Leonid Moroz and Volodymyr Samotyy

Efficient Floating-Point Division for Digital Signal Processing Application

Floating-point division is an expensive operation for processors in digital signal processing (DSP). The basis of the division operation is finding the reciprocal of the divisor. We present a reciprocal algorithm with four multiplications for single accuracy and six for double accuracy. The algorithm specifically includes bithack (known as *magic constant*) operations and floating-point addition, multiplication, and the fused multiply-add (fma) operation. The proposed approach improves two characteristics of the division process: accuracy and number of steps. This second characteristic has a direct impact on the division operation performance time and on the amount of equipment required for hardware implementation.

Challenges of floating-point division

Division is one of the more difficult of the four arithmetic operations (addition, subtraction, multiplication, and division) in processors, microcontrollers, and field-programmable gate arrays (FPGAs), which are used for DSP. Division or square root extraction can be used for fixed-point [1], [2] and floating-point [3]–[5] numbers. All modern DSP devices execute division in floating-point format. For example, TMS320F2807x Piccolo Microcontrollers have a built-in IEEE-754 single-precision floating-point unit (FPU) [4]. Microcontrollers, such as the STM32 Cortex-M4 and STM32 Cortex-M7, are equipped with single-precision FPUs for division [3]. However, many modern floating-point microcontrollers (such as those of the Cortex A53 and Cortex A57) have fast accurate reciprocal instructions like the SSE instructions in a central processing unit (CPU) for the initial approximations, which are then improved with the help of Newton–Raphson iterations. (SSE is a set of instructions for general purpose processors. These instructions give us the ability to perform, for example, reciprocal over several processor cycles.) Some modern FPGAs also have a set of single-precision floating-point blocks for add, multiply, and fma operations [5] that can be used to the build division units. Therefore, it is important to develop appropriate algorithms to perform division operations for platforms that support floating-point computing and that do not have a hardware division unit or the proper instructions for the division.

In this article, the reciprocal of the divisor's simple algorithms (which are the basis for division) are proposed. These algorithms belong to the group of iterative algorithms. They initially receive an approximation using the so-called magic constant and are then used to implement an iterative process with Newton–Raphson formulas. The reciprocal algorithms are suitable for software and hardware implementations, e.g., in microcontrollers that lack an FPU and in FPGAs.

Known algorithms

An algorithm with magic constant for the implementation of a reciprocal function $1/x$ for float-type number x was described for the first time in [6]. It consists of the following code:

```
1.float rcp_1(float x)
2.{
3.  int i=*(int*)&x;
4.  i=0x7f000000-i;
5.  float y=*(float*)&i;
6.  y=y*(2.0f-x*y);
7.  y=y*(2.0f-x*y);
8.  return y;
9.}
```

This code, written in C/C++, will be referred to as the *reciprocal*. In line 3, we convert bits of the variable x (type float) to a variable i (type int). In line 4, we determine an initial approximation (then subject to the iteration process) of the reciprocal, where $R = 0x7f000000$ is a magic constant for IEEE-754 implementations. In line 5, we convert bits of the variable i (type int) to a variable y (type float). Lines 6 and 7 contain two classic subsequent Newton–Raphson iterations.

If the maximum relative error of calculations is designated by $|\delta_{\max}|$, then the accuracy of this algorithm is only

Digital Object Identifier 10.1109/MSP.2018.2875977
Date of publication: 24 December 2018

$$|\delta_{\max}| = 2.44 \cdot 10^{-4},$$
$$\text{or } -\log_2(|\delta_{\max}|) = 12$$

correct bits. Better accuracy with magic constants is achieved in [7] and [8]: 16.5 correct bits. However, the improved algorithm with modified Newton–Raphson iterations from [9] is more precise:

```
float rcp_2(float x)
{
  int i=*(int*)&x;
  i=0x7ef311c3-i;
  float y=*(float*)&i;
  y=y*(2.00130856f-x*y);
  y=y*(2.00000084f-x*y);
  return y;
}
```

The accuracy of the algorithm is approximately $|\delta_{\max}| = 1.014 \cdot 10^{-6}$, or 19.9 correct bits. Graphics of the initial approximations and the relative errors of these algorithms are shown; e.g., the initial approximations, $(x \in [1,2))$, are shown in Figure 1. Relative errors of the initial approximation for both algorithms are shown in Figure 2. The described algorithms include only four multiplication operations.

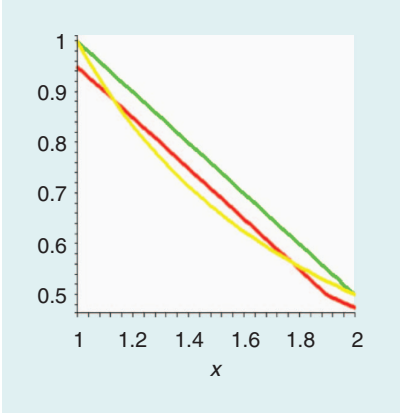

FIGURE 1. Initial approximations of the rcp_1 (green) and rcp_2 (red) algorithms. The yellow line indicates 1/*x*.

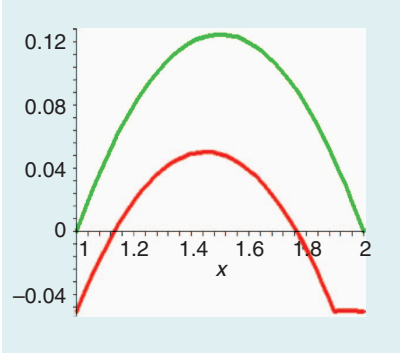

FIGURE 2. Relative errors of the initial approximations for the rcp_1 (green) and rcp_2 (red) algorithms.

The magic constant for double-type numbers is described in [7] (0x7fde9f73aabb2400) and [8] (0x7fde623822fc16e6). Fewer errors occur for the second constant at the $|\delta_{\max}| = 4.237 \cdot 10^{-11}$ level, which is 34.4 correct bits after three classic Newton–Raphson iterations (six multiplications), and at $|\delta_{\max}| = 2.22 \cdot 10^{-16}$, which is 52 correct bits after four classic Newton–Raphson iterations (eight multiplications).

This article discusses the construction of reciprocal algorithms with a diminished number of multiplications without loss of accuracy.

General theory of the algorithm

We now consider how the reciprocal algorithm works. Suppose we have a floating-point number

$$x = (-1)^{S_x} M_x \cdot 2^{E_x}, \tag{1}$$

where S_x is the sign ($S_x = \{0,1\}$, 0, 0 for positive numbers and 1 for negative numbers] and E_x is the order (exponent), which in general cases is determined by

$$E_x = \lfloor \log_2 x \rfloor = \text{floor}(\log_2 x). \tag{2}$$

The mantissa M_x is calculated according to

$$M_x = \frac{x}{2^{E_x}}. \tag{3}$$

It lies in the range $M_x = [1,2)$ and has the form $M_x = 1 + m_x$, where m_x is the fractional part of the mantissa. From this point on, to simplify the calculations, it will be assumed that $x > 0$ so that $S_x = 0$. The single-precision format of the IEEE-754 standard [11] uses a 32-bit register to store the binary representation of the number x:

- 1 bit (sign field) for S_x
- 8 bits (exponent field) for e_x, where $e_x = E_x + \text{bias}$ is a shifted order (shifted bias = 127 for single precision)
- 23 bits (mantissa field) for m_x.

The mantissa has a phantom (hidden) most significant bit that is not shown, so the appropriate 23-bit part of the 32-bit register stores only m_x. Because of this, the integer decimal number I_x corresponding to the binary representation of the number x in the IEEE-754 standard is depicted as

$$I_x = e_x \cdot N_m + m_x \cdot N_m = (e_x + m_x)N_m$$
$$= (\text{bias} + E_x + x \cdot 2^{-E_x} - 1) \cdot N_m. \tag{4}$$

$N_m = 2^{23}$ for single precision. The values of I_x could be considered as an approximation of the binary logarithm x.

If we take the binary logarithm of the number x, change its sign, and then convert it into a number y through the exponential function (base 2), we obtain the inverse of the value. This exact idea underlies the basis of the algorithm. However, float numbers in integer type are only coarse approximations of the binary logarithm x, so the inverse value obtained in this way can be considered only as an initial approximate reciprocal, which is clarified with the help of Newton–Raphson iterative equations.

The magic constant R is located in the 32-bit register as the positive integer number

$$R = Q \cdot N_m + T, \tag{5}$$

where Q is an integer number in the exponent field, and T is an integer number in the mantissa field; thus, $Q = 2 * \text{bias} - 1$. Next, the integer difference d is sought (the change of sign of the approximate binary logarithm of x occurs here):

$$d = R - I_x. \tag{6}$$

Then, the integer d is converted to a real number in the floating-point format, which will be the initial approximation y_0. The sequence of operations follows:

- a biased exponent should be found

$$e_y = \text{floor}\left(\frac{d}{N_m}\right) \tag{7}$$

- a nonbiased exponent should be identified

$$E_y = e_y - \text{bias} \tag{8}$$

- a mantissa fractional part should be found

$$m_y = \frac{d - E_y \cdot N_m}{N_m} = \frac{d}{N_m} - E_y \quad (9)$$

- an initial approximation y_0 should be found in the form

$$y_0 = (1 + m_y) \cdot 2^{E_y}. \quad (10)$$

The relative error of the initial approximation could be estimated from y_0 as

$$\delta_0 = y_0 x - 1. \quad (11)$$

After finding y_0, iteration is performed with the classic Newton–Raphson formula for the reciprocal,

$$y_n = y_{n-1}(2 - x y_{n-1}). \quad (12)$$

The strict mathematical model for the initial approximation of y_0 can be formed. First, it is necessary to write an expression of y_0. If we sequentially describe the converting of x to I_x, and the inverse converting d to y_0 using (1)–(10), then we conclude that the initial approximation of y_0 in the general case is described by

$$y_0 = (1 + t - 2^{E_x} x - E_x - E_y) \cdot 2^{E_y}, \quad (13)$$

where

$$t = \frac{T}{N_m}. \quad (14)$$

The expression (13) is a mathematical model for the formation of an initial approximation y_0 for number x, which is performed in the IEEE-754 standard.

Analysis of (13) shows that the approximation could be presented in the form

$$y_0 = (\alpha x + \beta \cdot 2^{E_y}), \quad (15)$$

where

$$\alpha = -2^{E_x}, \quad (16)$$
$$\beta = 1 + t - E_x - E_y, \quad (17)$$

and y_0 is the piecewise linear approximation of reciprocal.

Now a detailed analysis of (7) and (8) can be performed. It is possible to write

$$E_y = -E_x - 1 + \text{floor}(t - m_x). \quad (18)$$

Therefore, taking into account $0 \le t < 1$ and $0 \le m_x < 1$, the order E_y could have only two values:

$$E_y = -E_x - 1, \text{ when } m_x \le t \quad (19)$$

and

$$E_y = -E_x - 2, \text{ when } m_x > t. \quad (20)$$

From (15), (19), and (20), it follows that the interval $x \in [1, 2)$ is divided into two parts: $x \in [1, x_t)$ and $x \in [x_t, 2)$, where $x_t = 1 + t$. In the first part of the interval $x \in [1, x_t)$, E_y is described by (19), so the linear initial approximation is

$$y_{01} = 1 - \frac{x}{2} + \frac{t}{2}. \quad (21)$$

In the second part, $x \in [x_t, 2)$, E_y is determined by (20), so that

$$y_{02} = \frac{x}{4} + \frac{3}{4} + \frac{t}{4}. \quad (22)$$

For other values of x that lie in ranges that are multiples of (1, 2), conditions (19) and (20) are saved, and the initial approximation will be

$$y_{01} = \frac{1}{2}(1 - m_x + t)2^{-E_x}, \quad (23)$$

and

$$y_{02} = \frac{1}{4}(2 - m_x + t)2^{-E_x}. \quad (24)$$

Relative errors in this case will be the same as well for $x \in [1,2)$, and all performed calculations will be valid for negative values if the sign is taken into account.

Proposed algorithm

To provide full accuracy in single-precision format, the first iteration should be executed using a modification of (12):

$$y_1 = k_1 y_0 (2 + k_2 - x y_0). \quad (25)$$

The second iteration uses the classic formula

$$y_2 = y_1 (2 - x y_1). \quad (26)$$

The classic Newton–Raphson formula gives an exact result if an exact enough initial approximation y_0 is used in it. (The formula has quadratic convergence—the number of correct bits is doubled.) If the initial approximation is inexact (i.e., as in our case), then it is possible to modify a classic formula similar to (25) (in our case, $k_1 = 2$) to get the same exact result, as well as in the case described previously.

It is necessary to find optimal values of the magic constant R; its unknown parameter T, which is determined from $t - (14)$; and the coefficient k_2. Although the coefficient k_1 could also be optimized to minimize the error of the approximation similarly as for variables t and k_2, we have chosen to set $k_1 = 2$ because the multiplication by two can be efficiently implemented by addition or by increasing the exponent, without the need for floating-point multiplication. The following method should be used. After the first Newton–Raphson iterations with linear initial approximation y_0, the variable y_1 is described by a polynomial of the third order. To provide the smallest error, y_1 should be the best uniform approximation polynomial (in the sense of relative error).

The best uniform approximation for the reciprocal is the polynomial of the third order, P_3, in the form

$$P_3 = \alpha x^3 + \beta x^2 + \gamma x + \Psi, \quad (27)$$

where $x \in [a, b]$

$$\alpha = -\frac{128}{q}, \quad \beta = \frac{256(a+b)}{q},$$

$$\gamma = -\frac{32(5a^2 + 14ab + 5b^2)}{q},$$

$$\Psi = \frac{32(7ab^2 + 7a^2 b + a^3 + b^3)}{q},$$

and

$$q = a^4 + b^4 + ab(28(a^2 + b^2) + 70ab).$$

We obtained this polynomial by performing a minimization of the error, which will be detailed in a separate article. Because y_0 has two constituents, y_{01} and y_{02}, it is necessary to select values R and k_2 so that the total maximum relative error has a minimum value throughout the interval (1, 2). It is necessary to write (25) and (27) using the conditions

$$y_{11} = 2y_{01}(2 + k_{21} - x y_{01}),$$

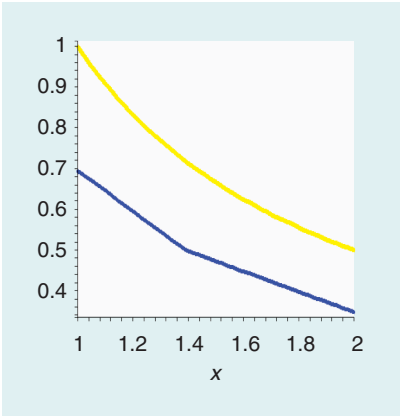

FIGURE 3. Initial approximations of rcp_3 algorithm (blue). The yellow line indicates 1/*x*.

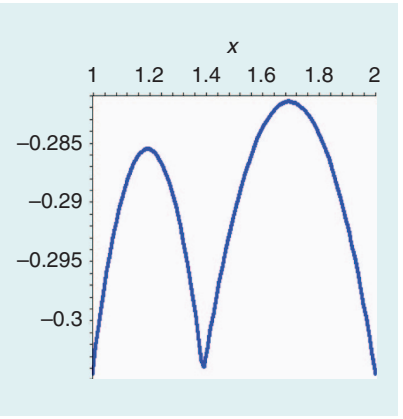

FIGURE 4. Relative errors of initial approximations for rcp_3 algorithm.

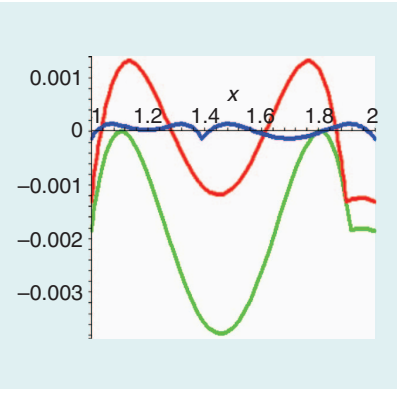

FIGURE 5. Relative errors of the first iteration for the rcp_1 (green), rcp_2 (red), and rcp_3 (blue) algorithms.

$$P_{31},\ x \in [a = 1, b = 1 + t],$$

and

$$y_{12} = 2y_{02}(2 + k_{22} - xy_{02}),$$
$$P_{32},\ x \in [a = 1 + t, b = 2].$$

Solving the two equations, $y_{11} = P_{31}$ and $y_{12} = P_{32}$, will yield the two pairs of meaning for t and k_2. The details are omitted here because of space limitations.

It can be shown that $t = 0.3911389\,72948632956$ ensures the lowest possible value of relative errors after the first iteration carried out by (25). With this the value t, the magic constant R for float numbers, will take the form 0x7eb210d8, and $k_2 = -0.585691542659989199$. Then, after rounding, (25) becomes

$$y_1 = 2y_0(1.41430846 - xy_0).$$

C code of the proposed algorithm

```
float rcp_3(float x)
{
  int i=*(int*)&x;
  i=0x7eb210d8-i;
  float y=*(float*)&i;
  y=y*(1.41430846f-x*y);
  y=y+y;
  float r=1.0f-x*y;
  y=y+r*y;
  return y;
}
```

The multiplication operation in line 2 is replaced by an addition operation. Figures 3 and 4 show graphics of the initial approximations y_{01}, y_{02}, and $1/x$ and the relative errors of the suggested algorithm accordingly. The large bias visible in Figure 3 is eliminated by the modified Newton–Raphson iteration (25) without additional mathematical operations. The maximum relative errors are $\delta^{+}_{\max} = 1.169 \cdot 10^{-7}$, $\delta^{-}_{\max} = -1.344 \cdot 10^{-7}$, or 22.89 correct bits. In this variant of the algorithm (rcp_3), the relative error for the first iteration has a minimum value ($\delta^{+}_{\max} = 1.341 \cdot 10^{-4}$ and $\delta^{-}_{\max} = -1.345 \cdot 10^{-4}$, or 12.86 correct bits). Relative errors of the first iteration for the three described algorithms are shown in Figure 5.

However, the most accurate results are given by the following implementation of the algorithm:

```
float rcp_4(float x)
{
  int i=*(int*)&x;
  i=0x7eb210da-i;
  float y=*(float)&i;
  y=y*(1.4143113f-x*y);
  y=y+y;
  float r=fmaf(y,-x,1.0f);
  y=fmaf(y,r,y);
  return y;
}
```

The second iteration is performed using the operation $fmaf(y, -x, 1.0f)$, that is equivalent to $1.0f - xy$. (The *fmaf* operation reduces errors due to use of the formula realization, e.g., in $z = y + r \cdot y$, the rounding is executed only once to receive the final result.) The *fmaf* use gives an opportunity to reduce the maximum relative errors of both signs to values $\delta^{+}_{\max} = 5.895 \cdot 10^{-8}$ and $\delta^{-}_{\max} = -7.608 \cdot 10^{-8}$, or 23.64 correct bits. In hardware implementation, multiplying by two makes it possible to use the second integer constant 0x7f3210da, which differs from the magic constant on 0x00800000 (i.e., the exponent increased by one).

The C code for hardware implementation of this algorithm in FPGA is

```
float rcp_5(float x)
{
  int i=*(int*)&x;
  int ii=0x7f3210da-i;
  i=0x7eb210da-i;
  float y=*(float*)&i;
  float yy=*(float*)&ii;
  y=yy*(1.4143113f-x*y);
  float r=fmaf(y,-x,1.0f);
  y=fmaf(y,r,y);
  return y;
}
```

The calculation of integer variables *i* and *ii* and their conversion in float can be conducted simultaneously in this case, and that algorithm includes only four multiplication operations without lookup table for initial approximation. The C code for double precision is

```
double rcp_6(double x)
{
  uint64_t i=*(uint64_
  t*)&x;
  uint64_t ii=0x7fe6421af
  0901626-i;
```

```
    i=0x7fd6421af0901626-i;
    double y=*(double*)&i;
    double yy=*(double*)&ii;
    y=yy*(1.4143084573400108
    -x*y);
    y=y*(2.0000000090062634
    -x*y);
    y=y+y*fma(y,-x,1.0);
    return y;
}
```

The algorithm has maximal relative errors in the whole range of normalized double-type numbers, $\delta^{+}_{\max} = 0$: $\delta^{-}_{\max} = -2.22 \cdot 10^{-16}$, or 52 correct bits.

Technical specifications of the computer running the experiments are as follows: Windows 7 64-bit operating system, Intel Pentium G850 with 2.90-GHz CPU, 4-GB memory using MS Visual C++ Compiler 12.0.

Experimental results for the microcontroller

Our reciprocal algorithm was tested in C++ on the Espressif (ESP) WROOM-32 microcontroller [10], which supports floating-point computing (addition, multiplication, and fma single-precision instructions) and does not have hardware reciprocal instructions. Relative errors and latency are shown in Table 1. The reciprocal algorithm for the single precision is 21% faster than usual float division 1.0f/x. If we conduct the first iteration as

$$y_1 = 2y_0(1.4143113 - xy_0) \qquad (28)$$

(see rcp_4_28 algorithm), the advantage will grow to 32%.

For the ESP microcontroller WROOM-32, multiplying by two (or number aliquot of two) occurs faster than any other real number that is not aliquot to two.

```
float rcp_4_28(float x)
{
  int i=*(int*)&x;
  i=0x7eb210da-i;
  float y=*(float)&i;
  y=2*y*(1.4143113f-x*y);
  float r=fmaf(y,-x,1.0f);
  y=fmaf(y,r,y);
  return y;
}
```

Table 1. The performance of reciprocal algorithms on ESP WROOM-32 microcontroller.

Single Precision	1.0f/x	Our rcp_4	Our rcp_4, (28)
$\delta^{-}_{\max}$	$-5.9558602 \times 10^{-8}$	$-7.6075395 \times 10^{-8}$	$-7.6075395 \times 10^{-8}$
$\delta^{+}_{\max}$	$+5.9604638 \times 10^{-8}$	$+5.8947094 \times 10^{-8}$	$+5.8947094 \times 10^{-8}$
Latency (ns)	327.31399	268.48363	255.90167

Summary

We have described simple floating-point division, which is based on the reciprocal of the divisor, and presented reciprocal algorithms with four multiplications for single precision and six multiplications for double precision. The proposed algorithms belong to the iterative algorithms group. They initially receive an approximation using the so-called magic constant and are then used to implement an iterative process with the Newton–Raphson formula. Improved accuracy of the proposed algorithms compared with known algorithms was achieved by reducing the magnitude scope of the relative errors of the initial approximations (the difference between the smallest and largest values) and, as a consequence, the modification of the first iterations. For all normalized floating-point numbers, both algorithms have maximal relative errors of $\delta_{\max} = 7.61 \cdot 10^{-8}$ and $\delta_{\max} = -2.22 \cdot 10^{-16}$ accordingly.

Authors

Leonid Moroz (moroz_lv@lp.edu.ua) received his Sc.D. degree in technical sciences from Lviv Polytechnic Institute, Ukraine, in 1978. He is a professor in the Department of Information Security, Institute of Computer Technologies, Automation and Metrology, Lviv Polytechnic National University, Ukraine. His research interests include computer arithmetic, numerical methods, and digital signal processing.

Volodymyr Samotyy (vsamotyy@pk.edu.pl) received his Sc.D. degree in technical sciences from Lviv Polytechnic Institute, Ukraine, in 1984. He is a professor in the Department of Automatic Control and Information Technology, Cracow University of Technology, Poland, and the Department of Information Security Management, Lviv State University of Life Safety, Ukraine. His research interests include evolutionary models, parametric optimization, information security, computer arithmetic, numerical methods, and digital signal processing.

References

[1] M. Allie and R. Lyons, "A root of less evil," *IEEE Signal Process. Mag.*, vol. 22, no. 2, pp. 93–96, 2005. doi: 10.1109/MSP.2005.1406500.

[2] F. Auger, Z. Lou, B. Feuvrie, and F. Li, "Multiplier-free divide, square root, and log algorithms," *IEEE Signal Process. Mag.*, vol. 28, no. 4, pp. 122–126, 2011.

[3] STMicroelectronics. (2016, May). Floating point unit demonstration on STM32 microcontrollers. Application note AN4044. [Online]. Available: https://www.st.com/content/ccc/resource/technical/document/application_note/10/6b/dc/ea/5b/6e/47/46/DM00047230.pdf/files/DM00047230.pdf/jcr:content/translations/en.DM00047230.pdf

[4] Texas Instruments. (2015, Oct.) TMS320F2807x Piccolo™ microcontrollers. SPRS902B. [Online]. Available: http://www.ti.com/lit/ds/symlink/tms320f28075.pdf

[5] Intel. (2017, 8 May). Intel Cyclone 10 GX device Overview. C10GX51001. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/documentation/grc1488182989852.html

[6] J. Blinn, "Floating-point tricks," *IEEE Comput. Graph. Appl. Mag.*, vol. 17, no. 4, pp. 80–84, 1997. doi: 10.1109/38.595279.

[7] K. Huang and Y. Chen, "Improving performance of floating point division on GPU and MIC," in *Proc. 15th Int. Conf. Algorithms and Architectures for Parallel Processing Part II*, 2015, vol. 9529, pp. 691–703.

[8] P. Khuong. (2011, Mar. 16). A magic constant for double float reciprocal. [Online]. Available: https://www.pvk.ca/Blog/LowLevel/software-reciprocal.html

[9] L. Moroz and A. Hrynchyshyn, "A fast calculation of function y=1/x with the use of magic constant," *Bull. Lviv Polytech. Nat. Univ.* (Automation, Measurement and Control Series 821, in Ukrainian), pp. 23–29, 2015.

[10] Espressif Systems. (2018). ESP32-WROOM-32 (ESP-WROOM-32) datasheet. Version 2.4. [Online]. Available: https://www.mouser.com/ds/2/891/esp-wroom-32_datasheet_en-1223836.pdf

[11] IEEE Computer Society. (1985). IEEE Standard for Binary Floating-Point Arithmetic. [Online]. Available: https://standards.ieee.org/standard/754-2008

SP