February 28, 2019

Lightweight approximations for division, exponentiation, and logarithm

Munenori Oizumi

Lightweight approximations for division, exponentiation, and logarithm

ABSTRACT

This disclosure describes low-complexity techniques for approximate division, approximate exponentiation, and approximate logarithm that are accurate to nearly ten percent. Per the techniques, division, exponentiation, and logarithm are expressed in terms of bit-shift and addition operations, which are low-complexity operations. Division, exponentiation and logarithm operations occur frequently in computer science, e.g., in image processing filters, etc. The techniques serve to speed up computations and to reduce silicon area footprint in such compute-intensive applications.

KEYWORDS

- Division
- Exponentiation
- Logarithm
- Approximation
- Noise filter
- Image processing

BACKGROUND

Division, exponentiation and logarithm operations occur frequently in computer science, e.g., in image processing filters, etc. Due to the relatively high complexity of these operations, applications that use these operations are frequently compute-intensive and require substantial amounts of silicon area when implemented as hardware IP.

DESCRIPTION

The techniques of this disclosure express the division, the exponentiation, and the logarithm operations in terms of lower complexity bit-shift and addition operations, thereby serving to speed up computations and reducing silicon area footprint in compute-intensive applications.

Approximate division

The division of a number *a* by *x* can be written as the multiplication of *a* by the reciprocal (1/*x*) of *x*. Therefore, it suffices to determine an approximate reciprocal of a number *x*.

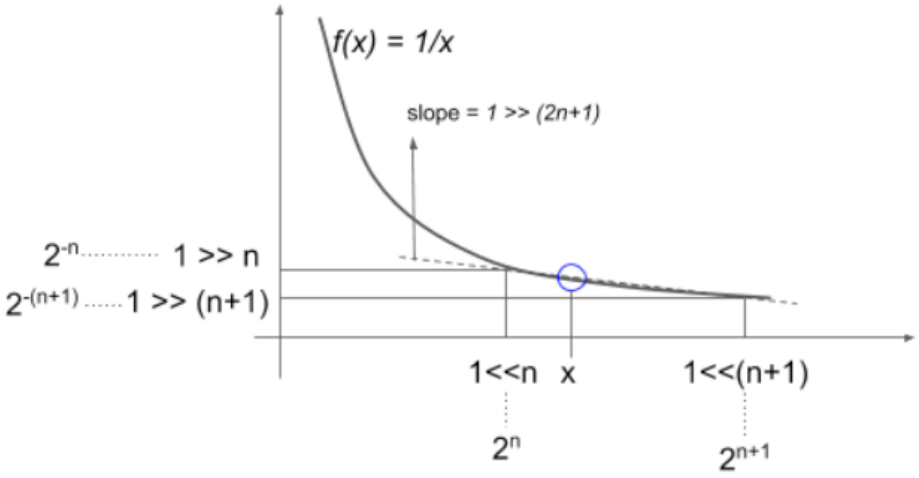

Fig. 1: Deriving an approximate reciprocal of a number *x*

Fig. 1 illustrates the reciprocal function, e.g., the graph of the function $f(x) = 1/x$. Per the techniques of this disclosure, the number *x* is binned between the two closest powers of 2, denoted 2^n and 2^{n+1}. In hardware, the determination of *n* is simple: it is the position of the highest "1" bit in the binary expansion of *x*. In binary notation, the bin boundary 2^n is calculated by left-shifting 1 *n* times, denoted 1<<*n*, and the bin boundary 2^{n+1} is calculated by left-shifting 1 *n*+1 times, denoted 1<<(*n*+1). The reciprocals of the bin boundaries are indicated on the Y-axis, and

are respectively 2^{-n} e.g., $1>>n$, and $2^{-(n+1)}$, e.g., $1>>(n+1)$. The absolute value of the slope of the line joining the bin boundaries is therefore given by

$$\begin{aligned} \text{slope} &= ((1 >> n) - (1 >> (n+1))) / ((1 << (n+1)) - (1 << n)) \\ &= (1 >> (n+1)) / (1 << n) \\ &= 1 >> (2n+1). \end{aligned}$$

The reciprocal $1/x$ is approximated using a first-order approximation, e.g., by subtracting from the reciprocal of the lower bin boundary the product of the slope (approximate derivative) and the distance between x and the lower bin boundary:

$$\begin{aligned} 1/x &\sim (1 >> n) - \text{slope} \times (x - (1 << n)) \\ &= (1 >> n) - (x - (1 << n)) >> (2n+1). \end{aligned}$$

Example

The reciprocal of $x = 5.328125 = 0b101.010101$ is to be computed. In this case, the position of the highest "1" bit, counting from zero, is $n = 2$. Therefore,

$$\begin{aligned} 1.0/5.328125 &\sim (1 >> n) - (x - (1 << n)) >> (2n+1) \\ &= (1>>2) - (5.328125 - (1 << 2)) >> (5) \\ &= 0.25 - (5.328125 - (4)) >> (5) \\ &= 0.25 - (1.328125) >> (5) \\ &= 0.25 - 0.04150 \\ &= 0.2085. \end{aligned}$$

The true value of $1/x$ is 0.18768, so that the approximation above yields an answer to an accuracy of nearly 10%.

For input in unsigned *k.m* format (Uk.m), if the location of the most significant "1" bit is n, the approximation reduces to

$$1.0/x \sim ((1<<2m) >> n) - (x - (1<<n)) << 2m >> (2n+1).$$

For example, in U8.8 format,

$$1.0/x \sim (1 << (2\times 8 - n)) - (x - (1 << n)) >> (2n + 1 - 2\times 8).$$

In this case, the maximum error, which occurs at $x = 3/2 \times (1<<n)$ is ⅛ (excluding quantization error).

Example

The reciprocal of $x = 2.75 = 0b010.11000000$ (in U3.8 format) is to be computed. In this case, k=3, m=8, and the position of the highest "1" is n=9 from the LSB (starting bit count from 0). Therefore,

$$
\begin{aligned}
1.0/2.75 &\sim ((1<<16) >> 9) - (x - (1<< 9)) << 16 >> 19 \\
&= (1 << 7) - (x - (1 << 9)) >> 3 \\
&= b000.10000000 - ((b010.11000000 - b010.00000000) >> 3) \\
&= b000.10000000 - (b000.11000000 >> 3) \\
&= b000.10000000 - b000.00011000 \\
&= b000.01101000 \\
&= 104 / 256 \\
&= 0.40625.
\end{aligned}
$$

The actual value of 1.0/2.75 is 0.363636, so that the error is about 10%.

In this manner, a relatively high complexity operation such as reciprocation, hence division, is computed with low-cost hardware operations, e.g., a few bit-shift and subtraction operations.

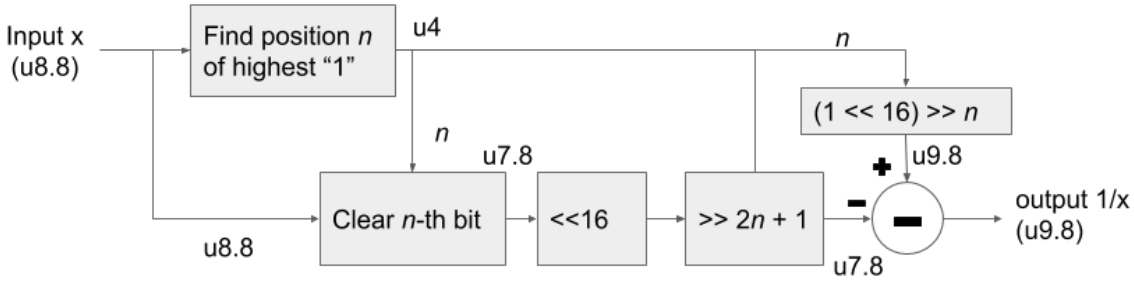

Fig. 2: Reciprocation circuit

A circuit or hardware schematic that computes the approximate reciprocal of a U8.8 number using the above techniques is illustrated in Fig. 2. As seen from Fig. 2, the reciprocal of

a number can be calculated with one operation to find the most significant 1, one subtraction, and a few bit shifts.

Approximate exponentiation

The exponent e^{-x} of a number x can be written as $e^{-x} = 2^{-x \log(e)}$, so that the exponent of x can be found by raising 2 to $-x\log_2 e = -1.4427x$. Therefore it suffices to determine an approximate value for 2 raised to x. The constant $\log_2 e = 1.4427$ can be pre-computed and stored, possibly at a lower resolution, e.g., $\log_2 e \sim 1.5$.

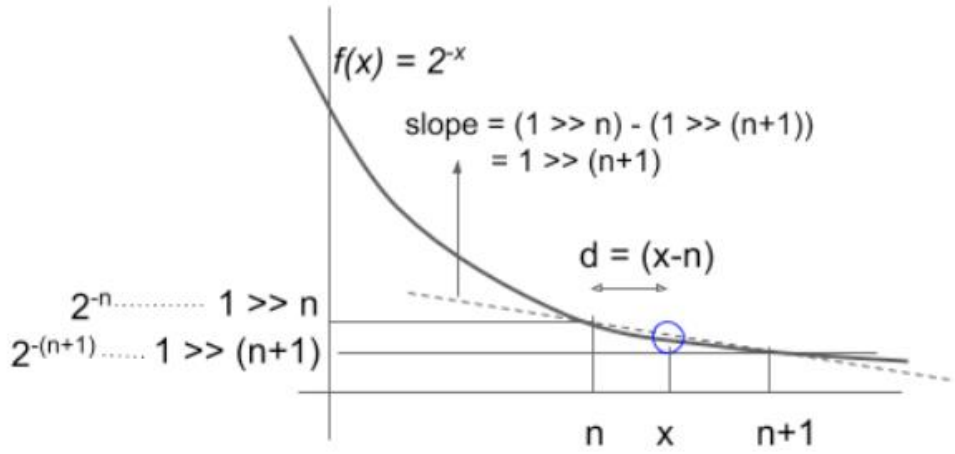

Fig. 3: Deriving an approximation to 2 raised to a number x

Fig. 3 illustrates the function $f(x) = 2^{-x}$. Per the techniques of this disclosure, the number x is binned between the two closest integers, denoted n and n+1. Equivalently, n is the integral part of x, and $x-n$, denoted d, is the fractional part of x. On the Y-axis, the corresponding bin boundaries are respectively 2^{-n}, e.g., 1>>n, and $2^{-(n+1)}$, e.g., 1>>(n+1).

The absolute value of the slope of the line joining the bin boundaries is therefore given by

$$\text{slope} = ((1>> n) - (1 >> (n+1)))$$
$$= 1 >> (n+1).$$

The function 2^{-x} is approximated using a first-order approximation, e.g., by subtracting from the integral part of x the product of the slope (approximate derivative) and the fractional part d of the x:

$$2^{-x} \sim (1>>n) - \text{slope} \times d$$
$$= (1>>n) - d >> (n+1).$$

Example

If $x = 5.328125$, then the integral part $n = 5$, and the fractional part $d = x - n = 0.328125$.

Therefore,

$$2^{-x} \sim (1>>n) - d >> (n+1)$$
$$= (1>>5) - 0.328125>> 6$$
$$= 0.0261$$

The true value of 2^{-x} is 0.0249, so that the approximation above yields an answer to an accuracy of better than 5%.

For input in unsigned *k.m* format (Uk.m), e.g., the number of integral bits being k and the number of fractional bits being m, the MSB k bits form the integral part n, and the LSB m bits form the fractional part $d = x - n$, so that the approximation reduces to

$$2^{-x} \sim 1<<(m-n) - d >> (n+1)$$
$$= ((1<<(m+1)) - d) >> (n+1).$$

In this manner, a relatively high complexity operation such as exponentiation is computed with low-cost operations, e.g., a few bit-shift operations and one subtraction operation.

The constant $\log_2 e = 1.4427$ that translates between 2^{-x} and e^{-x} is approximately 369 in U8.8 format, so that $x\log_2 e$ can be computed efficiently, e.g., using bit-shifts and additions, using

$$x\log_2 e \sim (369x+4)>>8.$$

Approximate logarithm

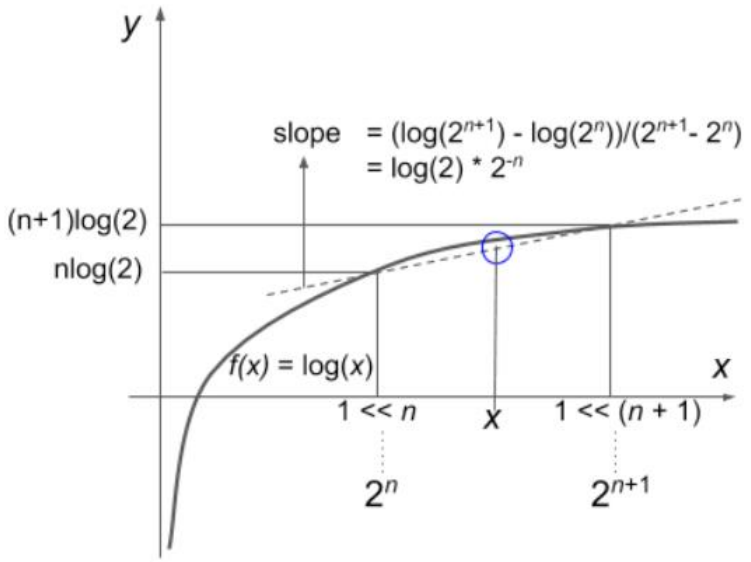

Fig. 4: Deriving an approximate logarithm of a number *x*

Fig. 4 illustrates the logarithm function, e.g., the graph of the function $f(x) = \log(x)$, where the base of the logarithm may be, e.g., 2, *e*, 10, or other convenient number. Per the techniques of this disclosure, the number x is binned between the two closest powers of 2, denoted 2^n and 2^{n+1}. In hardware, the determination of n is simple: it is the position of the highest "1" bit in the binary expansion of x. In binary notation, the bin boundary 2^n is calculated by left-shifting 1 n times, denoted $1<<n$, and the bin boundary 2^{n+1} is calculated by left-shifting 1 $n+1$ times, denoted $1<<(n+1)$. The logarithm of the bin boundaries are indicated on the Y-axis, and are respectively $n\log(2)$ and $(n + 1)\log(2)$. The slope of the line joining the bin boundaries is therefore given by

$$\text{slope} = (\log(2^{n+1}) - \log(2)) / (2^{n+1} - 2^n)$$
$$= \log(2) \times 2^{-n}.$$

The logarithm of x is approximated using a first-order approximation, e.g., by adding the product of the slope (approximate derivative) and the distance between x and the lower bin boundary to the logarithm of the lower bin boundary:

$$\log(x) \sim n \times \log(2) + \text{slope} \times (x - (1 << n)) \\ = \log(2) \times (n + (x - 2^n) \times 2^{-n}).$$

Note that log(2) is a constant, and can be pre-calculated and hardcoded with little overhead. For example, with 8-bit sub-decimal precision $\log_e(2)$ is approximated as 177/256, because 177/256 = 0.691…, which is close to the true value of $\log_e(2) = 0.693\ldots$.

Example

The logarithm of x = 5.328125 = 0b101.010101 is to be computed. In this case, the position of the highest "1" bit, counting from zero, is n = 2. Therefore,

$$\log_e(5.328125) \sim \log_e(2) \times (2 + (x - 2^2) \times 2^{-2}) \\ = \log_e(2) \times (2 + 1.328125 / 4) \\ = 0.6931 \times (2 + 1.328125 / 4) \\ = 1.616.$$

The true value of log(x) is 1.673, so that the approximation above yields an answer to an accuracy of nearly 4%.

For input in unsigned *k.m* format (Uk.m), if the location of the most significant "1" bit is n, the approximation reduces to

$$\log(x) \sim a \times (\ (\ (n - m) << n\ ) + x - (1<<n)\ ) >> n,$$

where a is approximation of log(2) in Uk.m format.

For example, in U8.8 format,

$$\log_e(x) \sim 177 \times (\ (\ (n - 8) << n\ ) + x - (1<<n)\ ) >> n.$$

Example

The logarithm of x = 5.328125 = 0b101.01010100 (in U3.8 format) is to be computed. In this case, k=3, m=8, and the position of the highest "1" is n=10 from the LSB (starting bit count from 0). Therefore,

$$\begin{aligned}\log(x) &\sim a \times (((n - 8) << n) + x - (1 << n)) >> n \\ &= 177 (((10 - 8) << 10) + 1364 - (1 << 10)) >> 10 \\ &= 412,\end{aligned}$$

which in U3.8 is the fraction 412/256 = 1.609375.

The actual value of log(5.328125) is 1.673, so that the error is about 4%.

In this manner, a relatively high complexity operation such as logarithm is computed with low-cost hardware operations, e.g., a few bit-shift and subtraction operations.

Second-order approximation of reciprocal with improved accuracy

The approximation to $1/x$ can be further improved by observing, as mentioned above, that the maximum error is $(1/12) \times (1 >> n - m)$, which occurs at $x = 1.5 \times (1 << n)$. The error can therefore be approximated with a second-order polynomial, as follows:

$$\text{error\_estimate} = 1/3 \times (x - (1 << n)) \times (((1 << (n + 1)) - x) >> (3n - m), \text{ if } 3n \geq m, \text{ and}$$

$$\text{error\_estimate} = 1/3 \times (x - (1 << n)) \times (((1 << (n + 1)) - x) >> (m - 3n), \text{ if } 3n < m.$$

This error estimate can be removed from the first-order approximation. Constants appearing in the approximation formulas, e.g., ⅓ in the above formula, can be pre-calculated. For example, ⅓ in U3.8 format is 0b000.01010101.

Example

The reciprocal of x = 2.75 = 0b010.11000000 in U3.8 (k=3, m=8) format is to be computed. In this case, most significant "1" bit, counting from LSB and starting bit-count at 0, is at position n=9. Using the first-order approximation, e.g.,

$$1.0/x \sim ((1<<2m) >> n) - (x - (1<<n)) << 2m >> (2n+1),$$

$$\begin{aligned} 1.0/2.75 &\sim ((1<<16) >> 9) - (2.75 - (1<< 9)) << 16 >> 19 \\ &= \text{b000.10000000} - (\text{b000.11000000} >> 3) \\ &= \text{b000.10000000} - \text{b000.00011000} \\ &= \text{b000.01101000} \\ &= 104 / 256 = 0.40625. \end{aligned}$$

The error estimate is given by

$$\begin{aligned} \text{error\_estimate} &\sim 1/3 \times (x - (1 << n)) \times ((1 << (n-1)) - x) >> (3n - m) \\ &= 1/3 \times 0.375 \times 0.625 / 4 \\ &= 0.01953125. \end{aligned}$$

Therefore the second-order approximation to the reciprocal of 2.75 is

$$\begin{aligned} 1.0/2.75 &\sim \text{1st order approximation} - \text{error\_estimate} \\ &= 0.40625 - 0.01953125 \\ &= 0.38671875. \end{aligned}$$

Comparing to the true value of 1.0/2.75 = 0.3636, the accuracy is better than 7%.

Second-order approximation of exponential with improved accuracy

It suffices to show the improvement to the approximation of 2^{-x}, since the conversion from e^{-x} to 2^{-x} has been explained already. For 2^{-x}, second-order error estimate term is

$$\text{error\_estimate}(x) = \text{max\_error} \times d \times ((1 << m) - d) \times 4 >> (2m + n) \text{ if } 2m + n \geq 0, \text{ and}$$

$$\text{error\_estimate}(x) = \text{max\_error} \times d \times ((1 << m) - d) \times 4 << (-2m - n) \text{ if } 2m + n < 0,$$

where n is the integral part of x ($n = x >> m$), d the is fractional part of x (LSB m-bit, $d = x - (n << m)$), and max_error is the error at $x = (n << m) + (1 << (m - 1))$, e.g., halfway between $x = n << m$ and $x = (n + 1) << m$. max_error is calculated as

$$\text{max\_error} = (3/4 - \text{sqrt}(2)/2) \times (1 << m).$$

Thus, a second-order approximation to 2^{-x} is

$$2^{-x} \sim (((1 << (m + 1)) - d) >> (n + 1)) - \text{error\_estimate}(x), \text{ if } n + 1 \geq 0, \text{ and}$$

$$\sim (((1 << (m + 1)) - d) << (- n - 1)) - \text{error\_estimate}(x), \text{ if } n + 1 < 0.$$

Example

For x = 1.3 (333 in U3.8 format), $2^{-1.3}$ = 0.40613.

1st order approximation of $2^{-1.3}$ is 0.421875 (108 in U3.8), and error is ~4%.

2nd order approximation of $2^{-1.3}$ is 0.40625 (104 in U3.8), and error is ~0.03%.

Second-order approximation of logarithm with improved accuracy

For log(x), second-order error estimate term is

$$\text{error\_estimate}(x) = \text{max\_error} \times d \times ((1 << n) - d) \times 4 >> 2n,$$

where n is highest '1' bit position in x, and $d = x - (1 << n)$. max_error is the error at $x = (1 << n) + (1 << (m - 1))$, e.g., halfway between $x = 1 << n$ and $x = 1 << (n + 1)$, which is calculated as max_error = (log(3/2) − log(2)/2) × (1 << m). For m = 8, max_error = 15.

Thus, a second-order approximation to log(x) is

$$\log(x) \sim a \times (((n - m) << n) + x - (1 << n)) >> n + \text{error\_estimate}(x)$$

Example

For x = 5.328125 (1364 in U3.8 format), log(x) = 1.673.

1st order approximation of log(x) is 1.613 (413 in U3.8 format), an error of ~4%.

2nd order approximation of log(x) is 1.664 (426 in U3.8 format), an error of ~0.53%.

Example application: proximity based noise filter in image processing

In image processing, there are situations in which reciprocal and exponential functions find use. For example, weight calculations in proximity-based noise filters, e.g., bilateral filters

or non-local mean noise filters, use the reciprocal and the exponential functions. The output image quality remains of good quality for sufficiently accurate approximations of these functions.

Proximity-based noise filters use weighted averaging to calculate the output pixel. The weight w is calculated using

$$w = \exp(-a*d/s),$$

where a is a tuning parameter, d is the distance between a target and a candidate, and s is a noise variance.

The divider and the exponential in this equation being compute-intensive operations, the techniques of this disclosure can be applied advantageously. Another problem is that weight w is calculated for every candidate in the search area. If the search area is a square of size N, the two reciprocal and exponentiation calculations are carried out N^2 times. A value of N between 5 and 16 is common in modern non-local mean filters, which makes a full-fledged hardware calculation of the reciprocal or exponential calculation prohibitive in area. Again, the techniques of this disclosure apply advantageously to optimize area in a hardware implementation of non-local mean filters.

(a) **(b)**

Fig. 5: An image processed with non-local mean filters: (a) using approximate division and exponentiation, and (b) using exact division and exponentiation

Fig. 5 illustrates an image processed with non-local mean filters using approximate division and exponentiation, per the techniques of this disclosure, in Fig. 5(a) and using exact division and exponentiation, in Fig. 5(b). As can be seen, there is hardly any difference between filtering using exact functions, which is compute-intensive, and filtering using approximate functions, which is of low compute complexity.

CONCLUSION

Division, exponentiation and logarithm operations occur frequently in computer science, e.g., in image processing filters, etc. The techniques of this disclosure serve to speed up computations or to reduce silicon area footprint in such compute-intensive applications by approximating division, exponentiation, and logarithm operations by a sequence of low-complexity bit-shift and addition operations.