantages of the continued fraction representation for rational arithmetic.

Continued fraction algorithms take simple continued fractions and compute a function returning a result in the form of a simple continued fraction. Assume we have a simple continued fraction $[x_0, x_1, x_2, \ldots]$ representing a number in our computer system and we want to compute a function $f(x)$. A continued fraction algorithm for $f(x)$ takes the digits x_i of the input fraction and produces the output digits o_i of the result.

$$[o_0, o_1, o_2 \ldots] = f\left([x_0, x_1, x_2, \ldots]\right) \tag{4.1}$$

The trivial rational function is the reciprocal. While for conventional number systems computing the reciprocal $f(x) = \frac{1}{x}$ is a complex operation, continued fractions are a natural representation for the reciprocal. Computing the reciprocal of a simple continued fractions consists of a single digit shift:

$$\frac{1}{[a_0; a_1, a_2, \ldots]} = \begin{cases} [0; a_0, a_1, a_2, \ldots] & \texttt{if } a_0 \neq 0 \\ [a_1; a_2, \ldots] & \texttt{if } a_0 = 0 \end{cases} \tag{4.2}$$

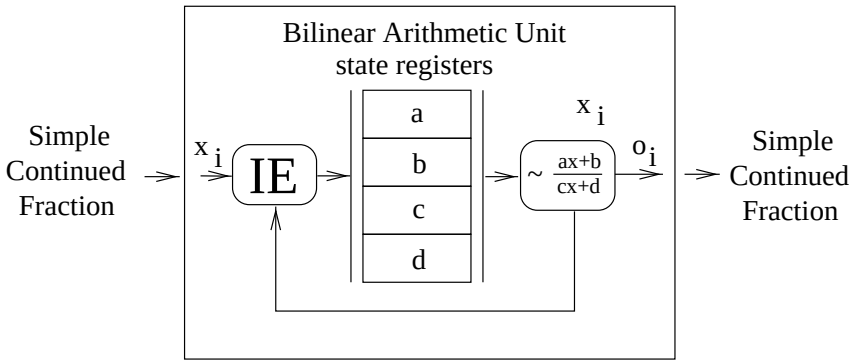


Figure 4.1: *The figure shows the state machine for the iteration equations (IE) of the positional algebraic continued fraction algorithm. The state-registers a, b, c, d hold at each iteration the values a_i, b_i, c_i, d_i. The algorithm selects the output digit o_i close to the current state $\frac{a_i x_i + b_i}{c_i x_i + d_i}$ shown without the indices in the figure above.*

A simple continued fraction multiplied by a constant c becomes:

$$[c \cdot a_0; \frac{a_1}{c}, c \cdot a_2, \frac{a_3}{c}, c \cdot a_4, \frac{a_5}{c}, c \cdot a_6, \ldots] \tag{4.3}$$

Next, let's examine the bilinear function $f(x) = \frac{ax+b}{cx+d}$ which is the simplest non-trivial rational function. In order to compute the bilinear function we use the algebraic algorithm proposed by Gosper[15].

At each iteration the algorithm consumes an input digit or produces an output digit. The amount of state kept by the algorithm stays constant and corresponds to the numbers of coefficients of the computed rational function. In the case of the bilinear function there are four state registers corresponding to the four coefficients a, b, c, and d.

The objective is to compute $[o_0, o_1, o_2 \ldots] = f([x_0, x_1, x_2, \ldots])$ where x_i's are the partial quotients of the simple input CF, and $o_i = f(x_i, \texttt{state})$ are the partial quotients of the output. $T(x)$ is the function that is stored in the state variables. The

general structure of the algebraic algorithm is shown in figure 4.1. The iteration equations (IE) transform the state from i to $i+1$ consuming an input digit and producing an output digit (Figure 4.2).

How do we obtain the iteration equations? At any given iteration the state of the entire computation is given by the state-registers, the remaining input digits and the produced output digits. The internal state changes on two events: consuming one input digit and producing one output digit. Continued fractions are add-divide structures. Thus, consuming an input digit amounts to an add-divide operation on the argument of the function that we want to compute. Producing an output digit amounts to an add-divide operation on the output of the function that we want to compute. The internal state represents the function that we want to compute. As a consequence, the transformation of the internal state can be described mathematically with the following transformations:

- to *consume* an input quotient x_i
  apply $T'(x) = T(x_i + \frac{1}{x})$.

- to *produce* an output quotient o_i
  apply $T''(x) = \frac{1}{T(x)-o_i}$.

We will use T' and T'' to derive the iteration equations below. $T(x)$ stands for the function that we want to compute–represented by the internal state registers. In the bilinear case $T(x) = \frac{ax+b}{cx+d}$ with state registers a, b, c, d. The number of coefficients and form of $T(x)$ stay constant across all iterations, thus leading to very efficient hardware structures. Why does the number of coefficients stay constant? The algorithm uses the fact that the homographic function is invariant over the basic CF transformations $T'(x)$ and $T''(x)$. In fact, there are also rational functions of higher degree that satisfy this requirement. This means that we can build arithmetic units for a whole family of functions. In this thesis we focus on rational functions.

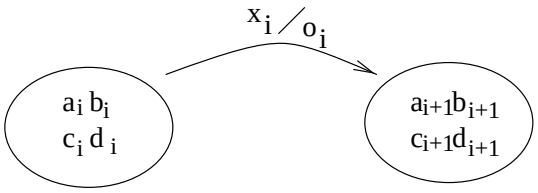


Figure 4.2: *The figure shows a slice of the transition diagram for the positional algebraic continued fraction algorithm. The iteration values a_i, b_i, c_i, d_i go over into the next state $a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1}$ absorbing an input digit x_i and producing an output digit o_i.*

In general, the algorithm can independently consumes an input quotient, or produces an output quotient at each iteration. However, ensuring that quotients are consumed and produced optimally requires the computation of a large error term which increases the overall computation time of the algebraic algorithm by an "order of magnitude"[45].

In order to avoid computing the error term in each iteration to determine if the algorithm should consume or produce a digit, Vuillemin[45] shows that in most common cases we can consume one input and produces one output at each iteration– the *positional algebraic algorithm.* Consuming and producing a digit at each iteration makes the computation more regular and eliminates the need to compute the error term at each iteration. All arithmetic units discussed in this thesis use the positional algebraic algorithm.

The following section presents three examples of rational functions, their iteration equations and corresponding output selection functions. These iteration equations can then be combined with the MFT digits resulting in the design of regular and efficient hardware arithmetic units for computing the rational functions. The following sections show the derivation of the iteration equations from figure 4.1.

Linear Fractional Transformation

Linear fractional transformations are also called homographic, or bilinear functions: $T_1(x) = \frac{ax+b}{cx+d}$. Given an input CF digit x_i and the current state a, b, c and d, the algorithm chooses[1] an output digit at each iteration $o_i \sim \frac{ax_i+b}{cx_i+d}$. In fact, it suffices to choose the next output digit o_i "close" to the real remainder $T(x) = \frac{ax_i+b}{cx_i+d}$ based on the current state a, b, c, and d. In the original algorithm with integer CF digits, choosing o_i "close" to the real remainder becomes a rounding operation.

Now let us derive the iteration equations for the bilinear transformation. The variable x consists of digits $[x_0; x_1, x_2, \ldots]$. The four coefficients determine the initial values of the four state registers a, b, c, d. The iteration equations for consuming one input digit x_i and producing one output digit o_i follow from applying T'(x) and T"(x) to the bilinear transform. $a_{i+1}, b_{i+1}, c_{i+1}, d_{i+1}$ denote the next state as a function of the previous state. The digit x_i is the current input digit and $o_i \sim \frac{a_i x_i+b_i}{c_i x_i+d_i}$ is the chosen output digit:

$$\texttt{next}-\texttt{state}\quad T_{i+1}(x) = \frac{a_{i+1}x + b_{i+1}}{c_{i+1}x + d_{i+1}} = \frac{1}{\frac{a_i(x_i+\frac{1}{x})+b_i}{c_i(x_i+\frac{1}{x})+d_i} - o_i} \tag{4.4}$$

After simplifying equation 4.4 to a bilinear form in x, the state iteration equations are obtained as the new coefficients:

$$\begin{aligned} \mathtt{a_{i+1}} &= \mathtt{c_i x_i + d_i} & \mathtt{b_{i+1}} &= \mathtt{c_i} \\ \mathtt{c_{i+1}} &= \mathtt{a_i x_i + b_i - o_i(c_i x_i + d_i)} & \mathtt{d_{i+1}} &= \mathtt{a_i - o_i c_i} \end{aligned} \tag{4.5}$$

The output digit is chosen in order to force c to zero. In the bilinear case, the state can be represented by the matrix $\underline{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$. The iteration equations above have the property that $|\underline{A}| = \text{constant}$[45]. As a consequence, forcing c to zero, forces

[1] "choose" refers to the fact that the algorithm can choose any output digit. The iteration equations adapt the state according to the chosen output quotient.

b to zero, d to one, and a to $|\underline{A}|$.

Quadratic Transformations

In the quadratic case the algorithm computes the transformation $T_2(x) = \frac{ax^2+bx+c}{dx^2+ex+f}$, with x as the current input digit and o as the current output digit (indices i are omitted for simplicity). The algorithm first chooses $o \sim \frac{ax^2+bx+c}{dx^2+ex+f}$, and afterwards updates the state registers as follows:

$$\begin{aligned} \mathtt{a}_{i+1} &= \mathtt{dx}^2 + \mathtt{ex} + \mathtt{f} & \mathtt{d}_{i+1} &= \mathtt{ax}^2 + \mathtt{bx} + \mathtt{c} - \mathtt{oa}_{i+1} \\ \mathtt{b}_{i+1} &= 2\mathtt{dx} + \mathtt{e} & \mathtt{e}_{i+1} &= \mathtt{b} + 2\mathtt{ax} - \mathtt{ob}_{i+1} \\ \mathtt{c}_{i+1} &= \mathtt{d} & \mathtt{f}_{i+1} &= \mathtt{a} - \mathtt{oc}_{i+1} \end{aligned} \tag{4.6}$$

In the quadratic case with two input variables x, y, the algorithm computes the transformation $T_3(x, y) = \frac{axy+bx+cy+d}{exy+fx+gy+h}$ where x,y are inputs to the following iteration equations for the state registers. For input digits x,y, and corresponding output digit $o \sim \frac{axy+bx+cy+d}{exy+fx+gy+h}$ the iteration equations are:

$$\begin{aligned} \mathtt{a}_{i+1} &= \mathtt{exy} + \mathtt{fx} + \mathtt{gy} + \mathtt{h} & \mathtt{e}_{i+1} &= \mathtt{axy} + \mathtt{bx} + \mathtt{cy} + \mathtt{d} - \mathtt{oa}_{i+1} \\ \mathtt{b}_{i+1} &= \mathtt{ex} + \mathtt{g} & \mathtt{f}_{i+1} &= \mathtt{ax} + \mathtt{c} - \mathtt{ob}_{i+1} \\ \mathtt{c}_{i+1} &= \mathtt{ey} + \mathtt{f} & \mathtt{g}_{i+1} &= \mathtt{ay} + \mathtt{b} - \mathtt{oc}_{i+1} \\ \mathtt{d}_{i+1} &= \mathtt{e} & \mathtt{h}_{i+1} &= \mathtt{a} - \mathtt{od}_{i+1} \end{aligned} \tag{4.7}$$

Quadratic units have more state variables than the bilinear unit. In addition, the iterations per variable require more operations. However, due to the available parallelism in the equations, minimal latency per iteration does not increase significantly.

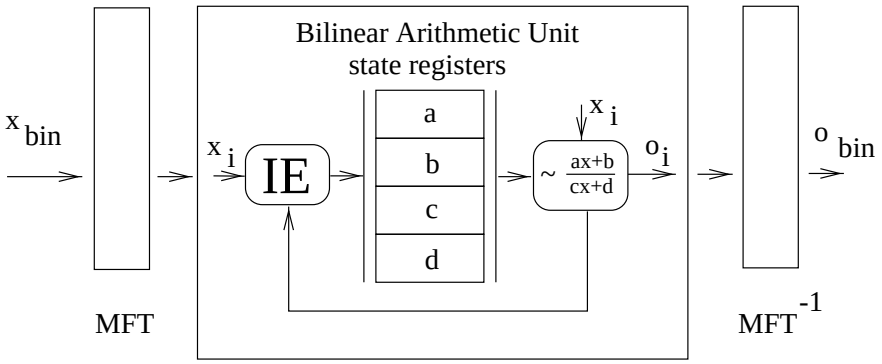


Figure 4.3: *The figure shows the state machine for the iteration equations (IE) of a bilinear arithmetic unit using the positional algorithm. The input digits enter the algorithm in the form of MFT digits. The algorithm selects the output digit o_i to fit the MFT digit value closest to the current state $\frac{ax+b}{cx+d}$.*

4.1 A 'shift and add' based rational arithmetic unit

Rational arithmetic units are candidate extensions for floating point division units. The following implementation focuses on the mantissa part of the floating point divider and shows the rational arithmetic unit for fixed-point rational numbers. This section describes the implementation of rational arithmetic units based on combining the iteration equations for algebraic algorithms developed above and the MFT introduced in the previous chapter.

The signed-digit binary M-log-Fraction (Corollary 3) combined with the positional algebraic algorithm results in the implementation specified below. Figure 4.3 shows the structure of the proposed arithmetic unit. We use the MFT to convert a binary number to the M-log-Fraction. The M-log-Fraction is fed into the positional algorithm. Finally, the inverse MFT converts the M-log-Fraction back to a binary number. More specifically, the input digit x_i and the output digit o_i are chosen according to the MFT. Rewriting the iteration equations with the particular x_i and

o_i leads to rational arithmetic units without any overhead for the MFT and inverse MFT. Next let us take a look at a detailed implementation of a bilinear arithmetic unit.

4.1.1 Implementing the Bilinear Function $\frac{ax+b}{cx+d}$

The bilinear (linear fractional) arithmetic unit (figure 4.3) computes output $o_{bin} = T(x_{bin}) = \frac{ax_{bin}+b}{cx_{bin}+d}$. The following steps detail the operations to compute $\frac{ax+b}{cx+d}$ given four binary coefficients a,b,c,d, and a *binary* input x_{bin}, producing a binary output o_{bin}:

1. Load state registers a, b, c, d with coefficient values. The coefficients are used as starting values a_0, b_0, c_0, d_0 for the state-registers

2. The input value x_{bin} is converted to a signed-digit representation with digits $t_i \in \{-1, 1\}$.

3. Combine neighboring digits t_i, t_{i-1} to form MFT digits according to the signed-digit MFT: for even digits : $x_{i=\texttt{even}} = -(t_i + \frac{t_{i-1}}{2})$, and for odd digits: $x_{i=\texttt{odd}} = (2t_i + t_{i-1})$, $t_0 = 0$.

4. For all CF digits x_i do

   (a) Compute the output CF digit $o_i \sim \frac{a_i x_i + b_i}{c_i x_i + d_i}$. Output digits o_i are restricted by the CF digit form required by the MFT. In fact, all we have to do is to choose the proper signed-digit s_i: for even digits : $o_{i=\texttt{even}} = -(s_i + \frac{s_{i-1}}{2})$, and for odd digits, $o_{i=\texttt{odd}} = (2s_i + s_{i-1})$, $s_0 = 0$. Using these identities we choose $s_i = \text{sign}(f(a, b, c, d, s_{i-1}))$ with $f(x)$ depending on the position i of the digit.

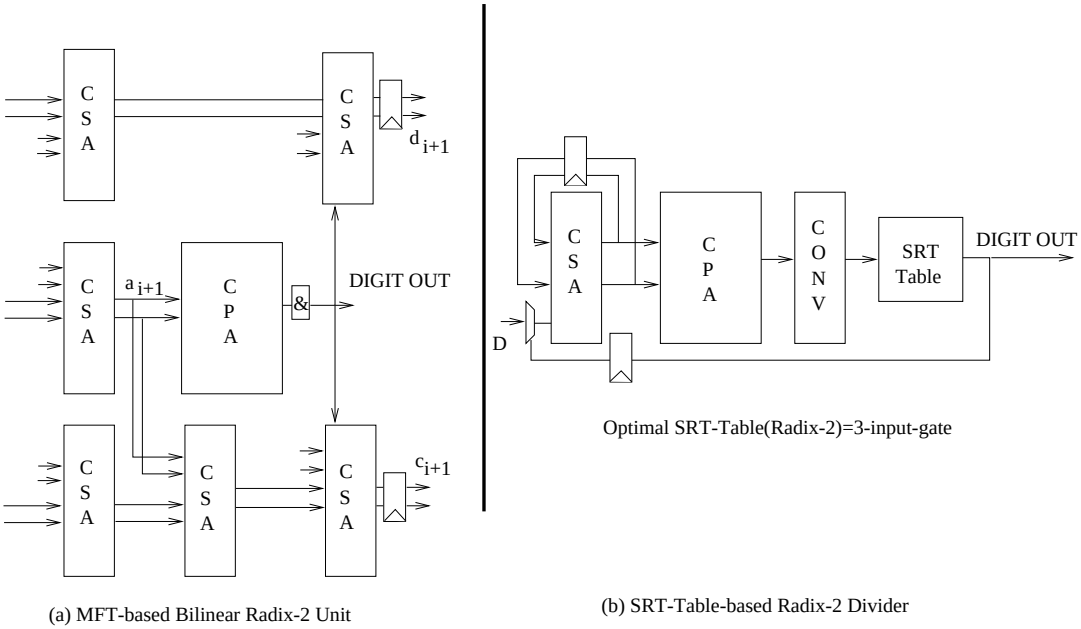


Figure 4.4: *The figure shows (a) a possible implementation of the iteration equations for a bilinear radix-2 unit based on the MFT, and (b) a general SRT-Table divider based on results from [41]. The sizes of the boxes indicate the relative, approximate VLSI-area relations. Note that* $b_{i+1} = c_i$ *and is therefore missing in the figure.*

(b) Update state variables with the iteration equations (IE) from equations 4.5:

$$\begin{aligned} \mathtt{a_{i+1}} &= \mathtt{c_i x_i + d_i} & \mathtt{b_{i+1}} &= \mathtt{c_i} \\ \mathtt{c_{i+1}} &= \mathtt{a_i x_i + b_i - o_i(c_i x_i + d_i)} & \mathtt{d_{i+1}} &= \mathtt{a_i - o_i c_i} \end{aligned} \tag{4.8}$$

5. Convert signed-digits s_i to the actual output value $o_{bin} = s_1 s_2 s_3 \ldots s_n = \frac{ax_{bin}+b}{cx_{bin}+d}$.

In the actual implementation the conversion between signed-digits and MFT digits can be absorbed into the iteration equations. To get a sense of the complexity of designing a rational arithmetic unit, figure 4.4 shows the micro-architecture of an iterative rational arithmetic unit. Carry-Save Adders (CSA) and Carry-Propagate Adders(CPA) are the building blocks of our design. For detailed explanation of Adder technology please refer to standard textbooks such as [3] or more advanced

texts on computer arithmetic [2][4][5][6]. CSA units mostly contribute to the area of a circuit and have very short delays: latency O(1). CPAs have significant delays due to the full propagation of the carry signal with latencies between $O(\log N)$ to $O(N)$ depending on the microarchitecture (N is the number of bits).

Figure 4.4 shows that the rational arithmetic unit for $T(x) = \frac{ax+b}{cx+d}$ has a delay, or cycle time, of O(CPA+ϵ), just as a regular divide unit. The additional functionality is bought with additional area for CSAs for a, d.

Similar structures for quadratic rational functions follow from iteration equations 4.6 and 4.7. Before comparing different rational arithmetic units we compare the efficiency of the proposed algorithm with state-of-the-art division.

4.2 An MFT-based Division Unit

Given the algorithm presented above, how do the rational arithmetic units introduced above compare to the state-of-the-art? As there is no directly competing arithmetic unit that computes any bilinear function[2], the next section examines the complexity of a division unit based on the bilinear unit. The division unit based on the MFT algorithm presented above allows us to compare the efficiency of the proposed method to state-of-the-art division.

The MFT-based division unit is a bilinear unit ($\frac{ax+b}{cx+d}$) with $a = c = 0$ and no input x. Thus, we compute the division $\frac{b}{d}$. Figure 4.5 shows the structural comparison between traditional division and MFT-based division. The MFT uses two feedbacks: one feedback of the current state register, and a second feedback of the previous, or delayed, state d_{i-1}.

We obtain the iteration equations starting with $T_0() = \frac{b}{d}$. As there are no input digits x_i to consume (both inputs are present at the start of computation), it is

[2]Ercegovac[52] proposes a general method that can also evaluate rational functions, but poses restrictions on the coefficients.

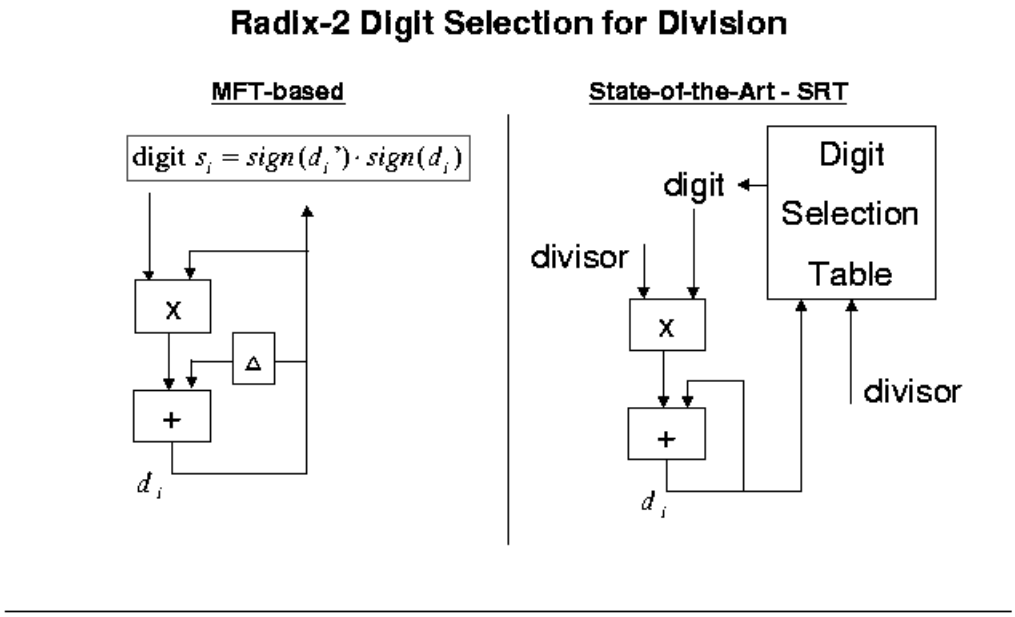


Figure 4.5: *The figure shows the structure of the MFT divider and the state-of-the-art SRT divider.* Δ *stands for an additional unit delay element.*

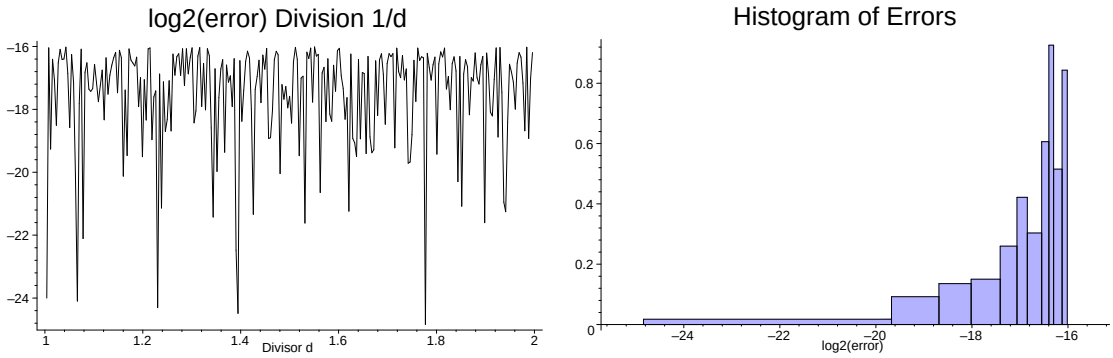


Figure 4.6: *The graphs plot the error for computing* 1/d *after 16 iterations with 16-bit precision at each iteration, not including exact results. (a) shows* $\log_2$(error) *for divisor* d *between one and two. (b) shows the distribution of error values with a histogram.*

sufficient to apply transformation $T_{i+1}() = \frac{1}{T_i() - o_i}$ for producing output digits.

$$d_{-1} = \mathtt{b} \qquad d_0 = \mathtt{d} \qquad d_{i+1} = d_{i-1} - o_i d_i \tag{4.9}$$

There is only one equation and one state register d since $b_{i+1} = d_i$. As before the output digits o_i based on the MFT are:

$$\begin{aligned} &\texttt{for even digits,} && o_{i=\mathtt{even}} = -(s_i + \frac{s_{i-1}}{2}), \\ &\texttt{for odd digits,} && o_{i=\mathtt{odd}} = (2s_i + s_{i-1}), \\ & && s_0 = 0. \end{aligned} \tag{4.10}$$

In fact, the iterations are reduced to shift-and-add operations. By unrolling the iterations, the shifts can be hardwired and the computation simplifies to a sequence of additions. A structural comparison of the resulting unit to state-of-the-art SRT-division[7][8](for recent improvements in SRT implementation see [41]) is shown in figure 4.7. The iterations 4.9 for computing $o_{bin} = \frac{\mathtt{b}}{\mathtt{d}}$ reduce to simple shift-and-adds

similar to conventional division as follows:

$$d_{-1} = \mathtt{b} \qquad d_0 = \mathtt{d} \qquad d_{i+1} = d_{i-1} + s_{i-1}2^{-1}d_i + s_i 2^0 d_i \tag{4.11}$$

The main difference to state-of-the-art division is that the state d_{i+1} depends on the **two** previous iterations.

Absorbing the inverse MFT into the next-output digit selection, the algorithm chooses the next signed-digit from $s_i = \{+1, -1\}$ by solving $o_i \sim \frac{d_{i-1}}{d_i}$ and equation 4.10 to:

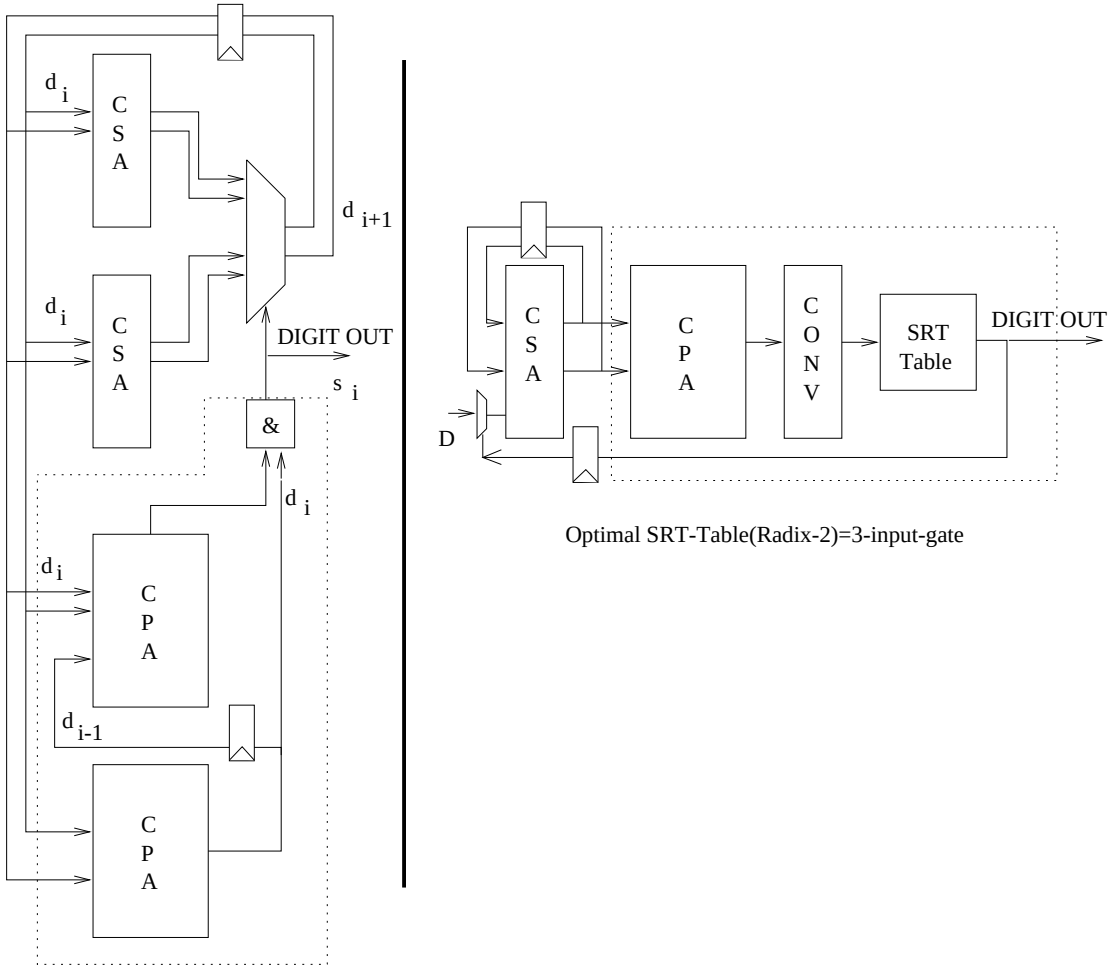
$$\texttt{for even digits,} \quad s_i = sign\left((-1)^{i-1} \cdot d_{i-1} - s_{i-1} \cdot 0.5 \cdot d_i\right) \cdot sign(d_i) \tag{4.12}$$

$$\texttt{for odd digits,} \quad s_i = sign\left((-1)^{i-1} \cdot d_{i-1} - s_{i-1} \cdot d_i\right) \cdot sign(d_i) \tag{4.13}$$

Equation 4.13 enables the algorithm to select the next signed digit with very little effort. The self-correcting feature of the algebraic algorithm guarantees convergence to the correct result.

For the conventional divider, a redundant digit set introduces some flexibility in the next-output digit selection function. As a consequence, a radix-2 SRT divider has a delay of 1 CSA plus a few gates to select the next digit. A similar approach can get rid of the CPAs for the MFT-based unit.

Figure 4.6 shows simulation results for the simple case $\mathtt{b} = 1$(computing $\frac{1}{\mathtt{d}}$). Operands are between one and two, like the mantissa of the floating point[9] representation. The figures show simulation results for an implementation of 16 iterations, 16-bit accurate iteration computations, targeting 16-bit accurate results ($\mathtt{error} < 2^{-16}$). Note that most results are close to the desired accuracy, i.e. for most divisors $2^{-17} < error < 2^{-16}$. This suggests a very high efficiency of the algorithm.



(a) MFT-based Radix-2 Divider

(b) SRT-Table-based Radix-2 Divider

Figure 4.7: *The figure shows (a) a regular, pipelined radix-2 divider based on the MFT, and (b) a general SRT-Table divider based on results from [41]. Broken lines en-capsule logic that can be collapsed to a few gates in case of a redundant digit representation.*

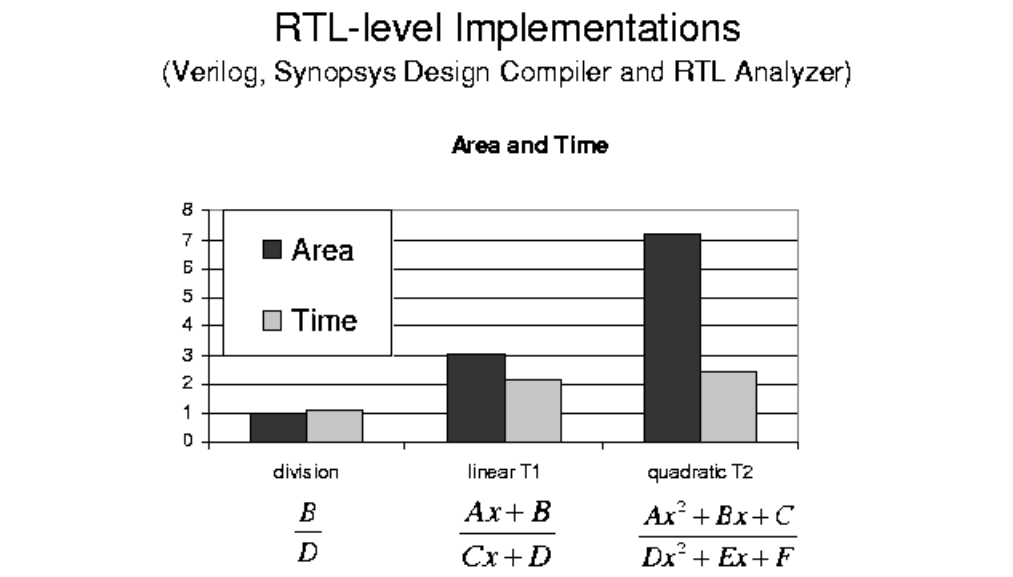


Figure 4.8: *The figure shows the area and time(latency) tradeoff for 'shift and add' based (class 1) rational arithmetic units implemented in Verilog and synthesized with a commercial synthesis tool. The results are shown relative to division.* $\frac{B}{D}$ *requires about 1 unit area and 1 unit of time.*

In conclusion, the complexity of the MFT-based divider is similar to the complexity of state-of-the-art division. The main difference is that the novel MFT-based divider scales up to higher degree rational arithmetic units and enables the extension of conventional floating point arithmetic units to higher-degree rational functions.

4.3 VLSI Implementation of Rational Arithmetic Units

The algebraic algorithm enables the computation of rational functions such as $T(x) = \frac{ax+b}{cx+d}, T(x) = \frac{ax^2+bx+c}{dx^2+ex+f}$, but also functions of multiple variables such as $T(x,y) = \frac{axy+bx+cy+d}{exy+fx+gy+h}$. The microarchitecture of these arithmetic units can be constructed from the iteration equations 4.5, 4.6 and 4.7.

In this section we implement division, linear, and quadratic rational arithmetic units at the RTL level using Verilog. The purpose is to get an understanding of the area-time tradeoff between the different degrees of the polynomials of rational functions. More specifically, Synopsys RTL-Analyzer[71] is used to obtain timing and area estimates for the rational arithmetic units.

Figure 4.8 shows the VLSI area and latency results for MFT-based division, linear, and quadratic fractional transformations. The arithmetic units are implemented in Verilog and synthesized with Synopsys Design Compiler[71]. With increasing degree N of the polynomials, area grows with at least $O(N^2)$ making implementations of higher-degree polynomials less attractive. Given enough area, the latency-growth of the proposed rational arithmetic units is less than $O(N)$.

All arithmetic units above use radix-2 digits and thus, for results with X bits of precision X iterations are required. In order to speed up computation, we investigate higher radix rational arithmetic algorithms next.

4.4 Higher Radix Rational Arithmetic

For conventional iterative division units, increasing the radix reduces the number of iterations per result. This section considers using an arbitrary radix r. The radix-r algorithms use redundant digit-sets such as $s_i = \{-(r-1), \ldots, r-1\}$.

In general, high radix algorithms (e.g. for division) reduce the number of iterations at the cost of additional area and, possibly, a longer delay per iteration. As a consequence, current state-of-the-art in SRT-division, does not scale well to efficient dividers much beyond radix-8. The MFT enables us to design efficient higher radix algorithms with an algebraic formulation of the selection functions, as shown for radix-2 in the previous sections.

The M-log-Fraction for the desired radix, shown below, leads to the design of the iterative algorithm for radix-r numbers(rrn). Let us compute the linear fractional transformation $y = \frac{ax+b}{cx+d}$ with $x_{rrn} = t_1 t_2 t_3 \ldots = [x_1, x_2, x_3, \ldots]$ and $y_{rrn} = s_1 s_2 s_3 \ldots = [y_1, y_2, y_3, \ldots]$. t_i, s_i are radix-r digits and x_i, y_i are the equivalent CF digits:

$$\begin{aligned} y_{rrn} &= s_1 r^{-1} + s_2 r^{-2} + s_3 r^{-3} + s_4 r^{-4} + \cdots \\ &= [0; y_1, y_2, y_3, \ldots] && (4.14) \\ y_1 &= \frac{r}{s_1} \qquad p_1 = s_1^{-1} \qquad \overline{p_1} \triangleq \frac{1}{p_1} = s_1 && (4.15) \\ y_i &= (-1)^{i-1} \cdot \left(p_i + r^{-1} \cdot \overline{p_{i-1}}\right) \cdot q_i && (4.16) \end{aligned}$$

with temporary variables

$$\begin{aligned} p_i &= \overline{p_{i-1}} \frac{s_{i-2}}{s_{i-1}} \qquad \overline{p_i} \triangleq \frac{1}{p_i} = p_{i-1} \frac{s_{i-1}}{s_{i-2}} && (4.17) \\ q_i &= \begin{cases} r & \texttt{i} = \texttt{odd} \\ 1 & \texttt{i} = \texttt{even} \end{cases} && (4.18) \end{aligned}$$

The next digit selection function for radix-r arithmetic is similar to the selection function in the radix-2 case:

$$y_i \sim \frac{a_i x_i + b_i}{c_i x_i + d_i} \tag{4.19}$$

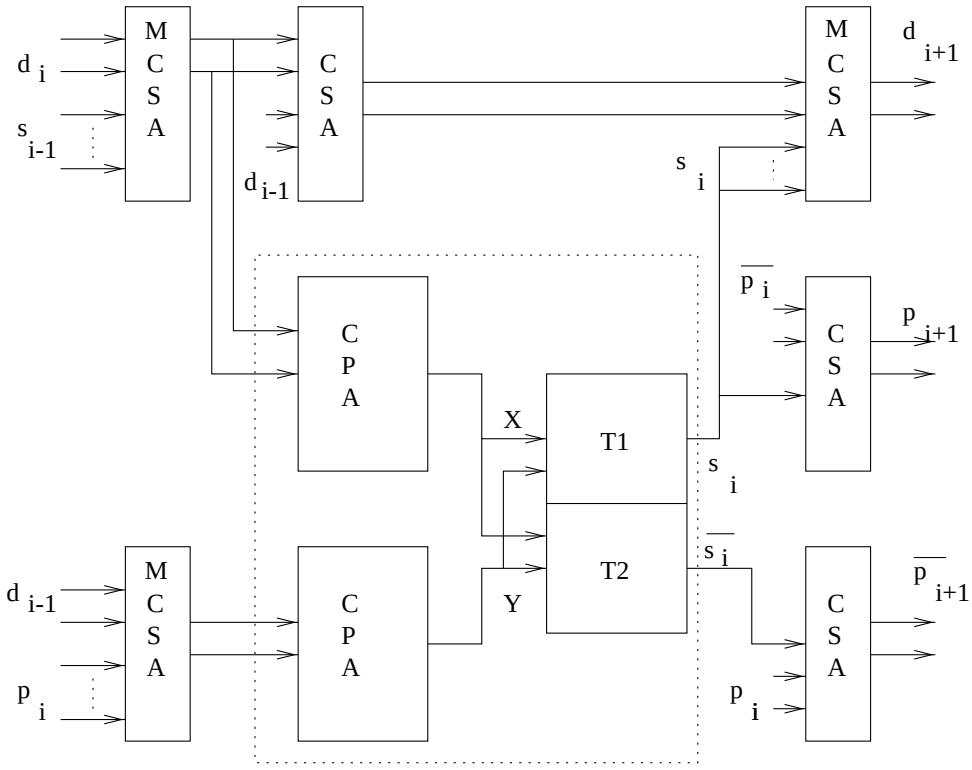


Figure 4.9: *The figure shows a radix-r base-implementation of the proposed division unit.* $\overline{s_i} \triangleq \frac{1}{s_i}, \overline{p_i} \triangleq \frac{1}{p_i}$, *MCSAs denote* $(\log r)$*-level CSA structures. Consequently, variables* $d, p,$ *and* $\overline{p}$ *hold full precision values. T1, T2 are small* $\log r$ *by* $\log r$ *tables. However, as in the case of radix-2, a redundant digit representation might enable us to reduce the complexity of choosing the next digit* s_i *to a few gates. The broken line en-capsules the area that could be simplified.*

Equations 4.19,4.16 lead to the next output digit s_i (radix-r) as a function of:

$$s_i = f(a_i, b_i, c_i, d_i, y_i) \tag{4.20}$$

The above equations are valid for all bilinear units. A special case of the bilinear unit is division. High-radix division is important for decreasing the number of iterations for the division operation. Below we show the next-digit selection function for high-radix division.

Figure 4.9 shows the microarchitecture of the above algorithm simplified to radix-r division. Tables $T1$ and $T2$ use the most significant bits of X and Y to choose the next output digit $s_i \sim \frac{X}{Y}$. In general, increasing the radix r, increases the number of levels of CSAs and increases the required precision in tables $T1$ and $T2$.

Formally, the next digit selection function from equation 4.20 becomes:

$$s_i \sim \frac{X}{Y} = \begin{cases} \frac{d_i s_{i-1} r}{-p_i d_{i-1} r - d_i} & i = \texttt{even} \\ \frac{d_i s_{i-1} r}{p_i d_{i-1} - d_i} & i = \texttt{odd} \end{cases} \tag{4.21}$$

with p_i computed by equation 4.17.

In summary, arbitrary radix-r rational arithmetic units are quickly increase in complexity. Even in the simplest case, division, retiring higher-radix digits requires substantial computational effort as shown in figure 4.9. The approach presented in this section can be used to derive high-radix rational arithmetic units for bilinear functions and quadratic rational functions leading to high-speed implementations of the rational arithmetic units introduced in this thesis.

4.5 Related Work

Early investigations found a very high potential [18][21][22] (very optimistic [23]) for continued fraction arithmetic. Vuillemin[45], Kornerup and Matula[25] formalize Gospers ideas and investigate software and hardware implementations. Additional continued fraction algorithms such as sorting continued fractions are summarized in [69].

The conventional way of computing rational functions is based on multiply-adds and a final division[34]. The advantage of the MFT method is that the most significant digit is produced immediately (with the delay of one iteration) given the first input digit. Figure 4.10 shows a direct comparison of the two options. The overall delay of the operations is similar, depending on the particular implementation. An advantage of the MFT-based design is a fast "first digit out", i.e. the first result digit is produced immediately after the first input digit is consumed. In addition, a single iteration step can be used in a loop to compute the entire function that would require multipliers, adders and a divider.

There are many shift-and-add based algorithms approximating elementary functions. One of the most popular 'shift-and-add' algorithms is CORDIC[56][57]. CORDIC algorithms were formally introduced by Volder in [56] and unified to compute elementary functions by Walther in [57]. CORDIC functional units compute up to two elementary functions at the same time. Given three arguments x, y, z, minor modifications to the CORDIC architecture compute function pairs such as:

$\{xcos(z) - ysin(z), ycos(z) - xsin(z)\}$

$\{x, y - xz\}$

$\{xcosh(z) - ysinh(z), ycosh(z) - xsinh(z)\}$

$\{\sqrt{x^2 + y^2}, z - tan^{-1}(y/x)\}$

$\{x, z - (y/x)\}$

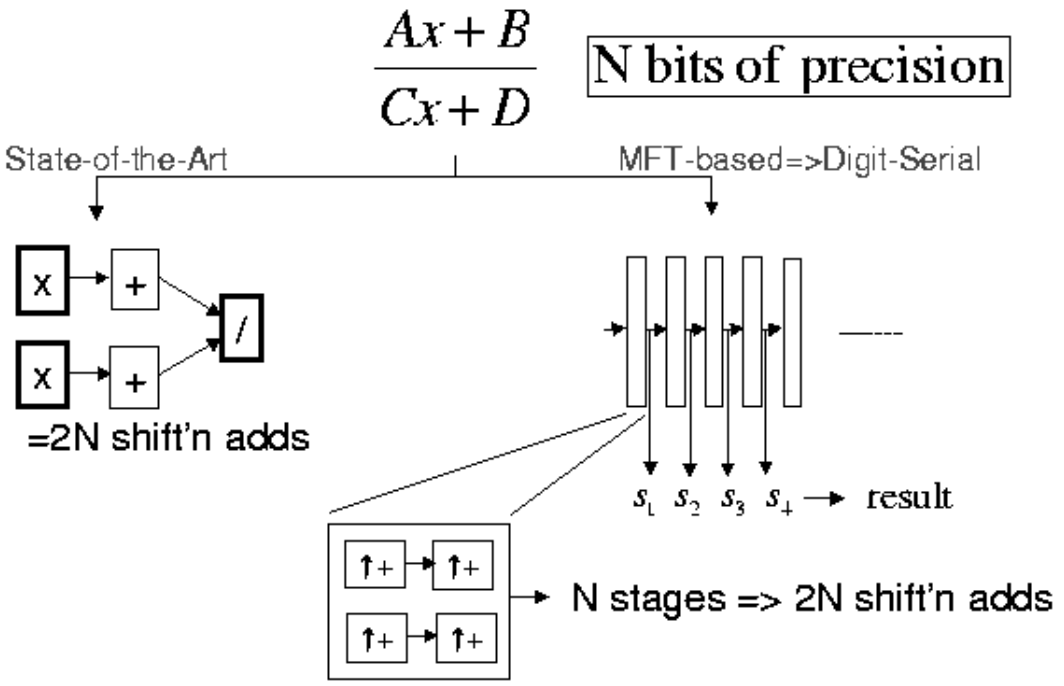


Figure 4.10: *The figure shows a conceptual comparison of the structures of computation for class 1 MFT-based arithmetic units computing a bilinear function, and the conventional approach of computing two multiply-adds followed by a final division. The box with the arrow and a plus stands for a shift-and-add step. Outputs s_i are the digits of the result in most-significant digit(MSD) first order.*

$\{\sqrt{x^2 - y^2}, z - tanh^{-1}(y/x)\}$

The fundamental mathematical principles behind CORDIC algorithms can be found in their scalar form in the work of T.C. Chen [58], as pointed out by Ahmed in his thesis [55]. Ahmed showed that if T.C. Chen's convergence computation technique is applied instead of real numbers (as assumed by Chen) to complex numbers one obtains the class of CORDIC algorithms. The real power of CORDICs lies in the ability to compute two functions simultaneously, where a conventional functional unit only computes one scalar function.

Although the set of functions that can be computed appears to be limited, CORDICs are very popular, especially in signal processing, due to the simplicity of implementation in hardware[29].

Ercegovac[51][52] generalizes digit-recurrence algorithms such as SRT division to matrices and vectors, creating the *E-method*. Ercegovac uses linear algebra to derive linear matrix equations that represent polynomial and rational expressions. The E-method is a 'shift-and-add' method solving the resulting linear system that represents the approximation. The required computation is done in linear time, i.e. producing one output digits per iteration, similar to the MFT-method proposed in this work. The E-method is a more general method capable of computing various expressions such as polynomial and rational approximations, given a specific set of restrictions. In the case of rational approximation, however, the E-method has very tight restrictions on the values of the coefficients.

Low latency, parallel arithmetic algorithms use substantially more VLSI area than the previous methods to obtain an approximation. Examples are table based methods such as bipartite tables[39][38] or multiplier/CSA-tree-based methods[37][36]. These parallel methods are more closely related to the multiplication-based algorithms presented in the next chapter.

In summary, the above chapter shows the details on how to combine the MFT and the algebraic algorithm to design rational arithmetic units based on shift-and-add primitives. The size of the arithmetic units grows very quickly with increasing order of the polynomials. Thus, higher order rational approximations require very large VLSI areas. Once the available area for arithmetic units is large enough to justify the use of multipliers, it makes sense to switch to multiplication-based algorithms. To address the computation of higher order rational approximations the next chapter looks at such multiplication-based arithmetic structures based on the MFT.

Chapter 5

Algorithm Class 2: Rational Approximation of Analytic Functions

As we have seen in the previous chapter, the area requirement of shift-and-add based rational arithmetic units grows very quickly with increasing order of the rational approximation. As a consequence, high-order rational approximations require a different solution. Of course it is possible to use shift-and-add based arithmetic units to compute multiple low-order rational approximations and combine them in a sum or a product. However, there is a faster way of computing high-order rational approximations if a multiplier is available. This chapter explores a multiplication-based algorithm based on the MFT and the algebraic algorithm that enables the evaluation of high- order rational approximations.

Which applications or rather functions require high-order rational approximations? In general, for most functions, rational approximations converge faster than polynomial approximations. Some functions are inherently difficult to approximate. A prominent "hard" function is the incomplete Gamma function which is used, for

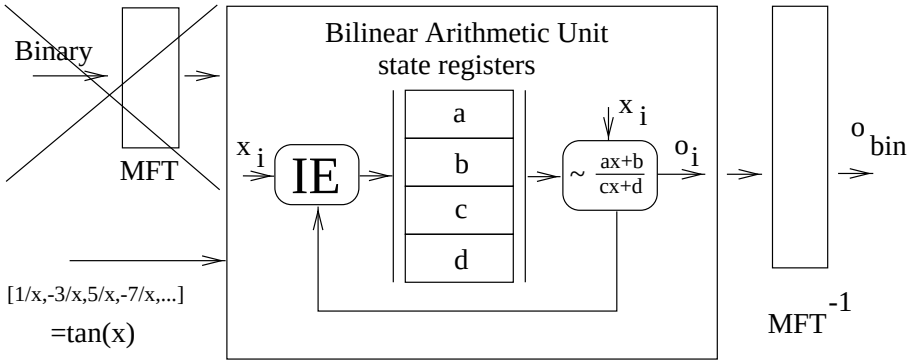


Figure 5.1: *The figure shows the state machine for the iteration equations (IE) evaluating a simple continued fraction. In this case the input digits are the digits of the continued fraction representing the rational approximation of the function.*

example, in signal and image processing applications.

A simple continued fraction representing a rational approximation of a function $f(x)$ replaces the input MFT in the MFT-based algebraic algorithm. The resulting arithmetic structure shown in figure 5.1 evaluates the simple continued fraction approximation yielding one output digit per iteration. As explained in chapter 1, continued fraction approximations are alternative representations for rational approximations. Some continued fraction approximation "gems" are known from literature (e.g. equations 1.7, 1.8,1.9,1.10). In general, any rational approximation can be converted to a simple continued fraction with linear CF digits using equivalence 1 (chapter 1) taken from Wall[63]:

$$H(z) = \frac{a_{00}z^n + a_{01}z^{n-1} + \cdots + a_{0n}}{a_{11}z^{n-1} + a_{12}z^{n-2} + \cdots + a_{1n}} \equiv [r_1 z + s_1, r_2 z + s_2, \cdots, r_n z + s_n] \qquad (5.1)$$

with all $a_{ij} \neq 0$, and $r_i \neq 0$.

5.1 A multiplication-based rational arithmetic unit

Let us focus now on evaluating simple continued fraction expansions (or rational approximations) with rational elements r_i, s_i to binary numbers using the positional algebraic algorithm from the previous chapters. The trivial *linear fractional transformation* $\frac{1 \cdot x + 0}{0 \cdot x + 1}$ computed with the positional algebraic algorithm can be used to convert a simple continued fraction expansion such as equation 5.1 to a binary M-log-Fraction. This is achieved by consuming the input digits of the CF expansion, and selecting the output digits according to the M-log-Fraction. Figure 5.1 shows the input digits of the function $\tan(x)$ being consumed by the arithmetic unit. Consuming an input digit requires a multiplication with the argument x. The output is chosen according to the inverse MFT. Thus, on the output side the computational effort remains a shift-and-add just like in the previous chapter.

The multiplicative algorithm shown in this section only makes sense for hard functions, i.e. functions which require many polynomial terms for accurate approximation. The area-time results for shift-and-add based arithmetic units in figure 4.8 show that the area of these arithmetic units grows very rapidly with the degree of the polynomials. It makes sense to use the multiplicative algorithm presented in this chapter when the area of a shift-and-add arithmetic unit exceeds the area required for multiplication.

Instead of fixing the shifts M_i, we use the initial MFT (Theorem 2 and compute at each iteration the next shift M_i *and* the signed-digit s_i. The resulting M-log-Fraction is a combination of Theorem 2 and Corollary 3. This small modification of the algorithm from the previous chapter increases the average speed of convergence. We use the following corollary of the MFT from Theorem 2:

Corollary 4 *Signed-digit M-log-Fraction with variable shifts. A binary number B_R with n digits, and $s_i \in \{+1, -1\}$ is equivalent to a simple continued fraction with n*

partial quotients as follows:

$$\begin{aligned} B_R &= s_1 2^{\beta_1} + s_2 2^{\beta_2} + s_3 2^{\beta_3} + s_4 2^{\beta_4} + \ldots + s_n 2^{\beta_n} \\ &\equiv [0; s_1 2^{M_1}, -(s_1 2^{-M_1} + s_2 2^{M_2}), (s_2 2^{-M_2} + s_3 2^{M_3}), -(s_3 2^{-M_3} + s_4 2^{M_4}), \\ &\quad \ldots \pm (s_{n-1} 2^{-M_{n-1}} + s_n 2^{M_n})] \end{aligned} \tag{5.2}$$

The corollary differs from previous M-log-Fraction by using signed-digits s_i *and* variable shifts M_i. The variable shifts M_i set the values of the β's. We use the corollary on the MFT above to represent the output $o_{bin} = [o_1, o_2, o_3, \ldots]$, and obtain $o_1 = s_1 2^{M_1}$, and for $i > 1$, $o_i = (-1)^{i-1}(s_i 2^{M_i} + s_{i-1} 2^{-M_{i-1}})$. The input fraction is not an M-log-Fraction. Instead, the input digits are taken from a continued fraction approximation as shown below.

As before, the algorithm is defined by the iteration equations 4.5 repeated below:

$$\begin{aligned} \mathtt{a_{i+1}} &= \mathtt{c_i x_i + d_i} & \mathtt{b_{i+1}} &= \mathtt{c_i} \\ \mathtt{c_{i+1}} &= \mathtt{a_i x_i + b_i - o_i(c_i x_i + d_i)} & \mathtt{d_{i+1}} &= \mathtt{a_i - o_i c_i} \end{aligned} \tag{5.3}$$

The iteration equations for the linear fractional algebraic algorithm lead to the following equations for s_i and M_i:

$$M_1 = \lfloor \log_2(x_1) \rfloor \tag{5.4}$$

$$s_1 = \mathtt{sign}(x_1) \tag{5.5}$$

Note that $a_0 = d_0 = 1$ and $b_0 = c_0 = 0$, simplifying M_1 and s_1. For $i > 1$ the equations for the output digits are:

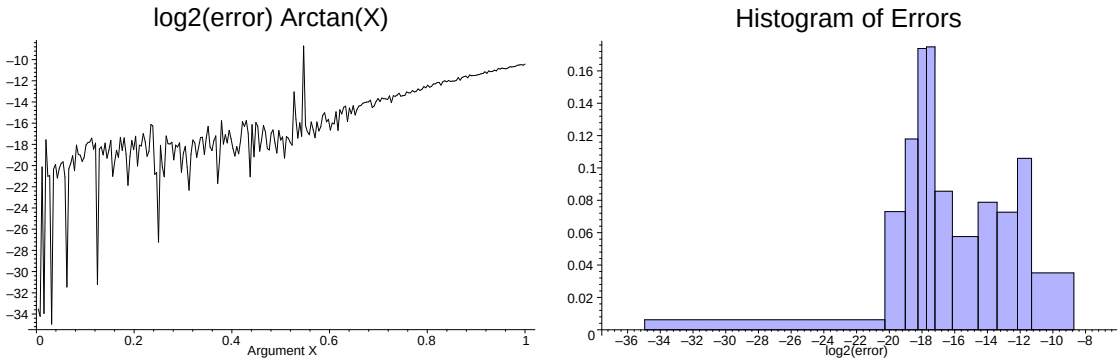
$$
\begin{aligned}
M_i &= \left\lfloor \log_2 \left| (-1)^{i-1} \left(\frac{ax_i + b}{cx_i + d} - s_{i-1} 2^{-M_{i-1}} \right) \right| \right\rfloor \\
&\sim \lfloor \log_2 |ax_i + b - s_{i-1} 2^{-M_{i-1}} (cx_i + d)| \rfloor - \lfloor \log_2 |cx_i + d| \rfloor \qquad (5.6) \\
s_i &= \texttt{sign}(ax_i + b - s_{i-1} 2^{-M_{i-1}} (cx_i + d)) \cdot \texttt{sign}((-1)^{i-1} (cx_i + d)) \qquad (5.7)
\end{aligned}
$$

Multiplication with x_i for the class 2 method requires a full precision multiply (NxN bits) while a multiplication with o_i just requires a shift-and-add. Thus, the 4 shift-and-add operations for equation 4.5 for the class 1 algorithm turn into 2 multiplications and 2 shift-and-adds. Instead of computing $\log(x)$ and then taking the floor $\lfloor x \rfloor$, we use a *Leading One Detect*(LOD)[42] circuit. Computing M_i requires 2 shift-and-adds and 2 LODs. The leading one detect circuit returns the position of the most left '1' of a binary number. As this is also the most significant '1', the integer value of the position of the left-most '1' is equal to $\lfloor \log_2(x) \rfloor$. A leading one detect circuit usually has a latency of less than one clock cycle, or one full carry-propagate.

The convergence of the evaluation algorithm increases to above one bit per iteration by computing an approximation to the optimal next shift (M_i).

Jones and Thron ([17], p. 202) give arctan(1) as an example of a function that converges much faster when approximated by a continued fraction compared to a Taylor approximation. In order to show the versatility of continued fraction approximations, the examples below also show a composite function, and the incomplete Gamma function. For simplicity, the examples use approximations with integer coefficients. For optimal results over an interval an optimal approximation with *minimax* coefficients can be obtained with Remez's method [43],[26].

Example 1 *Approximate* $\arctan(x)$ *for* x *between zero and one, with the continued fraction approximation given in [17], p. 202.*



(a) (b)

Figure 5.2: *The graphs show the error for computing arctan(x) after 16 iterations with 16-bit precision at each iteration. (a) shows* $\log_2(\texttt{error})$ *for arguments x between zero and one. (b) shows the distribution of error values with a histogram.*

$$arctan(x) = \left[0; \frac{1}{x}, \frac{3}{x}, \frac{5}{x}\left(\frac{1}{2}\right)^2, \frac{7}{x}\left(\frac{2}{3}\right)^2, \frac{9}{x}\left(\frac{3}{4}\right)^2, \ldots\right] \tag{5.8}$$

Figure 5.2 shows the results for 16 iterations with 16-bit precision at each iteration.

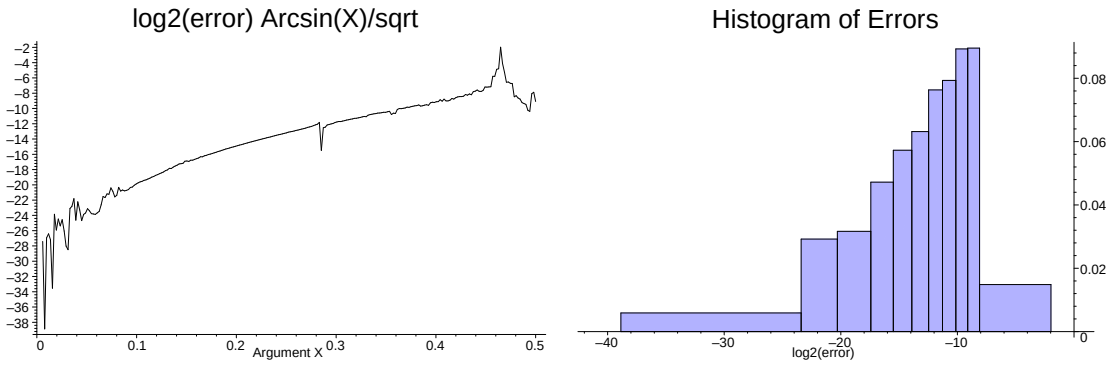
Example 2 *Approximate the normalized function* $\frac{\arcsin(x)}{\sqrt{1-x^2}}$ *for x between zero and one half, with the continued fraction approximation given in [17], p. 203.*

$$\frac{arcsin(x)}{\sqrt{1-x^2}} = \left[0; \frac{1}{x}, -\frac{3}{1\cdot 2}\cdot\frac{1}{x}, \frac{5}{1\cdot 2}\cdot\frac{1}{x}, \frac{7}{3\cdot 4}\cdot\frac{1}{x}\cdots\right] \tag{5.9}$$

Figure 5.3 shows the results for 16 iterations with 16-bit precision at each iteration.

Example 3 *Approximate the incomplete Gamma function* $\Gamma(a,x)$ *for* $a = \frac{1}{2}, x$ *between 10 and 100, with the continued fraction approximation given in [17], p. 348, and [35].*

$$\Gamma(0.5,x) = e^{-x}\cdot x^{0.5}\cdot\left[0; x, (0.5)^{-1}, x, (1.5)^{-1}, x, (2.5)^{-1}, x, (3.5)^{-1}\ldots\right] \tag{5.10}$$



(a) (b)

Figure 5.3: *The graphs show the error for computing* $\frac{arcsin(x)}{\sqrt{1-x^2}}$ *after 16 iterations with 16-bit precision at each iteration. (a) shows* $\log_2(\texttt{error})$ *for arguments* x *between zero and one half. (b) shows the distribution of error values with a histogram.*

Figure 5.4 shows the results for 16 iterations with 16-bit precision at each iteration. The figure shows increasing precision with increasing distance from $x = 0$*. Intuitively, the even quotients of the continued fraction,* x*, lead to pseudo-divisions by zero. In fact,* $\Gamma(0.5, x)$ *has a pole at zero.*

The actual results of the examples below show the convergence of the algorithm to the exact function, and thus includes also the convergence behavior of the rational approximation. Thus, we are dealing with two sources of error: the error of the rational approximation and the error of the evaluation of the rational approximation. As a consequence of the limitations posed by the convergence of the rational approximation, for some arguments X, overall convergence can drop below one bit per iteration. In the previous examples, this happens in example 1 for arguments larger than ~ 0.55, in example 2 for arguments larger than 0.2, and in example 3 for arguments smaller than ~ 25. Note that the algorithm degrades gracefully. Even when the rational approximation does not converge fast enough, the result stays as close to the rational approximation as possible.

The multiplication-based MFT algorithm enables the design of a general purpose

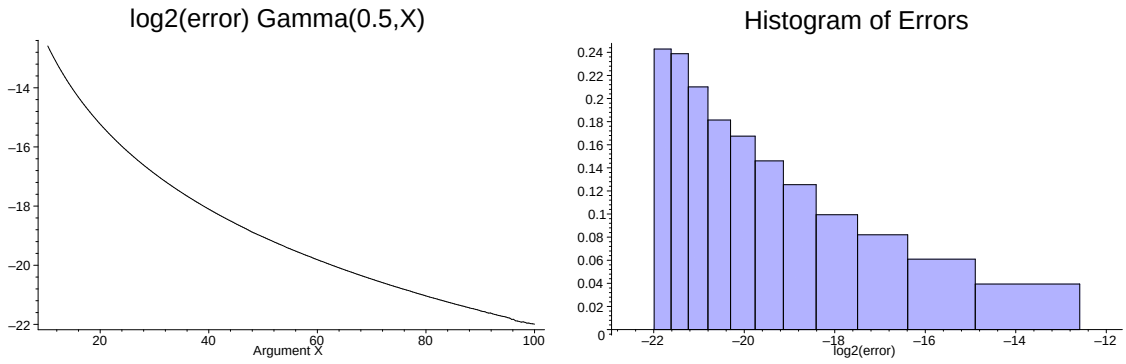


Figure 5.4: *The graphs plot the error for computing the incomplete Gamma function* $\Gamma(0.5, x)$ *after 16 iterations with 16-bit precision at each iteration. (a) shows* $\log_2(\texttt{error})$ *for arguments* x *between 10 and 100. (b) shows the distribution of error values with a histogram.*

rational approximation unit that can be used for the approximation of any analytical function. The resulting general-purpose arithmetic unit produces "fast first digit out" results, in the same manner as class 1 units.

5.2 Related Work

Rational approximations can be evaluated in continued fraction form, or by dividing two polynomials. Evaluating the continued fraction leads to the minimal number of operations, but requires many divisions. For an implementation technology where division takes significantly more time and/or resources than multiplication, it makes sense to evaluate two polynomials first, and then use only one division to obtain the result.

There are many options for evaluating polynomials with multiply-add structures. The various methods differ in the required hardware resources. A straightforward evaluation of polynomials uses the Horner scheme:

$$p(x) = a + bx + cx^2 + dx^3 = ((d'x + c')x + b')x + a' \tag{5.11}$$

Figure 5.5 compares conventional computation of rational approximation with the multiplication based MFT method (class 2). As before for class 1 both approaches have a similar latency. However, the MFT-based approach delivers a "fast first digit out" and offers a regular implementation.

Koren[34] uses two parallel multiply-add modules to compute rational approximations. The two polynomials are evaluated in parallel with Horner's scheme, followed by a division.

Knuth[64] obtains more complex structures to find the minimum number of operations for evaluating polynomials. A more regular scheme for parallel or pipelined execution with a performance between Knuth's optimum and the Horner scheme is proposed by Estrin[70][64]. The methods for evaluating polynomials are very efficient and could be used to evaluate the top and bottom polynomials of a rational approximations before the final divide. However, they require multiple multipliers and are hard to adapt to variable degree polynomials. Thus, these methods are less useful for a general-purpose evaluation unit for higher-degree rational approximations.

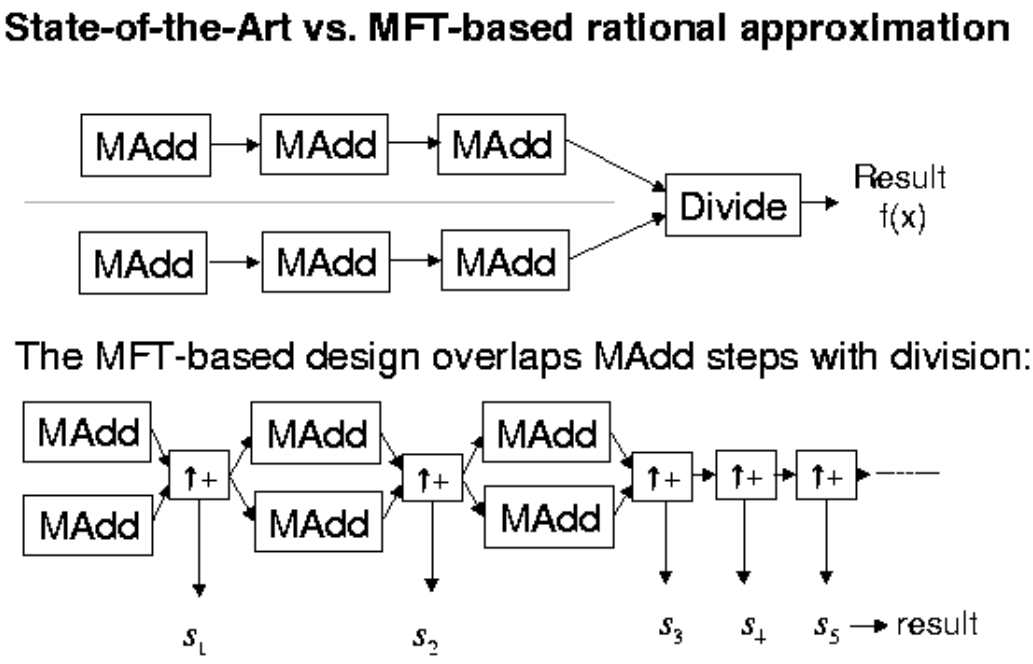


Figure 5.5: The figure shows a conceptual comparison of the structures of computation for the class 2 MFT-based arithmetic unit evaluating continued fraction approximations, and the conventional approach of evaluating a rational approximation with a series of multiply-adds(MAdd) and a final divide. The box with the arrow and a plus stands for a shift-and-add step. The example shown evaluates a rational approximation with third-degree polynomials. Outputs s_i are the digits of the result in most-significant digit(MSD) first order.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

We provide an effective algorithm to convert between binary numbers and continued fractions. The resulting transformation is called M-log Fraction Transform (MFT). The MFT provides a means to determine the optimal representation of binary numbers in the continued fraction space, and allows for error control.

The M-log-Fraction Transform(MFT), introduced in this work, enables the first practical implementation of continued fraction algorithms for computer systems. The MFT bridges the gap between continued fractions and the binary number representation, enabling the design of a new class of efficient rational arithmetic units.

From a distance the M-log-Fraction and MFT behave like a redundant representation of binary numbers utilizing the integer distances between '1's, similar to 0-runlength encoding (counting the '0's between the '1's of a binary number). Runlength encoding is a very fast way of converting between continued fractions and binary numbers reducing the conversion overhead by orders of magnitude. This fast conversion allows us to exploit the symmetries in the continued fraction space while computing and storing binary numbers.

The thesis explores two application areas of the MFT: 'shift and add' based rational arithmetic units (class 1), and multiplication-based digit-serial evaluation of higher degree rational approximations (class 2). The presented architectures are based on combining the MFT with a specific state-of-the-art continued fraction algorithm.

6.2 Future Work

Future work includes the application of the MFT to other continued fraction arithmetic algorithms such as the algorithms proposed by Jones and Thron [24][17] (see also [40] and [12]). [24] shows the application of continued fractions to problems in the frequency-domain of digital filters. A first step towards applying bilinear transformations to the time-domain of digital filters, inspired by [24], is proposed in [30].

In [30] linear FIR filters are extended by using bilinear function taps. Instead of a constant multiply, each "tap" consists of the bilinear function $\frac{ax+b}{cx+d}$ optimally implemented in hardware with class 1 MFT arithmetic units. The coefficients $a-d$ are found by starting with a linear filter ($a =$ optimal FIR coefficient, $b = c = 0$, and $d = 1$). A gradient search algorithm is used to improve the mean-square error of the frequency response. The resulting MFT-filter is non-linear (mostly in the stop-band), but stays closely related to the initial linear FIR filter.

Further research is required to identify and analyze the impact of the presented MFT arithmetic units on other numerically intensive application areas such as signal processing, multimedia processing, and computer graphics.

In computer graphics, for example, cubic surfaces ($f(x, y, z) = 0$) can be represented by rational parameterized forms with parameters u, v:

$$x = \frac{f_1(u,v)}{f_2(u,v)} \quad y = \frac{f_3(u,v)}{f_4(u,v)} \quad z = \frac{f_5(u,v)}{f_6(u,v)} \tag{6.1}$$

As more and more VLSI area is available for hardware acceleration of computer

graphics applications, MFT arithmetic units can be used to efficiently evaluate such cubic surfaces.

On the function approximation side, there is a connection between rational approximations, continued fraction approximations, and CORDIC arithmetic. The connecting paradigm seems to be the theory of linear transformations. CORDIC iteration can be written as (see [70] equation 6.6):

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 1 & -s_i 2^{-M} \\ s_i 2^{-M} & 1 \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \tag{6.2}$$

Interestingly, the M-log-Fraction can be written as:

$$\begin{pmatrix} x_{n+1} \\ y_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & -s_{i-1} 2^{M_{i-1}} \\ 1 & 1 + s_i 2^{M_i} \end{pmatrix} \begin{pmatrix} x_n \\ y_n \end{pmatrix} \tag{6.3}$$

The CORDIC iterations and M-log-Fractions appear to be special cases of a more general approximation space using linear transformations. An alternative approximation method, the *E-method*[52][51], uses linear transformations to evaluate polynomials, rational functions, and other expressions. However, evaluating rational approximations with the E-method is limited by restrictions on the values of the coefficients. Like in other scientific areas we are still missing a general unified theory for the three discussed methods.

On the implementation side, MFT arithmetic units promise to be an efficient alternative to state-of-the-art (floating point) division units. Current state-of-the-art SRT division is a result of decades of optimization and exploitation of redundant number representations. A similar effort is required for MFT arithmetic units. Of particular interest is the scalability of redundant, high-radix implementations of the arithmetic units proposed in this work. Future work will include VLSI implementation of MFT-based arithmetic units and a direct comparison to state-of-the-art division

units.

Digit-serial arithmetic units are also useful for a relatively young field of reconfigurable computing[31]. Reconfigurable, MFT-based rational arithmetic units are potential candidates for coarse-grain arithmetic cells. Arrays of rational arithmetic units offer the potential to speedup data-intensive and compute-intensive applications which are limited by arithmetic resources of a general-purpose processor.

Finally, the trend towards low-power and power-aware computing for mobile, ubiquitous and embedded communication and computation units can use high-level arithmetic units such as the rational arithmetic units presented in this thesis to minimize and/or adapt power consumption by using more powerful arithmetic units, enabling the computational elements to run at lower clock speeds.

Appendix A

Precision of Regular Continued Fraction Arithmetic

This appendix investigates hardware implementations of rational arithmetic units with inputs and outputs in regular continued fraction form, i.e. the digits or partial quotients of the input and output CF are small integers.

How many bits should we use to represent the integer digits of the regular CFs? Theorem 1 (Lochs) expresses the average behavior of integer CF digits:

(Lochs [66]) For almost all irrational numbers x and their approximation in the form of a regular continued fraction:

$$\lim_{n\to\infty} \frac{k_n(x)}{n} \simeq 0.9702. \tag{A.1}$$

with $k_n(x)$, the number of partial quotients, and n, the number of approximated decimal digits.

Theorem 1, repeated above, leads us to consider a *4+1-bit signed digit representation for partial quotients*. We know the distribution of the partial quotients of regular CFs from the previous section. By choosing 4-bits for the magnitude of integers to

represent one partial quotient we cover over 90% of the partial quotients of regular CFs. Including zero in the quotient digit set results in redundant CFs and allows us to handle quotient digit overflow by using Equivalence 5 below:

Equivalence 5 *(see [61][12][45]) For any fragment of a simple continued fraction with positive and/or negative partial quotients,*

$$[\ldots, a_i, 1, a_{i+2}, \ldots] \equiv [\ldots, a_i + 1, -a_{i+2} - 1, -a_{i+3}, -a_{i+4}, \ldots]$$
$$[\ldots, a_i, 0, a_{i+2}, \ldots] \equiv [\ldots, a_i + a_{i+2}, \ldots]$$

The algebraic algorithm introduced in chapter 4 computes functions such as $T_1(x) = \frac{ax+b}{cx+d}, T_2(x) = \frac{ax^2+bx+c}{dx^2+ex+f}$, but also functions of multiple variables such as $T_3(x, y) = \frac{axy+bx+cy+d}{exy+fx+gy+h}$. Given a *finite* 5-bit representation (including the sign) of quotient digits and *finite-size* state registers, we observe the following three main *sources of error*:

1. **Overflow of State Registers**: Representing the state registers with a finite number of bits leads to frequent overflow and limits the precision of the final result even with the use of floating-point-like arithmetic. Previous work[25] assumes infinite size registers[1].

2. **Virtual Singularities**: Even if there is no real division by zero, an intermediate input quotient may cause $cx+d = 0$ (in the linear case). Aborting the algorithm in case of division by zero introduces an error because the following part of the input is ignored.

[1]The paper[25] mentions that the results are true "given sufficient register lengths".

3. **Overflow of Quotient Digits**:The overflow of partial quotients creates Redundant Continued Fractions[2]. For 5-bit quotient digits, any quotient digit x with $|x| > 2^4$ results in a zero digit according to Equivalence 5. Thus, the CFs are not regular any more, and the algorithm may not converge to the right value. The following conditions also cause the algorithm to produce an error:

   - A CF ending with 1.
   - strings of 1s, e.g. $[\ldots, 1, 1, 1, 1, \ldots]$.
   - An end of the form $[\ldots, x, 0, x, 0, x, 0]$ or $[\ldots, x, 0, x, 0, x]$.

Given infinite resources and a converging, regular input CF, the positional algebraic algorithm converges to the exact result. However, the state registers either (1) quickly converge to the simplest form $\begin{pmatrix} |\underline{A}| & 0 \\ 0 & 1 \end{pmatrix}$ for T_1, or (2) diverge very rapidly towards infinity. In case (1) we obtain an exact result. Case (2) causes an overflow of the state registers and results in an approximate result.

The positional algebraic algorithm produces *almost always* exact results if all input quotients are regular CFs. "*Almost always*" is quantified in the results section below. However, the distribution of the values that are represented by CFs with integer digits is different from the uniform distribution of binary numbers (see 2).

Improved Positional Algebraic Algorithm

We propose the following *improvements* for the linear case,T_1, based on the three sources of error, explained above:

[2]An explanation of the distribution of regular continued fractions can be found in an article by Hall[14]. Hall proves that any rational number can be expressed as the sum or the product of two regular continued fractions with limited partial quotients, i.e. redundant continued fractions without zero quotients.

1. **Overflow of State Registers:** The *simple* solution is to shift the coefficients a to d to the right, and continue computation to a possibly non-exact result. Raney's results[16] suggest that we might not want to create exactly one output for each input. Raney's observations lead to a better solution to state register overflow: produce an output without consuming an input. The output can be chosen to decrease the values of the state registers without introducing errors.

2. **Virtual Singularities**: We choose the best possible value: $sign(ax+b)\cdot 2^4$.

3. **Overflow of Quotient Digits Creating Redundant Continued Fractions**: Redundant continued fractions allow us to pick from a variety of different continued fractions for the input to the transformation to avoid the cases that lead to non-exact results. We use equivalence 5 to convert between the various forms.

4. **Speedup** of the convergence of the transformation matrix $\underline{A}$ to $\begin{pmatrix} |\underline{A}| & 0 \\ 0 & 1 \end{pmatrix}$: In case $\left\lfloor \frac{a-1}{c} \right\rfloor = \frac{a-1}{c}$, we choose $o = \frac{a-1}{c}$, resulting in $d = 1$.

While the last feature does not address a particular source of error, it forces $\underline{A}$ faster to its simplest form where $c = 0, d = 1$. Examining equations 4.5 in more detail, we see that by choosing $o \sim \frac{ax+b}{cx+d}$, $c \to 0$. Intuitively, we also want to force $d \to 1$, in order to simplify $\underline{A}$.

For $|\underline{A}| = \pm 1, \underline{A}$ converges to the identity matrix $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. As soon as the identity matrix is reached, the tail of the input fraction is equal to the tail of the output fraction[12]; $\underline{A}$ does not change anymore, and the calculation can be terminated.

Below we show the improved **S**emi-**E**xact **P**ositional **A**lgorithm (`SEPA`) based on the positional algebraic algorithm for continued fraction arithmetic with finite state registers and redundant CFs.

SEPA Algorithm with Improvements:

```
 1. S[0]:={a,b,c,d}   //Init State Regs
 2. loop i,o from 0 to MAXLEN
 3.   if ((c*x[i]+d)==0.0)
 4.      Output[o]=sign(a*x[i]+b)*2^MAXVAL
 5.   else
 6.     if (c<>0)
 7.       if(frac((a-1)/(c))==0.0) and
 8.         (frac((a*x[i]+b)/(c*x[i]+d))<>0.0)
 9.           Output[o]=round((a-1)/c)
10.     else
11.       Output[o]=round((a*x[i]+b)/(c*x[i]+d))
12.   Compute Next State (S[o+1])
13.   if {state regs overflow}
14.     if (c<>0)
15.       Output[o]=round(a/c)
16.     else
17.       Output[o]=sign(a)*2^MAXVAL
18.   else
19.     i++; // consume input
20.   o++;   // produce output
21. endloop
22. return(Output)
```

The algorithm is shown for transformation T_1. `S[o]` stands for the state registers. Truncation of the output quotients to representable values is not shown for simplicity. `MAXLEN` is the maximal length of the input CF. `MAXVAL` is the maximal value of a

redundant CF quotient – chosen in case of a *virtual singularity*. Virtual singularities occur due to the truncation of CF quotients to integers.

In the *simple* case, lines `14-17` are replaced by a right-shift(division by a power of 2) of all state-registers.

Experimental Results

We use MapleV[26] to improve continued fraction arithmetic algorithms, and simulate various implementations. Exact arithmetic enables us to study the behavior of continued fraction arithmetic algorithms with arbitrary precision, limited only by computation time.

Running the simulations of the positional algebraic algorithm gives us more insight into its behavior and accuracy over a large set of inputs. We compare the *simple* algorithm to the improved `SEPA` algorithm described above. The simple version basically follows the standard algorithm, with a floating-point-like right-shift of all state registers on register overflow.

We classify individual results into **exact** and **non-exact** results. **Exact** results are results that match the result computed with Maple's exact arithmetic. **Non-exact** results differ from Maple's exact result by some error. We present the maximal error occurring within the non-exact results, and the average error, also within the non-exact results. Section A.2 deals with improving the maximal error with a correction term. The improvements suggested above are primarily chosen to minimize the percentage of non-exact results.

Example 4 *Linear fractional transformation* T_1*:*
$\underline{A} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ *with initial* a, b, c, d *between 1 and 15. Results are shown in figure A.1.*

The accuracy of the improved results does not depend much on state register size. For reasonably sized state registers 98.5% (1.5%) of results are (non-)exact. In

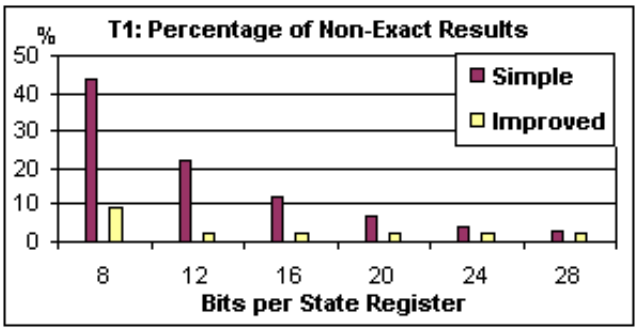


Figure A.1: **Example 4:**$T_1 = \frac{ax+b}{cx+d}$; Exact results match the input CF evaluated with T_1 using Maple's exact arithmetic. The "simple" results are obtained with "shift on overflow" of state registers.

addition, within the 1.5% of inputs that yield non-exact results, the *average* error is about 2^{-21}, and *maximal* error is 2^{-8}. The histogram of the distribution of error within non-exact results is shown in figure A.4.

Quadratic transformations create quadratic growth of state register values, resulting in a stronger dependence of precision on the size of the state registers.

As a consequence, overflow of state registers occurs more often, and the improvements that worked well in the linear case (T_1) fail to improve the performance in the quadratic case (results are shown without "improvements"). Still, with 28-bit registers, $\sim 75\%$ of inputs yield exact results. Average error of the non-exact computations is about $2^{-10} - 2^{-20}$ for the square function, and $2^{-20} - 2^{-25}$ for the chosen case $(1, 1, 1, 3, 2, 1)$, depending on state register size. In both cases maximal error is ~ 1 for state register sizes larger or equal to 12 bits.

Example 5 *Quadratic fractional transformation T_2:*

Figure A.2 shows cases:

- *square function* $\rightsquigarrow T_2 = \frac{x^2}{1}$.
- *chosen case* $\rightsquigarrow T_2 = \frac{x^2+x+1}{3x^2+2x+1}$.

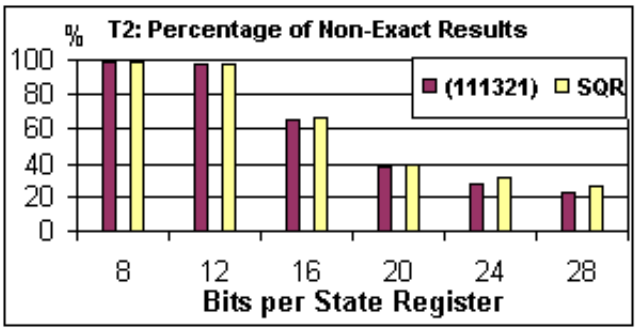


Figure A.2: **Example 5:**$T_2 = \frac{x^2}{1}$, $T_2 = \frac{x^2+x+1}{3x^2+2x+1}$; Exact results match the input CF evaluated with T_2 using Maple's exact arithmetic.

Example 6 *Quadratic fractional transformation of two input variables,* T_3*:*

- *Multiplication* $\rightsquigarrow T_3 = \frac{xy}{1}$.
- *Addition* $\rightsquigarrow T_3 = \frac{x+y}{1}$.

Results are shown in figure A.3.

As in the quadratic case with one input variable, the "improvements" of the linear case do not apply for the quadratic case T_3. Even with 28 bit registers, only about 70-80% of the results are exact. However, for the two cases shown in figure A.3 the average error of the non-exact results is about 2^{-20} for register sizes larger or equal to 16 bits.

A.1 Simple Continued Fraction Inputs

Simple continued fractions consist of partial quotients $\in \mathcal{R}$. We use the identity transformation $T_1 = \frac{1 \cdot x+0}{0 \cdot x+1}$ to convert simple continued fractions with rational partial quotients to redundant continued fractions – implicitly evaluating a continued fraction expansion, in our case $\tan(x)$.

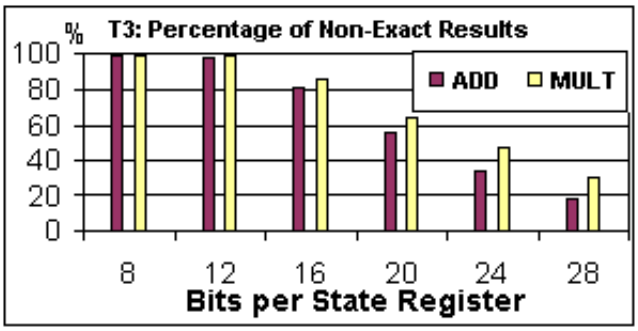


Figure A.3: **Example 6:**$T_3 = \frac{xy}{1}$, and $T_3 = \frac{x+y}{1}$. Exact results match the input CF evaluated with T_3 using Maple's exact arithmetic.

Example 7 *We evaluate* $\tan(x)$ *(from equation 1.7) with the identity transformation* $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, $T_1 = \frac{x}{1}$.

We observe a dependence of the accuracy of the final result on the accuracy of the input quotient. In fact, simulations show that the *average* error of the result is close to the precision of the state registers. *Maximal* error is roughly the square-root of the average error (i.e. half the bits).

Note that for a simple continued fraction input the algorithm does not produce any exact results. Accuracy is now not limited by state register overflow, as much as by the loss of accuracy from truncation of fractional digits.

It appears reasonable to expect that a converging simple continued fraction at the input would improve the accuracy (convergence) of the output. Convergence theorems for regular CFs are given in standard CF literature [12][17][63] etc. Surprisingly, the following convergent, positive, simple continued fraction expansion of $\tan(x)$, obtained from equation 1.7, fails to improve the precision of the final result.

$$\tan(x) = [0; \frac{1}{x} - 1, 1, \frac{3}{x} - 2, 1, \frac{5}{x} - 2, 1, \frac{7}{x} - 2, 1, \ldots] \tag{A.2}$$

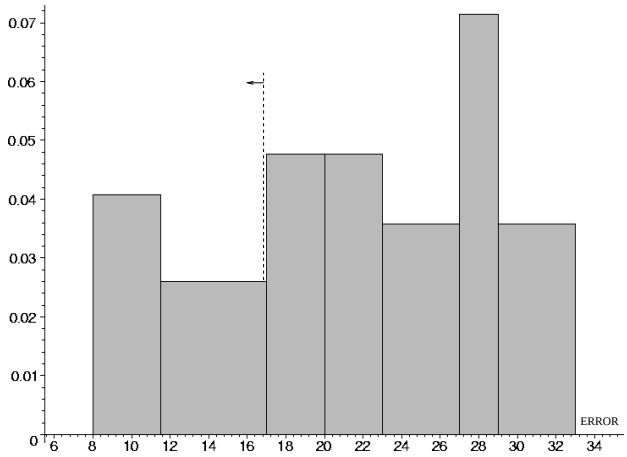


Figure A.4: The figure shows the histogram of error of non-exact results (logarithmic x-axis) for $T_1 = \frac{ax+b}{cx+d}$ (example 1). Boxes contain equal amounts of data points.

As in the case of continued fractions with integer quotients, we find no simple dependence between the convergence of the input, and the exactness of the output.

A.2 Final Optimization

Within the non-exact results, the proposed algorithm has a very low *average* error, but a relatively high *maximal* error. We discuss the final optimization of example 4 from above.

Figure A.4 shows the histogram of the distribution of the error within the 1.5% of non-exact results in the case of T_1 with 12-bit state registers, as shown in Example 1 above. We see that the error is almost uniformly distributed. In order to guarantee 16 bits of precision, we have to find a correction value for a small set of about 300 non-exact input values (out of 64K possible 16-bit inputs). A small programmable array such as a table, PLA, etc., indexed with a subset of input bits holds the 300 correction values. In case of a non-exact result, the corresponding correction value is added to the final result.

In conclusion, finite resources limit the achievable precision of continued fraction arithmetic. The proposed improvements make it feasible to obtain exact results in 98.5% of cases for the linear fractional transformation (T_1), even with relatively small registers – making the algorithm interesting for implementation in hardware. Quadratic transformations create quadratic growth of state register values, resulting in a stronger dependence on the size of the state registers. A large percentage of non-exact results make it unrealistic to guarantee a specific precision with reasonable resources.

The examples analyzed in this appendix form the beginning of a bit-level understanding of *algebraic algorithms* with regular continued fractions for rational arithmetic. The major issues for the regular CF algorithms to be practical in the framework of real computer systems:

- The **Redundant Continued Fraction** representation limits the achievable precision by limiting the maximal range of partial quotients. Extending the maximal value of a quotient with "0" quotients leads to unacceptable growth of the number of partial quotients.

- **Quadratic growth** of state register values for quadratic transformations limits the predictability and precision of quadratic arithmetic units.

- **Conversion** of regular continued fractions to and from binary numbers limits the performance and applicability of current regular continued fraction arithmetic.

All three problems are solved by using the M-log-Fraction Transform introduced in this thesis.

Appendix B

Historical Notes on Continued Fractions in Arithmetic

The fundamental ideas behind continued fractions can be traced back to Euclid's algorithm for finding the greatest common divider (GCD) of two integers. Cataldi[59] is first to mention continued fractions in 1613. The theory of continued fractions has its roots in the works of Euler[60], Lagrange[61], Legendre, Lambert, and many other mathematicians during previous centuries.

This century Khinchin[62] studies regular continued fractions as a number representation, and gives the distribution of values of partial quotients found by Kuzmin in 1928. Wall[63] summarizes the analytic theory of continued fractions with an emphasis on convergence and function theory. The best discussion of the theory of continued fractions is given by Perron[12].

Homographic function evaluation with simple continued fractions, as discussed in this thesis, was first studied by Hurwitz ([13], 1896). Hall[14] follows up on Hurwitz's work, and Raney[16] shows a simplified algorithm based on linear algebra and finite state machines.

Meanwhile Knuth[64] considers continued fractions for semi-numerical algorithms.

Gosper[15] notes how to evaluate homographic functions to CFs, given CF inputs. Vuillemin[45] formalizes and extends Gosper's work to the *algebraic* and *positional algebraic* algorithms. Kornerup[25] investigates a hardware implementation of Gosper's algorithm. A more theoretical use of CFs and the 2-dimensional, quadratic, rational form is suggested by Potts. Potts[68] extends the notion of representing exact real numbers with expression trees of tensors (rational forms).

Bibliography

[1] M.J. Flynn, *Computer Architecture, Pipelined and Parallel Processor Design,* Jones and Bartlett Pubs., Inc., 1995.

[2] S. Waser, M.J. Flynn, *Introduction to Arithmetic for Digital Systems Designers,* Holt, Rinehard & Winston, New York, 1982.

[3] N.H.E. Weste, K. Eshraghian, *Principles of CMOS VLSI Design,* Addison-Wesley Pubs, 1993.

[4] K. Hwang, *Computer Arithmetic, Principles, Architecture and Design,* John Wiley and Sons, New York, 1978.

[5] I. Koren, *Computer Arithmetic Algorithms,* Prentice Hall, 1993.

[6] A. Omondi, *Computer Arithmetic Systems,* Prentice Hall, 1994.

[7] J.E. Robertson, *A new class of digital division methods,* IRE Trans. Electron. Computers, EC-5, p.65-73, June 1956.

[8] K. D. Tocher, *Techniques of Multiplication and Division for Automatic Binary Computers,* Quarterly Journal of Mechanics and Applied Mathematics, Vol. 11, p. 364-384, 1958.

[9] IEEE, *ANSI/IEEE Std 754-1985, IEEE Standard for Binary Floating-Point Arithmetic,* 1985.

[10] J.L. Hennessy, D.A. Patterson, *Computer Architecture: A Quantitative Approach,* Morgan Kaufmann, San Mateo, CA, 1990.

[11] C. Arnold, *Formal Continued Fractions Solutions of the Generalized Second Order Riccati Equations, Applications,* Numerical Algorithms 15, p. 111, J.C. Baltzer A.G., 1997.

[12] O. Perron, *Die Lehre von den Kettenbrüchen,* Band I,II , Teubner Verlag, Stuttgart, 1957.

[13] A. Hurwitz, *Über die Kettenbrüche, deren Teilnenner arithmetische Reihen bilden,* Vierteljahrsschrift d. naturforsch. Gesellschaft, Zürich, Jahrgang 41, 1896.

[14] M. Hall, *On the sum and product of continued fractions,* Ann. of Math. 48, 966-993, 1947.

[15] R.W. Gosper, R. Schroeppel, M. Beeler, *HAKMEM,* Continued Fraction Arithmetic, MIT AI Memo 239, Feb. 1972.

[16] G.N. Raney, *On Continued Fractions and Finite Automata,* Math. Ann. 206, 265-283, 1973.

[17] W.B. Jones, W.J. Thron, *Continued Fractions: Analytic Theory and Applications,* Encyclopedia of Mathematics and its Applications, Vol. 11, Addison-Wesley, Reading, Mass., 1980.

[18] A. Bracha-Barack, *Application of Continued Fractions for Fast Evaluation of Certain Functions on a Digital Computer,* IEEE Trans. on Computers, March 1974.

[19] M.D. Ercegovac, K.S. Trivedi, *On-line algorithms for division and multiplication,* IEEE Trans. on Computers, C-26(7), 1977.

[20] M.D. Ercegovac, K.S. Trivedi, *On-line arithmetic: An Overview,* SPIE Real Time Signal Processing VII, p. 86-93, Bellingham, Washington, 1984.

[21] J.E. Robertson, K.S. Trivedi, *The Status of Investigations into Computer Hardware Design Based on the Use of Continued Fractions,* IEEE Trans. on Computers, Vol. C-22, No. 6, June 1973.

[22] J.E. Robertson, K.S. Trivedi, *On the Use of Continued Fractions for Digital Computer Arithmetic,* IEEE Trans. on Computers, July 1977.

[23] R.B. Seidensticker, *Continued Fractions for High-Speed and High-Accuracy Computer Arithmetic,* IEEE Symposium on Computer Arithmetic, 1983.

[24] W.B. Jones, A. Steinhardt, *Digital Filters and Continued Fractions,* Lect. Notes In Math, No. 932, p. 129, Springer-Verlag, Berlin, 1982.

[25] P. Kornerup, D.W. Matula, *An algorithm for redundant binary bit-pipelined rational arithmetic,* IEEE Trans. on Computers, Vol. 39, No. 8, Aug. 1990.

[26] Waterloo Maple Inc., *Maple V,*
http://www.maplesoft.com/.

[27] American Mathematical Society, *MathSciNet,*
http://ams.rice.edu/mathscinet/.

[28] O. Mencer, M. Morf, M. J. Flynn, *Precision of Semi-Exact Redundant Continued Fraction Arithmetic for VLSI,* SPIE Advanced Signal Processing Algorithms, Architectures, and Implementations IX, Denver, July 1999.

[29] O. Mencer, L. Semeria, M. Morf, J.M. Delosme, *Application of Reconfigurable CORDIC Architectures,* The Journal of VLSI Signal Processing, Special Issue: VLSI on Custom Computing Technology, Kluwer, March 2000.

[30] O. Mencer, M. Morf, A. Liddicoat, M. J. Flynn, *Efficient Digit-Serial Rational Function Evaluation and Digital Filtering Applications,* Asilomar Conference on Signals, Systems, and Computers, California, Nov. 1999.

[31] W.H. Mangione-Smith, B. Hutchings, D. Andrews, A. DeHon, C. Ebeling, R. Hartenstein, O. Mencer, J. Morris, K. Palem, V. Prasanna, H. Spaanenburg, *Seeking Solutions in Configurable Computing,* IEEE Computer Magazine, December 1997.

[32] O. Mencer, M. Morf, M. Flynn, *Hardware Software Tri-Design of Encryption for Mobile Communication Units*, IEEE ICASSP, Seattle, May, 1998.

[33] O. Mencer, M. Morf, M. J. Flynn, *PAM-Blox: High Performance FPGA Design for Adaptive Computing*, IEEE Symposium on FPGAs for Custom Computing Machines, Napa Valley, CA, 1998.
http://arithmetic.stanford.edu/PAM-Blox/

[34] I. Koren, O. Zinaty, *Evaluating Elementary Functions in a Numerical Coprocessor Based on Rational Approximations,* IEEE Trans. on Computers, Vol. 39, No. 8, Aug. 1990.

[35] A.N. Khovanskii, *The application of continued fractions and their generalizations to problems in approximation theory,* 1956, translated by Peter Wynn, Groningen, Netherlands, P. Noordhoff, 1963.

[36] R. Stephanelli, *A suggestion for a high-speed parallel binary divider,* IEEE Trans. on Computers, Vol. 21, No. 1, Jan. 1972.

[37] E.M. Schwarz with M. Flynn, *High-Radix Algorithms for High-Order Arithmetic Operations,* PhD Thesis, E.E. Dept., Stanford, Jan. 1993.

[38] M. Schulte and J. Stine, *Symmetric Bipartite Tables for Accurate Function Approximation,* IEEE Symposium on Computer Arithmetic, 1997.

[39] D. Das Sarma and D.W. Matula *Faithful Bipartite ROM Reciprocal Tables,* IEEE Symposium on Computer Arithmetic, 1995.

[40] L. Lorentzen, H. Waadeland, *Continued Fractions with Applications,* Studies in Computational Mathematics 3, North-Holland, 1982.

[41] S. Oberman with M.J. Flynn, *Design issues in high performance floating point arithmetic units* PhD thesis, Dept. of Electrical Engineering, Stanford, Nov. 1996.

[42] N. Quach, M.J. Flynn, *Leading One Detection – Implementation, Generalization, and Application* Techical Report, CSL-TR-91-463, Stanford, March 1991.

[43] E.Y. Remez, *General Computational Methods of Chebyshev Approximation,* Kiev, 1957. (see also L.A. Lyusternik, *Computing Elementary Functions,* Pergamon Press, 1965.)

[44] E.E. Schwartzlander, A.G. Alexpoulos *The sign-logarithm number system,* IEEE Trans. on Computers, Dec. 1975.

[45] J.E. Vuillemin, *Exact Real Computer Arithmetic with Continued Fractions,* IEEE Trans. on Computers, Vol. 39, No. 8, Aug. 1990.

[46] G. Wei-Bo, Y. Hong, *Rational Approximants for Some Performance Analysis Problems,* IEEE Trans. on Computers, Vol. 44, No. 12, Dec. 1995.

[47] D.W. Matula, P. Kornerup, *Finite precision rational arithmetic: Slash number systems* IEEE Trans. on Computers, C-34(1):3-18, Jan. 1985.

[48] J. Fandrianto, *Algorithm for high speed shared radix 8 division and radix 8 square-root,* IEEE Symposium on Computer Arithmetic, Sept. 1989.

[49] C. Farnum, *Compiler Support for Floating-Point Computation,* Software Practices and Experience, vol. 18, no. 7, 1988.

[50] M.D. Ercegovac, T. Lang, *Division and Square Root, Digit-Recurrence Algorithms and Implementations,* Kluwer Academic Pubs, Massachusetts, 1994.

[51] M.D. Ercegovac, *A General Method for Evaluation of Functions and Computation in a Digital Computer,* Ph.D. thesis, Dept. of Computer Science, University of Illinois, Urbana-Champaign, 1975.

[52] M.D. Ercegovac, *A General Hardware-Oriented Method for Evaluation of Functions and Computations in a Digital Computer,* IEEE Trans. on Computers, C-26, No. 7, July 1977.

[53] A. Avizienis, *Signed-digit number representations for fast, parallel arithmetic,* IRE Trans. Electron. Comput., EC-10, pp. 389-400, Sept. 1961.

[54] T.C. Chen, *Automatic Computation of Exponentials, Logarithms, Ratios and Square Roots,* IBM Journal of Research and Development, July 1972.

[55] H.M. Ahmed, *Signal Processing Algorithms and Architectures,* PhD Thesis (with M. Morf), E.E. Dept., Stanford, June 1982.

[56] J.E. Volder, *The CORDIC Trigonometric Computing Technique,* IRE Trans. on Electronic Computers, Vol. EC-8, No. 3, pp. 330-334, Sept. 1959.

[57] J.S. Walther, *A Unified Algorithm for Elementary Functions,* AFIPS Conf. Proc., Vol. 38, pp. 379-385, 1971.

[58] T.C. Chen, *Automatic Computation of Exponentials, Logarithms, Ratios and Square Roots,* IBM Journal of Research and Development, July 1972.

[59] P.A. Cataldi, *Trattato del modo brevissimo di trovare la radice quadra delli numeri, et regole di approssimarsi di continuo al vero nelle radici dei numeri non quadrati, con le cause et inventioni loro,* Bologna, (Italy), 1613, see [12].

[60] L. Euler, *Introductio in analysin infinitorum I,* 1748, translated into English, Springer Verlag, New York, 1980.

[61] J.L. Lagrange, *Additions aux éléments d'algebre d'Euler,* 1798.

[62] A. Khinchin, *Continued Fractions,* 1935, translated from Russian, The University of Chicago Press, 1964.

[63] H.S. Wall, *Analytic Theory of Continued Fractions,* Chelsea Publishing Company, Bronx, N.Y., 1948.

[64] D.E. Knuth, *The Art of Computer Programming, Vol 2, Seminumerical Algorithms,* Addison-Wesley, Reading, Mass., 1969.

[65] B.C. Berndt, *Ramanujan's Notebooks,* Springer-Verlag, 1998.

[66] C. Faivre, *On decimal and continued fraction expansion of a real number,* Acta Arithmetica LXXXII.2(1997).

[67] C.T. Fike, *Computer evaluation of mathematical functions,* Englewood Cliffs, N.J., Prentice-Hall, 1968.

[68] P.J. Potts with A. Edalat, *Exact real arithmetic using Möbius transformations,* PhD Thesis, Imperial College, London, March 1999.

[69] P. Flajolet, B. Vallee, *Continued fraction algorithms, functional operators, and structure constants* Theor.Comp.Sci. 194, 1-34, Elsevier Science, 1998.

[70] J.M. Muller, *Elementary Functions, Algorithms and Implementation,* Birkhaeuser, Boston, 1997.

[71] SYNOPSYS Products, *RTL Analyzer, http://www.synopsys.com/products/*

[72] WWWebster Dictionary, *Merriam-Webster's Collegiate Dictionary, http://www.m-w.com/cgi-bin/dictionary*