The 4th IEEE International Symposium on Wireless Systems within the International Conferences on Intelligent Data Acquisition and Advanced Computing Systems
20-21 September, 2018, Lviv, Ukraine

An Effective Floating-Point Reciprocal

Leonid Moroz [1], Volodymyr Samotyy [2], Oleh Horyachyy [3]
[1,3] Institute of Computer Technologies, Automation, and Metrology, Lviv Polytechnic National University, 79013
[2] Department of Automatic Control and Information Technology, Cracow University of Technology, 31155
[2] Department of Information Security Management, Lviv State University of Life Safety, 79000
moroz_lv@lp.edu.ua, vsamotyy@pk.edu.pl, oleh.y.horiachyi@lpnu.ua

***Abstract*—This paper describes a simple and accurate floating-point reciprocal algorithm based on two modified Newton-Raphson iterations with a magic constant as the initial approximation. It can be effectively implemented on platforms with no FPU support since it uses just addition, multiplication and fused multiply-add operations.**

***Keywords*—reciprocal algorithm; division; floating point arithmetic; Newton-Raphson method; magic constant**

I. Introduction

Such operations as division and square root which are frequently used in multimedia, communication and digital signal processing (DSP) applications [1] are very expensive in terms of computations. In particular, division and square root are important in matrix decomposition algorithms such as QR, LU, and Cholesky for MIMO (Multiple-Input, Multiple-Output) wireless communication systems [2]-[3].

Usually, for fast execution of such operations (both for fixed-point numbers [4] and floating-point numbers [5]-[8]) in DSP processors and microcontrollers mathematical floating-point coprocessors (FPUs) are used. Floating-point unit, which includes special hardware instructions for floating-point operations (addition, subtraction, multiplication, multiply-accumulate, fused multiply-add, division, square root, etc.), can be separated from the CPU [5] or integrated with it [7]. But, unfortunately, not all devices have full hardware floating-point support. "On an FPU-less processor, all these operations are done by software through the C compiler library" [5] using available hardware instructions. Among the most common and fast floating-point operations implemented in hardware are single precision floating-point addition, multiplication and fused multiply-add (FMA). Nowadays, modern Field Programmable Gate Arrays (FPGAs) also have single precision floating-point DSP blocks for these three basic operations [9].

In this paper, we propose a simple and accurate reciprocal algorithm which can be used to perform division and is computed in large quantities in matrix operations. The main advantage of this algorithm is that it can be used to improve the performance of such FPU-less processors or other hardware platforms which support only three floating-point operations – addition, multiplication, and FMA.

II. Theoretical Basis of the Algorithm

The basic idea behind the implementation of a floating-point reciprocal algorithm which uses a magic constant is that if we take binary logarithm of a real number x with the opposite sign and then by making the inverse operation translate it into a number y using exponential function then we get the inverse of the value x. However, since the real number given in IEEE 754 standard format is a binary quasi-logarithm of x, then the inverse value y obtained in this way can only be considered as a certain initial approximation of the reciprocal function $1/x$. In order to reduce the error of this approximation, the formula contains some corrective value – a magic constant. The resulting initial approximation is improved by using iterative Newton-Raphson equations to obtain higher accuracy results. A detailed mathematical description of the algorithm is given in [10], here we present only the main theoretical results for a single-precision format (type *float*).

A. Initial Approximation

Suppose we have a floating-point number x in the form of a normalized number

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

where S_x denotes the sign, $S_x = \{0,1\}$, 0 is defined for positive numbers and 1 – for negative numbers; E_x is the order of magnitude (exponent), which is generally defined by the formula

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

M_x – the mantissa, which is located within the range $M_x = [1,2)$, is calculated according to the formula

$$M_x = \frac{|x|}{2^{E_x}}. \quad (3)$$

In IEEE 754 standard mantissa is represented in form $M_x = 1 + m_x$, where m_x is the fractional part of the mantissa while integer part is omitted. Besides that, the exponent E_x is stored with some offset $e_x = E_x + bias$. For single-precision format $bias = 127$. Thus, the numbers of type *float* in IEEE 754 standard (32 bits) are represented as follows

31	30… …23	22… …0
S_x	e_x	m_x
1 bit	8 bits	23 bits

Henceforth, for simplicity, we will assume that $x > 0$, that is $S_x = 0$.

Integer decimal number I_x, which corresponds to the binary representation of x in IEEE 754 standard, can be expressed by the formula (4).

$$\begin{aligned} I_x &= e_x \cdot N_m + m_x \cdot N_m = (e_x + m_x)N_m = \\ &= (bias + E_x + x \cdot 2^{-E_x} - 1) \cdot N_m, \end{aligned} \quad (4)$$

where $N_m = 2^{23}$ for single precision.

The magic constant R is given as a positive integer number

$$R = Q \cdot N_m + T \quad (5)$$

representing a floating point value in the 32-bit register. Here $Q = e_R = 2 \cdot bias - 1$ – exponent part and T – mantissa part of integer value R. Let's also denote by t the value of $t = m_R = \frac{T}{N_m}$.

The initial approximation y_0 of the inverse to x value is obtained from the expression $I_{y_0} = R - I_x$ and is represented as a floating point number

$$y_0 = (1 + m_y) \cdot 2^{E_y}. \quad (6)$$

As it was shown in [10], in a general case, y_0 is described by the equation

$$y_0 = (-x \cdot 2^{-E_x} + 1 + t - E_x - E_y) \cdot 2^{E_y}, \quad (7)$$

where m_y and E_y are the fractional part of the mantissa and the unbiased exponent of the initial approximation respectively.

The analysis of (7) shows that y_0 can be regarded as a piecewise linear approximation of the function $1/x$

$$y_0 = (\alpha x + \beta) \cdot 2^{E_y}, \quad (8)$$

where

$$\alpha = -2^{-E_x}, \quad (9)$$
$$\beta = 1 + t - E_x - E_y. \quad (10)$$

Let's describe the behavior of the function y_0 on the interval $x \in [1,2)$. It can be broken down into two parts: $x \in [1, x_t)$ and $x \in [x_t, 2)$, where $x_t = 1 + t$. As a result, for the interval $x \in [1, x_t)$ the linear initial approximation y_0 will be the following [10]

$$y_{01} = -\frac{x}{2} + 1 + \frac{t}{2}, \quad (11)$$

and for interval $x \in [x_t, 2)$

$$y_{02} = -\frac{x}{4} + \frac{3}{4} + \frac{t}{4}. \quad (12)$$

It should be mentioned that for any other interval $[2^k, 2^{k+1}), k \in \mathrm{Z}$, formulas (11) and (12) for the initial approximation y_0, and also the value of the absolute and the relative error, will have a similar shape as for $x \in [1,2)$. In addition, all the above results are also valid for $S_x = 1$, when $x < 0$.

B. Modified Newton-Raphson Iterations

In order to ensure the increased accuracy of the algorithm, we propose to perform the first and the second iterations of Newton-Raphson according to formulas

$$y_1 = k_1 y_0 (2 + k_2 - x y_0) \quad (13)$$
$$y_2 = y_1 (2 - x y_1). \quad (14)$$

Formula (13) represents the modified Newton-Raphson iteration for $y = 1/x$, and (14) – iteration in classical form. Taking into account that when using type *float* rounding errors become significant, especially in the second iteration, so we can recommend:

- to rewrite (14) as $r = 1 - xy_1; y_2 = y_1 + y_1 r$ how it was suggested by Schulte et al. in [1];
- to use fused multiply-add function $FMA(a,b,c) = ab + c$. This will increase the accuracy of the fused operation by performing just one rounding and also can accelerate it when the appropriate hardware instruction is used.

The next step will be to find optimal values of the magic constant R and the coefficients k_1 and k_2. To accomplish that, the following analytical approach was used in this study. After conducting the first iteration with a linear initial approximation y_0 the y_1 variable takes the form of third-order polynomials. These polynomials should have the best uniform approximation to provide the lowest relative error. The following value was chosen $t = \sqrt{2} - 1$, and respectively $x_t = 1 + t = \sqrt{2}$, since such t ensures the lowest possible value of relative errors after the first iteration in form (13). For this t the magic constant takes the value $R = 0\text{x7eb504f3}$.

Now, having t, we can find the values of coefficients k_1 and k_2 using the aforementioned polynomials

$$k_1 = 1.940908883185;$$
$$k_2 = -0.56433982822018.$$

Thus, after rounding (13) looks like

$$y_1 = 1.94091 \cdot y_0 (1.43566 - xy_0). \quad (15)$$

III. Proposed Reciprocal Algorithm

A. Basic Structure

After taking into account all the recommendations, the final C/C++ algorithm for type *float* is

```
float reciprocal_1_f (float x){
   int i = *(int*)&x;
   i = 0x7eb504f3 - i;
   float y = *(float*)&i;
   y = 1.94091f*y*fmaf(-x, y, 1.43566f);
   float r = fmaf(y, -x, 1.0f);
   y = fmaf(y, r, y);
   return y;
}
```

As you can see, the algorithm *reciprocal_1_f* requires only 2 multiplications and 3 FMA operations. Note that in order to get the high performance it is necessary to use hardware-implemented FMA instructions that are available in many CPUs [6], [11], FPUs [5], [12], microcontrollers [13] and FPGAs [9]. To give an example, for the Intel platform the C++ compiler can automatically translate functions *fmaf* from *cmath* library into hardware instructions such as *vfmadd132ss* or *vfnmadd132ss* [14]. The Xtensa architecture has the hardware instructions *madd.s* and *msub.s* respectively for fused addition and subtraction [13].

Our numerical experiments on these two platforms show that using such instructions the maximum relative errors of the algorithm *reciprocal_1_f* will be $\delta^+_{\max} = 5.895382 \times 10^{-8}$; $\delta^-_{\max} = -7.166542 \times 10^{-8}$, which corresponds to 23.73 correct bits of the result.

Here is another version of the first iteration which gives similar results after two iterations ($\delta_{\max} = 7.1744 \times 10^{-8}$).

```
y = 1.940909f*y*fmaf(-x,y,1.4356601f); (16)
```

After the first iteration using (16), this algorithm will have almost aligned results $\delta^+_{\max} = 1.1173 \times 10^{-4}$; $\delta^-_{\max} = -1.1170 \times 10^{-4}$ (13.13 correct bits). It's more accurate than known instruction *rcpss* in Intel [11], [14], which has errors $\delta^+_{\max} = 3.0023 \times 10^{-4}$; $\delta^-_{\max} = -2.9993 \times 10^{-4}$ (11.7 correct bits), though, it is obvious that it concedes in the speed of execution.

Note that we use the following formulas to calculate the maximum relative errors

$$\delta^+_{\max} = \max_{x \in [1,2)} (x \cdot y - 1); \; \delta^-_{\max} = \min_{x \in [1,2)} (x \cdot y - 1); \quad (17)$$

and the overall error

$$\delta_{\max} = \max_{x \in [1,2)} | x \cdot y - 1 | \quad (18)$$

can be used to evaluate the accuracy of the algorithms

$$p = -\log_2(\delta_{\max}). \quad (19)$$

B. Improved Algorithm

Obviously, due to rounding errors in type *float,* especially for two iterations, the given above theoretical values of the magic constant R, coefficients k_1 and k_2 in practice will not be optimal. It turns out that they can be refined using computational methods and, accordingly, the values of the maximum relative errors for the algorithm *reciprocal_1_f* can be reduced. To do that, we have used the following method:

1) as the initial values of the parameters R^0, k_1^0 and k_2^0 we take corresponding theoretical coefficients; for each parameter, we select intervals of their change such that $R^0 \in [R_{\min}, R_{\max})$, $k_i^0 \in [k^i_{\min}, k^i_{\max})$, $i = \overline{1,2}$;

2) randomly select triples $\left\langle R^0, k_1^0, k_2^0 \right\rangle$, where R^0 has type *int* and k_i^0 – type *float*, looking for a triple whose value of maximum error $\delta_{\max}$ on the whole interval $x \in [1,2)$ is minimal;
3) apply the method of 3D optimization based on the algorithm [15] to improve the parameters obtained in step 2.

The calculation was performed on a computer with processor Intel Core i7-7700HQ (2.8 GHz, 16 Gb RAM) using GNU compiler (GCC 4.9.2) for C++ with options: *–std=c++11 –mfma –Os –ffp=contract=on.*

As a result we have following algorithm

```
float reciprocal_2_f (float x){
  int i = *(int*)&x;
  i = 0x7eb53567 - i;
  float y = *(float*)&i;
  y = 1.9395974f*y*fmaf(-x, y, 1.436142f);
  float r = fmaf(y, -x, 1.0f);
  y = fmaf(y, r, y);
  return y;
}
```

The algorithm *reciprocal_2_f* has relative errors $\delta_{\max}^{+} = 5.901984\times10^{-8}$; $\delta_{\max}^{-} = -6.861453\times10^{-8}$ or 23.8 correct bits. It is just 4.26 percent decrease in maximum relative error $\delta_{\max}$ compared to *reciprocal_1_f.*

IV. Main Results

We have tested these reciprocal algorithms on a microcontroller ESP-WROOM-32 created by Espressif Systems. ESP-WROOM-32 is a powerful combined Wi-Fi/Bluetooth/BLE module with two low-power Xtensa 32-bit microprocessors [16]. It can be effectively used as a node in WSAN (Wireless Sensor and Actuator Network) and for data processing. The algorithms were compiled and uploaded to the microcontroller using Arduino IDE with default parameters: *–std=gnu++11 –Os –ffp-contract=fast.*

In Table I you can see the results of measuring the execution time (latency) and maximum relative error of algorithms *reciprocal_1_f* and *reciprocal_2_f* comparing with the regular *float* division (1.0f/x) and other known reciprocal algorithms from [17] and [18]. Time execution was evaluated using *chrono* C++ library. It should be noted that this architecture has no separate hardware instructions for reciprocal or division operations (like *rcpss* or *divss* in Intel) [13]. That's why all listed approximate algorithms with magic constant are much faster than division. But it provides the best accuracy (all 24 bits are correct). Our algorithms have slightly worse precision, but they are much faster (just 256 ns). On the other hand, other reciprocal algorithms from the table when compared with the proposed ones have similar latency but they give a smaller number of correct bits.

TABLE I. Comparison of Reciprocal Algorithms With Magic Constant on ESP-WROOM-32 Microcontroller

	1.0f/x	***reciprocal_1_f***	***reciprocal_2_f***	***alg. 1*** [17]	***alg. 2*** [18]	***alg. 3*** [18]
$\boldsymbol{\delta}_{\max}^{-}$	-5.9558602e-08	-7.1665418e-08	-6.8614526e-08	-1.7593574e-07	-1.4569250e-07	-1.1882202e-07
$\boldsymbol{\delta}_{\max}^{+}$	5.9604638e-08	5.8953816e-08	5.9019840e-08	1.7441791e-07	1.7478635e-07	5.4937601e-08
Precision, bits	24	23.73	23.8	22.44	22.45	23
Latency, ns	327.314	255.902	255.902	243.325	260.091	255.902

V. Conclusions

In this paper, a very effective and simple reciprocal algorithm for single precision floating-point numbers in IEEE 754 format is presented. Instead of look-up table (LUT) and operations with different parts of the numbers, this approximation algorithm uses a clever magic constant trick for initial value followed by two modified Newton-Raphson iterations. First, we determined such theoretical parameters of the algorithm which provide the best accuracy and then tried to improve them using numerical 3D optimization approach. The proposed reciprocal algorithm was designed for hardware platforms (CPU and microcontrollers) which can take advantage of using fast FMA operation and doesn't depend on the existence of hardware instructions such as *rcpss*. Also, we believe that the current algorithm can be effectively implemented in FPGA both using fixed and floating-point arithmetic. Our future work will include using the same approach to construct reciprocal and inverse square root algorithms also for higher precision data types (such as type *double*) so that they can be used to design an effective implementation of decomposition algorithms for microcontrollers and FPGAs.

References

[1] M. Schulte, J. Stine and K. Wires, "High-speed reciprocal approximations," *Conference Record of the Thirty-First Asilomar Conference on Vol. 2. IEEE*, 1997, pp. 1–5.

[2] A. Irturk, B. Benson, N. Laptev and R. Kastner, "Architectural optimization of decomposition algorithms for wireless communication systems," *WCNC 2009. IEEE*, 2009, pp. 1–6.

[3] F. Lemaitre, B. Couturier and L. Lacassagne, “Cholesky factorization on SIMD multi-core architectures,” *Journal of Systems Architecture*, vol. 79, pp. 1–15, 2017.

[4] M. Allie and R. Lyons, “A root of less evil,” *IEEE Signal Processing Magazine*, vol. 22, no. 2, pp. 93–96, 2005.

[5] STMicroelectronics N.V., “Floating point unit demonstration on STM32 microcontrollers. Application note AN4044,” 2016.

[6] M. Cornea, “Intel® AVX-512 instructions and their use in the implementation of math functions,” 2015.

[7] Texas Instruments Incorporated, “TMS320F2807x Piccolo™ microcontrollers. SPRS902B,” 2015.

[8] B. Parhami, *Computer arithmetic algorithms and hardware architectures*, 2nd ed, Oxford University Press, New York, 2010.

[9] Intel Corporation, “Intel® Cyclone® 10 GX device overview. C10GX51001,” 2018.

[10] L. Moroz and A. Hrynchyshyn. “A fast calculation of function y=1/x with the use of magic constant,” *Automation, Measurement and Control*, LPNU, N 821, pp. 23–29, 2015. (In Ukrainian).

[11] A. Fog, “Instruction tables: lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs,” 2018.

[12] ARM, “Cortex-A5™ floating-point unit. Technical reference manual. Revision: r0p1,” 2010.

[13] Tensilica, Inc., “Xtensa instruction set architecture (ISA). Reference manual,” 2018.

[14] “x86 and amd64 instruction reference,” *Felixcloutier.com*, 2018. [Online]. Available: http://www.felixcloutier.com/x86/index.html. [Accessed: 10- May- 2018].

[15] A. Yalcin, *Multidimensional direct search method*. Chapter 09.03, pp. 1–5.

[16] Espressif Systems, “ESP32-WROOM-32 (ESP-WROOM-32) datasheet. Version 2.4,” 2018.

[17] K. Huang and Y. Chen, “Improving performance of floating point division on GPU and MIC,” in *15th International Conference on Algorithms and Architectures for Parallel Processing*, part II, vol. 9529, 2015, pp. 691–703.

[18] L. Moroz, A. Hrynchyshyn and Y. Miretska. “Simple floating point division algorithms,” *Automation, Measurement and Control*, LPNU, N 852, pp. 35–38, 2016. (In Ukrainian).