A Multi-Table Approach to Floating-Point Function Approximation

Lucas M. Dutton
McMaster University
duttonl@mcmaster.ca

Christopher Kumar Anand
McMaster University
anandc@mcmaster.ca

Robert Enenkel
IBM Canada
enenkel@ca.ibm.com

Silvia Melitta Müller
IBM Germany
smm@de.ibm.com

***Abstract*—AI, analytics and database performance depend on two types of hot functions: linear algebra, and non-linear functions. In this paper, we describe a novel method of accelerating the approximation of non-linear functions, with worked-out examples for divide, reciprocal, square root, and reciprocal square root, all using three tables. These functions are important enough that they are included in IEEE and language standards, and form the basis of more complex functions such as activation functions in AI, hash functions used in databases, and optimization algorithms. The wide range of applications imposes different requirements on implementations in general-purpose hardware. For example, AI applications are sensitive to performance but tolerate lower precision, but other applications would fail without high accuracy and precision. We can meet both demands using a sequence of small table lookups and narrow integer multiplications to create low-latency, but highly accurate, approximations. For the studied functions, this approach can eliminate the need for iterative refinement or polynomial approximations in the single-precision floating point case, both of which entail longer latencies. For higher precision, the iteration depth or polynomial degree could be reduced. We use circuit block diagrams to demonstrate how multiple table lookups and small-integer multiplications can work in parallel to calculate piecewise polynomial approximations. This can result in lower latencies for single-precision computations than currently achieved by reciprocal and reciprocal-square-root estimates—e.g., frsqrtes on POWER—making the proposed hardware a drop-in replacement (and improvement) for current estimate implementations. We estimate an overall execution latency of four cycles, when fully pipelined, for the fixed-point core of a fixed or floating point unit of a current-generation, general-purpose CPU. We also discuss extensions to other floating point and fixed point precisions. In a companion artifact, we include C software emulations of the proposed hardware blocks and related tables for all four functions.**

***Index Terms*—accurate table method, divide, floating point, reciprocal, square root.**

I. Introduction

Algebraic functions are used in numerous computing applications. Activation functions in AI applications such as sigmoid have widened the scope of design for such functions. Algorithms for evaluating algebraic functions typically consist of three steps: 1. Computing an initial approximation from an input, 2. Refining the approximation to a desired accuracy, and 3. Rounding the output to the target precision. As the refinement step receives the initial approximation as input, the refinement algorithm's properties depend on the approximation's quality. An important design decision must be made between the accuracy of the approximation and the time taken to calculate the approximation.

State-of-the-art approximation algorithms (Step 1) heavily rely on lookup tables to return accurate approximation results. This is also performed in a three-step process: 1. Performing a range reduction on the input, 2. Using bits in the input as a key to extract the value from a lookup table, and 3. performing a polynomial approximation of the function near the table value. Designers are concerned with the table design and its impact on overall accuracy.

In this paper, we propose a novel approach to generating an approximation for a wide range of elementary functions, by which we mean $1/x$, $1/\sqrt{x}$, e^x, $\log(x)$, $\sin(x)$ and other functions which have a symmetry which can be used for range reduction, and continuous second derivatives [1]. Our contributions are as follows:

- We present an algorithm that achieves 22 bits of accuracy without polynomials, iterative algorithms or floating point hardware instructions. It acheives low latency by using parallel table lookups using the leading fractions bits as and index, and also in parallel does small-integer multiplications, using other bitfields in the input (Section IV).
- We show how to build the algorithm out of circuit blocks common in floating point units in Section V, and estimate the overall execution time of the implementation. The block diagram shows that we can achieve an overall execution time of four cycles when fully pipelined.
- We evaluate the accuracy of our method using a circuit emulation written with C, and present the results in Section VI.

Our discussion and results rely on the concept of units in last place (ulp), and all error bounds are presented using ulp errors as a unit of measurement. The unfamiliar reader is encouraged to explore the definitions given in Harrison [2], Muller [3] and Kahan [4]. Our ulp error measurements follow closely the definition given in Cornea-Hasegan et al. [5].

II. Background

Table-based methods of evaluating continuously twice-differentiable elementary functions in a large domain follow a standard recipe. First, a range reduction step is performed, reducing the initial problem to evaluating a function in one or several smaller intervals. In each of these intervals, the function can be approximated by a polynomial whose coefficients

are tabulated, or we can store the value of the function at some point in the interval, and use range-reduction identities applicable to the function of interest (e.g trigonometric identities) to reduce the problem to evaluating the function near zero.

The task of choosing the size of these intervals is a design tradeoff made by experiment.

1) Large intervals translate to smaller tables and a simpler range reduction step. However, this translates to polynomials of larger degrees, thus requiring longer evaluation times.
2) Small intervals would make polynomial evaluation faster, but require larger tables and possibly several range reduction steps.

Gal [6]'s accurate tables method tabulates the function at values that are very close to a machine number. Such tables can store values in precisions close to the target output precision, and are highly accurate. With accurate tables, one must deal with the Table Maker's dilemma [7], which states that there does not exist a general method to predict how many bits to round to some preassigned number of bits correctly for a given function. It also requires full-precision floating-point calculations, and cannot compete with dedicated hardware on speed.

III. Related Work

Table-based approaches to compute approximations for elementary functions are widely used. For existing approximation algorithms, the design of the table must be taken into consideration. Factors such as the size of the table, the width of the table (i.e. the bit length of the table values), and the interval of the table, are design parameters that are chosen in conjunction with the associated series iteration to yield as many accurate values as possible.

Gal [6]'s accurate tables method is closely related to our work. It uses a specially chosen lookup table and interpolation involving several subintervals to generate last-bit accurate values of elementary functions, and can be combined with a final series approximation as well. Tang [8, 9, 10, 11] proposed a "table-driven" method for several elementary functions, providing a framework for implementing these functions using table-lookup algorithms, and also provides numerical analysis on their error bounds.

Kucukkabak and Akkas [12] describes an implementation of a reciprocal unit, which computes double precision floating-point reciprocals in eleven clock cycles. The implementation makes use of a $2^{10} \times 20$ bits table followed by two Newton-Raphson iterations. Our method is similar in its design, but eschews the use of polynomials, and instead of a single large table we use three compact tables.

Since then, other authors have published improved results. Chen et al. [13] uses $2^7 \times 16$ bits tables, and the latency has been reduced to 10 cycles. Habegger et al. [14] reduced the number of Newton-Rhapson iterations to one second-order polynomial.

A survey paper by Muller [1] discusses methods for approximating elementary functions, including a discussion on the bipartite table method, which involves splitting the input into separate parts and using different tables based on the computation desired. An example taken from his paper is given as follows: Given a $3k$-bit input x, it is split into three k-bit numbers x_0, x_1 and x_2, and instead of using a $3k$-address-bit table for storing $f(x)$, they use $2k$-address-bit tables, storing the values $A(x_0, x_1) = f(x_0 + 2^{-k}x_1)$ and $B(x_0, x_2) = 2^{-2k}x_2 \cdot f'(x_0)$. Then $f(x)$ is approximated by summing the two values. This differs from our method in two ways: 1. The values stored in the S-table and A-table are error values instead of tabulated values from a function, and 2. all three tables use the same lookup index.

The methods above can be implemented in both hardware and software. Wong and Gogo [15] proposed a method that is specialized for hardware implementations, in the double-precision floating-point format. They use a "rectangular multiplier", which is a 16×56 multiplier that truncates the result to 56 bits.

There are also non-table-based methods to compute approximations to functions. Digit recurrence algorithms such as SRT division as proposed in MacSorley [16], Robertson [17], Tocher [18] have been implemented in many modern floating-point units, and have been analyzed in works such as Harris et al. [19].

IV. Design

Our design aims to provide approximations for single-precision floating-point elementary functions, here illustrated for reciprocal. It contains three lookup tables, each fulfilling a different purpose in the algorithm. Subsequent steps involve correction steps using the values from the lookup tables in the calculation of the approximation.

A. Tables

Figs. 1 and 2 illustrate the interaction between the different table values. First, we derive the lookup key i from the input x. The same lookup key is used to index each table, producing values `t[i]`, `s[i]` and `a[i]`.

1) The T-table consists of regularly-spaced fixed-point values of the function interval;
2) The S-table stores the fixed-point coefficient of linear correction relative to the values of the T-table;
3) The A-table stores the fixed-point coefficient of quadratic correction relative to the coefficient of the S-table

1) T-table: The first table, referred to as the T-table, includes values from the function of interest in a predetermined interval. For example, an interval within one binade such as $[0.5, 1)$, $[1, 2)$, $[2, 4)$, or a combination of two such adjacent intervals are chosen, depending on the function of interest. We typically compute T-table values to have a few extra bits of precision to improve rounding later in the algorithm.

2) S-table: The second table, referred to as the S-table, contains values approximately equal to the difference between adjacent T-table values, i.e. $S[i] \approx T[i+1] - T[i]$, with the table value chosen to improve accuracy. The S-table value is

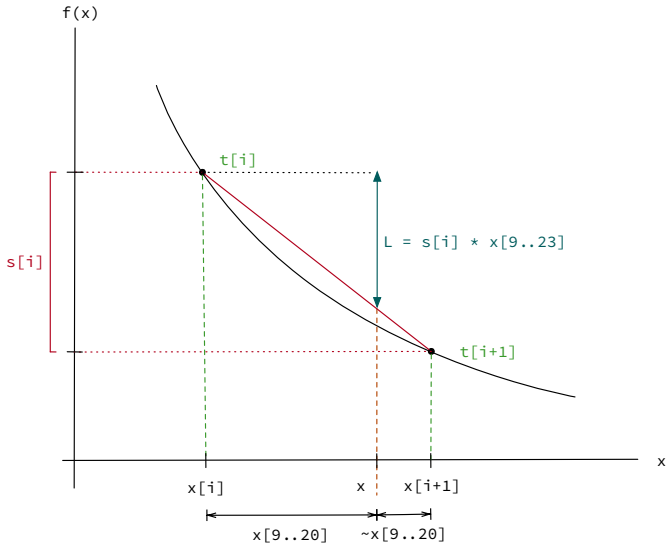

Fig. 1: A visual representation of how the first two table values for reciprocal are used to calculate a fixed-point, piecewise linear function. The leading 8 bits of the fraction determine the interval number $i = x[1..8]$. The first table value $t[i]$ is an approximation of reciprocal at $x[1..8]$, the value represented by the first 8 fraction bits of the input x. The table value $s[i]$ is approximately the negative of the difference between $t[i]$ and $t[i+1]$, i.e., approximately the height of the chord between the values at either end of the interval of approximation. The linear approximtion at x, is calculated as $t[i] - s[i] * x[9..23]$.

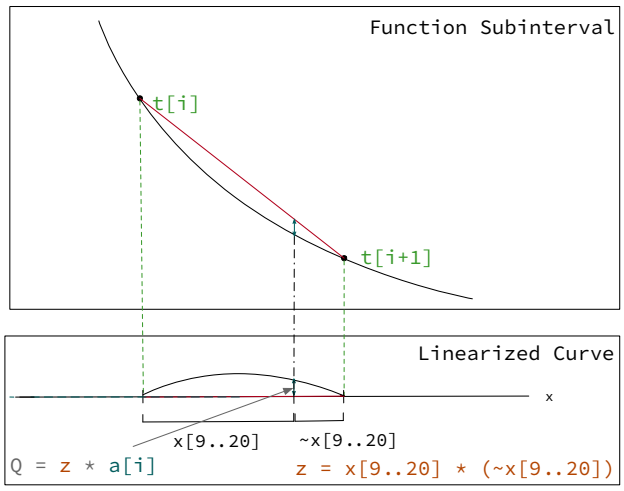

Fig. 2: Visualization of the quadratic correction to reciprocal. The top graph shows the error in the linear approximation being corrected. The bottom graph shows how $x[9..20]$, interpreted as fractional bits determine the relative position of x in the interval $[x[i], x[i+1])$ measured from the left, while the complemented bits, when interpreted as unsigned fraction bits are the relative distance measured from the right. The product of the two is the unscaled quadratic correction, and the table value $a[i]$ is the scaling of the correction.

used in the calculation of a linear correction to the T-table value.

3) A-table: The third and final table, referred to as the A-table, includes a small correction applied to the S-table value, and represents a quadratic approximation to the residual error of the function on the subinterval. For $1/x$ and $1/\sqrt{x}$, the second derivative is positive, so the residual is greatest near the middle of the subinterval, and decreases monotonically to either side. In Fig. 2, the correction is seen to be quadratic, because it is the product of the distance between the evaluation point and the ends of the interval, and taking the residual at the middle point provides a good estimate for the A-table value.

One advantage we gain from this design is the sizes of the tables are the same, due to their relationship with each other, so the lookup circuits could be shared, and the tables could be merged. The values stored in the S-table and A-table are *correction* values—thus the final result is calculated not from a polynomial evaluation, but rather by adding parallelly computed correction values to the T-table value. Due to the nature of the correction applied, the function in question must be continuous in the chosen interval, and should be well-approximated by its quadratic Taylor series. Fortunately, most elementary functions including sqrt, exp and div can use this method as they do have these qualities, and are infinitely differentiable.

The task is to choose the size and widths of the tables. The widths can be chosen independently. For a fixed accuracy target, including the quadratic correction allows the tables to be smaller than they would be with only a linear correction. Table size and width will vary with the function and accuracy requirements. For single-precision reciprocal, we determined empirically that the following table sizes are sufficient:

- T-table: 26-bit wide, 256 values
- S-table: 18-bit wide, 256 values
- A-table: 5-bit wide, 256 values

Note that the T-table is wider than the target precision, i.e., 24 bits, while the other tables are significantly narrower.

B. Algorithm

Algorithm 1 embodies the entire procedure of calculating the approximation given a single-precision input x.

Algorithm 1 Three-table Procedure

Input: x a fixed-point single-precision number
Output: $y = \mathtt{approx}(f(x))$

$t \leftarrow T[x[1..8]]$
$s \leftarrow S[x[1..8]]$
$a \leftarrow A[x[1..8]]$
$z \leftarrow x[9..20] * (\sim x[9..20])$ ▷ (1)
$Q \leftarrow z * a$ ▷ (2)
$L \leftarrow s * x[9..23]$ ▷ (3)
$y \leftarrow t - L - Q$ ▷ (4)

The algorithm can be explained in terms of traditional table-based methods. First, the range reduction step is encapsulated

in the "Require" clause of Algorithm 1. We can decompose the single-precision input into its exponent and fraction, and work within a selected interval of interest, using the fraction in our fixed-point representation and saving the exponent to be combined at a later step. Converting to and from IEEE floating-point is already performed efficiently in commercial processors, and hence outside the scope of this paper.

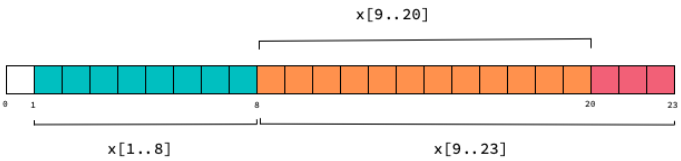

Fig. 3: The bit-field layout of the fixed-point input to reciprocal. The important bit-fields we extract are labeled in the figure: Bits $x[1..8]$, $x[9..20]$ and $x[9..23]$.

Fig. 3 shows the layout of the fixed-point input and associated bit-field indexing. We use the notation $x[i..j]$ to mean bit-field indexing of x from bit i to j, inclusive on both ends, where $i = 1$ is the most-significant fraction bit of x. A total of 3 different bit-fields are used in Algorithm 1: $x[1..8]$ is used as the index to all three tables, $x[9..20]$ is used in the "quadratic" term (precisely, its product with its one's complement), and $x[9..23]$ is used in the computation for L. These values are chosen in accordance to the table design in Section IV-A; for example, $x[1..8]$ can only have 256 possible values, which corresponds to the size of the tables.

Variables t, s and a correspond to the values looked up from the T-table, S-table and A-table respectively. The rest of the algorithm consists of a sequence of calculations:

1) The variable z is an approximation of the quadratic function $(x-x_i)(x_{i+1}-x)$, which is zero at the boundaries of the interval, and positive inside the interval. The actual computation we use is the multiplication of $x[9..20]$ (in place of $x - x_i$) with the one's complement $\sim x[9..20]$ (in place of $x_{i+1} - x$). This is an approximation because we ignore the lowest-order bits, and because the one's complement differs from the negation by the addition of a 1. This is visualized in Fig. 2.
2) The quadratic correction Q is then obtained by multiplying with a, the coefficient obtained from the A-table. The key idea behind this mechanism can be explained by looking at the quadratic curve in Fig. 2, and noticing that the roots are at the endpoints of the interval. The product is largest when x is at the midpoint of the interval.
3) The linear correction L is calculated by taking the product of s and $x[9..23]$. We use bits 9-23 of x that have not already been used to index the tables. The value L effectively moves along the linear slope given by s.
4) The approximation y is then computed by subtracting the linear and quadratic corrections L and Q from t.

Algorithm 1 is presented in a way that clearly demonstrates the computation of the linear and quadratic correction, and how they relate to the final result. Later in Section V, we rearrange the computation to optimize the execution time of the algorithm.

C. Table Search

Although the three tables have a geometrical interpretation as being a stepwise approximation and linear and quadratic corrections to that approximation, we do not need to calculate the table values by solving algebraic equations. Instead, we can use the initial function value in each interval as a candidate T-value, the difference of neighbouring T-values as a candidate S-value, and $1/4$ of the difference between the function and the linear approximation at the midpoint of the interval as a candidate A-value. We then perform an exhaustive search in an interval of these candidate values, using the best combination of table values based on the maximum absolute error on the corresponding subinterval.

D. Extension to Integers

Our method as described applies to single-precision floating-point elementary functions. A key feature of the algorithm is that intermediate calculations are all performed in a fixed-point format, leaving the exponent to be handled separately.

It is this feature that allows us to extend this algorithm to work on high-precision integers. This extension does not modify the existing algorithm, only requiring the input to be coerced into the format expected by the algorithm, and a subsequent polynomial refinement step (typically a Taylor polynomial, where the order can be adjusted based on accuracy requirements and the integer precision).

The conversion of a unsigned 64-bit integer input to the fixed-point format expected by Algorithm 1, and be performed using count-leading-zeros and shift operations. Negative inputs can be handled after taking the absolute value.

E. Extension to other Floating-Point Formats

The most common use case when applying this method to other floating-point formats is to calculate IEEE double-precision floating-point functions. Unlike in Section IV-D, the input does not have to be converted; we only need to extract a portion of the fractional bits of the input to be passed into the approximation algorithm, and the double-precision output can then be computed by refining the approximation using a polynomial or iterative method. By starting with at least 22 bits of accuracy, the polynomial order or iteration count will be reduced.

V. Hardware Implementation

Algorithm 1 can be translated to an efficient hardware implementation with circuit blocks, as shown in Fig. 4 for reciprocal.

- The lookup blocks of the T-table, S-table and A-table are done in parallel, and derive the lookup key from the same input bits, i.e. $x[1..8]$. Also in parallel, the multiplication of $x[9..20]$ and $\sim x[9..20]$ is computed, with the latter operand taken as the one's complement

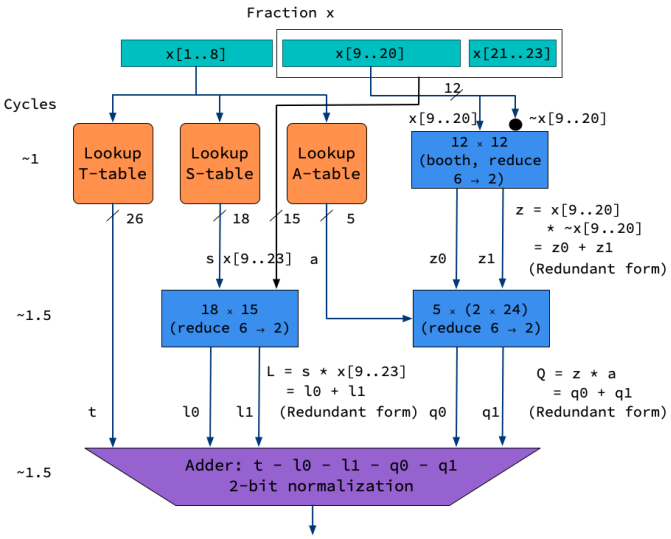

Fig. 4: Circuit diagram of the Three-table method applied to recip. The three dark-blue rectangles are Booth multipliers which produce results in redundant form which are used by the next stage in that format. The black dot indicates negation of the input.

of $x[9..20]$. The multiplication block can utilize Booth's algorithm [20] and provides the products as two vectors in a carry-save format. These operations are executed within approximately one cycle.

- The 18-bit linear correction term s and bits $x[9..23]$ are given as inputs to the next multiplication block. This multiplication is also performed in carry-save format, producing vectors $l0$ and $l1$ as results. In parallel, the 5-bit quadratic correction term a and the two output vectors of the multiplication block in the previous cycle are multiplied, producing vectors $q0$ and $q1$ in carry-save format. Both multiplications can be performed in approximately 1.5 cycles.
- The linear correction L, consisting of vectors $l0$ and $l1$, and the quadratic correction Q, consisting of vectors $q0$ and $q1$, are provided to the final adder block, which computes the output y to the approximated function. The addition also takes approximately 1.5 cycles to compute.

The hardware implementation is efficient, parallelizing multiple operations, and uses common arithmetic circuits and algorithms such as the carry-save format and Booth's multiplication algorithm. What is not covered is the handling of the sign and exponent, which is done separately and can be combined with the result y. Denormalized floating-point numbers can also be handled at the cost of approximately 1 additional cycle.

A. Bit-alignment of Operations

The hardware diagram of Fig. 4 specifies the bit-widths of each operation from one circuit block to another. To understand the C-code emulation, the artifact includes comments about the place values for intermediate values, and a visualization of how they align. All bit-shifts are by constant amounts, so the C operations may not need any additional gates.

Note that the details of bit-alignment and bit-shifting are not present in the circuit diagram in Fig. 4, but all bit-alignment and bit-shift operations are by constant amounts, so they can be implemented in hardware without a penalty to execution time and circuit area.

Function	Input range	Ulp error (SP)	comparable†
recip	$[1, 2)$	$[-1, 3]$	2.5
sqrt	$[1, 4)$	$[-1.5625, 0.4375]$	1.8
rsqrt	$[1, 4)$	$[-0.625, 2.75]$	1.3
div	$[1, 2)$	$[-4, 4]$	3.0

TABLE I: Functions and their ulp error bounds in single precision, showing the input range tested. The error bounds were determined by exhaustive testing for all functions except divide, for which random testing was used. †POWER 7 Vector MASS single-precision library absolute Ulp error (https://www.ibm.com/support/pages/node/318393)

VI. Evaluation

The algorithm in Section IV and the hardware implementation in Section V both use single-precision reciprocal as the running example. We have also implemented this method for single-precision division, square root and reciprocal square root as a software emulation in C code. We also created test suites that compute the approximation for inputs within a chosen interval, and reports the ulp error of each test. All results are tested with the standard `clang` compiler.

In the artifact, we provide C code to illustrate this method applied to single- and double-precision floating point, as well as test code. Comments in the code map variables to the quantities in Fig. 4. The table sizes and other factors are parametrized in the code to facilitate experiments when designing for applications with different precision and accuracy requirements. By using 128-bit integers, the code can simulate up to 64-bit floating point computations.

In a separate preprint, we describe a simple procedure to correct errors in all of the functions in this paper resulting in correctly rounded results [21]

The extensions as described in Section IV-D and Section IV-E have also been implemented for division and square root, for double-precision floating point numbers, `int64` and `int128`, by computing a Newton iteration to refine the initial approximation.

VII. Conclusions and Future Work

We have presented a multi-table approach to approximating elementary functions. This method uses three separate tables, providing an initial approximation, a linear correction term, and a quadratic correction term respectively. The approximation result is then computed using the correction terms and a series of calculations, without the use of a polynomial.

The proposed hardware implementation is efficient, requiring an estimated four cycles to compute an approximation

when fully pipelined. It is flexible as it applies to all elementary math functions, and implementable using parallel circuits in existing architectures.

Future work should (1) extend this method for other functions, such as exponential and logarithm, and extend to both higher and lower precisions; and (2) give better estimates for latency by implementing in verilog or vhdl. We have not implemented lower precision tables, but many AI applications have lower requirements on accuracy and prcision, and the existing method could be applied to produce simpler or faster circuits. The existing method can also be modified to accommodate AI-specific floating point formats such as bfloat16 [22] and DLFloat [23].

ACKNOWLEDGEMENT

Thanks to Sharon Wu for contributing to the development of the test programs.

REFERENCES

[1] J.-M. Muller, "Elementary functions and approximate computing," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2136–2149, 2020.

[2] J. Harrison, "A Machine-Checked Theory of Floating Point Arithmetic," in *Theorem Proving in Higher Order Logics*, Y. Bertot, G. Dowek, L. Théry, A. Hirschowitz, and C. Paulin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, vol. 1690, pp. 113–130.

[3] J.-M. Muller, "On the definition of ulp (x)," Ph.D. dissertation, INRIA, LIP, 2005.

[4] W. Kahan, "A Logarithm Too Clever by Half," https://people.eecs.berkeley.edu/~wkahan/LOG10HAF.TXT, 2004.

[5] M. Cornea-Hasegan, R. Golliver, and P. Markstein, "Correctness proofs outline for Newton-Raphson based floating-point divide and square root algorithms," in *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No.99CB36336)*. Adelaide, SA, Australia: IEEE Comput. Soc, 1999, pp. 96–105.

[6] S. Gal, "Computing elementary functions: A new approach for achieving high accuracy and good performance," in *Accurate Scientific Computations*, ser. Lecture Notes in Computer Science, W. L. Miranker and R. A. Toupin, Eds. Berlin, Heidelberg: Springer, 1986, pp. 1–16.

[7] V. Lefèvre, C. N. S. D. Lyon, J.-M. Muller, J.-m. Muller, A. Tisserand, and A. Tisserand, "The Table Maker's Dilemma," 1998.

[8] P.-T. P. Tang, "Table-driven implementation of the exponential function in ieee floating-point arithmetic," *ACM Transactions on Mathematical Software (TOMS)*, vol. 15, no. 2, pp. 144–157, 1989.

[9] ——, "Table-driven implementation of the logarithm function in ieee floating-point arithmetic," *ACM Transactions on Mathematical Software (TOMS)*, vol. 16, no. 4, pp. 378–400, 1990.

[10] P. T. P. Tang, "Table-driven implementation of the expm1 function in ieee floating-point arithmetic," *ACM Transactions on Mathematical Software (TOMS)*, vol. 18, no. 2, pp. 211–222, 1992.

[11] ——, "Table-lookup algorithms for elementary functions and their error analysis," Argonne National Lab., IL (USA), Tech. Rep., 1991.

[12] U. Kucukkabak and A. Akkas, "Design and implementation of reciprocal unit using table look-up and newton-raphson iteration," in *Euromicro Symposium on Digital System Design, 2004. DSD 2004.* IEEE, 2004, pp. 249–253.

[13] D. Chen, B. Zhou, Z. Guo, and P. Nilsson, "Design and implementation of reciprocal unit," in *48th Midwest Symposium on Circuits and Systems, 2005.* IEEE, 2005, pp. 1318–1321.

[14] A. Habegger, A. Stahel, J. Goette, and M. Jacomet, "An efficient hardware implementation for a reciprocal unit," in *2010 Fifth IEEE International Symposium on Electronic Design, Test & Applications*. IEEE, 2010, pp. 183–187.

[15] W.-F. Wong and E. Gogo, "Fast hardware-based algorithms for elementary function computations using rectangular multipliers," *IEEE Transactions on Computers*, vol. 43, no. 3, pp. 278–294, 1994.

[16] O. L. MacSorley, "High-speed arithmetic in binary computers," *Proceedings of the IRE*, vol. 49, no. 1, pp. 67–91, 1961.

[17] J. E. Robertson, "A new class of digital division methods," *IRE transactions on electronic computers*, no. 3, pp. 218–222, 1958.

[18] K. D. Tocher, "Techniques of multiplication and division for automatic binary computers," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 11, no. 3, pp. 364–384, 1958.

[19] D. L. Harris, S. F. Oberman, and M. A. Horowitz, "Srt division architectures and implementations," in *Proceedings 13th IEEE Sympsoium on Computer Arithmetic*. IEEE, 1997, pp. 18–25.

[20] A. D. Booth, "A signed binary multiplication technique," *The Quarterly Journal of Mechanics and Applied Mathematics*, vol. 4, no. 2, pp. 236–240, 1951.

[21] L. M. Dutton, C. K. Anand, R. Enenkel, and S. M. Müller, "Inexactness and correction of floating-point reciprocal, division and square root," *arXiv preprint arXiv:2404.00387*, 2024.

[22] D. Kalamkar, D. Mudigere, N. Mellempudi, D. Das, K. Banerjee, S. Avancha, D. T. Vooturi, N. Jammalamadaka, J. Huang, H. Yuen, J. Yang, J. Park, A. Heinecke, E. Georganas, S. Srinivasan, A. Kundu, M. Smelyanskiy, B. Kaul, and P. Dubey, "A Study of BFLOAT16 for Deep Learning Training," Jun. 2019.

[23] A. Agrawal, S. M. Mueller, B. M. Fleischer, X. Sun, N. Wang, J. Choi, and K. Gopalakrishnan, "DLFloat: A 16-b Floating Point Format Designed for Deep Learning Training and Inference," in *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, Jun. 2019, pp. 92–95.