Hardware Fixed-Point 2D and 3D norms

Romain Bouarah, Florent de Dinechin
INSA Lyon - Inria, CITI (UR3720), 69621 Villeurbanne, France
{romain.bouarah, florent.de-dinechin}@insa-lyon.fr

***Abstract*—This article studies the hardware implementation of the Euclidean norm in 2 and 3 dimensions with fixed-point inputs and outputs. It compares the CORDIC shift-and-add algorithm to a "naive" architecture combining squarers, adders and square root, with a common specification: faithful accuracy. This specification is used in both cases to determine bounds on architectural parameters such as the number of CORDIC iterations and the bit-width of internal data-paths. Several architectural variants of the "naive" architecture are investigated. Their relevance domains are discussed based on synthesis results on FPGAs. For two dimensions, CORDIC has lower area but longer latency than a well-researched naive version. 3D variants of CORDIC, however, are worse than the naive architecture both in area and delay.**

***Index Terms*—Computer arithmetic, hardware operator, Euclidean norm, fixed point, CORDIC**

I. Introduction

The Euclidean norm appears in many application domains such as graphics, signal and image processing, and scientific simulations. It is used among others to compute Euclidean distance, to normalize vectors, or to convert from rectangular to polar coordinates. Each application has specific precision and performance requirements. To match these needs, this work studies two parametric techniques for computing fixed point Euclidean norms in hardware. The first is CORDIC, a classic fixed-point iterative computation. The second is derived from the definition of the N-dimensional norm:

$$\|(X_1, ..., X_N)\|_2 = \sqrt{\sum_{i=1}^{N} X_i^2} \tag{1}$$

It is therefore a naive combination of squarers, adders and square root. However this article attempts to implement this combination as cleverly as possible.

A. Common notations and setup

For the description of unsigned (resp. signed) fixed-point numbers, we use the notations $\text{ufix}(m, \ell)$ (resp. $\text{sfix}(m, \ell)$) where m and ℓ are integers denoting bit weights: the most significant bit (MSB) has weight 2^m, the least significant bit (LSB) has position 2^ℓ, and $m \geq \ell$ [4]. These bits are included, so the size of an $\text{sfix}(m, \ell)$ or $\text{ufix}(m, \ell)$ is $m - \ell + 1$ bits in both cases. The value 2^ℓ is also the unit in the last place (ulp) of the format. For $\text{sfix}(m, \ell)$, the most significant bit of weight 2^m is the sign bit.

This work was partially supported by the PEPR IA HOLIGRAIL project of the Agence Nationale de la Recherche, ANR-23-PEIA-0010

For any positive scaling factor s, we have $\sqrt{(sx)^2 + (sy)^2 + (sz)^2} = s\sqrt{x^2 + y^2 + z^2}$. We therefore consider without loss of generality that the fixed-point inputs are in the interval $[-1, 1)$. This corresponds to an input format $\text{sfix}(0, \ell_{\text{in}})$ where $\ell_{\text{in}} < 0$.

With inputs in $[-1, 1)$, the outputs belong to the interval $[0, \sqrt{2}]$ for the 2D norm and $[0, \sqrt{3}]$ for the 3D norm. The norm is obviously unsigned, and its MSB is a bit of weight 2^0 in both cases. Its format is therefore $\text{ufix}(0, \ell_{\text{out}})$ where ℓ_{out} is a parameter controlling the accuracy of the output. Note that when $\ell_{\text{in}} = \ell_{\text{out}}$, the input and output bit-width are identical.

B. Scope of this work

The operators studied here are parameterized by ℓ_{in} and ℓ_{out}, and their specification is to always return faithful (or last-bit accurate) results [6] [4, p. 76]: they have to return one of the two fixed-point numbers closest to the exact (often irrational) value of the norm. Equivalently, the difference between the returned fixed-point number and the exact norm should be strictly smaller than $2^{\ell_{\text{out}}}$.

An error analysis attempts to optimize architectural parameters while respecting this specification. This analysis is target independent and the resulting architectures are equally suited to ASICs and FPGAs. However, our comparisons target FPGAs only, and any conclusion drawn from these comparisons may not apply to ASICs (although the methodology will). In detail, all the results in this article have been obtained using Vivado 2024.1 targeting a Xilinx/AMD Kintex7, part xc7k70tfbv484-3, post synthesis. All the operators are combinatorial and can be pipelined, but due to lack of space only combinatorial results are reported.

This work is implemented in the FloPoCo core generator[1] as the Fix2DNorm and Fix3DNorm operators. It reuses many of the existing FloPoCo components.

C. State of the art

The CORDIC algorithm (**CO**ordinate **R**otation **DI**gital **C**omputer) was introduced by Volder in 1959 to compute trigonometric functions and norms using rotations [25]. An extensive reference for all CORDIC variants is [14]. The simplest CORDIC variants use only shift-and-add operations, and are well suited to FPGAs thanks to their support of fast-carry additions. This work focuses on this approach, contributing a rigorous method for determining the minimal datapath sizes that enable faithful rounding. Radix-4 [24] and

[1] www.flopoco.org, git branch articles/2025-arith-norms

2576-2265/25/$31.00 ©2025 IEEE
DOI 10.1109/ARITH64983.2025.00015

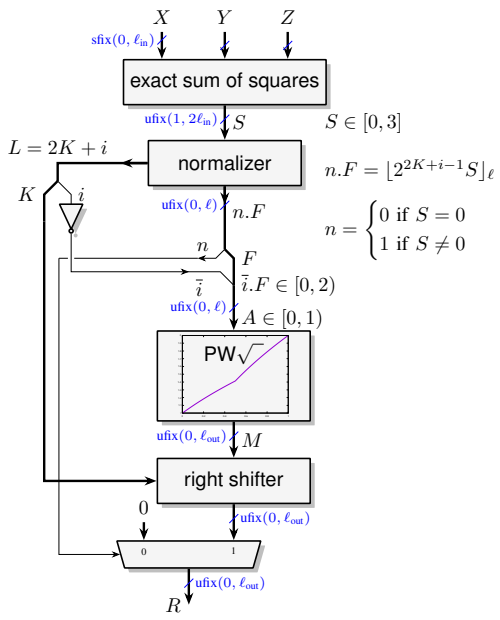


Fig. 1: High-level architecture of naive solution

higher [1] variants are left for future work – we discuss them briefly in Section V.

Beyond CORDIC, a handful of works have developed digit-recurrence techniques focused on the Euclidean norm in floating point [22] or in an online context [12]. In terms of cost and latency, these architectures are expected to be quite similar to high-radix CORDIC, and are also discussed in Section V.

The 2D norm has also been used in the evaluation of a generic method for two-input function evaluation [17]. However the results reported are really not competitive with the specialized methods studied here.

II. Naive architecture for 2D and 3D

This solution is naive in the sense that it consists in computing the sum of squares then taking the square root of the sum.

For n-bit signed inputs ($n = 1 - \ell_{in}$), the exact squares are $2n$ bits, and their sum S is $2n + 1$ bits. This entails two problems. Firstly, in the typical case where the inputs and output have the same size n, we have internally to compute the square root of a $2n$-bit number. Secondly, this large input can be arbitrarily close to 0, where the square root has a singularity (infinite derivative). This prevents the use of polynomial approximation methods and SRT [7] techniques. A solution to both problems is a range reduction to a kind of internal floating-point format. We first describe it, then analyze its numerical behaviour before discussing architectural variants for the subcomponents.

A. Overview

The proposed implementation is shown in Fig. 1.

It begins with an exact sum of squares, described in more details in Section II-C. The exact $S = X^2+Y^2(+Z^2)$ belongs to $[0, 2]$ in the 2D case and $[0, 3]$ in the 3D case: its format is ufix$(1, 2\ell_{in})$ in both cases.

A first idea is to compute the leading zero count (LZC) L of S, write it $L = 2K+i$ where i is its LSB, shift the input by $2K$ to obtain a quasi-normalized mantissa $S' \in [1, 4)$, and evaluate $M \approx \sqrt{S'}$. Then we have $\sqrt{S} \approx 2^K M$. An issue with this approach is that the interval $[1, 4)$ is not hardware-friendly. For instance if we tabulate $\sqrt{S'}$, one fourth of the address space is unused. The following is a slightly more complex approach that solves this issue.

The normalizer component is a combined leading zero counter and shifter [4, p. 323] which computes a floating-point representation of the fixed-point input. If it were exact, it would output a normalized mantissa S' of the form $1.F$ (a ufix$(0, 2\ell_{in} - 1)$) and a leading zero count L with the identity

$$S = 2^{-L+1} S' \tag{2}$$

where the $+1$ is due to the MSB position being 1 in the input.

However, we use a truncating variant that outputs a ufix$(0, \ell)$, in other words it keeps only the $1-\ell$ most significant bits. The value of l will be determined in Section II-B. The truncated mantissa output of the normalizer is thus an ufix$(0, \ell)$ number $n.F$:

$$n.F = \lfloor S' \rfloor_\ell = \lfloor 2^{L-1} S \rfloor_\ell \tag{3}$$

Here n, the leading bit, is zero only if S was itself zero: this bit controls a mux that returns 0 in this case (bottom of Fig. 1).

Otherwise, $n = 1$, the value of $n.F$ is $1+F$ with $F \in [0, 1)$. We also decompose L as $L = 2K + i$, where i is the least significant (parity) bit of L, such that

$$\sqrt{S} \approx \sqrt{2^{-2K-i+1}(1 + F)} = 2^{-K} \sqrt{2^{1-i}(1 + F)}. \tag{4}$$

Now the factor 2^{-K} can be implemented by a right shifter.

As $1 - i$ is a boolean (the complement of i), from there on we have two cases [3, 13]:

$$\begin{cases} \text{if } i = 0 & \sqrt{S} \approx \sqrt{2^{-2K+1}(1 + F)} = 2^{-K} \sqrt{2(1 + F)} \\ \text{if } i = 1 & \sqrt{S} \approx \sqrt{2^{-2K}(1 + F)} = 2^{-K} \sqrt{1 + F} \end{cases} . \tag{5}$$

FloPoCo offers various function approximation techniques for a fixed-point function defined on [0,1). To leverage them, we need to combine the two cases above in a single function. To this purpose, replacing n with $\bar{i} = 1 - i$ yields a number $\bar{i}.F \in [0, 2)$. This bit vector can be interpreted as a number $A \in [0, 1)$ whose value is:

$$\begin{cases} \text{if } i = 0 \ (\bar{i} = 1) & A = 0.5 + F/2 \text{ hence } F = 2A - 1 \\ \text{if } i = 1 \ (\bar{i} = 0) & A = F/2 \text{ hence } F = 2A \end{cases} \tag{6}$$

and finally, combining (5) and (6), the piecewise function on $[0, 1)$ that cover both cases is

$$f(A) = \begin{cases} \sqrt{4A} & \text{when } A \geq 0.5 \quad (\text{case } i = 0) \\ \sqrt{1 + 2A} & \text{when } A < 0.5 \quad (\text{case } i = 1) \end{cases} \tag{7}$$

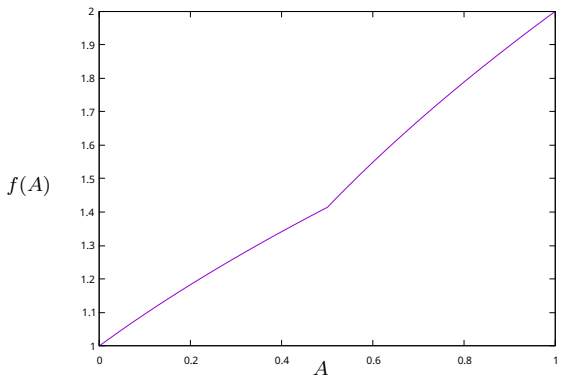


Fig. 2: The piecewise square root used in the naive norm

A plot of this piecewise function is given in Fig. 2. Note that the inverter on i could be saved, but then function $f(A)$ is no longer continuous and monotonic, which crashes the existing approximation code in FloPoCo. Indeed the function is actually entered as

$$f(A) = t(A)\sqrt{4A} \;+\; (1-t(A))\sqrt{1+2A} \tag{8}$$

where $t(A)$ is an approximation to the Heaviside step function: $t(A) = \dfrac{1}{1+e^{C(A-0.5)}}$ with C a large constant. This tinkering is numerically expensive[2], but works well. The PW$\sqrt{\ }$ block in Fig. 1 is either a correctly rounded table, or a faithful piecewise approximation using either multipartite tables or piecewise Horner polynomials of degree 2 to 4. The discontinuity of the derivative at $f(0.5)$ is not a problem as soon as these piecewise methods split the input interval $[0,1)$ in at least 2 sub-intervals.

B. Error analysis and parameter optimization

Let R be the final value returned by the architecture as depicted in Fig. 1. The architecture is faithful if

$$|R - \sqrt{S}| < 2^{\ell_{\text{out}}} \tag{9}$$

(reminding that S is the exact sum of the squares of the inputs). We have here three sources of error:

- The truncation in (3) can be rewritten
$$1 + F = 2^{2K+i-1}S - \delta_F \quad \text{with} \quad 0 \le \delta_F < 2^{\ell} \;. \tag{10}$$
- M is an approximation to the square root:
$$M = f(A) + \delta_M \tag{11}$$
with either $|\delta_M| \le 2^{\ell_{\text{out}}-1}$ (correct rounding), or $|\delta_M| < 2^{\ell_{\text{out}}}$ (faithful rounding).
- The final shift also discards bits:
$$R = 2^{-K}M - \delta_{\text{shift}} \quad \text{with} \quad 0 \le \delta_{\text{shift}} \le 2^{\ell_{\text{out}}} - 2^{\ell_{\text{out}}-K}. \tag{12}$$

[2]Some of the larger operators take several hours to produce, which is strictly due to this pseudo-threshold function.

The error bound on δ_{shift} reads as follows: if $K = 0$, there is no shift, therefore $\delta_{\text{shift}} = 0$; if $K = 1$ we lose at most one bit of weight $2^{\ell_{\text{out}}-1}$; if $K = 2$ we lose at most two bits whose maximum value is $2^{\ell_{\text{out}}} - 2^{\ell_{\text{out}}-2}$; and so on.

Now we may rewrite

$$R - \sqrt{S} \;=\; R - 2^{-K}M \;+\; 2^{-K}(M - 2^K\sqrt{S}). \tag{13}$$

We remark that as soon as $|M - 2^K\sqrt{S}| < 2^{\ell_{\text{out}}}$, we will have

$$\begin{aligned} |R - \sqrt{S}| \;&\le\; |R - 2^{-K}M| \;+\; 2^{-K}|M - 2^K\sqrt{S}| \\ &<\; 2^{\ell_{\text{out}}} - 2^{\ell_{\text{out}}-K} \;+\; 2^{\ell_{\text{out}}-K} \end{aligned} \tag{14}$$

and our objective (9) will be satisified. Our new objective is therefore that M is faithful to $2^K\sqrt{S}$:

$$\text{Objective:} \quad |M - 2^K\sqrt{S}| < 2^{\ell_{\text{out}}} \tag{15}$$

This is rewritten

$$\begin{aligned} M - 2^K\sqrt{S} \;&=\; M - f(A) \;+\; f(A) - 2^K\sqrt{S} \\ &=\; \delta_M \;+\; f(A) - 2^K\sqrt{S} \quad . \end{aligned} \tag{16}$$

Here $f(A) = \sqrt{2^{1-i}(1+F)}$ exactly, as the computation of A out of (i, F) is implemented with exact bit operations. The only error source in $f(A) - 2^K\sqrt{S}$ is the truncation error captured in (10). We may thus rewrite

$$\begin{aligned} f(A) - 2^K\sqrt{S} \;&=\; \sqrt{2^{1-i}(1+F)} - 2^K\sqrt{S} && (17)\\ &=\; \sqrt{2^{2K}S - \delta_F} - 2^K\sqrt{S} \text{ using (10)} \\ &=\; 2^K(\sqrt{S - \delta_F} - \sqrt{S}) && (18)\\ &=\; 2^K\sqrt{S}(\sqrt{1 - \frac{\delta_F}{S}} - 1) && (19)\\ &=\; 2^K\sqrt{S}(-\frac{\delta_F}{2S} + o((\frac{\delta_F}{2S})^2)) && (20)\\ &\approx\; -2^{K-1}\frac{\delta_F}{\sqrt{S}} \;=\; \frac{-\delta_F}{\sqrt{2^{2K-2}S}} && (21) \end{aligned}$$

where by defintion of K, $\sqrt{2^{2K-1+i}S} \in [1,2)$ hence $\sqrt{2^{2K-2+i}S} \in [1/\sqrt{2}, \sqrt{2})$ hence $\sqrt{2^{2K-2}S} > 1/\sqrt{2}$. Finally $|f(A) - 2^K\sqrt{S}| \;<\; \sqrt{2}\cdot\delta_F < \sqrt{2}\cdot 2^{\ell}$.

Back to our objective (15), it can be achieved by taking $\ell = \ell_{\text{out}} - 1$ when $f(A)$ is correctly rounded ($\delta_M < 2^{\ell_{\text{out}}-1}$). When $f(A)$ is faithful, a sensible choice would be to use $\ell = \ell_{\text{out}} - 2$ to minimize the number of input bits to $f(A)$, and require the implementation of $f(A)$ to ensure $\delta_M < 2^{\ell_{\text{out}}} - \sqrt{2}\cdot 2^{\ell_{\text{out}}-2} \approx 0.64 \cdot 2^{\ell_{\text{out}}}$.

However, for some reason, a faithful $f(A)$ with $\delta_M < 2^{\ell_{\text{out}}}$ works just as well with $\ell = \ell_{\text{out}} - 2$: exhaustive testing of up to 12 input bits ($\ell_{\text{in}} \ge -11$) for 2D norms, and up to 8 input bits ($\ell_{\text{in}} \ge -7$) for 3D norms have shown all these operators to be faithful despite the above error bound being $1.35 \cdot 2^{\ell_{\text{out}}}$ in this case. We welcome help by readers who could help us prove this.

C. Architectural variants for the sum of squares

The exact sum of squares can be implemented in several ways. Of course it may use multipliers and adders. On FPGAs, this solution makes perfect sense for the precision domain where this computation fits in a DSP block. Just to take two examples, the DSPs in recent AMD FPGAs include a 18x25 or 18x27 multiplier followed by a 48-bit adder: they can compute a sum of two (resp. 3) squares of up to 18-bit inputs in two (resp. three) DSPs and no LUT. A DSP in the recent Intel/Altera FPGAs can compute a sum of two products of 18x18 bits, so a sum of two squares up to 18 bits fits in a single DSP without any logic.

All the other options consume no DSP resources, and Table I compares them for small precisions.

The first solution is simply to tabulate the squares. For an exact square of n-bit inputs (in our case $n = 1 - \ell_{\text{in}}$) this is a table of $2^n \times 2n$ bits. With 6-input LUTs, we expect for n=8 such a table to consume $2^{8-6} \times 16 = 64$ LUTs. Synthesis tools are able to compress this in 32 LUTs, exactly half: they exploit the symmetry in the square of a signed input. In Table I the lines "TablesSigned" report the cost of adding the outputs of 2 or 3 such tables.

This symmetry can also be explicited by computing $|X|$ and squaring this. However in our setup, X can represent -1, but not 1. Therefore, $|X|$ can take the value 1: its format must be ufix$(0, \ell_{\text{in}})$, so the size in bits of $|X|$ must be the same as that of X. However, it is possible to decompose (as in Fig. 3) $|X| = m + L$ where m is the MSB of $|X|$ and L consists of all the lower bits: $L \in [0, 1)$ is an ufix$(0, \ell_{\text{in}})$. Now m is only used to represent the value 1 and in this case $L = 0$. Conversely, when $m = 0$, then $X^2 = L^2$. Therefore, the architecture of Fig. 3 reduces the square of an sfix$(0, \ell)$ to the square of an ufix$(-1, \ell)$.

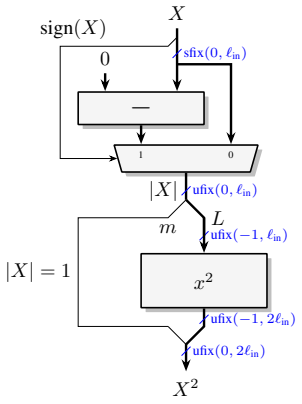


Fig. 3: Reduction of the square from signed to unsigned

Removing one bit from the input reduces the size of the x^2 table size by a half, at the cost of the subtracter and mux of Fig. 3 (on FPGAs, synthesis tools will fuse them in a single row of $1 - \ell_{\text{in}}$ LUTs) . The actual benefit (lines "TablesUnsigned" Table I) is much less since the synthesis tools could already exploit the corresponding symmetry.

The square tables lend themselves well to Lossless Differential Table Compression (LDTC) [11][4, p. 497], which comes for free with FloPoCo (lines "Tables*Compressed").

A last option is to use dedicated squarers [4, p. 443], which have been studied quite extensively [2, 8, 26]. As the current implementation in FloPoCo does not work for signed inputs, the architecture of Fig. 3 is used.

TABLE I: Comparison of DSP-free variants for the sum of squares

Table generated with the FloPoCo `FixSumOfSquares` operator

2D	4 bits	6 bits	8 bits	10 bits
TablesSigned	11 L / 2.12 ns	24 L / **2.17** ns	75 L / **2.63** ns	266 L / **3.22** ns
TablesUnsigned	11 L / 2.12 ns	**23** L / 2.21 ns	64 L / 3.02 ns	369 L / 3.4 ns
TablesSignedCompressed	11 L / 1.87 ns	28 L / 2.41 ns	61 L / 2.96 ns	201 L / 3.72 ns
TablesUnsignedCompressed	11 L / 2.12 ns	29 L / 2.63 ns	70 L / 3.47 ns	132 L / 4.15 ns
Multipliers	11 L / 2.12 ns	24 L / **2.17** ns	110 L / 4.09 ns	180 L / 5.06 ns
Squarers (unsigned)	11 L / 2.12 ns	27 L / 2.72 ns	**54** L / 3.59 ns	**96** L / 4.18 ns
3D	4 bits	6 bits	8 bits	10 bits
TablesSigned	18 L / 2.17 ns	37 L / **2.24** ns	108 L / **2.82** ns	397 L / **3.53** ns
TablesUnsigned	19 L / 2.12 ns	40 L / 2.52 ns	150 L / 3.59 ns	500 L / 4.05 ns
TablesSignedCompressed	22 L / 2.31 ns	44 L / 2.85 ns	95 L / 3.46 ns	306 L / 3.92 ns
TablesUnsignedCompressed	19 L / 2.12 ns	44 L / 2.94 ns	115 L / 3.83 ns	194 L / 4.34 ns
Multipliers	16 L / 2.88 ns	38 L / 3.04 ns	171 L / 4.86 ns	278 L / 5.82 ns
Squarers (unsigned)	18 L / 2.2 ns	**36** L / 3.23 ns	**85** L / 4.06 ns	**137** L / 4.72 ns

As each of the squarers involves a bit-array compression, the sum of squarers could be implemented with a single bit array compression (a concept called "merged arithmetic" [21, 23] [4, p. 151]). This is not working at publication time and therefore not reported in Table I, but we invite our reader to look up this option in their current FloPoCo.

As Table I shows, the squarers are the best option for area as soon as the input width reaches 8 bits, and the tables stay relevant a bit further if latency is the main criterion. Asymptotically, squarers grow quadratically whereas tables grow exponentially, therefore squarers should be used.

As an exact result is needed, approximate methods (such as the piecewise polynomial approximations used in [18]) were not studied.

D. Architectural variants for the square root

The current code also offers a choice for the PW$\sqrt{\ }$ block. Their impact can be observed in Table III.

The first variant is, again, a plain table, possibly compressed. It is correctly rounded, which (Section II-B) enables a saving of one bit on the parameter ℓ. Still, it is only relevant for inputs smaller than 8 bits.

The other methods are only faithfully rounded. Piecewise polynomial evaluation, here of degree 1 to 4 [3, 13] scale to arbitrary sizes. For a given size, the degree controls a balance between memory for the coefficients and DSP resource consumption for the polynomial evaluator (here in Horner form).

The multipartite table method [4, p. 503] is an optimization of the piecewise linear (degree 1) approximation where the products are distributed and tabulated. This method has known many evolutions [5, 6, 10, 11, 15, 19, 20] and the current code uses one of the most recent variants [11]. Below 16 bits, it consumes fewer resources than the corresponding degree-1 approximation. At 16 bits and more, the quadratic-scaling multipliers again win against exponential-scaling tables.

Since the architectures use an internal floating-point representation, we also, of course, investigated the use of a variation of a floating-point square root. FloPoCo's square root is a basic radix-2 SRT implementation whose iterative structure is quite comparable to 2D CORDIC. Indeed, this operator alone has a latency comparable to 2D CORDIC for the same precision, and CORDIC is already slower than Naive, so this would be the slowest variant. However, the SRT square root is also smaller

TABLE II: Breakout of the area and delay of the Naive version (using the best-area variant from Table III)

	8 bits	12 bits	16 bits	24 bits
X^2+Y^2	54 L / 3.59 ns	166 L / 5.39 ns	286 L / 5.82 ns	644 L / 6.38 ns
$X^2+Y^2+Z^2$	85 L / 4.06 ns	242 L / 6.00 ns	419 L / 6.36 ns	954 L / 6.97 ns
normalizer	37 L / 3.35 ns	70 L / 4.21 ns	82 L / 4.32 ns	146 L / 5.36 ns
PW$\sqrt{}$	27 L / 2.79 ns (Multipartite)	77 L / 3.34 ns (Multipartite)	195 L / 4.37 ns (degree 1)	626 L / 8.62 ns (degree 2)
right shifter	17 L / 1.81 ns	27 L / 1.83 ns	39 L / 2.11 ns	64 L / 2.12 ns

than the best implementation of PW$\sqrt{}$ (for instance for 16 bits 188 L versus 195, for 24 bits 376 L versus 626). However, as we will see, 2D CORDIC is much smaller than the naive method for these sizes, and a small improvement in PW$\sqrt{}$ will not be enough for the naive method to catch up. In short, a naive 2D norm using SRT square root will be both slower and larger than CORDIC. It still remains interesting to explore in the 3D case as a low-area, long latency, multiplierless option.

E. Area and delay breakout for the naive version

Table II provides a breakout of the area and delay of each component in the best solution for a few sizes. In the polynomial approximators for 16 and 24 bits, the multipliers are synthesized in logic to ease the comparison. We observe that the main components are the sum of squares and the square root, with very comparable costs for 2D.

III. CORDIC 2D

A. Overview

The 2D norm can be computed by the vectoring mode of CORDIC:

$$\begin{cases} x_0 & \in [0,1] \\ y_0 & \in [-1,1] \\ \theta_0 & = 0 \end{cases} \tag{22}$$

$$\begin{cases} d_n & = +1 \quad \text{if } y_n < 0 \text{ else } -1 \\ x_{n+1} & = x_n - 2^{-n} d_n y_n \\ y_{n+1} & = y_n + 2^{-n} d_n x_n \\ \theta_{n+1} & = \theta_n - d_n \arctan(2^{-n}) \end{cases} \tag{23}$$

Iterations converge as follows

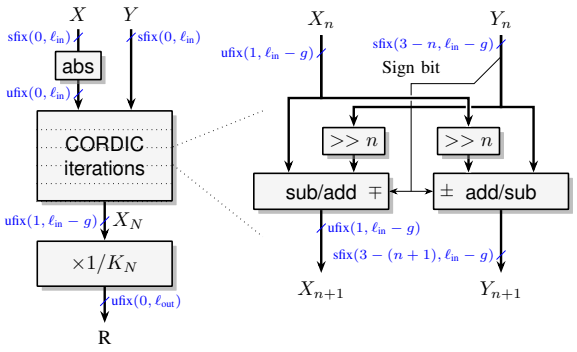
$$\begin{cases} \lim\limits_{n\to\infty} x_n & = K \cdot \sqrt{x_0^2 + y_0^2} \\ \lim\limits_{n\to\infty} y_n & = 0 \\ \lim\limits_{n\to\infty} \theta_n & = \arctan\left(y_0/x_0\right) \end{cases} \tag{24}$$

where

$$K_n = \prod_{i=0}^{n} \sqrt{1 + 2^{-2i}} \tag{25}$$

$$K = \lim_{n\to\infty} K_n \approx 1.646 \tag{26}$$

The corresponding architecture is shown in Fig. 4a. First, the absolute value of X is taken, so that CORDIC iterations converge. Subsequently, N unrolled CORDIC iterations are performed. The hardware implementation of a CORDIC iteration is depicted in Fig. 4b. Finally, X_N is divided by the scale factor K_N and the architecture returns the norm in R.



(a) Overview (b) one iteration

Fig. 4: CORDIC 2D abstract architecture

B. Error analysis and datapath sizing

The rounding error is controlled by adding g guard bits to the internal computation datapath. Let

- R the final value returned by the architecture.
- X_n the value of x_n actually computed by the architecture at iteration n.

The overall error of the architecture is defined as:

$$\delta_{\text{total}} = \|(x_0, y_0)\|_2 - R \tag{27}$$

$$= \underbrace{\|(x_0, y_0)\|_2 - \frac{x_N}{K_N}}_{=\delta_{\text{approx}}} + \underbrace{\frac{x_N}{K_N} - \frac{X_N}{K_N}}_{=\delta_{\text{round}}} + \underbrace{\frac{X_N}{K_N} - R}_{=\delta_{\text{mult}}} \tag{28}$$

a) Approximation error: Let γ_n be the angle between the vector (x_n, y_n) and the x-axis.

$$\delta_{\text{approx}} = \|(x_0, y_0)\|_2 - \frac{x_N}{K_N} \tag{29}$$

$$= \|(x_0, y_0)\|_2 - \frac{K_N \cos(\gamma_N)\, \|(x_0, y_0)\|_2}{K_N} \tag{30}$$

$$= \|(x_0, y_0)\|_2 \cdot (1 - \cos(\gamma_N)) \tag{31}$$

$$\leq \sqrt{2}(1 - \cos(\gamma_N)) \tag{32}$$

Moreover,

$$\begin{aligned} 1 - \cos(\gamma_N) &= 1 - \cos(\gamma_0 - \theta_N) && \text{(definition of } \theta_N) \\ &\leq 1 - 1 + \frac{(\gamma_0 - \theta_N)^2}{2} \\ &\leq \frac{1}{2}\left(\sum_{k=N}^{\infty} \arctan\left(2^{-k}\right)\right)^2 && \text{see [16]} \\ &\leq \frac{1}{2}\left(\sum_{k=N}^{\infty} 2^{-k}\right)^2 \\ &\leq 2^{-2N+1} \end{aligned}$$

Finally,

$$\delta_{\text{approx}} \leq \sqrt{2}(1 - \cos(\gamma_N)) \tag{33}$$

$$\leq 2^{-2N+3/2} \tag{34}$$

b) Rounding errors: The architecture truncates intermediate results, so it computes:

$$\begin{cases} \widetilde{d}_n & = +1 \quad \text{if } Y_n < 0 \text{ else } -1 \\ X_{n+1} & = X_n - \widetilde{d}_n \cdot \lfloor 2^{-n} Y_n \rfloor_{\ell_{\text{in}}-g} \\ Y_{n+1} & = Y_n + \widetilde{d}_n \cdot \lfloor 2^{-n} X_n \rfloor_{\ell_{\text{in}}-g} \end{cases} \tag{35}$$

An issue is that $\widetilde{d}_n$ (the sign used in the architecture) may differ from d_n (the sign in the mathematical recurrence) due to rounding errors leading to different signs between y_n and Y_n. Of course this can only happen if y_n and Y_n are close to 0. We remark that the X datapath computes in all cases $X_{n+1} = X_n + \lfloor 2^{-n} |Y_n| \rfloor_{\ell_{\text{in}}-g}$.

The rounding error for X_{n+1} is defined as:

$$\delta^x_{n+1} = X_{n+1} - x_{n+1} \tag{36}$$

$$= X_n - \widetilde{d}_n \cdot \lfloor 2^{-n} Y_n \rfloor_{\ell_{\text{in}}-g} - (x_n - 2^{-n} d_n y_n) \tag{37}$$

$$= X_n - x_n + 2^{-n}(d_n y_n - \widetilde{d}_n Y_n) + 2^{-n}\widetilde{d}_n Y_n - \widetilde{d}_n \cdot \lfloor 2^{-n} Y_n \rfloor_{\ell_{\text{in}}-g} \tag{38}$$

Let us therefore define the rounding error on the Y datapath:

$$\delta^y_n = \widetilde{d}_n Y_n - d_n y_n = |y_n| - |Y_n| \quad . \tag{39}$$

With this definition, (38) leads to:

$$\left|\delta^x_{n+1}\right| \leq |\delta^x_n| + 2^{-n} |\delta^y_n| + 2^{\ell_{\text{in}}-g} \tag{40}$$

and for $\left|\delta^y_{n+1}\right|$:

$$\delta^y_{n+1} = |y_{n+1}| - |Y_{n+1}| \tag{41}$$

$$= \left|y_n + 2^{-n} d_n x_n\right| - \left|Y_n + \widetilde{d}_n \cdot \lfloor 2^{-n} X_n \rfloor_{\ell_{\text{in}}-g}\right| \tag{42}$$

$$= \left|-d_n |y_n| + 2^{-n} d_n x_n\right| - \left|-\widetilde{d}_n |Y_n| + \widetilde{d}_n \cdot \lfloor 2^{-n} X_n \rfloor_{\ell_{\text{in}}-g}\right| \tag{43}$$

$$= \left|\,|y_n| - 2^{-n} x_n\right| - \left||Y_n| - \lfloor 2^{-n} X_n \rfloor_{\ell_{\text{in}}-g}\right| \tag{44}$$

Taking the absolute value and using the reverse triangle inequality yield:

$$\left|\delta^y_{n+1}\right| = \left|\left|\,|y_n| - 2^{-n} x_n\right| - \left||Y_n| - \lfloor 2^{-n} X_n \rfloor_{\ell_{\text{in}}-g}\right|\right| \tag{45}$$

$$\leq \left||y_n| - 2^{-n} x_n - |Y_n| + \lfloor 2^{-n} X_n \rfloor_{\ell_{\text{in}}-g}\right| \tag{46}$$

$$\leq \big||y_n| - |Y_n|\big| + \left|2^{-n} X_n - 2^{-n} x_n\right| + \left|\lfloor 2^{-n} X_n \rfloor_{\ell_{\text{in}}-g} - 2^{-n} X_n\right| \tag{47}$$

$$\leq |\delta^y_n| + 2^{-n} |\delta^x_n| + 2^{\ell_{\text{in}}-g} \tag{48}$$

Let $\overline{\delta^x_n}$ and $\overline{\delta^y_n}$ be, respectively, error bounds on δ^x_n and δ^y_n:

$$\begin{cases} \overline{\delta^x_0} = \overline{\delta^y_0} & = 0 \\ \overline{\delta^x_{n+1}} & = \overline{\delta^x_n} + 2^{-n}\overline{\delta^y_n} + 2^{\ell_{\text{in}}-g} \\ \overline{\delta^y_{n+1}} & = \overline{\delta^y_n} + 2^{-n}\overline{\delta^x_n} + 2^{\ell_{\text{in}}-g} \end{cases} \tag{49}$$

By induction, $\forall n\ \overline{\delta^x_n} = \overline{\delta^y_n} = 2^{\ell_{\text{in}}-g} \cdot \alpha_n$ where:

$$\begin{cases} \alpha_0 & = 0 \\ \alpha_{n+1} & = \alpha_n(1 + 2^{-n}) + 1 \end{cases} . \tag{50}$$

It is easier to determine the number of guard bits required with this expression as g does not appear in the definition of α_n.

c) Multiplication rounding errors: The multiplication by $1/K_N$ uses FixRealKCM, one of the constant multipliers from FloPoCo. It can be configured with an ulp error bound parameter $t \in \left]\frac{1}{2}, 1\right]$ so that the faithful architecture obeys

$$|\delta_{\text{mult}}| < t \cdot 2^{\ell_{\text{out}}} \ . \tag{51}$$

d) Determining the parameters of the architecture: The architectural parameters are N, the number of iterations, g, the number of guard bits, and t above. Our objective is to ensure $|\delta_{\text{total}}| < 2^{\ell_{\text{out}}}$ at the least possible hardware cost. As

$$|\delta_{\text{total}}| = |\delta_{\text{approx}} + \delta_{\text{round}} + \delta_{\text{mult}}| \tag{52}$$

$$\leq |\delta_{\text{approx}}| + |\delta_{\text{round}}| + |\delta_{\text{mult}}| \tag{53}$$

a sensible set of constraints (among an infinity – determining the actual optimal is beyond the scope of this work) is:

$$|\delta_{\text{approx}}| < \frac{1}{8} 2^{\ell_{\text{out}}} \tag{54}$$

$$|\delta_{\text{round}}| < \frac{1}{8} 2^{\ell_{\text{out}}} \tag{55}$$

$$|\delta_{\text{mult}}| < \frac{3}{4} 2^{\ell_{\text{out}}} \tag{56}$$

From equations (54) and (34), N must satisfy:

$$2^{-2N+3/2} < \frac{1}{8} 2^{\ell_{\text{out}}} \tag{57}$$

hence the smallest value of N verifying this inequality is

$$N = \left\lceil \frac{-\ell_{\text{out}} + 9/2}{2} \right\rceil \quad . \tag{58}$$

The recurrence from (49) is then evaluated up to this N. The rounding errors verify:

$$|\delta_{\text{round}}| = \frac{1}{K_N} |x_N - X_N| = \frac{1}{K_N} \delta^x_N \tag{59}$$

$$\leq \frac{1}{K_N} \overline{\delta^x_N} = \frac{2^{\ell_{\text{in}}-g}}{K_N} \cdot \alpha_N \tag{60}$$

To satisfy inequation (55), the following constraint holds on g

$$\frac{2^{\ell_{\text{in}}-g}}{K_N} \cdot \alpha_N < \frac{1}{8} 2^{\ell_{\text{out}}} \tag{61}$$

thus, the smallest value of g that ensures last-bit accuracy is

$$g = \lceil \log_2(\alpha_N) - \log_2(K_N) \rceil + \ell_{\text{in}} - \ell_{\text{out}} + 3 \quad . \tag{62}$$

Finally, from (56) and (51), the value of t in (56) should obviously be

$$t = \frac{3}{4} \ . \tag{63}$$

e) Sizing of the Y_n datapath: From the analysis, the X_n number format will be ufix$(1, \ell_{\text{in}} - g)$. For the Y_n datapath, notice that

$$|y_n| = K_n |\sin(\gamma_n)| \, \|(x_0, y_0)\|_2 \tag{64}$$

$$\leq K_n |\gamma_n| \, \|(x_0, y_0)\|_2 \tag{65}$$

$$\leq 2 \cdot 2^{-n+1} \cdot 2^{1/2} \tag{66}$$

$$\leq 2^{-n+3} \tag{67}$$

thus the number format will be sfix$(3 - n, \ell_{\text{in}} - g)$ for $n \geq 3$.

TABLE III: Comparison of CORDIC and the Naive method variants

Table generated with the FloPoCo `Fix2DNorm` and `Fix3DNorm` operators

2D	4 bits	6 bits	8 bits	12 bits	16 bits	24 bits
CORDIC	55 L / 6.46 ns	87 L / 8.4 ns	141 L / 10.88 ns	**268** L / 14.29 ns	**415** L / 16.62 ns	**813** L / 22.21 ns
NaivePlainTable	34 L / **4.2** ns	**76** L / **5.88** ns	178 L / **8.18** ns	1000 L / **12.22** ns	14898 L / **13.69** ns	N/A
NaiveMultiPartite	**31** L / 4.39 ns	93 L / 6.66 ns	**130** L / 8.74 ns	324 L / 12.88 ns	677 L / 14.33 ns	5926 L / **17.62** ns
NaivePiecewiseHorner1	48 L / 5.8 ns	90 L / 8.27 ns	161 L / 9.72 ns	353 L / 13.55 ns	602 L / 14.58 ns	2441 L / 18.83 ns
NaivePiecewiseHorner2	51 L / 6.12 ns	137 L / 10.27 ns	228 L / 11.55 ns	471 L / 16.32 ns	757 L / 17.77 ns	1479 L / 20.82 ns
NaivePiecewiseHorner3	N/A	151 L / 9.94 ns	280 L / 13.26 ns	N/A	850 L / 20.76 ns	1641 L / 23.56 ns
3D	4 bits	6 bits	8 bits	12 bits	16 bits	24 bits
CORDIC	906 L / 12.99 ns	N / A	1880 L / 23.55 ns	2786 L / 30.25 ns	3951 L / 36.82 ns	N / A
NaivePlainTable	**38** L / **4.3** ns	103 L / **6.29** ns	200 L / **8.61** ns	1066 L / **12.77** ns	15038 L / **14.17** ns	N/A
NaiveMultiPartite	**38** L / **4.3** ns	**89** L / 8.03 ns	**150** L / 9.09 ns	**408** L / 13.3 ns	817 L / 14.81 ns	6236 L / **18.46** ns
NaivePiecewiseHorner1	55 L / 5.85 ns	102 L / 8.58 ns	185 L / 10.02 ns	433 L / 14.08 ns	**741** L / 15.1 ns	**2749** L / 19.68 ns
NaivePiecewiseHorner2	60 L / 6.33 ns	142 L / 10.49 ns	250 L / 11.98 ns	550 L / 16.69 ns	896 L / 18.19 ns	1777 L / 21.54 ns
NaivePiecewiseHorner3	N/A	N/A	300 L / 13.69 ns	N/A	991 L / 21.16 ns	1915 L / 24.52 ns

IV. CORDIC 3D

CORDIC can be generalized to d dimensions [9]. This section focuses on $d = 3$. As in CORDIC 2D, the idea in CORDIC 3D is to align the initial vector with the $(1, 0, 0)$ axis. To do so, the vectoring operation is decomposed into elementary rotations around two possible axes. The iterative equations are controlled by two variables that encode the axis and the rotation direction.

$$\begin{cases} x_0 & \in [-1, 1[\\ y_0 & \in [-1, 1[\\ z_0 & \in [-1, 1[\end{cases} \tag{68}$$

$$\begin{cases} \delta_n & = +1 \quad \text{if } x_n y_n \leq 0 \text{ else } -1 \\ \lambda_n & = +1 \quad \text{if } x_n z_n \leq 0 \text{ else } -1 \\ x_{n+1} & = x_n - 2^{-2n+1} x_n + \delta_n 2^{-n+1} y_n + \lambda_n 2^{-n+1} z_n \\ y_{n+1} & = y_n - \delta_n 2^{-n+1} x_n - \delta_n \lambda_n 2^{-2n+1} z_n \\ z_{n+1} & = z_n - \lambda_n 2^{-n+1} x_n - \delta_n \lambda_n 2^{-2n+1} y_n \end{cases} \tag{69}$$

We first investigated the potential of this architecture for 4, 8, 12, and 16 bits. For these input sizes, we manually determined by trial and error the smallest possible number of iterations and guard bits ensuring a faithful operator (based on exhaustive tests for small sizes and one million random test cases for larger sizes). Operators with these parameters were then synthesized, and this is the data we report in Table III. As these results leave no hope that this approach could be competitive with the naive approach, we did not feel compelled to provide an error analysis or investigate further optimization.

Note that even the simpler solution of a sequence of two rotations (chaining two CORDIC 2D) will be much cheaper than CORDIC 3D (even accounting for the fact that extra accuracy is needed in the intermediate result). However it will have longer latency and more area than the naive approach, so we didn't investigate it either.

V. Results and conclusions

Table III shows the area/latency trade-off of the 2D and 3D norm variants implemented here. The short conclusion is that 2D CORDIC is smaller but slower than Naive 2D, while in 3D the naive approach is the one to use.

The naive variants can trade LUTs for DSP blocks. It makes sense at 16 bits and above, for instance the 286 L / 5.82 ns for the sum of squares in Table II can be replaced with 2 DSP blocks and 0 LUTs / 4.47ns. Due to lack of space we do not explore this trade-off in detail.

These operators also work for different input and output sizes, since the error analysis includes these cases. Table IV illustrates the overhead of increasing the accuracy of the norm for a fixed input precision. This may be useful when the norm is taken directly on the inputs of a complex datapath that needs to work with internal higher precision.

VI. Future work

Future work includes an extension to floating-point norms. For the naive version, the cost will probably be comparable: we will save the normalizer and final shifter component (the latter being replaced with a 1-bit normalization and rounding unit). However, the input squares will have to be shifted based on to their exponent differences before being added. Alternatively, the inputs can be converted to a common fixed-point format before entering one of the proposed fixed-point architectures. In both cases the truncation in this initial shift is a new source of error, requiring an update of the error analysis.

A fixed-point in, floating-point out norm would probably be a cheap variant of the proposed naive architecture, again just replacing the final shifter with a simpler rounding unit. A reciprocal norm (evaluating $1/\sqrt{X^2 + Y^2 + Z^2}$) is another simple variant (essentially changing the function f) but it mostly makes sense with a floating-point output.

There are many more methods to investigate, including higher-radix CORDIC [1, 24]. These are expected to have lower latency but higher area than CORDIC due to the need for prescaling, additional look-up tables to read the microrotation angles, and the non-constant scale factor – the combined cost of the higher-radix micro-rotations themselves should

TABLE IV: Norms with 8-bit inputs and wider output

output width	8 bits	12 bits	16 bits
2D CORDIC	141 L / 10.88 ns	216 L / 13.28 ns	311 L / 15.65 ns
2D Naive	130 L / 8.74 ns	188 L / 9.54 ns	363 L / 9.94 ns
3D Naive	150 L / 9.09 ns	211 L / 9.98 ns	385 L / 10.38 ns

be comparable. Due to this overhead they probably become relevant only for precisions higher than the ones targeted here.

We have focused on norms, but another advantage of 2D CORDIC is that a single iteration can perform a full polar/cartesian coordinate conversion (both ways). This useful operator should also be explored.

References

[1] Elisardo Antelo, Tomas Lang, and Javier D. Bruguera. "Very-high radix circular CORDIC: Vectoring and unified rotation/vectoring". In: *IEEE Transactions on Computers* 49.7 (2000), pp. 727–739.

[2] Andreas Böttcher, Martin Kumm, and Florent de Dinechin. "Resource Optimal Squarers for FPGAs". In: *Field-Programmable Logic and Applications (FPL)*. IEEE, Aug. 2022.

[3] Florent de Dinechin, Mioara Joldes, Bogdan Pasca, and Guillaume Revy. "Multiplicative square root algorithms for FPGAs". In: *Field-Programmable Logic and Applications (FPL)*. 2010, pp. 574–577.

[4] Florent de Dinechin and Martin Kumm. *Application-Specific Arithmetic*. Springer, 2024.

[5] Florent de Dinechin and Arnaud Tisserand. "Multipartite Table Methods". In: *IEEE Transactions on Computers* 54.3 (2005), pp. 319–330.

[6] Debjit Das Sarma and David W. Matula. "Faithful Bipartite ROM Reciprocal Tables". In: *12th Symposium on Computer Arithmetic*. Ed. by S. Knowles and W.H. McAllister. IEEE, 1995, pp. 17–28.

[7] Miloš D. Ercegovac and Tomás Lang. *Digital Arithmetic*. Morgan Kaufmann, 2004.

[8] Shuli Gao, Noureddine Chabini, Dhamin Al-Khalili, and J M Pierre Langlois. "FPGA-Based Efficient Design Approaches for Large Size Two's Complement Squarers". In: *Journal of Signal Processing Systems* 58.1 (2008), pp. 3–15.

[9] Shen-Fu Hsiao and J.-M. Delosme. "Householder CORDIC algorithms". In: *IEEE Transactions on Computers* 44.8 (Aug. 1995), pp. 990–1001.

[10] Shen-Fu Hsiao, Chia-Sheng Wen, Yi-Hau Chen, and Kuei-Chun Huang. "Hierarchical Multipartite Function Evaluation". In: *Transactions on Computers* 66.1 (2017), pp. 89–99.

[11] Shen-Fu Hsiao, Po-Han Wu, Chia-Sheng Wen, and Pramod Kumar Meher. "Table Size Reduction Methods for Faithfully Rounded Lookup-Table-Based Multiplierless Function Evaluation". In: *Transactions on Circuits and Systems II* 62.5 (2015), pp. 466–470.

[12] Zhijun Huang and Milos D Ercegovac. "FPGA implementation of pipelined on-line scheme for 3-D vector normalization". In: *Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2001, pp. 61–70.

[13] Claude-Pierre Jeannerod, Hervé Knochel, Christophe Monat, and Guillaume Revy. "Computing floating-point square roots via bivariate polynomial evaluation". In: *IEEE Transactions on Computers* 60.2 (Feb. 2011), pp. 214–227.

[14] Pramod K. Meher, Javier Valls, Tso-Bing Juang, K. Sridharan, and Koushik Maharatna. "50 Years of CORDIC: Algorithms, Architectures, and Applications". In: *IEEE Transactions on Circuits and Systems I : Regular papers* 56.9 (2009), pp. 1893–1907.

[15] Jean-Michel Muller. "A Few Results on Table-Based Methods". In: *Reliable Computing* 5.3 (1999), pp. 279–288.

[16] Jean-Michel Muller. *Elementary functions, algorithms and implementation, 3rd Edition*. Birkhaüser, 2016.

[17] Shinobu Nagayama, Tsutomu Sasao, and Jon T Butler. "A systematic design method for two-variable numeric function generators using multiple-valued decision diagrams". In: *IEICE TRANSACTIONS on Information and Systems* 93.8 (2010), pp. 2059–2067.

[18] J. A. Piñeiro, J. D. Bruguera, and J.-M. Muller. "Faithful Powering Computation Using Table Look-Up and a Fused Accumulation Tree". In: *15th Symposium on Computer Arithmetic*. IEEE, 2001, pp. 40–47.

[19] James E. Stine and Michael J. Schulte. "The Symmetric Table Addition Method for Accurate Function Approximation". In: *Journal of VLSI Signal Processing* 21.2 (1999), pp. 167–177.

[20] David A. Sunderland, Roger A. Strauch, Steven S. Wharfield, Henry T. Peterson, and Christopher R. Role. "CMOS/SOS Frequency Synthesizer LSI Circuit for Spread Spectrum Communications". In: *IEEE Journal of Solid-State Circuits* 19.4 (1984), pp. 497–506.

[21] Earl E. Swartzlander. "Merged Arithmetic". In: *IEEE Transactions on Computers* C-29.10 (1980), pp. 946–950.

[22] Naofumi Takagi and Seiji Kuwahara. "A VLSI Algorithm for Computing the Euclidean Norm of a 3D Vector". In: *IEEE Transactions on Computers* 49.10 (2000), pp. 1074–1082.

[23] Ajay K. Verma, Philip Brisk, and Paolo Ienne. "Data-Flow Transformations to Maximize the Use of Carry-Save Representation in Arithmetic Circuits". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.10 (2008), pp. 1761–1774.

[24] Julio Villalba, Emilio L Zapata, Elisardo Antelo, and Javier D. Bruguera. "Radix-4 vectoring CORDIC algorithm and architectures". In: *Journal of VLSI signal processing systems for signal, image and video technology* 19 (1998), pp. 127–147.

[25] Jack Volder. "The CORDIC Computing Technique". In: *IRE Transactions on Electronic Computers* EC-8.3 (1959), pp. 330–334.

[26] Simin Xu, Suhaib A. Fahmy, and Ian V. Mcloughlin. "Efficient Large Integer Squarers on FPGA". In: *Field-Programmable Custom Computing Machines*. 2013, pp. 198–201.