**International Journal of Power Electronics and Drive System (IJPEDS)**
Vol. 11, No. 3, September 2020, pp. 1153~1164
ISSN: 2088-8694, DOI: 10.11591/ijpeds.v11.i3.pp1153-1164 

# Review of fast square root calculation methods for fixed point microcontroller-based control systems of power electronics

**Anton Dianov[1], Aleksey Anuchin[2]**
[1] Digital Appliances Business, Samsung Electronics, Republic of Korea
[2] Electrical Drives Department, Moscow Power Engineering Institute, Russia

**Article Info**

***Article history:***

Received Sep 23, 2019
Revised Jan 9, 2020
Accepted Feb 16, 2020

***Keywords:***

Approximate computing
Approximation algorithms
Newton method
Numerical methods
Root mean square

**ABSTRACT**

Square root calculation is a widely used task in real-time control systems especially in those, which control power electronics: motors drives, power converters, power factor correctors, etc. At the same time calculation of square roots is a bottle-neck in the optimization of code execution time. Taking into account that for many applications approximate calculation of a square root is enough, calculation time can be decreased with the price of precision of calculation. This paper analyses existing methods for fast square root calculation, which can be implemented for fixed point microcontrollers. It discusses algorithms' pros and cons, analyses calculation errors and gives some recommendations on their use. The paper also proposes an original method for fast square root calculation, which does not use hardware acceleration and therefore, is suitable for implementation at a variety of modern Digital Signal Processors, which have high-speed hardware multipliers, but do not have effective dividers. The maximum relative error of the proposed method is 3.36% for calculation without division, and can be decreased to 0.055% using one division operation. Finally, the most promising methods are compared and results of their performance comparisons are depicted in tables.



***Corresponding Author:***

Anton Dianov,
Digital appliances business, Samsung Electronics,
129, Samsungro, Youngtong-gu, Suwon, Goenggi-do, 16677, Republic of Korea.
Email: anton.dianov@gmail.com

## 1. INTRODUCTION

Modern control systems of power electronics have extended functionality and large code to be executed. At the same time the most of mass produced devices are subject for cost optimization, therefore developers frequently select cheap microcontrollers, which are not powerful. The bulk of the code including math functions and model of control objects are run every sampling period of the system, which is typically equal to modulation period. The modulation period of the majority of power electronics control systems lies in the interval of 4 - 20 kHz, while operating frequency of the cost-effective microcontroller belong to the range of 40 - 80 MHz, which gives 2,000-20,000 processor cycles per modulation period. High power devices operate at lower modulation frequencies, while lower power electronics and devices with lower inductances, which are widely used in home appliances, automotive and industry, operate at higher modulation frequencies. The price of high power electronics is also high, so increase of microcontroller's cost in 1 - 2 $ does not impact significantly the total price of device, but the cost of the low power units is low, and the pressure form competitors make every cent significant. So developers of power electronics put

cheap Digital Signal Processors (DSP) in their solutions and then try to optimize software to be able run in the selected DSP. One of the main milestones in the way of optimization is square root calculation routine.

Square root operation is frequently used in many control systems of power electronics and digital signal processing algorithms. Tasks with square roots can be divided into two groups. Tasks from the first group are executed every sampling period and process newly sampled data. Tasks from the second group are executed rarely (with a frequency less than 120 Hz), when measurement results are needed and process data from multiple samples. First group includes voltage and current vector transformations and normalizations, models of control objects, observers, etc. Second group includes calculation of Root Mean Squares (RMS) and Total Harmonic Distortion (THD), least squares based algorithms, etc. Square root operation is also intensively used in electrical drives, where it is involved into motor models calculation, Maximum Torque Per Ampere (MTPA) control [1], Maximum Torque Per Voltage (MTPV) control, field weakening [2], various observers, Power Factor Correctors (PFC), etc.

Authors of this paper faced with the same problem, which arose, when 80 MHz DSP was substituted with 64 MHz DSP and software operating at 16 kHz failed due to the MCU overload. Analysis of the software revealed several bottle-necks, one of which was square root calculation routine called several times per sampling period and two times in the 60 Hz interrupt. At the same time implementation of the square root calculation routine used digit-by-digit method, which was not perfect and took a lot of processor cycles. Therefore, authors concentrated their efforts on the optimization of time used for square root calculation.

Let's formulate a problem for the discussion. For a given number S, it is necessary to find the number X so that:

$$\sqrt{S} \approx X. \tag{1}$$

Modern DSPs for cheap control systems of power electronics are typically 16 or 32-bit with 10 or 12-bit ADC. Therefore, majority of variables in control algorithms are 16-bit and intermediate results of calculations are 32-bit variables. Consequently, X is expected to be 16-bit and S is expected to be 32-bit.

The overwhelming majority of DSPs are designed for fast multiplication with addition, but has no hardware acceleration for the division operation. Consequently, implementation of the division operation is slow and typically takes as many processor cycles as number of bits of the result. For a 16-bit result, one division operation takes about 16 cycles and significantly slows down calculations. Thus, it is desired to minimize the number of divisions in the square root calculation algorithm.

In such a way our purpose is to accelerate the square root calculation with the price of decreased tolerance. As square root operation is typically used for the processing of measured signals corrupted by noise, calculation errors should be similar to the measuring errors, which are typically 0.5 – 5%. However some cases demand higher precision, so calculation methods must be able to decrease errors by running one to two additional iterations. Therefore, the calculation method must quickly calculate the square root with errors in the range of 0.5 – 5% and must have fast convergence to be able to decrease errors in one to two additional iterations.

Basically, the algorithms for square root evaluation can be divided into two main groups: iterative methods and approximation by real functions. Iterative methods comprise three classes: direct methods, algorithms based on the Newton-Raphson formula and normalization techniques [3].

Authors of [4–13] reported hardware implementation of non-iterative methods. These algorithms are mainly designed for low-resolution data (8-bit), but operate extremely fast and can be used for pre-processing of sampled data Such methods need additional hardware and cannot be used in general DSP-based algorithms. Authors of [14] used parallel calculation in reported speed increase in more than 38%.

The uncommon approach for square root calculation was reported in [16]. Authors proposed to use iterative execution of the nonlinear Infinite Impulse Response (IIR) filter to obtain the square root of the given number. This method does not involve division operation, but intensively uses multiplication with addition operations.

This paper is divided into five sections. In Section II the authors explain existing methods for the fixed point square root calculation, and discuss their pros and cons. Section III proposes a new method and discusses its tolerance. Experimental results and performance of the most promising methods are given in the Section IV. Section V contains conclusions.

## 2. EXISTING METHODS

The authors of [3] give classification and extensive review of the general square root calculation methods, but not all of them are suitable for fixed point implementation. Paper [3] can be extended with [15] and [17], which review and analyse only fixed point algorithms.

There are several square root calculations methods, which can be considered to be fast. Some of them utilize hardware features of processor core and can be implemented in the limited number of DSPs, while others are general methods, which do not depend on processor core. This paper reviews the most prospective methods and discusses their pros and cons.

### 2.1. Digit-by-digit calculation

This method for square root calculation at fixed point DSPs, is the most popular as it is easy to understand and implement. Despite relatively slow convergence, many engineers still pay attention to this method, often implementing it in hardware [18] and software [19]. This method can be considered as a basic method and can be used for the evaluation of other methods.

For better understanding the algorithm implementation in binary system, let's consider its operation with decimal numbers, which is often used for manual calculation of square roots with paper and pencil.

Suppose $X_k$ is the k-th digit of X, so:

$$X = X_k \cdot 10^k + X_{k-1} \cdot 10^{k-1} + \ldots + X_1 \cdot 10 + X_0 \quad (2)$$

The most significant digit $X_k$ is the first approximation of the square root X and can be easily found as the highest integer which satisfies:

$$X_k^2 \cdot 10^{2k} \leq S \quad (3)$$

The next digit $X_{k-1}$ of square root $X$ can be found using the following inequality:

$$S \geq (X_k \cdot 10^k + X_{k-1} \cdot 10^{k-1})^2 \quad (4)$$

which transforms into:

$$S - X_k^2 \cdot 10^{2k} \geq X_{k-1} \cdot 10^{2(k-1)}(20 \cdot X_k + X_{k-1}) \quad (5)$$

Similarly, the next digit can be calculated as:

$$S - \left( \underbrace{X_k^2 10^4}_{\substack{\text{Leftmost pair} \\ \text{subtrahend}}} + \underbrace{\left(20X_k + X_{k-1}\right)X_{k-1} \cdot 10^2}_{\text{Next pair subtrahend}} \right) \cdot 10^{2(k-2)} \geq \left( 20 \cdot \left( \underbrace{10X_k + X_{k-1}}_{\text{Root at previous step}} \right) + \underbrace{X_{k-2}}_{\text{New digit}} \right) \cdot \underbrace{X_{k-2}}_{\text{New digit}} \cdot 10^{2(k-2)} \quad (6)$$

This approach can be used recursively during the next steps to obtain the necessary number of digits. Since every digit of the result X is multiplied by 102k, it's easier to separate S into pairs of digits and calculate Xk for every pair. If S contains odd number of digits, a 0 must be added to the left. Thereafter, every pair is combined with previous step remainder and new digit satisfying:

$$R \geq (20 \cdot X_k + X_{k-1})X_{k-1} \quad (7)$$

must be found.

Let's denote root calculated at the current step as "R" and remainder of the initial value as "Rem". R is initialized with zero, and every iteration of the algorithm calculates the next digit of the root. The flowchart of the algorithm is shown in Figure 1:

Let's illustrate this algorithm with the following example: The square root of 54756 is:
So answer is 234.

This method can be extended to binary numbers. A binary system is simpler for calculation because the greatest number can be found just by comparing numbers. The only difference is that condition (7) transforms into:

$\sqrt{054756}$ } Given number

$-\frac{05}{04}$ } The first digit is 2 because $\underbrace{2^2}_{4} \le 05 < \underbrace{3^2}_{9}$

$-\frac{147}{129}$ } The second digit is 3 because $\underbrace{(20\cdot 2+3)\cdot 3}_{129} \le 147 < \underbrace{(20\cdot 2+4)\cdot 4}_{176}$

$-\frac{1856}{1856}$ } The third digit is 4 because $\underbrace{(20\cdot 23+4)\cdot 4}_{1856} \le 1856 < \underbrace{(20\cdot 23+5)\cdot 5}_{2325}$

$$R \ge (100_b \cdot X_k + X_{k-1})X_{k-1}7 \tag{8}$$

because the radix number changed from 10 to 2.
The next example shows root calculation in the binary system.

$\sqrt{100010100001_b}$ } Given number $(2209_d)$

$-\frac{10}{01}$ } First digit is 1 because $01_b \le 10_b$

$-\frac{100}{000}$ } Second digit is 0 because $\underbrace{(100_b \cdot 1_b + 1_b)1_b}_{101_b} \ge 100_b$

$-\frac{10010}{01001}$ } Third digit is 1 because $\underbrace{(100_b \cdot 10_b + 1_b)1_b}_{1001_b} \le 10010_b$

$-\frac{100110}{010101}$ } Fourth digit is 1 because $\underbrace{(100_b \cdot 101_b + 1_b)1_b}_{10101_b} \le 100110_b$

$-\frac{1000100}{0101101}$ } Fifth digit is 1 because $\underbrace{(100_b \cdot 1011_b + 1_b)1_b}_{101101_b} \le 1000100_b$

$-\frac{1011101}{1011101}$ } Sixth digit is 1 because $\underbrace{(100_b \cdot 10111_b + 1_b)1_b}_{1011101_b} \le 1011101_b$

So, result is 101111b = 47d

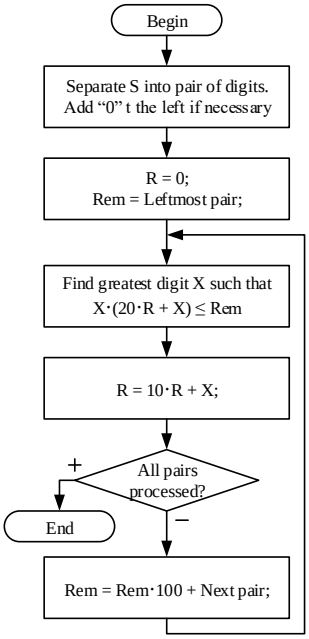


Figure 1. Flowchart of digit-by-digit method

This method does not use specific hardware and its execution time does not significantly depend on the processor type. The advantages of this method are precise results (LSB or half LSB if remainder is analysed), simple implementation and absence of division. It calculates root digit by digit, starting from the most significant, so calculations can be stopped before processing of full number S, if acceptable tolerance is reached.

However, the most significant disadvantage of the explained method is calculation time. Despite this, digit-by-digit method is frequently used in the control systems, where the processor is not highly loaded. It was also proposed for implementation in hardware and the authors of [19] enhanced this idea to calculation of other functions and implemented on FPGA.

### 2.2. Polynomial approximation

Square root function is difficult for approximation because its derivative rapidly changes close to zero, which results in high approximation errors. Therefore, square root can be successfully approximated only at the limited interval.

Polynomial approximation is proposed in [20] and recommended by engineers from Analog Devices to use with their DSPs. They propose to approximate square root function f(x) = x0.5 at the interval of [0.5..1], with fifth order polynomial:

$$\sqrt{x} \approx 0.1121216 \cdot x^5 - 0.536499 \cdot x^4 + 1.106812 \cdot x^3 - 1.34491 \cdot x^2 + 1.454895 \cdot x + 0.2075806 \quad (9)$$

Since approximating polynomial gives appropriate results in the limited interval, the initial number must be scaled to fall inside it. After calculation of the square root of the scaled number, the resulting value must be multiplied by the square root of the scaling value to obtain the correct answer. The authors of [20] published assembler code of the proposed method and claimed maximum execution time of 75 cycles.

This essential scaling operation is not simple for many DSPs and takes time, however processors from Analog Devices have hardware units called an exponent detector, which calculates the amount of the redundant sign bits and shifts initial numbers, so calculation is accelerated. Moreover, sign operation of the exponent detector limits the range of input value by 32768, which is restricting in most cases. Evaluation of the high power polynomial increases calculation error and impacts tolerance of the result. So this method is not recommended for implementation.

### 2.3. Table approximation

Lookup table-based methods are common for function approximation. They are relatively fast, but use a lot of memory to increase tolerance. Bipartite and multipartite methods are excellent examples of table lookups and are described in [18]. Authors propose inserting two lookup tables into the data paths and use one table for the initial seeds, while another one should be used for adding a small correcting value.

The approximation error can also be reduced by interpolation between table points (e.g. linear), but it increases the calculation time and diminishes the main advantage of the method - speed. The flowchart of function approximation with variable segments table and linear interpolation between points is depicted in Figure 2. It indicates that calculation contains one division, which significantly slows down calculation.

The calculation time of the function approximated with table can be significantly decreased, if the table consists of equal segments and a number of them is to the power of two. A flowchart of the corresponding algorithm is depicted in Figure 3. It indicates that calculation was simplified and division had been excluded.

Table approximation perfectly works for the smooth function with smooth derivative and outputs higher errors for the function with rapidly changing derivatives. A typical example of functions, which can be approximated easily, is sine and cosine.

As stated above, square root is not easy for approximation, because its derivative rapidly changes close to zero. Therefore, variable step table approximation or pre-scaling must be used. For this specific reason, Figure 3 contains two scaling blocks.

On the one hand, the tolerance of this method mainly depends on the size of the table and in some cases the size is unacceptably large. On the other hand, this method needs pre-scaling before table is read and after, which diminishes its main advantage - calculation speed.

### 2.4. Newton's method

This method is very popular for numerical calculation because it is simple and has fast convergence. It can be successfully used for the approximation of various functions and calculation operations [21]. Let's consider its application to the calculation of square roots.

Finding X, which is square root of S is the same as solving:

$$x^2 - S = 0 \quad (10)$$

According to Newton's method, the next step approximation xi+1 of the function root is:

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad (11)$$

where xi is root approximation at the current step. The calculation process can be illustrated by drawing series of secant lines to parabola as shown in Figure 4. The iterative calculation is stopped, when the necessary accuracy has been reached:

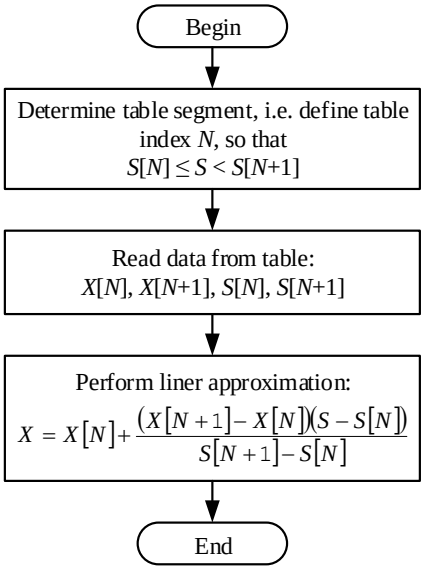


Figure 2. Flowchart of function approximation with variable segments table and linear interpolation between points

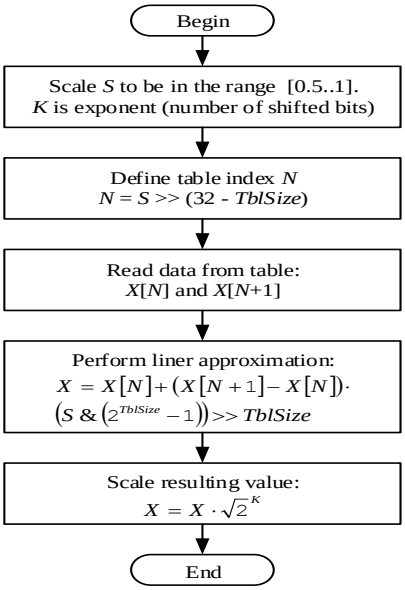


Figure 3. Flowchart of square root approximation with equally segmented table with size of $2^{\text{TblSize}}$ and linear interpolation between points

$$|x_{i+1} - x_i| \leq \varepsilon \tag{12}$$

However, for simpler calculation, the iterative process can be limited to fixed number of iterations. Thus, the function given in (10), using (11) transforms into:

$$x_{i+1} = x_i - \frac{x_i^2 - S}{2x_i} = \frac{1}{2}\left(x_i + \frac{S}{x_i}\right) \tag{13}$$

This equation is also known as the Babylonian method [22]. It has quadratic convergence and allows us to calculate square root with low number of operations. However, each iteration of the method contains division, which takes a significant amount of processor time

Properly selected initial value can significantly accelerate calculation, so an initial guess x0 is very important. We suggest to locate square root by finding n, so that square root belongs to the interval:

$$2^n \leq X < 2^{n+1} \tag{14}$$

This step can be easily implemented in the DSPs and provides advantage of using the power of two, therefore division at the first step can be substituted with bit shift. Then the initial value can be selected during this interval, using any criteria. Typical initial values are middle point of the interval:

$$x_0 = 3 \cdot 2^{n-1} \tag{15}$$

which is the most probable number, and value, which gives symmetrical relative error at interval borders:

$$x_0 = \frac{4}{3} \cdot 2^n \tag{16}$$

Using (15) as an initial value and combining with (13) results:

$$x_1 = \frac{1}{2}\left(3 \cdot 2^{n-1} + \frac{S}{3 \cdot 2^{n-1}}\right) = 3 \cdot 2^{n-2} + \frac{S}{3 \cdot 2^n} \tag{17}$$

where division can be substituted by the bit shift. However, the following steps involve division and are performed according to (13).

Let's calculate the error for every iteration in order to define the number of required steps. The relative error is higher, when X lies at the interval's lower border: X=2n.
Relative error of initial approximation:

$$\delta_0 = \frac{x_0}{X} - 1 = \frac{3 \cdot 2^{n-1}}{2^n} - 1 = \frac{1}{2} \quad (18)$$

First step approximation:

$$x_1 = 3 \cdot 2^{n-2} + \frac{2^{2n}}{3 \cdot 2^n} = \frac{9 \cdot 2^{n-2} + 2^n}{3} = \frac{13 \cdot 2^n}{12} \quad (19)$$

Relative error of the first step approximation:

$$\delta_1 = \frac{x_1}{X} - 1 = \frac{13 \cdot 2^n}{12 \cdot 2^n} - 1 = \frac{1}{12} \approx 8.3\% \quad (20)$$

Second step approximation:

$$x_2 = \frac{1}{2}\left(\frac{13 \cdot 2^n}{12} + \frac{12 \cdot 2^{2n}}{13 \cdot 2^n}\right) = \frac{313 \cdot 2^n}{312} \quad (21)$$

Relative error of the second step approximation:

$$\delta_2 = \frac{x_2}{X} - 1 = \frac{313 \cdot 2^n}{312 \cdot 2^n} - 1 = \frac{1}{312} \approx 0.32\% \quad (22)$$

Third step approximation:

$$x_3 = \frac{1}{2}\left(\frac{313 \cdot 2^n}{312} + \frac{312 \cdot 2^{2n}}{313 \cdot 2^n}\right) = \frac{195313 \cdot 2^n}{195312} \quad (23)$$

Relative error of the third step approximation:

$$\delta_3 = \frac{x_3}{X} - 1 = \frac{1}{195312} \approx 5.1 \cdot 10^{-6}\ \% \quad (24)$$

This tolerance is typically sufficient, and square root calculation needs three iterations, which involve only two divisions.

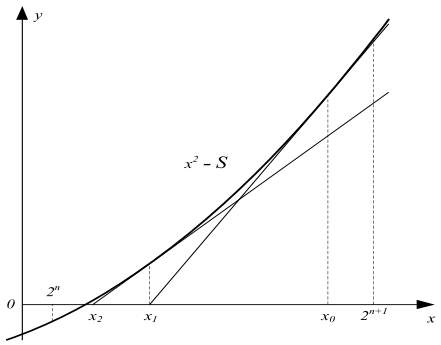


Figure 4. Geometrical interpretation of the Newton's method

### 2.5. A two-variable iterative method

This method was proposed by the authors of [23] for one of the first computers. It is applicable to the square root calculation of the number S satisfying 0 < S <3, so it also needs scaling of the initial number.

This method calculates two numbers a and c at every iteration, where a converges to square root of S and c converges to 0.

These numbers are initialized as:

$$a_0 = S; \qquad c_0 = S - 1 \tag{25}$$

and at every iteration they are updated as:

$$\begin{aligned} a_{i+1} &= a_i - 0.5 \cdot a_i c_i \\ c_{i+1} &= 0.25 \cdot c_i^2 (c_i - 3) \end{aligned} \tag{26}$$

It should be noted that for faster calculation S can be scaled to satisfy 0.5 < S <2.

The maximum relative error occurs when S = 2. Maximum relative errors for each calculation step are:

$$\begin{aligned} &\delta_0 = \sqrt{2} - 1 \approx 41.4\%; \qquad \delta_1 = \frac{1}{\sqrt{2}} - 1 \approx -29\% \\ &\delta_2 = \frac{5}{4\sqrt{2}} - 1 \approx -11.6\%; \qquad \delta_3 \approx -1.94\% \\ &\delta_4 \approx -0.056\% \end{aligned} \tag{27}$$

This method, as Newton's method, has quadratic convergence and typically 3 – 4 iterations of (26) is sufficient. This calculation method does not contain division operation and time delays inherent to it.

The problem of this method is multiplication of 32-bit numbers, which is typically slower than 16-bit number multiplication. It also truncates 64-bit numbers to 32-bit, producing calculation errors, which tend to be accumulated.

### 2.6. Goldschmidt's algorithm

One more interesting and prospective method is Goldschmidt's algorithm. Its usual description reflects a hardware orientation with difficulties of implementation in software, however the author of [24] suggests a software-friendly way of computing, which is described below.

Let y0 be a suitably good approximation to $1/\sqrt{S}$ , such as:

$$\frac{1}{2} \leq S \cdot y_0^2 \leq \frac{3}{2} \tag{28}$$

Initialize variables:

$$\begin{aligned} x_0 &= S \cdot y_0 \\ h_0 &= 0.5 \cdot y_0 \end{aligned} \tag{29}$$

And then iterate as follows:

$$\begin{cases} r_{i-1} = 0.5 - x_{i-1} \cdot h_{i-1} \\ x_i = x_{i-1} + x_{i-1} \cdot r_{i-1} \\ h_i = h_{i-1} + h_{i-1} \cdot r_{i-1} \end{cases} \tag{30}$$

Calculation can be stopped, when ri is sufficiently low or after fixed number of iterations. Calculated values converge to:

$$\begin{aligned} &\lim_{i \to \infty} x_i = \sqrt{S} \\ &\lim_{i \to \infty} 2h_i = \frac{1}{\sqrt{S}} \end{aligned} \tag{31}$$

The maximum relative error occurs when,

$$y_0 = \frac{1}{\sqrt{2 \cdot S}} \tag{32}$$

And maximum relative errors for each calculation steps are:

$$\delta_0 = \frac{1}{\sqrt{2}} - 1 \approx -29\%; \qquad \delta_1 = \frac{5}{4\sqrt{2}} - 1 \approx -11.6\%$$
$$\delta_2 = \frac{355}{256\sqrt{2}} - 1 \approx -1.94\%; \qquad \delta_3 \approx -0.056\% \tag{33}$$

The author of [24] proves that Goldsmith's algorithm, as Newton's method has quadratic convergence, but has an advantage of absence of division operation. The feature of the algorithm is that it simultaneously calculates square root and reciprocal square root. However, if one of these values is not needed, the corresponding equation in (30) can be excluded.

Equations in (30) can be effectively implemented for large numbers of DSPs, which have hardware units for multiplication with addition. However, it should be noted that variables in (30) have significantly different orders, therefore a lot of attention must be paid to their representation.

The disadvantage of this method is the necessity of initial approximation (28), which is quite difficult. The paper [24] suggests using small lookup tables for this purpose, but it takes additional memory. Another drawback of this method is propagation of errors due to rounding. The author of [24] claims that high precision results are usually achieved by starting with Goldschmidt's algorithm and then switching to the self-correcting Newton's iterations. However, for our purpose the tolerance of original algorithm is sufficient.

## 3. PROPOSED METHOD

After analysis of advantages and disadvantages of the discussed methods, the authors proposed a combined method, which is far superior to other algorithms.

Authors propose to define an initial root interval as described in (14). Then, initial approximation can be made by drawing the secant between the ends of the root interval and calculation of secant intersection with abscissa axis, Figure 5.

The equation of the secant line drawn over two points $[x_1, y_1]$ and $[x_2, y_2]$ is:

$$y(x) = \frac{y_1 - y_2}{x_1 - x_2} x + \frac{y_2 x_1 - y_1 x_2}{x_1 - x_2} \tag{34}$$

Its root can be found as:

$$x = \frac{y_1 x_2 - y_2 x_1}{y_1 - y_2} \tag{35}$$

Assuming $x_1 = 2^n$, $x_2 = 2^{n+1}$ and combining (10) with (35) results initial guess:

$$x_0 = \frac{(2^{2n} - S)2^{n+1} - (2^{2n+2} - S)2^S}{(2^{2n} - S) - (2^{2n+2} - S)} = \frac{1}{3}\left(2^{n+1} + \frac{S}{2^n}\right) \tag{36}$$

Let's calculate the maximum error Δxmax. From Figure 6 error function for the approximation with secant is:

$$\Delta x(x) = x_0 - x = \frac{1}{3}\left(2^{n+1} + \frac{x^2}{2^n}\right) - x = \frac{x^2}{3 \cdot 2^n} - x + \frac{2^{n+1}}{3} \tag{37}$$

It has maximum at the point:

$$x_{\varepsilon \max} = 3 \cdot 2^{n-1} \tag{38}$$

And maximum absolute error is:

$$\Delta x_{\max} = -\frac{1}{12} \cdot 2^n \tag{39}$$

Figure 7 illustrates error function and depicts its peak point. Maximum relative error is:

$$\delta_0 = \frac{\Delta x_{\max}}{x} = \frac{-\frac{1}{12} \cdot 2^n}{3 \cdot 2^{n-1}} = -\frac{1}{18} \approx -5.6\% \tag{40}$$

As can be seen from (40), the error is negative, so initial guess $x_0$ can be improved by adding shift.

$$x_0 = \frac{1}{3}\left(2^{n+1} + \frac{S}{2^n}\right) + A \tag{41}$$

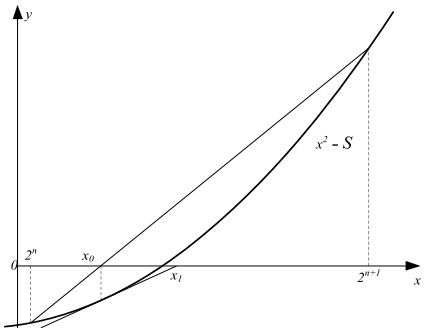


Figure 5. Geometrical interpretation of the proposed method

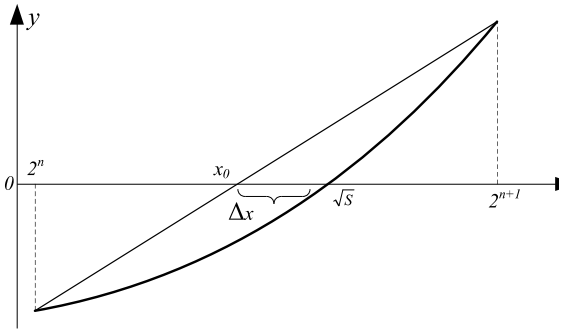


Figure 6. Error of the approximation

In order to have equal positive and negative absolute errors, offset value A should be 1/24·2n and (41) transforms into:

$$x_0 = \frac{1}{3}\left(2^{n+1} + \frac{S}{2^n}\right) + \frac{2^n}{24} = \frac{1}{3}\left(17 \cdot 2^{n-3} + \frac{S}{2^n}\right) \tag{42}$$

If we need same values of the relative errors, offset value *A* should be $0.0336735 \cdot 2^n$ and (41) transforms into:

$$x_0 = \frac{1}{3}\left(2^{n+1} + \frac{S}{2^n}\right) + 0.0336735 \cdot 2^n = \frac{1}{3}\left(1.05051025 \cdot 2^{n+1} + \frac{S}{2^n}\right) \tag{43}$$

And from (40) maximum relative error is:

$$\delta_0 = \frac{x_0}{\Delta x_{\max}} - 1 \approx 3.36\% \tag{44}$$

This error is almost twice less than the first step error in the Newton's method. If precision of (44) is not sufficient, one more calculation step should be performed. It could be performed according to (13), which is simple for implementation, but involves one division operation.

Let's find the maximum error, which corresponds to X=2n.

Initial approximation:

$$x_0 = \frac{1}{3}\left(1.05051025 \cdot 2^{n+1} + \frac{2^{2n}}{2^n}\right) = 1.0336735 \cdot 2^n \tag{45}$$

First step approximation:

$$x_1 = \frac{1}{2}\left(1.0336735 \cdot 2^n + \frac{2^{2n}}{1.0336735 \cdot 2^n}\right) \approx 1.0005485 \cdot 2^n \tag{46}$$

Relative error of the first step approximation:

$$\delta_1 = \frac{x_1}{X} - 1 = \frac{1.0005485 \cdot 2^n}{2^n} - 1 \approx 0.055\% \tag{47}$$

Which is sufficient for a wide range of engineering applications.

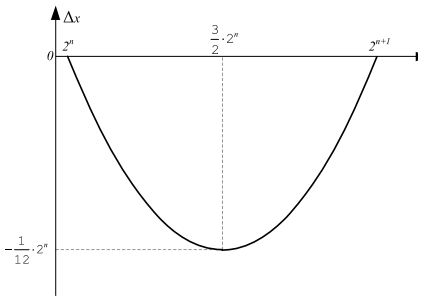


Figure 7. Error function of the proposed method

## 4. EXPERIMENTAL RESULTS

For the performance analysis, the most prospective algorithms and digit-by-digit method as a reference, have been selected. Due to the fact that the proposed algorithm is not iterative, it has been enhanced with one iteration of Newton's method in order to obtain higher precision.

Calculation time, operations and the number of processor cycles were calculated for the lower precision calculation (about 2%) and higher precision calculation (about 0.05%). The results were put into Table 1. and Table 2 for the lower and higher precision calculations, respectively. These tables contain main operations and do not include data initialization, data movement and branching, which strongly depend on the compiler optimization and conveyer mechanism of the microprocessor. The real calculation time of the selected methods is typically greater at 8-15 cycles depending on the optimization settings.

For the experimental results Cortex-M3 based microprocessor has been selected, because this core is widely used and does not have specific hardware acceleration. Timings of instruction execution were taken from the technical reference [25]. For calculation of execution time we considered the worst case scenarios, e.g. according to [25] division operation takes 2-12 cycles, depending on the data, so we used 12 for method speed evaluation.

Table 1. Complexity of calculation methods with lower precision

| Characteristics | Number of | Maximum | Operations | | | | | Number of |
|---|---|---|---|---|---|---|---|---|
| Method | iterations | error | +,-,&,\|,^ | <<, >> | Compare | × | ÷ | cycles |
| Digit-by-digit | 16 | $8 \cdot 10^{-6}$ % | 65 | 64 | 17 | 0 | 0 | 146 |
| Newton-Raphson | 2 | 0.32 % | 9 | 9 | 4 | 0 | 2 | 46 |
| Two variables method | 3 | 1.94 % | 11 | 13 | 4 | 9 | 0 | 37 |
| Goldsmith's algorithm | 2 | 1.94 % | 11 | 11 | 5 | 7 | 0 | 34 |
| Proposed method | 0 | 2.86 % | 6 | 7 | 4 | 1 | 0 | 18 |

Table 2. Complexity of calculation methods with Higher precision

| Characteristics | Number of | Maximum | Operations | | | | | Number of |
|---|---|---|---|---|---|---|---|---|
| Method | iterations | error | +,-,&,\|,^ | <<, >> | Compare | × | ÷ | cycles |
| Digit-by-digit | 16 | $8 \cdot 10^{-6}$ % | 65 | 64 | 17 | 0 | 0 | 146 |
| Newton-Raphson | 3 | $5 \cdot 10^{-6}$ % | 10 | 10 | 4 | 0 | 3 | 60 |
| Two variables method | 4 | 0.056 % | 13 | 16 | 4 | 12 | 0 | 45 |
| Goldsmith's algorithm | 3 | 0.056 % | 14 | 14 | 5 | 10 | 0 | 43 |
| Proposed method | 1 | 0.04 % | 7 | 8 | 4 | 1 | 1 | 32 |

All the multiplications were considered as 32-bit operations, which generally take 1 cycle. Every multiplication operation is supposed to involve one more shift operation for scaling of the result. Some algorithms operate with numbers of different ranges, so they may need 64-bit multiplication, which is executed in 3-7 cycles, but this issue was not considered here.

Initial scaling, which finds n satisfying (14), is performed by four stages of comparison and includes maximum 4 comparison, 3 shifts and 4 additions. Initial conditions for Goldsmith's algorithms are tougher, so it needs one more stage and takes 5 comparison, 3 shifts and 5 additions.

Experimental results prove feasibility of the proposed method and indicate that it is fastest among reviewed algorithm. Furthermore, its simplicity makes implementation in hardware possible, resulting in high speed with acceptable tolerance.

## 5. CONCLUSION

This paper analyses the existing methods for square root calculations at fixed points DSPs, and proposes an original method for the approximate calculus with acceptable tolerance. The maximum relative errors for several iterations are calculated, so the number of steps can be selected, depending on the demanded tolerance of the result. The complexity and performance of the proposed method and competitive methods were both checked and compared. The experimental results indicate significant leadership of the proposed method, thus it is recommended for implementation in new algorithms or hardware.

## REFERENCES


[1] A. Dianov, N.-S. Kim, S.-M. Lim, "Sensorless starting of horizontal axis washing machines with direct drive," *2013 International Conference on Electrical Machines and Systems (ICEMS)*, pp. 1 – 6, 2013.

[2] A. Dianov, S.-T. Lee, "Novel IPMSM drive for compact washing machine," *INTELEC 2009 - 31st International Telecommunications Energy Conference*, pp. 1 – 7, 2009.

[3] P. Montuschi, P.M. Mezzalama, "Survey Of Square Rooting Algorithms", *IEEE Proceedings - Computers and Digital Techniques*, vol. 137, issue 1, pp. 31–40, 1990.

[4] T. Sutikno, "A Simple Strategy To Solve Complicated Square Root Problem In DTC For FPGA Implementation", *IEEE Symposium on Industrial Electronics and Applications (ISIEA)*, pp. 691–695, 2010.

[5] R. Rodriguez, J. Rodriguez, R.A. Gomez, "Fast Square Root Calculation For DTC Magnetic Flux Estimator", *IEEE Latin America Transactions*, vol. 12, issue 2, pp. 112–115, 2014.

[6] M.H. Sheu, S.H. Lin, "Fast Compensative Design Approach For The Approximate Squaring Function", *IEEE Journal of Solid-State Circuits*, vol. 37, issue 1, pp. 95–97, 2002.

[7] A.A. Hiasat, H.S. Abdel-Aty-Zohdy, "Combinational Logic Approach For Implementing An Improved Approximate Squaring Function", *IEEE Journal of Solid-State Circuits*, vol. 34, issue 2, pp. 236–240, 1999.

[8] A. Eshraghi, T.S. Fiez, K.D. Winters, T.R. Fischer, "Design Of A New Squaring Function For The Viterbi Algorithm", *IEEE Journal of Solid-State Circuits*, vol. 29, Issue 9, pp. 1102–1107, 1994.

[9] A. Vázquez, J.D. Bruguera, "Iterative Algorithm And Architecture For Exponential, Logarithm, Powering, And Root Extraction", *IEEE Transactions on Computers*, vol. 62, issue 9, pp. 1721–1731, 2013.

[10] J.M.P. Langlois, D. Al-Khalili, "Carry-Free Approximate Squaring Functions With O(N) Complexity And O(1) Delay", *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 53, issue 5, pp. 374–378, 2006.

[11] L. Wang, H. Zhang, K.C.L. Wong, P. Shi, "A Reduced-Rank Square Root Filtering Framework for Noninvasive Functional Imaging of Volumetric Cardiac Electrical Activity," *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 533-536, 2009.

[12] S.F. Hsieh, K.J.R. Liu, K. Yao, "A Unified Square-Root-Free Approach for QRD-Based Recursive-Least-Squares Estimation," *IEEE Transactions on Signal Processing*, vol. 41, issue 3, pp. 1405-1409, 1993.

[13] V.K. Jain, G.E. Perez, J.M. Wills, "Novel Reciprocal and Square-Root VLSI Cell: Architecture and Application To Signal Processing," *Proc. International Conference on Acoustics, Speech, and Signal Processing*, vol. 2, pp. 1201-1204, 1991.

[14] Y. Wang, J. Wang, "Fast Square-Root Detection Algorithm for V-BLAST," *Proc. International Conference on Wireless Communications, Networking and Mobile Computing*, pp. 1340-1343, 2007.

[15] N. Takagi, K. Takagi, "A VLSI Algorithm for Integer Square-Rooting," *Proc. International Symposium on Intelligent Signal Processing and Communications*, pp. 626-629, 2006.

[16] N. Mikami, M. Kobayashi, Y. Yokoyama, "A New DSP-Oriented Algorithm For Calculation Of The Square Root Using A Nonlinear Digital Filter", *IEEE Transactions on Signal Processing*, vol. 40, issue 7, pp. 1663–1669, 1992.

[17] A. Seth, W.-S. Gan, "Fixed-Point Square Roots", *2012 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1725–1728, 2012.

[18] J. Chen, J.E. Stine, "Optimizations Of Bipartite Memory Systems For Multiplicative Divide And Square Root", *48th Midwest Symposium on Circuits and Systems*, vol. 2, pp. 1458–1461, 2005.

[19] M. D. Ercegovac, "On Digit-By-Digit Methods for Computing Certain Functions", *Forty-First Asilomar Conference on Signals, Systems and Computers*, pp. 338–342, 2007.

[20] Edited by A. Mar, "Function Approximation," in "Digital signal processing applications using the ADSP-2100 family", Prentice Hall, NJ, USA: Englewood Cliffs, pp. 57–61, 1992.

[21] A. Anuchin, V. Kozachenko, V. Kulmanov, D. Shpak, "Optimization Of The Division Operation For Real-Time Control Systems", *International Siberian Conference on Control and Communications (SIBCON)*, pp. 1–4, 2015.

[22] O. Kosheleva, "Babylonian Method Of Computing The Square Root: Justifications Based On Fuzzy Techniques And On Computational Complexity", *Annual Meeting of Information Processing Society*, pp. 1–6, 2009.

[23] V. Wilkes, D. J. Wheeler and S. Gill, "Programs for An Electronic Digital Computer", Second edition. Addison-Wesley Publishing Company, 1957.

[24] P. Markstein, "Software Division and Square Root Using Goldschmidt' S Algorithms", *in Proceeding of 6th Conference on Real Numbers and Computers*, pp. 146–157, 2004.

[25] ARM Ltd., Cortex-M3 Technical Reference Manual, Rev. r2p0, http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0337h/index.html, Sep 6th 2019.