Logarithms in Computer Arithmetic

by Vahe Vartan

December 15, 2006

Abstract

In this paper, the reader is presented with three algorithms to compute the logarithm of a positive real number. The first two that are presented are essentially the same: the first is works for any base logarithm, while the second is designed specifically to calculate the base-2 logarithm. Our final algorithm is executed with the use of a table of values, which can be used to increase the speed of calculating non-base-2 logarithms (i.e. $\ln(x)$, the natural logarithm) in hardware. Along the way, we will note the advantages and disadvantages of each algorithm.

1 Introduction

The logarithm, like other transcendental functions, is introduced purely as a function with an analytical solution process. For example, when students are asked to exactly calculate $\log_{10}(x)$ (by hand), only the integer part of the solution can be solved with ease by simply left-shifting the decimal point until there is only one digit to its left.

Example 1. Given $\log_{10}(143)$, we see that $143. \rightarrow 1.43$ (2 left-shifts). We therefore guess that $\log_{10}(143) \approx 2$.

We will demonstrate an extension of this idea to arbitrary precision. For our first two algorithms, we will do this using surprisingly simple algebraic manipulation. For our last, we will introduce a more complex algorithm that can directly compute any logarithm via hardware.

The reader should note that numbers that are not labeled are assumed to be part of the decimal system. Numbers in binary are labeled with the subscript BIN.

2 The Simple Algorithm: In Base-d

2.1 Deriving the Algorithm

The simple algorithm is perhaps not as simple as one would like it to be. An exponent must be computed at each iteration of the algorithm; this can be particularly difficult if we are using a large or non-integer base for our logarithm. We will be calculating one digit at a time; this indicates that this algorithm linearly converges to the solution.

Let x_0 be a positive real number. Let $a_{o.}a_1a_2a_3\cdots$ be the radix-d solution to $\log_d(x_0)$. Note that [1]:

$$a_{o.}a_1a_2a_3\cdots = \sum_{i=0}^{\infty} a_i \cdot d^{-i} = a_0 + \frac{a_1}{d} + \frac{a_2}{d^2} + \frac{a_3}{d^3} + \dots .$$

Therefore, we have [1]:

$$x_0 = d^{a_0 + \frac{a_1}{d} + \frac{a_2}{d^2} + \frac{a_3}{d^3} + \cdots} = d^{a_0} \cdot d^{\frac{a_1}{d} + \frac{a_2}{d^2} + \frac{a_3}{d^3} + \cdots}. \tag{1}$$

Now we can extract the a_0 term from this equation to get [1]:

$$\exists p \in \mathbb{Z} \quad \text{s.t.} \quad d^p \leq x_0 < d^{p+1} \text{ where } a_0 = p. \tag{2}$$

To continue this process, we alter Equation (1) and define x_1 as follows [1]:

$$x_1 = \left(\frac{x_0}{d^{a_0}}\right)^d = d^{a_1 + \frac{a_2}{d} + \frac{a_3}{d^2} + \cdots}.$$

We can repeat the process in Equation (2), with x_1 to get a_1. We can then define a recurrence as follows:

$$\begin{aligned} x_0 &= d^{a_0+\frac{a_1}{d}+\frac{a_2}{d^2}+\frac{a_3}{d^3}+\cdots} \\ x_{n+1} &= \left(\frac{x_n}{d^{a_n}}\right)^d. \end{aligned} \tag{3}$$

If we look closer at x_n, we see that this value can actually be written easily in a floating point representation if we let $M_n = d^{\frac{a_{n+1}}{d}+\frac{a_{n+2}}{d^2}+\cdots}$:

$$x_n = d^{\frac{a_{n+1}}{d}+\frac{a_{n+2}}{d^2}+\cdots} \cdot d^{a_n} = M_n \cdot d^{a_n}.$$

Therefore, our algorithm is as follows:

$$\begin{aligned} x_n &= M_n \times d^E \text{ where } a_n = E \\ x_{n+1} &= (M_n)^d. \end{aligned} \tag{4}$$

Remark 2. To achieve our floating point representation, we need only normalize our x_n value, i.e.

$$x_n = b_i \cdots b_1 b_0 . b_{-1} \cdots = b_i . b_{i-1} \cdots \times d^i,$$

where $0 \leq i < d$ (note that $a_i = i$). This is a linear time algorithm since we shift our decimal point left until $b_i < d$.

Given all of the above, it is easy to see that our algorithm computes

$$\log_d(x_0) = a_0 + a_1 \cdot d^{-1} + a_2 \cdot d^{-2} + \ldots.$$

We can demonstrate our algorithm with an example.

2.2 An Example

Example 3. Let $\log_{10}(1343) = a_0.a_1a_2a_3\ldots$. Find a_0, a_1, a_2, a_3 in decimal.

We get the following:

$$\begin{aligned} x_0 &= 1343 && = 1.343 \times 10^3 && \Rightarrow a_0 = 3 \\ x_1 &= (1.343)^{10} && = 1.9088 \times 10^1 && \Rightarrow a_1 = 1 \\ x_2 &= (1.9088)^{10} && = 6.42101 \times 10^2 && \Rightarrow a_2 = 2 \\ x_3 &= (6.42101)^{10} && = 1.19128 \times 10^8 && \Rightarrow a_3 = 8 \end{aligned}$$

Our algorithm's answer is 3.128 while the actual answer to 4 digits of accuracy is 3.128. We see that each iteration adds one digit of accuracy. Therefore, our algorithm converges to the actual answer in a linear fashion.

2.3 Disadvantages

The main disadvantage of this algorithm is that for large d, we have to compute large exponents. This would be very troublesome if we chose to implement a base-10 logarithm into hardware: we would have to compute the 10th power every iteration. We could do this by multiplying a number by itself 10 times; not only is this time-consuming, but what if we decide to change our base, d, to another value, such as e, Euler's constant? We would be taking exponents of non-integer values - this would be extremely difficult to implement in hardware. In short, we would like to devise a new algorithm that does not have the aforementioned problems.

3 The Simple Algorithm: In Base-2

3.1 Setting the Ground Rules

As is usually the case with hardware implementation, it is the most expedient to define our algorithm in binary. We will choose to compute the base-2 logarithm. Furthermore, we will assume that the value we are taking the logarithm of, x_0, is such that $1 \leq x_0 \leq 2$.

Remark 4. We will speak shortly upon how we can calculate *any* base-2 logarithm, given the above restriction of $x_0 \in [1,2]$. Say we are given a number y such that

$$\{y \in \mathbb{R} \mid y \in (0,1) \text{ or } y \in (2,\infty)\},$$

and we want to solve $\log_2(y)$. We can rewrite y in binary floating point representation, $y = b_i.b_{i-1}\cdots \times 2^i$, as done in the previous section. Finally, we have the following:

$$\begin{aligned} \log_2(y) &= \log_2(b_i.b_{i-1}\cdots \times 2^i) \\ &= \log_2(b_i.b_{i-1}\cdots) + \log_2(2^i) \\ &= \log_2(b_i.b_{i-1}\cdots) + i. \end{aligned}$$

We see that $1 \leq b_i.b_{i-1}... \leq 2$ as desired and so, we have shown that we can calculate the base-2 logarithm of any positive real number if we know how to calculate $\log_2(x_0)$, where $1 \leq x_0 \leq 2$.

3.2 The Computer-Friendly Algorithm

First, we will define $z_n = z_{n-1} + a_n \cdot 2^{-n}$ and $z_0 = 0$, where z_n is the nth approximation of $\log_2(x_0)$, and, just as before, a_n is the nth digit answer of our algorithm. Notice that x_n from the Equation (3) is of the form $x_n = 2^p$, $0 \leq p \leq 1$ for $p \in \mathbb{R}$. It follows that $\log_2(x_n) = p$; if $p \geq \frac{1}{2}$, then the next digit in $\log_2(x_n)$, represented in binary by a_{n+1}, must be equal to one. Conversely if $p < \frac{1}{2}$, a_{n+1} must be equal to zero. Therefore, we can see that [3]:

$$\begin{aligned} a_{n+1} = 0 &\Leftrightarrow x_n < \sqrt{2} \\ a_{n+1} = 1 &\Leftrightarrow x_n \geq \sqrt{2}. \end{aligned}$$

Notice that when $a_{n+1} = 0$, $x_{n+1} = x_n^2$ and similarly, when $a_{n+1} = 1$, $x_{n+1} = \frac{x_n^2}{2}$ from Equation (3). Using all of these properties we get [3]:

$$\begin{aligned} x_n^2 < 2 &\Rightarrow x_{n+1} = x_n^2 \text{ and } z_{n+1} = z_n \\ x_n^2 \geq 2 &\Rightarrow x_{n+1} = \frac{x_n^2}{2} \text{ and } z_{n+1} = z_n + 2^{-(n+1)}. \end{aligned}$$

A line-by-line algorithm for n-bits of accuracy is given below:

```
x_0 ← input
z_0 ← 0
for i = 0 to n − 1
{
   x_{i+1} ← (x_i)^2
   z_{i+1} ← z_i
   if (x_{i+1} ≥ 2):  x_{i+1} ← x_{i+1} / 2
                      z_{i+1} ← a_{i+1} + 2^{−(i+1)}
}
```

We will demonstrate this algorithm through an example.

3.3 Another Example

Example 5. Compute $\log_2(1.1_{\text{BIN}})$ to 6-bits of accuracy.

We use our algorithm above as follows:

1. Assign z_0 to 0. Let $i = 0$.
2. Assign x_{i+1} to $(x_i)^2$.
3. Assign z_{i+1} to z_i.

4. Check to see if the term x_{i+1} is greater than or equal to 2. If it is then:
   a) Left-shift the decimal place in x_{i+1} by one digit ($\equiv$ divide by 2).
   b) Assign the $-(i+1)$th bit in z_{i+1} to 1.
5. Do steps 2-4 with $i = i+1$ until $i = n$.

We have $x_0 = 1.1_{\text{BIN}}$ and we execute the above steps to get:

$$\begin{array}{rrcll}
(x_0)^2 = & 10.010000_{\text{BIN}} \geq 10_{\text{BIN}} & \Rightarrow & x_1 = 1.001000_{\text{BIN}} \text{ and } & z_1 = .1_{\text{BIN}} \\
(x_1)^2 = & 1.010001_{\text{BIN}} < 10_{\text{BIN}} & \Rightarrow & x_2 = 1.010001_{\text{BIN}} \text{ and } & z_2 = .10_{\text{BIN}} \\
(x_2)^2 = & 1.100110_{\text{BIN}} < 10_{\text{BIN}} & \Rightarrow & x_3 = 1.100110_{\text{BIN}} \text{ and } & z_3 = .100_{\text{BIN}} \\
(x_3)^2 = & 10.100100_{\text{BIN}} \geq 10_{\text{BIN}} & \Rightarrow & x_4 = 1.010010_{\text{BIN}} \text{ and } & z_4 = .1001_{\text{BIN}} \\
(x_4)^2 = & 1.101001_{\text{BIN}} < 10_{\text{BIN}} & \Rightarrow & x_5 = 1.101001_{\text{BIN}} \text{ and } & z_5 = .10010_{\text{BIN}} \\
(x_5)^2 = & 10.101101_{\text{BIN}} \geq 10_{\text{BIN}} & \Rightarrow & x_6 = 1.010110_{\text{BIN}} \text{ and } & z_6 = .100101_{\text{BIN}}
\end{array}$$

The actual answer to $\log_2(1.1_{\text{BIN}})$, to the same 6-bit accuracy, is $.100101_{\text{BIN}}$ and so, it is evident that we are actually adding 1-bit of accuracy per iteration. This implies that this algorithm converges to the actual answer in linear time, which is expected considering that this algorithm is a specialized version of our abstract base-d algorithm.

3.4 Advantages and Disadvantages

This algorithm is simple and requires little work: each iteration requires one multiply and possibly one left-shift. We can see that this algorithm requires much less work than one that would be conducted for a non-base-2 logarithm.

The disadvantage of this algorithm is that it can only calculate the base-2 logarithm. In general, base-10 and base-e logarithms are desired. There is, however, a quick fix for this and it is explained in the next section.

3.5 Calculating Different Logarithms with Log_2

The logarithm of any base is proportional to the logarithm of any other base. If we are given a base-d logarithm and would like to make it a base-c logarithm, we can convert it by doing the following:

$$\log_c(x) = \frac{\log_d(x)}{\log_d(c)}.$$

Therefore, if we have the answer to a logarithm in base-d, we can obtain the answer to the logarithm in base-c by dividing our base-d answer by $\log_d(c)$ (i.e. for $c = 2, d = 10, x = 8$ we get $\log_2(8) = \frac{\log_{10}(8)}{\log_{10}(2)}$).

Specifically for $\log_2$, if we want to convert to $\log_d$, we do the following operation:

$$\log_d(x) = \frac{\log_2(x)}{\log_2(d)}.$$

If we wanted to implement this into hardware, we would need a table of values for $\log_2(d)$. To calculate base-10 and natural logarithms we would respectively need table values for $\log_2(10)$ and $\log_2(e)$. We have therefore shown that we can calculate non-base-2 logarithms with a simple divide operation, as long as we have the necessary table value stored.

4 Calculating the Logarithm with Table Values

4.1 A Different Approach

The proceeding algorithm is derived in a completely different manner. The simple algorithm was derived with simple algebraic manipulation; the complex algorithm, as we will call it, utilizes limits. We would like to develop a direct hardware implementation of the natural logarithm, $\ln(x)$, in this section.

4.2 The Complex Algorithm

We want to calculate $\ln(x_0)$. We begin by presenting the following recursive formulas [2]:

$$\begin{aligned} x_{i+1} &= x_i \cdot b_i \\ y_{i+1} &= y_i - \ln(b_i). \end{aligned} \tag{5}$$

We let x_0 be the input of the log function, and $y_0 = 0$, while y_{i+1} is meant to ultimately equal the output of the log function. The idea behind these equations is that we will choose b_i such that $\lim_{i\to\infty} x_{i+1} = 1$. It follows that $x_0 \prod_{i=0}^{\infty} b_i = 1$ and so, we have [2]:

$$\prod_{i=0}^{\infty} b_i = \frac{1}{x_0}.$$

We let $b_i = (1 + s_i 2^{-i})$, where $s_i \in \{-1, 0, 1\}$, and assert that this causes convergence, if we choose each s_i correctly. We find the following inequality from convergence [2]:

$$\begin{aligned} \prod_{i=0}^{\infty} (1 - 2^{-i}) \;\leq\; \frac{1}{x_0} \;&\leq\; \prod_{i=0}^{\infty} (1 + 2^{-i}) \\ 0.29 \;\leq\; \frac{1}{x_0} \;&\leq\; 4.77 \\ 0.21 \;\leq\; x_0 \;&\leq\; 3.46 \end{aligned}$$

From the above, we can see that the domain of x_0, the input, is between 0.21 and 3.46; we may not stray outside of these bounds when taking $\ln(x_0)$. The main issue, however, is the selection of s_i. There are many complex ways of selecting the s_i values, but we will employ a simple, one-sided algorithm.[2] If $x_0 < 1$, then $s_i \in \{0, 1\}$; if $x_0 > 1$, then $s_i \in \{-1, 0\}$; finally, if $x_0 = 1$, then $s_i = 0$. We will select as follows:

$$\begin{aligned} \text{For } x_0 < 1 \quad &\text{if } x_i \cdot (1 + 2^{-i}) \leq 1 \quad s_i = 1 \\ &\text{else} \qquad\qquad\qquad\quad s_i = 0 \\ \text{For } x_0 > 1 \quad &\text{if } x_i \cdot (1 - 2^{-i}) \geq 1 \quad s_i = -1 \\ &\text{else} \qquad\qquad\qquad\quad s_i = 0 \end{aligned}$$

Note that when $s_i = 0$, we actually multiply x_{i+1} by 1.

We will now demonstrate the use of the y_i recurrence formula from Equation (5). Notice that [3]:

$$\begin{aligned} \lim_{i\to\infty} y_{i+1} &= y_0 - \sum_{i=0}^{\infty} \ln(b_i) \\ &= y_0 - \ln\left(\prod_{i=0}^{\infty} b_i \right) \\ &= y_0 - \ln\left(\frac{1}{x_0} \right) \\ &= y_0 + \ln(x_0). \end{aligned}$$

We know $y_0 = 0$ and so, we have $y_{i+1} \to \ln(x_0)$ as $i \to \infty$. Also note that the values for $\ln(1 + s_i 2^{-i})$ and $(1 + s_i 2^{-i})$ must be stored in a table with values of size n-bits (i.e. a ROM chip), where n is the desired bit precision.[2] If we decide we want n-bits of accuracy to the right of the decimal place, we set our upper bound in both the summation and product to $n - 1$, i.e.

$$y_n = y_0 - \sum_{i=0}^{n-1} \ln(b_i)$$

$$x_n = x_0 \prod_{i=0}^{n-1} b_i.$$

We will demonstrate the algorithm in an example.

4.3 The Last Example

Example 6. Compute $\ln(0.627)$.

We use the above algorithm with 5-bit accuracy to get the following:

$$\begin{array}{llllll}
x_0 = & 0.627 & y_0 = & 0 & s_0 = 0 & \Leftarrow 0.627(1+2^{-0}) > 1 \\
x_1 = & 0.627(1+0\cdot 2^{-0}) = 0.627 & y_1 = & y_0 = 0 & s_1 = 1 & \Leftarrow 0.627(1+2^{-1}) \leq 1 \\
x_2 = & 0.627(1+1\cdot 2^{-1}) = 0.941 & y_2 = & y_1 - \ln(1+1\cdot 2^{-1}) = -0.405 & s_2 = 0 & \Leftarrow 0.941(1+2^{-2}) > 1 \\
x_3 = & 0.941(1+0\cdot 2^{-2}) = 0.941 & y_3 = & y_2 = -0.405 & s_3 = 0 & \Leftarrow 0.941(1+2^{-3}) > 1 \\
x_4 = & 0.941(1+0\cdot 2^{-3}) = 0.941 & y_4 = & y_3 = -0.405 & s_4 = 1 & \Leftarrow 0.941(1+2^{-4}) \leq 1 \\
x_5 = & 0.941(1+1\cdot 2^{-4}) = 0.999 & y_5 = & y_4 - \ln(1+1\cdot 2^{-4}) = -0.466 & s_5 = 0 & \Leftarrow 0.999(1+2^{-5}) > 1
\end{array}$$

After five iterations we have $-0.466 = -0.01110_{\text{BIN}}$ as our answer; the actual answer to 5-bit accuracy is also -0.01110_{BIN}. Just like our two previous algorithms, we converge to the actual value linearly.

4.4 The Advantages and Disadvantages of the Complex Algorithm

The complex algorithm's only advantage is that it directly calculates the natural logarithm through a hardware implementation with no conversion. We could, in fact, use this implementation to calculate *any* base logarithm via hardware with no conversion. This method, much like the others, requires one iteration for each bit of accuracy and has linear convergence.

The disadvantages of this algorithm are very evident. First, we must have a table of values to look up; in our simple $\log_2$ algorithm, we need only one table value and one more division at the end of our algorithm to convert from $\log_2(x)$ to $\ln(x)$. Second, we require more work per step (two multiplications and possibly one subtraction).

5 Conclusion

We have examined three algorithms to compute the logarithm function; the last two, specifically, can be readily implemented into a hardware implementation. In conclusion, we can see that the simple $\log_2$ requires the least amount of work and is the easiest and fastest algorithm. Our complex algorithm is interesting, but does not provide us with a hardware implementation that is as reasonable as the simple $\log_2$ one. Through a simple and complex analysis of the logarithm, the author hopes that the reader holds a greater understanding of this transcendental function.

Bibliography

[1] M. Goldberg. Computing logarithms digit-by-digit. *BRICS*, RS-04-17:1–6, 2004.

[2] I. Koren. *Computer Arithmetic Algorithms*. A K Peters, Natick, MA, 2002.

[3] J.C. Majithia and D Levan. A note on base-2 logarithm computations. *IEEE Proc.*, 61:213–214, 1973.