Non-sequential Division
Michael Lunglmayr, Member, IEEE,

Abstract
The division operation is important for many areas of data processing. Especially considering today’s demand for hardware accelerators for machine learning algorithms, there is a high demand for an efficient calculation of the division function, e.g. for averaging operations or the online calculation of activation functions. For such algorithms, which are often iterative in nature, one would like to have a non-sequential way of calculating the division operation. The work presents such an approach. It is based on an efficient way of calculating the reciprocal operation, based on a low complexity approximation combined with a correction function. The described approach allows approximating the division operation (with errors that can be made arbitrarily low), within one clock cycle using only low hardware requirements (although one might reasonably use more clock cycles to increase the clock frequency). These hardware requirements are scale-able depending on the desired precision. We show results obtained by synthesis and hardware simulations demonstrating the low complexity and high clock speed achievable with the described method.

Index Terms
Division, Computation architecture, average calculation, digital hardware

I. INTRODUCTION
Division is considered to be the most complex of all four basic arithmetic operations [1]. This complexity is mainly reflected in the effort required for calculating the reciprocal operation. If one has the reciprocal $1/x$ of a number $x$, the result of a division can be obtained by multiplication with such a reciprocal. The use-cases of efficient reciprocal operations are manifold. Especially for iterative algorithms, where the division or reciprocal operation often has to be performed within an iteration, one would like to have a fast reciprocal operation. This is crucial for algorithms where the operand of the reciprocal operation depends on results calculated in previous iterations. Such dependencies often hinder an efficient pipelining of calculation units. Examples are algorithms where divisions are used to calculate stopping criteria [2] or architectures using non-linear functions [3] based on divisions. Often hardware dividers are avoided or approximated in such architectures [2]–[4] as ”division consumes too much cycles and area” [4] or are delegated to a connected CPU [5].

One of the reasons is that, typically, algorithms for division/reciprocal are sequential in nature. Classical algorithms for implementing a division are often based on digit recurrence algorithms (e.g. see [6] and the references therein), while algorithms for applications where speed is more important than precision often rely on iterative algorithms. Such algorithms are typically either based on Newton’s method [7]–[9] or on Goldschmidt’s algorithm [10]–[12]. Other approaches use lookup tables [13] or combine lookup tables with iterative methods [14,15]. In this work, we present an approach for a non-sequential reciprocal calculation, that can be calculated within one clock cycle (as we demonstrate in Sect. V; however, one might want to spend more clock cycles to increase the maximum clock frequency). The proposed method does not require Lookup tables and does not require any scaling of the input (as e.g. some implementations do that rely on Newton’s method [7]). We will describe the approach that uses a simple approximation of the reciprocal function in combination with a correction function that is represented by a polynomial (where the degree of the polynomial defines the hardware requirements as well as the precision of the result). This correction function will, depending on the input value $x$ of the
reciprocal, give a correction factor that is applied to the output of the (more crude) approximation of the reciprocal. We will show that, with one correction function, the whole number range can be covered. In its simplest form, this will allow calculating the reciprocal operation using only three multiplications (in addition to other, less complex operations) with absolute errors below $10^{-3}$ for small input values $x$ and with errors smaller than $10^{-7}$ for high input values $x$. By increasing the number of multiplications, i.e. the degree of the correction polynomial, the precision can be further increased.

In this paper, first, the approximate reciprocal calculation is described. Then we derive the correction function and show how this correction function can be approximated by polynomials. We analyze the precision that can be achieved with this concept. We then present architectures that have been designed and optimized for correction polynomials of degrees 2 and 4 and finally show synthesis results demonstrating the low complexity and high clock speeds achievable with this method.

II. APPROXIMATE RECIPROCAL CALCULATION

In [16], we proposed a low complexity approximation of the reciprocal function. We will repeat the main principle, as it is utilized for this approach, for the reader’s convenience. The idea of the approximation is the following. The function $1/x$ is linearized over the intervals $[2^z, 2^{z+1}]$ using $z = \lfloor \log_2(x) \rfloor$ for each positive value $x$. For simplicity, we assume in this work that $x > 1$. Although the presented approach also works for positive values smaller than one (for negative values, one can convert to positive values first and then correct the sign of the output accordingly), the number ranges change depending on whether $x$ is smaller or larger than 1. Assuming $x > 1$, the result conveniently fits into a fixed point format with only one bit in front of the comma, (the so-called fractional $1/(B-1)$ format of $B$-bits) that is often used in signal processing. For easier readability, we will further assume that in each of the following equations $z$ is always $\lfloor \log_2(x) \rfloor$ of the $x$ appearing in same the equation.

Fig. 1 graphically shows the mentioned principle of the approximation $y_l$. The approximation of Fig. 1 can be calculated as

$$y_l(x, C) = (C - x2^{-z})2^{-(z+1)},$$

using the value $C = 3$ [16]. Investigating the relative error between the exact calculation and this approximation showed that this error followed a curve of similar shape in each interval $[2^z, 2^{z+1}]$.  

![Fig. 1. Piecewise approximation of 1/x. Image from [16].](image)
This lead to the idea of correcting the approximation by multiplication with correction factors \( \gamma_x(x) \) such that

\[
\gamma_x(x)(3 - x2^{-z})2^{-(z+1)} = 1/x.
\]  

(2)

III. CORRECTION FUNCTION

Fig. 2 graphically shows the correction factors over \( x \). For an interval \([2^z, 2^{z+1}]\), one can calculate such factors as giving the correction function

\[
\gamma_x(x) = \frac{1}{x(3 - x2^{-z})2^{-(z+1)}}
\]  

(3)

Using the fact, that each number \( x \) of an interval \([2^z, 2^{z+1}]\), \( z \in \mathbb{Z} \) can be represented by a factor \( a \in [0, 1] \) times the lower interval border (plus offset): \( x = a2^z + 2^z \), with

\[
a = (x - 2^z)/2^z = x2^{-z} - 1,
\]  

(4)

one can formulate the correction function for \( a \in [0, 1] \) as

\[
\gamma(a) = \frac{2}{3(1 + a) - (1 + a)^2}.
\]  

(5)

To obtain the values of this function, one would require to calculate a reciprocal function, which would of course not be very helpful if one’s aim is to use it in an approximation of the reciprocal. One can, however, approximate the correction function (5) with polynomials. Fig. 3 shows a
plot of (5) as well as a least squares (LS) fitted polynomial of degree 2, $p_2(a)$. For polynomials $p_d(a)$ of higher degrees $d$, one would not see a difference to $\gamma(a)$ in this figure. For this reason, we plotted the errors for polynomials of higher degrees $|\gamma(a) - p_d(a)|$ in Fig. 4. Higher degrees than plotted in Fig. 4 are limited by the precision of the double floating-point format.

![Graph showing the correction function and polynomial approximation of degree 2.](image)

Fig. 3. Correction function and polynomial approximation of degree 2.

The coefficients for all polynomials in this work have been obtained by sampling $a$ at Chebyshev nodes stretched to fit the interval $[0, 1]$: 

$$a = (\text{Re}(e^{i\theta}) + 1)/2. \quad (6)$$

We used $\theta \in [-\pi, -\pi + 1 \cdot 10^{-5}, -\pi + 2 \cdot 10^{-5}, \ldots, -1 \cdot 10^{-5}, 0]$ in this work. The reason for using Chebyshev nodes for sampling the correction function is to reduce edge effects at the interval border [17]. By weighting the errors at the interval borders stronger, the approximation results in a relatively constant error over the whole interval $[0, 1]$ (as can also be seen in Fig. 4). If one uniformly samples $a$ for the LS fit, the errors would increase at the interval borders 0 and 1, respectively. Tab. I shows the coefficient vectors of the polynomials in the format $[c_d, c_{d-1}, c_0]$ describing a polynomial of the form 

$$p_d(a) = \sum_{j=0}^{d} c_j a^j. \quad (7)$$

Using these correction polynomials, one can evaluate the errors (compared to $1/x$) when being applied to the approximation $y_l(x, 3)$. Such errors are plotted in Fig. 5. As one can see from this figure, even with a correction polynomial of degree 2 for $x$ values larger than 1.6 one is always below an absolute error $10^{-3}$ (the maximum absolute error at $x > 1$ is $1.684e - 3$). Each
Fig. 4. Absolute approximation errors for polynomials of different degrees $d$

degree increase by 2, reduces the error by approximately a factor of 35. The relative error can be obtained by multiplying the error of Fig. 5 with $x$ (it then gives the error related to $1/x$). It is shown in Fig. 6 for different degrees of the polynomials. Here, one can see that, for a polynomial of a given degree, the relative error has about the same magnitude for all $x$ values (although for better visibility, we only plotted $x$ values up to 256, the same levels can be observed for larger values as well).

IV. HARDWARE ARCHITECTURES

For the author’s applications, the correction polynomials of degrees 2 and 4 are of most interest. Considering the error described in the sections before, these correction polynomials allow calculating the reciprocal with approximately 10 bits precision and better when using the degree 2 polynomial and approximately 16 bits precision when using the degree 4 polynomial (as described above, the absolute precision improves for large values of $x$).

For these two polynomials the architectures described in this section have been developed. If one has a look at the coefficients of the polynomial of degree 2, one can see that the coefficients $c_2$ and $c_1$ are nearly equal in absolute value. Assuming the values to be equal (the difference in magnitude is of order $10^{-9}$) one can reformulate $p_2(a)$ as

$$p_2(a) = c_2(a - 0.5)^2 + c_0 - 0.25c_2,$$

where the constant can be combined into $C' = c_0 - 0.25c_2$. This allows incorporating this correction polynomial efficiently in the approximate $1/x$ architecture of [16], as it is schematically shown in Fig. 7. Here the input $a = x2^{-z} - 1$ of the polynomial is already combined with the subtraction of 0.5 of (8), resulting in the subtraction with 1.5 before the squaring operation.
In contrast to [16], we incorporated a multiplication of a dividend input \( w \) already in the shown architecture. Instead of calculating
\[
w \cdot \left( p_d(x^{2^{-z}} - 1) (C - x^{2^{-z}}) 2^{-(z+1)} \right),
\]
(9)
one can do the multiplication \( w 2^{-(z+1)} \) within the architecture. This can reduce the bit length for the multiplier combining the now shifted \( w \) value with the rest of the division operation, especially for applications where \( w \) has a large number of bits in front of the comma (that can be reduced after the right shift by \( z + 1 \)). An example application where this is convenient is when using the architecture for calculating averages over \( N \) fractional numbers. In this scenario, \( x = N \) and, because \( w \) is a sum of fractional numbers, \( w 2^{-(z+1)} \) is then also a fractional number.

The polynomial of degree 4 of Tab. 1 can be factored into the following form:
\[
0.209150199411479 \cdot \\
(3.0616168632399 - 2.500018461800448a + a^2) \cdot \\
(1.561598389171924 + 0.5000184489913662a + a^2)
\]
(10)
In this form, the coefficients in front of \( x \) can be rounded to 0.5 and \(-2.5 \) without a noticeable effect on the approximation error (at least not for the synthesized bit lengths of this work). This allows performing the multiplications with these values using a shift operation and two shift and one add operation, respectively. The developed architecture for the polynomial of degree 4 is shown in Fig. 8.

In Fig. 7 and Fig. 8 the gray rectangles specify register positions for the implementation using 4 – 5 clock cycles for calculating the result, as is described in the next section. For the one clock cycle variants, only the registers before the output were kept, all other registers have been removed.
Fig. 5. Absolute errors between $1/x$ and corrected approximation $y_l(x, 3)$ using polynomials of different degrees $d$.

V. IMPLEMENTATION RESULTS

Fig. 9 shows absolute errors obtained by VHDL simulations of the implemented architecture, for the architectures of Fig. 7 and Fig. 8 respectively. The bit lengths for this architecture have been chosen for the application scenario of calculating the average of up to $2^{15}$ 16-bit fractional values, i.e. $x$ was chosen to have only 16 bits in front of the comma, and $w$ had 16 bits before as well as after the comma. Tab. 11 shows the synthesis results for this scenario using Quartus Prime 19.1. All internal calculations of the architectures have been performed with 17 bits after the comma. The results shown in Fig. 9 have been obtained for $w = 1$. For reference, we plotted the results when calculating $1/x$ in double precision rounded to the next 16-bit fractional value (optimal 16-bit). As one can see from Fig. 9, the architecture for the polynomial of degree 4 practically achieves a comparable precision to the optimal 16-bit solution (except for $x$ values up to 4, where the absolute error is slightly larger). Using larger bit lengths one can even increase the precision (with higher hardware costs) for large $x$ values, as the results of Fig. 5 demonstrate. For the degree 2 architecture, one can achieve about 10-bit precision for low $x$ values and up to 16-bits precision for large $x$ values.

In Tab. 11, we show synthesis results for two architectures of Fig. 7 and Fig. 8 for the Intel FPGA Stratix V (SGSMD5K2F40C2), respectively. When using all intermediate registers as drawn in the figures, the architecture of Fig. 7 requires 4 clock cycles for calculation and the architecture of Fig. 8 requires 5 clock cycles. This might be an option when one can utilize the pipelining capabilities of the architectures. For iterative algorithms, where the input of a division unit often depends on values that are only known after completing an iteration one might want to reduce...
Fig. 6. Relative errors between $1/x$ and corrected approximation $y_l(x, 3)$ using polynomials of different degrees $d$.

| Table II | Synthesis results for Stratix V: 5GSMD5K2F40C2 |
|----------|-----------------------------------------------|
| Clock cycles | degree 2 | degree 4 |
| ALMs (of 172,600) | 167 | 170 | 227 | 210 |
| Registers (of 706,560) | 157 | 66 | 215 | 70 |
| DSP blocks (of 1,590) | 4 | 4 | 5 | 5 |
| Fmax Slow 900mV 85C (MHz) | 334.78 | 116.78 | 251 | 78.77 |

the number of clock cycles. To show the results for the most extreme reduction of registers in the architectures, the synthesis results are shown for the architectures without intermediate registers (1 clock cycle) as well. The obtain correct timing results, the inputs of the architectures have been registered as well (this accounts for the higher number of registers in the synthesis results). When using the described architecture as part of a larger architecture, such registers will be typically present in the entity feeding the inputs of the division unit. As one can see from these results, the maximum clock frequency drops by about a factor of 3 for the one clock cycle architectures, compared to when using $4 - 5$ clock cycles. For single clock cycle implementations, one obtains a maximum clock frequency higher than 110 MHz for degree 2 and a maximum clock frequency of 78.77 MHz for degree 4. The clock frequencies of the $4 - 5$ clock cycle variant, however, are higher than the clock frequency than can be typically achieved
for advanced data processing architectures (e.g. machine learning algorithms) when using this FPGA, according to the author’s experience.

VI. COMPARISON WITH STATE-OF-THE-ART

In Tab. III, we collected three synthesis results of efficient reciprocal FPGA implementations reported in the literature (achieving 16-bit precision). The comparison can be performed most fairly with [7] as a comparable FPGA is used in this work, while e.g. a VIRTEX-7 FPGA is typically rated for higher clock speeds. When comparing the described architecture with these results, one can see that comparable FPGA utilization is achieved (the author wants to point out that the comparison algorithms are only for calculating the reciprocal while the architecture of this work already includes the multiplication with the dividend \( w \)). Comparing the speed, one can see that the proposed method leads to up to 3 times faster implementations (when comparing with the degree 4 variant of this work) than the state-of-the-art methods.

VII. CONCLUSION

We presented a concept for non-sequential division. It is based on an efficient way of approximating the reciprocal operation by piecewise linearization with node points as powers of two. Using a correction function, that can be approximated with a single polynomial for the whole number range allows implementing the division with high precision and low hardware requirements within one clock cycle.
Fig. 8. Architecture using correction polynomial of degree 4.

REFERENCES

[1] U. Meyer-Baese, *Digital signal processing with field programmable gate arrays*. Springer Verlag, 2004.
[2] A. Ibrahim and M. Valle, “Real-time embedded machine learning for tensorial tactile data processing,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 65, no. 11, pp. 3897–3906, 2018.
[3] S. Zheng, P. Ouyang, D. Song, X. Li, L. Liu, S. Wei, and S. Yin, “An ultra-low power binarized convolutional neural network-based speech recognition processor with on-chip self-learning,” *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 12, pp. 4648–4661, 2019.
[4] G. C. Cardarilli, L. D. Nunzio, R. Fazzolari, M. Panella, M. Re, A. Rosato, and S. Span, “A parallel hardware implementation for 2-d hierarchical clustering based on fuzzy logic,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 68, no. 4, pp. 1428–1432, 2021.
[5] L. Du, Y. Du, and M.-C. F. Chang, “A reconfigurable 64-dimension k-means clustering accelerator with adaptive overflow control,” *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 67, no. 4, pp. 760–764, 2020.
[6] S. F. Obermann and M. J. Flynn, “Division algorithms and implementations,” *IEEE Transactions on Computers*, vol. 46, no. 8, pp. 833–854, Aug 1997.
[7] A. Rodriguez-Garcia, L. Pizano-Escalante, R. Parra-Michel, O. Longoria-Gandara, and J. Cortez, “Fast fixed-point divider based on newton-raphson method and piecewise polynomial approximation,” in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec 2013, pp. 1–6.
[8] M. P. Vestias and H. C. Neto, “Revisiting the newton-raphson iterative method for decimal division,” in *2011 21st International Conference on Field Programmable Logic and Applications*, Sep 2011, pp. 138–143.
[9] E. Libessart, M. Arzel, C. Lahuec, and F. Andriulli, “A scaling-less newton-raphson pipelined implementation for a fixed-point reciprocal operator,” *IEEE Signal Processing Letters*, vol. 24, no. 6, pp. 789–793, June 2017.
[10] R. E. Goldschmidt, *Applications of division by convergence*, Cambridge, MA, USA, jun 1964.
Fig. 9. Absolute errors between $1/x$ and hardware implementations of degrees 2 and 4.

### TABLE III

| Work reported in | FPGA | Slices | LUT | Registers | DSP blocks | Clock cycles | Clock frequency |
|------------------|------|--------|-----|-----------|------------|---------------|-----------------|
| [9]              | VIRTEX-4 SX35 | 347   | 372 | 568       | 7         | 25            | 294.1           |
| [9]              | VIRTEX-7 690T |   not reported | 111 | 240       | 6-10       | 20-31         | 740             |
| [9]              | Stratix-V 5SGXMA7 | 339   |       | 73       | 5         | 3             | 68.62           |

[11] M. J. Flynn, “On division by functional iteration,” *IEEE Transactions on Computers*, vol. C-19, no. 8, pp. 702–706, Aug 1970.

[12] M. D. Ercegovac, L. Imbert, D. W. Matula, J. . Muller, and G. Wei, “Improving goldschmidt division, square root, and square root reciprocal,” *IEEE Transactions on Computers*, vol. 49, no. 7, pp. 759–763, July 2000.

[13] V. K. Jain, G. E. Perez, and J. M. Wills, “Novel reciprocal and square-root vlsi cell: architecture and application to signal processing,” in *Proceedings* ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing, April 1991, pp. 1201–1204 vol.2.

[14] H. C. Neto and M. P. Vestias, “Very low resource table-based fpga evaluation of elementary functions,” in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Dec 2013, pp. 1–6.

[15] J. Pineiro and J. D. Bruguera, “High-speed double-precision computation of reciprocal, division, square root, and inverse square root,” *IEEE Transactions on Computers*, vol. 51, no. 12, pp. 1377–1388, Dec 2002.

[16] M. Lunglmayr and O. Polder, “Fast approximate reciprocal approximations for iterative algorithms,” 2020. [Online]. Available: [https://arxiv.org/abs/2007.06241](https://arxiv.org/abs/2007.06241)
[17] J.-M. Muller, *Elementary Functions - Algorithms and Implementation*. Birkhäuser, 2016.