FPGA-based Low Latency Square Root CORDIC Algorithm

Mariusz Węgrzyn[1], Stepan Voytusik[2], and Nataliia Gavkalova[3]

[1]*Cracow University of Technology, Cracow, Poland,*
[2]*Lviv Polytechnic National University, Lviv, Ukraine,*
[3]*Warsaw University of Technology, Warsaw, Poland*

https://doi.org/10.26636/jtit.2025.1.1950

Abstract — The coordinate rotation digital computer (CORDIC) algorithm is a popular method used in many fields of science and technology. Unfortunately, it is a time-consuming process for central processing units (CPUs) and graphics processing units (GPUs), and even for specialized digital signal processing (DSP) solutions. The CORDIC algorithm is an alternative for Newton-Raphson numerical calculation and for the FPGA based resource-expensive look-up-table (LUT) method. Various modifications of the CORDIC algorithm allow to speed up the operation of hardware in edge computing devices. With that context taken into consideration, this article presents a fast and accurate square root floating point (SQRT FP) CORDIC function which can be implemented in field programmable gate arrays (FPGAs). The proposed algorithm offers low-complexity, decent accuracy and speed, and is sufficient for digital signal processing (DSP) applications, such as digital filters, accelerators for neural networks, machine learning and computer vision applications, and intelligent robotic systems.

Keywords* — *computer vision, CORDIC algorithm, FPGA, numerical methods, reconfigurable computing systems

1. Introduction

The current methods by means of which the square root (SQRT) calculation approach is implemented in hardware continue to suffer from numerous drawbacks that limit their practical use. Software developers and researchers of many real-time DSP applications face challenges related to computational accuracy and speed [1], as well as optimization of hardware resources required to run square root algorithms [2]. In the Newton-Raphson numerical method that is commonly used for computing the square root, the precision level depends on the initial guess and requires significant computational resources, due to its reliance on iterative multiplication [3], [4].

Alternative multiplicative methods have a quadratic type of convergence, and thus may speed up the computation process. These methods perform a number of iterations of a fused multiply-add (FMA) operation, with the latency of a single FMA being in the range of 3 and 6 cycles [5]. For a low-precision SQRT computation, look-up table or low-degree polynomial approximation methods can be applied [1]. However, high demand for FPGA resources is an additional disadvantage here. This problem has been partially solved by iterative or digit-recurrence methods presented in [6], [7], which are characterized by linear convergence.

In order to overcome the abovementioned issues, the coordinate rotation digital computer (CORDIC) algorithm was proposed [8]–[10].

The main drawback of the CORDIC method [11], [12] is its low speed resulting from the use of linear convergence, where only one correct bit of the result per iteration is given. The number of iterations depends on the precision of the required data. Therefore, latency is a main research problem, especially when the algorithm operates on large bit-width vectors [13].

To simplify hardware implementation, a CORDIC method with angle recoding has been proposed in [7], [10], reducing computational complexity to two iteration equations only. Other articles focus on improving the speed of the method, i.e. reducing the number of iterations, proposing hybrid structures that use, sequentially, three different methods: table-based + CORDIC + piece-wise linear multiplication (linear approximation) [5], [14]–[17].

Another proposal to accelerate the CORDIC algorithm consisted in the introduction of the pre-rotation technique [13]. Moreover, this algorithm can be executed in various forms: classical [12], using higher-order iterative formulas [16], [18], without recoding [13], and with angle recoding [19], [20]. The simplest solution – in terms of hardware implementation – is CORDIC with angle recoding [10], [18]. However, a significant drawback lies in the large amounts of memory (LUT type) required for large values of m (a table of size not less than $2^{\frac{m}{3}} \times m$ bits is needed). Furthermore, the output multipliers are implemented in the {-1, 1} basis, preventing the use of multipliers that are part of the DSP blocks in modern FPGA devices.

The CORDIC algorithm is widely applied to calculate various mathematical functions, including SQRT [4], [10], [21]. An example of the application of the new numerical method for SQRT calculations is presented in [22], where an efficient design of a Kalman filter is implemented. The authors plan to apply the CORDIC algorithm to improve digital filters for telecommunications applications. Thus, we have the intention of developing further effective methods for the calculation of trigonometric functions using the CORDIC style. The said algorithm can also be relied upon to calculate nonlinear functions, such as $\mathrm{th}(x)$, which are useful for implementing neural networks. Its other applications include low-resource microprocessors without the floating point unit (FPU), as our algorithm converts from floats to integers and all calculations are executed on integer-type numbers, with the result them being converted back to the floating point type.

The goal of this work is to propose a new efficient SQRT floating point type CORDIC algorithm with low latency and moderate FPGA resource requirements. Our proposal utilizes an original methodology of converting floating numbers to integers and vice versa, in order to optimize the use of FPGA hardware resources and to improve calculation efficiency.

The article is organized as follows. Section 2 reviews various methodologies and hardware implementations of square root algorithms. Section 3 introduces the proposed methodology. Section 4 introduces the proposed algorithm and describes the details of the FPGA implementation, while Section 5 presents and discusses the results achieved with the use of our solution. Section 6 presents the conclusions.

2. Related Works

The CORDIC concept was introduced by Volder in 1959 [12], then the solution was extended to calculate elementary functions, including the square root [16]. CORDIC owes its low hardware resource-related requirements to the fact that one iteration may be performed using basic shift and addition commands. A low-complex design methodology is introduced in [4] for the computation of square root ($\sqrt{x}$) and division ($\frac{x}{z}$) using circular CORDIC reuse. This method reduces the area overhead for biomedical applications. Here, the square root is computed using the derivative Newton-Raphson (NR) formula, and a technique consisting in dividing input x into different segments is applied. Solutions [4], [23] also perform micro-rotations to predict rotation directions.

Paper [23] proposed a pipeline-parallel unified CORDIC architecture to perform square root computing and several basic functions using floating point numbers. Article [24] proposes a complex square root computation method independent of angle in the CORDIC style.

Article [11] presents the advantages of the square root CORDIC radix-10 FPGA implementation method. The solution covers both fixed- and floating-point versions with different reconfigurable number of digits, thus different precision versions – according to IEEE 754-2008 standard – are implemented. Therefore, the number of iterations is configurable (13–25). The authors of [25] redesigned the core that employs an iterative non-restoring algorithm that converges closer to the result after every iteration. Articles [9], [11], [18], [26] described the advantages and disadvantages of the various radix-2, radix-4 and radix-10 CORDIC algorithms. These solutions reduce the number of iterations and thus can speed up the calculation process [16], [18].

Article [21] proposes the radix-4 CORDIC algorithm for computing roots and powers of various orders. This solution is divided into three phases, where each stage is completed by a different class of the modified radix-4 CORDIC algorithm. The degree of complexity is reduced by precomputing the scale factor for initial iterations and by employing scaling-free rotations for later iterations. Other papers [27], [28] describe precomputation of rotation direction to achieve a latency improvement, using the radix-4 architecture.

The CORDIC IP core v6.0 developed by Xilinx [29] implements a generalized CORDIC algorithm to iteratively solve trigonometric equations, hyperbolic and square root equations, etc. There are two architectural configurations available: a fully parallel configuration with single-cycle data throughput at the higher expense of silicon die size, and a word serial implementation with multiple-cycle throughput, but with the advantage of low silicon usage.

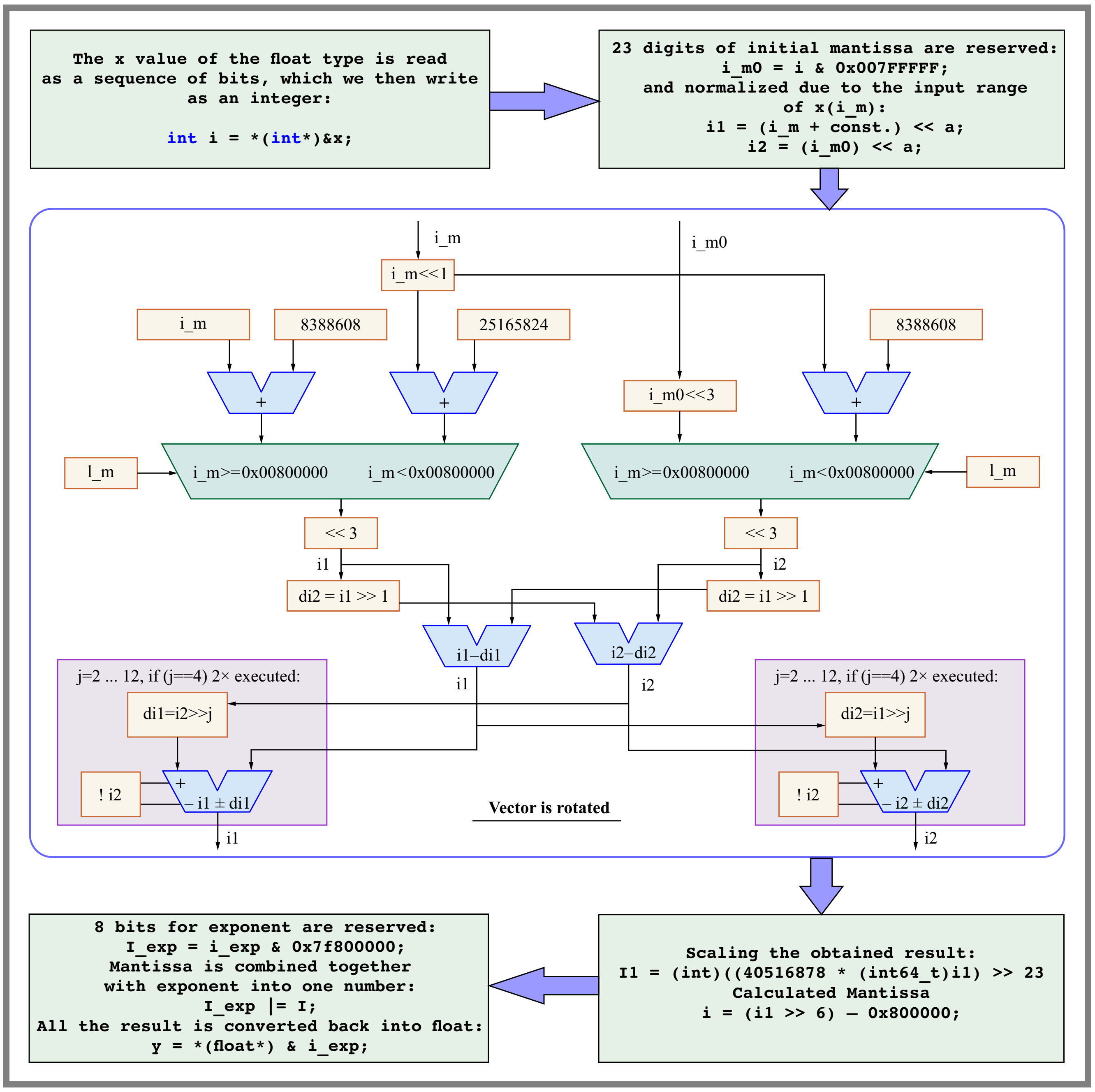
Alternatively, paper [10] incorporates the square root function into the existing FP multiplication/division fused unit to reduce the hardware resources required. The Taylor series expansion algorithm with powering units, which exhibits the highest performance, was implemented in the hardware.

A parallel CORDIC core with N bit output width has a latency of N cycles and produces a new output every cycle. A word serial CORDIC core with N bit output width has a latency of N cycles and produces a new output every N cycles. In practice, the CORDIC square root function of the Xilinx IP core generator can be useful for the estimation of the DC motor rotor flux [10], [28]. For FPGA implementation, tools provided by FPGA providers are presented in [13]. For example, the Xilinx System Generator (XSG) simulation tool was applied that can easily make the direct translation into hardware of control algorithms without knowing any hardware description language (HDL) for implementing solutions developed in [13]. This is a high-level tool for designing high-performance DSP systems in the Simulink environment.

Similarly, our methodology uses the Xilinx Vitis HLS tool to optimally translate the proposed algorithm into HDL C language. In such a method, the hardware implementation is optimal for any FPGA chips and knowledge of the details of the FPGA architecture is not required.

3. Description of the Proposed Method

In this Section, the application of the CORDIC algorithm for calculating the floating-point square root is described and the open-source code is provided. All known CORDIC square root implementations have been used for integer or fixed point [8]–[10], [14], [26] calculations. The proposed algorithm effectively combines such techniques as convert fp x into an integer i, splitting integer i into a mantissa and an exponent, using the fast bitwise masking operation & (instead of the slow frexpf), combining the converted mantissas and exponents of the result into one number using the same fast bitwise operation (instead of slow ldexpf). It should also be noted that the high accuracy and speed of our algorithm are obtained without the use of the pace-hindering correct rounding operation. Other advantages include the lack of division operations and the presence of only one multiplication opera-



```
The x value of the float type is read
as a sequence of bits, which we then write
as an integer:

int i = *(int*)&x;
```

```
23 digits of initial mantissa are reserved:
i_m0 = i & 0x007FFFFF;
and normalized due to the input range
of x(i_m):
i1 = (i_m + const.) << a;
i2 = (i_m0) << a;
```

```
Scaling the obtained result:
I1 = (int)((40516878 * (int64_t)i1) >> 23
Calculated Mantissa
i = (i1 >> 6) – 0x800000;
```

```
8 bits for exponent are reserved:
I_exp = i_exp & 0x7f800000;
Mantissa is combined together
with exponent into one number:
I_exp |= I;
All the result is converted back into float:
y = *(float*) & i_exp;
```

Fig. 1. Low-latency square root CORDIC algorithm.

tion to scale the result. Moreover, the algorithm achieves the expected accuracy in the entire range of normalized floating-point numbers. In this article, we use the following approach to calculate the square root of floating-point numbers [30]. IEEE-754 floating point formats are as follows:

$$-1^{S_x} \cdot M_x \cdot 2^{e_x - bias}. \tag{1}$$

where, for the square root function, $S_x = 0$.
Therefore, it is worth noting that we will use exclusively a biased exponent e_x, and mantissa $M_x \in [1, 2)$. For computing a square root, we have to represent argument x by two separate numbers – exponent and mantissa. Next, the operation presented in Eq. (2) or Eq. (3) is performed:

$$\sqrt{M_x}\, 2^{\frac{e_x - bias}{2}} \quad \text{, if } e_x - bias \text{ is an even number} \tag{2}$$

and:

$$\begin{aligned} &\sqrt{2M_x}\, 2^{\frac{e_x - bias - 1}{2}} \\ &= \sqrt{2}\,\sqrt{M_x}\, 2^{\frac{e_x - bias - 1}{2}} \quad \text{, if } e_x - bias \text{ is an odd number .} \end{aligned} \tag{3}$$

From these formulas, one may conclude that the mantissa must be placed in two intervals $M_x \in [1, 2)$ and $2M_x \in [2, 4)$. It can be seen from the above equations that it is necessary to calculate the square root of mantissa M_x. For this reason, we suggest using the CORDIC algorithm. The classical CORDIC algorithm for calculating the square root gives the following equations (vector mode):

$$x_{i+1} = x_i - \sigma_i \, y_i \, 2^{-i} \tag{4}$$

$$y_{i+1} = y_i - \sigma_i \, x_i \, 2^{-i}$$

$$x_0 = M_x + 0.25, \;\; y_0 = M_x - 0.25$$

$$M_x \in [0.03, 2.33]$$

$$\sqrt{M_x} \approx x_{m+1} P'$$

where P' is the scaling factor, taking into account the repetitions of some iterations (4.4 and 13.13 for example). We have observed that from Eq. (5). It is necessary to expand the range of values to $M_x \in [1, 4)$ instead of 0.03–2.33. Therefore, we proposed to change the constant from 0.25 to 1.0. In this case, we have:

$$x_0 = M_x + 1, \;\; y_0 = M_x - 1\,, \tag{5}$$

$$\sqrt{4\,M_x} \approx 2\,x_{m+1}\,P'\,.$$

In the proposed algorithm, all conversions must be performed in the integer format, as required by CORDIC. The main steps of the new algorithm are presented in Fig. 1 and are summarized as follows:

1) Input argument x is given as a single precision floating point number.
2) x is converted into integer i.
3) Using `0x7f800000` and `0x00ffffff` masks, i is split into two parts: biased exponent i_exp and i_m – the fractional part of the mantissa with the youngest bit of the biased exponent acting as the oldest bit of the mantissa. This bit indicates which range of the mantissa is considered: M_x or $2M_x$. If $i_m \geqslant$ `0x00800000`, then the mantissa is in the $[1, 2)$ range, otherwise in the $[2, 4)$ range.
4) This allows us to set the appropriate values of the exponent of the result – see Eqs. (2) and (3).
5) The initial values of numbers i_1 and i_2 are introduced, corresponding to variables x_0 and y_0 of the CORDIC algorithm for SQRT calculation; see Eq. (5). Depending on the range of the mantissa, M_x or $2M_x$, these are the initial values of x_0 and y_0. These constants are the newly proposed elements of the presented algorithm.
6) In addition, in the CORDIC algorithm, three additional bits are used to increase the accuracy of calculating the square root of the mantissa, up to the available 23-24 bits of mantissa available in float-type numbers.
7) The result obtained is scaled by integer multiplication: $(\mathrm{int})((40516878 * (int64_t)i_1) >> 23)$.

4. Proposed Algorithm

The aforementioned methodology is implemented in the algorithm presented in Fig. 2.

```
float Sqrt_Cordic_1 (float x)
{ float y;
 int32_t i_m, i_exp;
 int32_t i, i1, i2, i_m0, i_m1, di1, di2,j;
 i = *(int*)&x;
 i_exp = i & 0x7f800000;
 i_m = i & 0x00ffffff;
 i_m0 = i & 0x007fffff;
  if (i_m>= 0x00800000){
      i1=(i_m+8388608)<<3; i2=i_m0<<3}
     else{i1=((i_m<<1)+25165824)<<3;
       i2=((i_m<<1)+ 8388608)<<3;}
  di1 = (i2 >> 1); di2 = (i1 >> 1);
  i1 = i1 - di1; i2 = i2 - di2;
 for (j = 2; j <=12; j++ )
  {di1 = (i2 >> j); di2 = (i1 >> j);
  if (i2 >= 0) { i1 = i1 - di1; i2 = i2 - di2;}
     else { i1=i1+di1; i2=i2+di2; }
  if (j= =4){ di1 = (i2 >> j); di2 = (i1 >> j);
    if (i2 >= 0) { i1 = i1 - di1; i2 = i2 - di2;}
      else { i1=i1+di1; i2=i2+di2; }
  }
  }
i1 = (int)(( 40516878* (int64_t)i1)>> 23) ;
i=(i1>>6)- 8388608;
i_exp=(i_exp + 0x3f800000)>>1;
i_exp = i_exp &0x7f800000;
i_exp |= i;
y = *(float*)&i_exp;
  return y;
}
```

Fig. 2. Square root floating-point function algorithm.

The proposed floating point CORDIC function was implemented on several families of FPGAs. For this purpose, the Xilinx Vitis HLS automatic synthesis and implementation software were used, generating Verilog code at the RTL level. The project was implemented by utilizing basic FPGA resources, such as LUT-based logic, multipliers from DSP blocks, flip-flops (FFs), etc.

The algorithm generates results from 2 clock cycles for the fastest FPGAs (i.e. Versal, Virtex-7 Ultra Plus, Kintex-7 Ultra Plus), up to 8 clocks for the slowest FPGA Spartan-7. The CORDIC function written in C, as presented in this paper, is easily implementable on FPGA chips and requires a small amount of resources.

The Xilinx FPGA 7 family is optimized for low-power applications requiring serial transceivers, fast DSP and high logic throughput. The logic is based on real 6-input LUT technology, configurable as distributed memory. DSP slices are designed with a 25×18 multiplier, a 48-bit accumulator, and a pre-adder for high performance filtering, including optimized symmetric coefficient filtering [31].

The Artix-7 family includes up to 215 K logic cells, 13 Mb RAM block, and 740 DSP slices. The more sophisticated families, i.e. Kintex-7 and Virtex 7, include: in the case of Kintex-7 – up to 478 K logic cells, 34 Mb RAM block, and 1920 DSP slices. In the case of Virtex-7 – up to 1955 K logic cells, 68 Mb RAM block, and 3600 DSP slices.

The AMD Xilinx Zynq UltraScale+ MPSoC family is based on the UltraScale MPSoC architecture. This series integrates a 64-bit quad-core or dual-core Arm Cortex-A53 and dual-core Arm Cortex-R5F-based processing system (PS) and Xilinx programmable logic (PL) UltraScale architecture in

Tab. 1. Results of logic synthesis of SQRT FP CORDIC using Xilinx Vitis HLS tool.

FPGA	T [ns]	No. of cycles	No. of DSP	No. of LUT	LUT usage	No. of FF	FFs usage
Artix-7	70	7	3	2878	6.7%	642	35.96%
Kintex-7	50	5	3	2870	7.00%	493	0.60%
Kintex-7U	40	4	2	2871	0.43%	395	0.03%
Kintex-7 UP	20	2	2	2860	1.76%	252	0.08%
Spartan-7	80	8	3	2886	36.06%	697	4.63%
Virtex-7	50	5	3	2870	1.40%	438	0.11%
Virtex UP	20	2	2	2860	0.72%	155	0.02%
Zynq-7000	70	7	3	2878	16.35%	607	1.72%
ZynqUP	30	3	2	2866	3.26%	269	0.15%
Versal	20	2	2	2698	2.40%	230	0.01%

Tab. 2. Results of FPGA implementation of SQRT FP CORDIC by Xilinx Vitis HLS.

FPGA	T [ns]	No. of cycles	No. of DSP	DSP usage	No. of LUT	LUT usage	No. of FF	FF usage
Artix-7	70	7	4	8.88%	982	12.28%	416	2.60%
Kintex-7	50	5	4	1.70%	862	2.10%	262	0.32%
Kintex-7U	40	4	2	0.04%	862	0.13%	210	0.02%
Kintex-7 UP	20	2	2	0.15%	1042	0.64%	199	0.06%
Spartan-7	80	8	4	20.00%	976	12.2%	464	2.9%
Virtex-7	50	5	3	0.27%	861	0.42%	232	0.05%
Virtex UP	20	2	2	0.09%	858	0.22%	82	0.01%
Zynq-7000	70	7	4	5.0%	1099	6.24%	468	1.33%
ZynqUP	30	3	2	0.27%	905	1.03%	136	0.08%
Versal	20	2	2	0.1%	810	0.72%	122	0.007%

a single device. Also included are on-chip memory, multiport external memory interfaces, and a large set of peripheral connectivity interfaces. The Zynq UltraScale+ MPSoC family includes up to 1.14 M logic cells, in particular 522 K CLB LUT, 1 M CLB flip-flops, 984 Mb RAM block, and 2 M DSP slices [32].

5. Achieved Results

The maximum negative and positive errors achieved by the proposed method are –1.7001956E–07 and 1.0053241E–07, respectively. Additionally, we can observe four special cases for normalized numbers:

- zero $x = 0.0$; $y = 7.6664669522108749\text{E–}20$,
- min $x = 1.17549421069244107548702\text{9E–}38$; $y = 1.0842021078620191\text{E–}19$,
- max $x = 3.40282346638528859811704\text{2E+}38$, $y = 1.8446742974197924\text{E+}19$,
- $x = \infty$; $y = 1.8446744073709552\text{E+}19$.

We checked the relative errors over the full range of normalized single-precision floating-point numbers using the $\mathrm{nextafterf}()$ function for the C++ code of our algorithm, comparing the results with the `sqrt` function of the `cmath` library. $\mathrm{nextafterf(a, b)}$ is a function defined in the C++ cmath library. Return the next representable value after x in the y direction. The relative error was calculated as:

$$dr = (\text{double})\frac{y}{\sqrt{x}} - 1 \, .$$

The y results from the C++ code were compared with the results obtained in FPGA – the integer number results were always the same.

Table 1 presents the execution time of the proposed square root floating point (FP) CORDIC algorithm and FPGA resources utilized after logical synthesis on the chips. Table 2 collects the same results after FPGA implementation by means of the Xilinx Vitis HLS tool that provides optimization of the resources used.

First, one of the widely used FPGA Artix-7 chips was chosen. The clock frequency was set to a default value of 100 MHz, because many other FPGAs can operate at this frequency. This makes it easier to compare the achievements of FPGAs originating from different families. The execution time and the resources used after the logical synthesis using the Xilinx Vitis tool for Artix-7 are presented in Tab. 1. Our floating point CORDIC function required 7 clock cycles to generate output, i.e. 70 ns.

Implementing the proposed function on the smallest available chip of the Artix-7 family (`xc7a12t-cpg238-3`) required 4 (8.8%) built-in DSP blocks, 982 (12.28%) LUTs, and 416 (2.6%) flip-flops. Xilinx does not disclose the results of the own implementation of the CORDIC v. 6.0 IP on Artix-7 FPGAs [29].

Next, the Kintex-7 FPGA was tested. The proposed function took 5 clock cycles (50 ns) to complete. Implementation of the algorithm on the smallest chip of the Kintex-7 family (`xc7k70t-fbv676-3`) required 4 (1.7%) built-in DSP blocks, 862 (2.10%) LUTs, and 262 (0.32%) flip-flops. According to Xilinx [29], their square root in CORDIC IP occupied 2184 LUTs, and 1328 flip-flops on the Kintex-7 chip, which is approximately 2.5 times more LUTs, and 5 times more flip-flops than in the proposed solution.

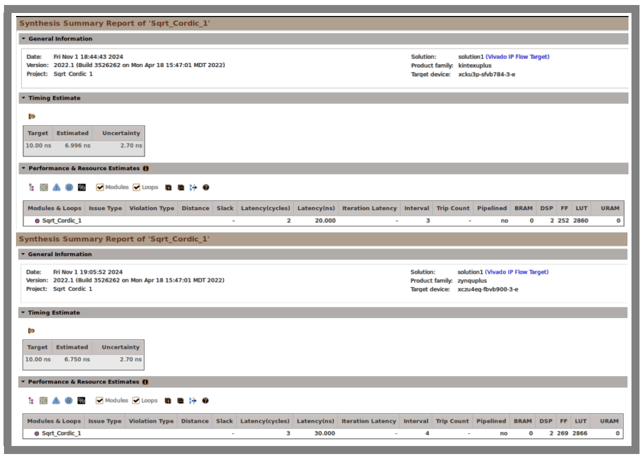
Furthermore, we implemented the algorithm on the Kintex-7 Ultra (Kintex-7U) FPGA. The execution took 4 clock cycles (40 ns) on the smallest chip of the Kintex-7 Ultra family (`xcku115-flva1517-3-e`) and required 2 (0.04%) built-in DSP blocks, 862 (0.13%) LUTs, and 210 (0.02%) FFs. According to Xilinx [29], their SQRT function occupied 2278 LUTs and 1336 flip-flops, which is approximately three times more than for the proposed solution.

The last FPGA variant of the Kintex-7 family was Kintex-7 Ultra Plus (Kintex-7 UP). In this case, the algorithm was executed in 2 clock cycles (20 ns) only. It means that implementation on the smallest chip of the Kintex-7 Ultra Plus family (`xcku3p-sfvb784-3-e`) required 2 (0.15%) built-in DSP blocks, 1042 (0.64%) LUTs, and 199 (0.06%) FFs. The Xilinx solution uses 2208 LUTs and 1334 FFs. This is also approximately 2.12 times LUTs more and 6.7 times more flip-flops than for the proposed solution.

Tables 1 and 2 illustrate that the proposed function implemented on different FPGAs of the Kintex-7 family occupied a similar amount of resources.

Moreover, we also tested the Spartan-7 FPGA (`xc7s15ftgb196-2`), a chip with a slightly different and older architecture when compared to the previous family. In this case, the proposed function achieved the worst result of 8 clock cycles (80 ns), using a small portion of the resources available, i.e. 4 (20%) built-in DSP blocks, 976 (12.2%) LUTs, and 464 (2.9%) flip-flops. This result is a consequence of the lower quantity of resources in this chip: 20 DSP, 8000 LUTs, and 16000 FFs. Xilinx does not disclose the results achieved while implementing CORDIC v. 6.0 IP on Spartan-7 FPGAs [29].

The Virtex-7 series FPGA is optimized for the best performance and capacity and is used in the verification `xc7vx330t-ffv1761-3` chip, which offers abundant re-



Synthesis Summary Report of 'Sqrt_Cordic_1'

General Information

Date: Fri Nov 1 18:44:43 2024
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022)
Project: Sqrt Cordic 1
Solution: solution1 (Vivado IP Flow Target)
Product family: kintexuplus
Target device: xcku3p-sfvb784-3-e

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	6.996 ns	2.70 ns

Performance & Resource Estimates

Modules ✓ Loops ✓

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
Sqrt_Cordic_1				-	2	20.000	-	3	-	no	0	2	252	2860	0

Synthesis Summary Report of 'Sqrt_Cordic_1'

General Information

Date: Fri Nov 1 19:05:52 2024
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:47:01 MDT 2022)
Project: Sqrt Cordic 1
Solution: solution1 (Vivado IP Flow Target)
Product family: zynquplus
Target device: xczu4eg-fbvb900-3-e

Timing Estimate

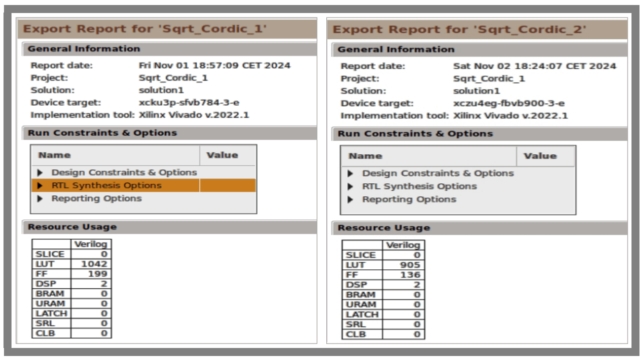
Target	Estimated	Uncertainty
10.00 ns	6.750 ns	2.70 ns

Performance & Resource Estimates

Modules ✓ Loops ✓

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
Sqrt_Cordic_1				-	3	30.000	-	4	-	no	0	2	269	2866	0

Fig. 3. Results of the synthesis of Xilinx Vitis HLS on: Kintex Ultra Plus FPGA (top) and Zynq Ultra Plus FPGA (bottom).



Export Report for 'Sqrt_Cordic_1'

General Information

Report date: Fri Nov 01 18:57:09 CET 2024
Project: Sqrt_Cordic_1
Solution: solution1
Device target: xcku3p-sfvb784-3-e
Implementation tool: Xilinx Vivado v.2022.1

Run Constraints & Options

Name	Value
Design Constraints & Options	
RTL Synthesis Options	
Reporting Options	

Resource Usage

	Verilog
SLICE	0
LUT	1042
FF	199
DSP	2
BRAM	0
URAM	0
LATCH	0
SRL	0
CLB	0

Export Report for 'Sqrt_Cordic_2'

General Information

Report date: Sat Nov 02 18:24:07 CET 2024
Project: Sqrt_Cordic_1
Solution: solution1
Device target: xczu4eg-fbvb900-3-e
Implementation tool: Xilinx Vivado v.2022.1

Run Constraints & Options

Name	Value
Design Constraints & Options	
RTL Synthesis Options	
Reporting Options	

Resource Usage

	Verilog
SLICE	0
LUT	905
FF	136
DSP	2
BRAM	0
URAM	0
LATCH	0
SRL	0
CLB	0

Fig. 4. Results of implementation of Xilinx Vitis HLS on Kintex Ultra Plus FPGA (left) and on Zynq Ultra Plus FPGA (right).

sources and achieves top performance in the Xilinx FPGA family. The execution time of the proposed function was 5 clock cycles (50 ns) with the usage of only 3 (0.27%) built-in DSP blocks, 861 (0.42%) LUTs, and 232 (0.05%) FFs. According to Xilinx [29], their SQRT solution occupied 2177 LUTs and 1328 FFs, which is approximately 2.5 times more LUTs, and 5.7 times more FFs than in the case of our solution.

The Virtex-7 Ultra Plus (`xcvu3pCIV-ffvc1517-3-e`) algorithm took only two clock cycles (20 ns) to complete. The function occupies merely 2 DSP blocks (0.09%), a low number of 858 LUTs (0.22%), and 82 flip-flops (0.01%). Compared to Xilinx [29], their SQRT IP occupied 2242 LUTs and 1342 flip flops, which is approximately 2.6 times more LUTs and 16 times more FFs than in the proposed solution.

On Zynq Ultra Plus (`xczu4eg-fbvb900-3-e`) FPGA the algorithm required only 2 built-in DSP blocks (0.27%), 905 LUTs (1.03%), and 136 FFs (0.08%). The function was executed within 3 clock cycles (30 ns), while Xilinx [29] SQRT occupied 2177 LUTs, and 1330 FFs, which is about 2.4 times more LUTs, and 9.7 times more flip-flops compared to proposal.

We also tested an older version of the Zynq-7000 (`xc7z010-clg225-3`) FPGA chip, with the execution taking 7 clock cycles (70 ns) and using 4 built-in DSP blocks (5%), 1099 LUTs (6.24%) and 468 flip-flops (1.33%). Compared to Xilinx [29], their SQRT occupied 2177 LUTs and

1330 flip-flops, which is approximately 2 times more LUTs, and 2.8 times more FFs than in our proposal.

Finally, we implemented the algorithm in the most advanced chip family, namely the Versal.

The implementation required a negligible amount of resources of the smallest Versal chip (`xcvc1902-viva1596-3HP-e-S`), i.e. only 2 DSP blocks (0.1%), 810 (only 0.72%) LUTs, and 122 FFs (0.007% of all FFs available). The algorithm required only 2 clock cycles (20 ns) to complete. In comparison to Xilinx, their solution occupied 1968 LUTs and 1454 FFs in this FPGA, which is approximately 2.43 times more LUTs and 11.9 times more flip-flops than in the case of our proposal. All the results of logic synthesis performed using the Xilinx Vitis HLS software are presented in Tab. 1, while Tab. 2 illustrates the results of the FPGA implementations of the proposed CORDIC function.

These results have been optimized due to the architectural details of each selected FPGA. Details concerning the time of the signal's propagation through a CLB for the basic families of FPGAs, retrieved from Xilinx data sheets, are presented in Tab. 3. Data on the operating speed of FPGAs utilized in the experiments highlights the time efficiency of the proposed algorithm in the context of its complexity.

Example results of Xilinx Vitis HLS synthesis on Kintex Ultra Plus FPGA are shown in Fig. 3, while results of Xilinx Vitis HLS implementation on Kintex Ultra Plus FPGA are presented in Fig. 4.

The usage of FPGA resources required to implement CORDIC-based modules usually exceeds 1100 LUTs and 1000 flip-flops. Our solution achieved a result that was 20% better. The CORDIC structure [33] was of the combined iterative three-stage or multistage variety and required slightly more resources than our solutions inside the Kintex-7 FPGA. However, only the sum of LUTs and flip-flops together is given, and unlike our algorithm, those need ROM memory. The execution time ranges from 60 ns to 190 ns, meaning it is longer than the time achieved by us (50 ns on Kintex-7).

An FPGA implementation of the CORDIC floating point SQRT performed on Virtex-7 in the iterative pipeline-parallel version, as presented in [23], occupied 708 LUTs. However, it required weighting the scale factors by applying an additional 25-bit fixed-time expensive multiplication to generate final results. The implementation of the complex SQRT method proposed in [24] on the Virtex-6 FPGA occupied 6852 LUTs and generated a latency of 38 clock cycles. The square root calculated on the Virtex-7 Ultra Plus FPGA by the Radix-10 CORDIC algorithm presented in [11] required from 1193 (7 digit version) to 5796 LUTs (34 digits version) and from 339 (7 digits) up to 1481 (34 digits) flip-flops, depending on the number of precision digits. Latency ranged from 10 clock cycles for 7 digits to 37 clock cycles. It was the worst result when compared to the application of our proposal on the same FPGA family (2 clock cycles).

Our solution also consumed approximately 25% less FPGA resources. As a further example, a faster radix-4 root CORDIC algorithm with a 40-bit precision level required 9324 LUTs when implemented on the Virtex-6 FPGA [21]. This is a hyperbolic CORDIC utilizing Taylor's approximation. The concurrent radix-4 CORDIC solution proposed in [26] was implemented in the Spartan-6 FPGA and occupied 6840 LUTs on this FPFA, with a latency of 68 ns. For the older FPGA version (Spartan-3E), 4508 LUTs were used and latency equaled 80 ns.

Despite the lower amount of hardware resources required, the relatively high latency of [10] was achieved for his high-performance SQRT circuit based on the Taylor series. The authors of [25] focused on maximizing operating frequency as well as reducing static and dynamic power levels. However, their implementation of the FP SQRT occupied, for instance, 804 basic and 971 enhanced LUTs of the Virtex-5 FPGA. Our result of 861 LUTs achieved on the Virtex-7 FPGA ranks us in the middle of their range.

One may notice that sophisticated solutions require similar to proposed bit of precision amount of FPGA resources and a higher number of clock cycles to complete specific functions.

Implementation of fixed-point SQRT CORDIC solutions is usually more frequent in FPGAs. Therefore, it is easier to find many more publications about fixed-point algorithms implemented in FPGAs. However, the proposed FP solution often achieves a lower latency level than many of its fixed-point counterparts, displaying a similar or lower demand for FPGA resources.

Tab. 3. Signal propagation time through configurable logic block.

FPGA	Artix-7	Kintex-7	Spartan-7	Virtex-7	Zynq-7000
Combinatorial [ns]	0.94	0.58	1.05	0.58	0.94
Sequential [ns]	0.47	0.32	0.53	0.32	0.47
CLB set and hold [ns]	0.59	0.36	0.66	0.36	0.59
Set/reset [ns]	0.53	0.52	0.78	0.52	0.53
DSP I/O [ns]	4.06	3.44	4.65	3.44	4.06

6. Conclusions

In this article, a new algorithm is presented for CORDIC square root computation for FP numbers. The original methodology of converting floating numbers to integers and back allows to optimize the usage of FPGA hardware resources and lowers the efficiency of the calculation. We achieved a very short computation time of two clock cycles on Ultra Scale Plus Xilinx FPGAs from the Kintex-7 and Virtex-7 series. The same result was achieved on the Versal FPGA. The usage of FPGA resources by the proposed solution is similar to or lower than that of the more sophisticated optimization methods presented in the literature.

The authors' contribution to the field can be summarized as follows:

- FPGA floating point CORDIC SQRT circuit for normalized numbers in the single precision IEEE754 format,
- Maximum relative error of 1.7E–7,
- Relatively simple theory, easily implementable on FPGAs,
- Low average implementation latency on widespread FPGAs,
- Very low implementation latency on Ultra Plus Families of FPGAs,
- No division operation, only one integer multiplication operation (to scale the result),
- Decent accuracy over the entire range of normalized FP numbers,
- Lower or similar utilization of FPGA resources compared with other solutions.

Our future research will focus on methods relied upon to perform fixed-point CORDIC computations of several basic functions. We currently work on a new approach to angle recoding allowing to flexibly adjust the memory table size and the number of CORDIC iterations.

References

[1] L. Moroz, V. Samotyy, M. Wegrzyn, and U. Dzelendzyak, "Efficient Floating-point Square Root and Reciprocal Square Root Algorithms", *11th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, Cracow, Poland, 2021 (https://doi.org/10.1109/IDAACS53288.2021.9660872).

[2] A. Hasnat *et al.*, "A Fast FPGA Based Architecture for Computation of Square Root and Inverse Square Root", *Devices for Integrated Circuit (DevIC)*, Kalyani, India, 2017 (https://doi.org/10.1109/DEVIC.2017.8073975).

[3] Z. Kokosinski *et al.*, "Fast and Accurate Approximation Algorithms Computing Floating Point Square Root", *Numerical Algorithms*, 2024 (https://doi.org/10.1007/s11075-024-01932-7).

[4] S. Mopuri, S. Bhardwaj, and A. Acharyya, "Coordinate Rotation-based Design Methodology for Square Root and Division Computation", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 66, pp. 1227–1231, 2019 (https://doi.org/10.1109/TCSII.2018.2878599).

[5] R. Shukla and K.C. Ray, "Low Latency Hybrid CORDIC Algorithm", *IEEE Transactions on Computers*, vol. 63, pp. 3066–3078, 2014 (https://doi.org/10.1109/TC.2013.173).

[6] M.D. Ercegovac and T. Lang, *Division and Square Root Digit-recurrence Algorithms and Implementations*, Norwell: Kluwer Publishers, 240 p., 1994 (ISBN 9780792394389).

[7] Y.H. Hu and S. Naganathan, "An Angle Recoding Method for CORDIC Algorithm Implementation", *IEEE Transactions on Computers*, vol. 42, pp. 74–79, 1993 (https://doi.org/10.1109/12.192217).

[8] E. Antelo, T. Lang, and J. Bruguera, "Very-high Radix Circular CORDIC: Vectoring and Rotation/vectoring", *IEEE Transactions on Computers*, vol. 49, pp. 727–739, 2000 (https://doi.org/10.1109/12.863043).

[9] E. Antelo, J. Villalba, J.D. Bruguera, and E. Zapata, "High Performance Rotation Architectures Based on Radix-4 CORDIC Algorithm", *IEEE Transactions on Computers*, vol. 46, pp. 855–870, 1997 (https://doi.org/10.1109/12.609275).

[10] T.-J. Kwon and J. Draper, "Floating-Point Division and Square Root Implementation using a Taylor-series Expansion Algorithm with Reduced Look-up Tables", *2008 51st Midwest Symposium on Circuits and System*, Knoxville, USA, 2008 (https://doi.org/10.1109/MWSCAS.2008.4616959).

[11] Martín Vázquez, Marcelo Tosini, Lucas Leiva., "Radix-10 Restoring Square Root for 6-input LUTs Programmable Devices", *Circuits Systems and Signal Processing*, vol. 40, pp. 2335–2360, 2021 (https://doi.org/10.1007/s00034-020-01571-y).

[12] J.E. Volder, "The CORDIC Trigonometric Computing Technique", *IEEE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, 1959 (https://doi.org/10.1109/TEC.1959.5222693).

[13] J.-G. Mailloux, S. Simard, and R. Beguenane, "FPGA Implementation of Induction Motor Vector Control using Xilinx System Generator", *6th WSEAS International Conference on Circuits, Systems, Electronics, Control & Signal Processing*, Cairo, Egypt, 2007.

[14] M. Garrido, P. Källström, M. Kumm, and O. Gustafsson, "CORDIC II: A New Improved CORDIC Algorithm", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 63, pp. 186–190, 2016 (https://doi.org/10.1109/TCSII.2015.2483422).

[15] S. Srinivasan *et al.*, "Split-path Fused Floating Point Multiply Accumulate (FPMAC)", *21th IEEE Symposium on Computer Arithmetic*, Austin, USA 2013 (https://doi.org/10.1109/ARITH.2013.32).

[16] J.S. Walther, "A Unified Algorithm for Elementary Functions", *Proc. of AFIPS Joint Computer Conferences*, vol. 38, pp. 385–389, 1971 (https://doi.org/10.1145/1478786.1478840).

[17] S. Wang, V. Piuri, and E.E. Swartzlander, "Hybrid CORDIC Algorithms", *IEEE Transactions on Computers*, vol. 46, no. 11, pp. 1202–1207, 1997 (https://doi.org/10.1109/12.644295).

[18] P.-T. Vo-Thi, T.-T. Hoang, C.-K. Pham, and D.-H. Le, "A Floating-point FFT Twiddle Factor Implementation Based on Adaptive Angle Recoding CORDIC", *2017 International Conference on Recent Advances in Signal Processing Telecommunications & Computing (SigTelCom)*, Da Nang, Vietnam, 2017 (https://doi.org/10.1109/SIGTELCOM.2017.7849789).

[19] A. Madisetti, A.Y. Kwentus, and A.N. Willson, "A 100 MHz, 16-b, Direct Digital Frequency Synthesizer with 100-dBc Spurious-free Dynamic Range", *IEEE Journal of Solid-State Circuits*, vol. 34, no. 8, pp. 1034–1043, 1999 (https://doi.org/10.1109/4.777100).

[20] D. Timmermann, H. Hahn, and B. Hosticka, "Low Latency Time CORDIC Algorithms", *IEEE Transactions on Computers*, vol. 41, pp. 1010–1015, 1992 (https://doi.org/10.1109/12.156543).

[21] M. Woźniak *et al.*, "Radix 4 CORDIC Algorithm Based Low Latency and Hardware Efficient VLSI Architecture for *N*th Root and *N*th Power Computations", *Scientific Reports*, vol. 13, art. no. 20918, 2023 (https://doi.org/10.1038/s41598-023-47890-3).

[22] R. Dutt and A. Acharyya, "Low-complexity Square-root Unscented Kalman Filter", *Circuits, Systems, and Signal Processing*, vol. 42, pp. 6900–6928, 2023 (https://doi.org/10.1007/s00034-023-02437-9).

[23] B. Li *et al.*, "A Unified Reconfigurable Architecture Based on CORDIC Algorithm Floating-point Arithmetic", *2017 International Conference on Field Programmable Technology (ICFPT)*, Melbourne, Australia, 2017 (https://doi.org/10.1109/FPT.2017.8280166).

[24] S. Mopuri and A. Acharyya, "Low-complexity and High-speed Architecture Design Methodology for Complex Square Root", *Circuits, Systems, and Signal Processing*, vol. 40, pp. 5759–5772, 2021 (https://doi.org/10.1007/s00034-021-01738-1).

[25] S. Suresh, S.F. Beldianu, and S.G. Ziavras, "FPGA and ASIC Square Root Designs for High Performance and Power Efficiency", *2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors*, Washington, USA, 2013 (https://doi.org/10.1109/ASAP.2013.6567588).

[26] M.A. Darshan, "A High Performance and Low Latency FPGA Implementation of CORDIC Algorithm", *International Journal of Scientific & Engineering Research*, vol. 4, no. 8, 2013 (ISSN 22295518).

[27] T.-B. Juang, S.-F. Hsiao, and M.-Y. Tsai, "Para-CORDIC: Parallel CORDIC Rotation Algorithm", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 51, pp. 1515–1524, 2004 (https://doi.org/10.1109/TCSI.2004.832734).

[28] T. Juang, "Low Latency Angle Recoding Methods for the Higher Bit-width Parallel CORDIC Rotator Implementations", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 55, pp. 1139–1143, 2008 (https://doi.org/10.1109/TCSII.2008.2002566).

[29] Xilinx, "CORDIC v6.0 LogiCORE IP Product Guide", 2021.

[30] F. de Dinechin, M. Joldes, B. Pasca, and G. Revy, "Multiplicative Square Root Algorithms for FPGAs", *2010 International Conference on Field Programmable Logic and Applications*, Milan, Italy, 2010 (https://doi.org/10.1109/FPL.2010.112).

[31] AMD Xilinx, "7 Series FPGAs Data Sheet: Overview DS180 (v2.6.1)", product specification, 2020.

[32] AMD Xilinx, "Zynq UltraScale+ MPSoC Data Sheet: Overview DS891 (v1.10)", product specification, 2022.

[33] M. Qin *et al.*, "A Low-latency RDP-CORDIC Algorithm for Real-time Signal Processing of Edge Computing Devices in Smart Grid Cyber-physical Systems", *Sensors*, vol. 22, art. no. 7489, 2022 (https://doi.org/10.3390/s22197489).

Mariusz Węgrzyn, Ph.D.
Faculty of Electrical and Computer Engineering
https://orcid.org/0000-0002-6938-2954
E-mail: mariusz.wegrzyn@pk.edu.pl
Cracow University of Technology, Cracow, Poland
https://www.pk.edu.pl

Stepan Voytusik, D.Sc.
Department of Information Technology Security
https://orcid.org/0000-0003-4234-3303
E-mail: voytusik.b@gmail.com
Lviv Polytechnic National University, Lviv, Ukraine
https://lpnu.ua/en

Nataliia Gavkalova, Prof.
Faculty of Mechanical and Industrial Engineering
https://orcid.org/0000-0003-1208-9607
E-mail: nataliia.gavkalova@pw.edu.pl
Warsaw University of Technology, Warsaw, Poland
https://eng.pw.edu.pl