# Solidity implementation of binary logarithm for 128-bit numbers using the Cole-Dickinson algorithm

by the PowerPool team

October 1, 2023

## 1 Introduction

Certain applications, like evening out the ranges of values that can differ wildly, call for implementation of a logarithm. However, efficient logarithm implementations on Solidity are few and far between. In particular, we known of no efficient implementation of logarithm for 128-bit numbers. In this short paper, we endeavor to demonstrate the algorithm we have come up with, which, to the best of our knowledge, is novel.

## 2 The Cole-Dickinson logarithm

The Cole-Dickinson binary logarithm algorithm, perhaps not as well-known as it'd deserve, reads as published by Sean Anderson [cite bithacks]:

```
uint32_t v; // find the log base 2 of 32-bit v
int r;      // result goes here

static const int MultiplyDeBruijnBitPosition[32] =
{
  0, 9, 1, 10, 13, 21, 2, 29, 11, 14, 16, 18, 22, 25, 3, 30,
  8, 12, 20, 28, 15, 17, 24, 7, 19, 27, 23, 6, 26, 5, 4, 31
};

v |= v >> 1; // first round down to one less than a power of 2
v |= v >> 2;
v |= v >> 4;
v |= v >> 8;
v |= v >> 16;

r = MultiplyDeBruijnBitPosition[(uint32_t)(v * 0x07C4ACDDU) >> 27];
```

Listing 1: Cole-Dickinson algorithm

Now, the working principles of the algorithm are not immediately obvious. Firstly, we do a cascading bit-shift-or. The purpose of this is to produce a

(zero-padded) value of $2^k - 1$ nature, so a zero-padded sequence of all ones. It may seem that we need to also bit-shift-or with numbers of bits that are not themselves powers of two (say, with 3), but this is not the case, as the sequential nature of the transform allows us to skip all shift values that are representable as a sum of previously encountered ones (so, e.g, a right shift-or with a value of 2 on a sequence that has already undergone right shift-or with a value of 1 has this operation with a value of 3 already baked in, as $1 + 2 = 3$). Afterwards, the only thing done is determination of a lookup table index, wherefrom the logarithm is recovered. This determination is done by multiplying our sequence with a weird number and truncating it to five most significant bits to obtain a number in the [0; 31] range (of which the table is a permutation).
The idea of using de Bruijn sequences for indexing is due to Leiserson, Prokop, and Randall [cite them]. Now, a de Bruijn sequence of order $k$ is a $2^k$-length binary sequence in which each possible $k$-length bit string appears exactly once as a contiguous subsequence, wrap-around permitted. This naturally implies that if a power of 2 (call it $2^n$) is multiplied by the de Bruijn sequence, this is equivalent to shifting the de Bruijn sequence left by $\log_2 n$ bits. There are $2^k$ such shifts. By the properties of de Bruijn sequences, the top $k$ bits will then uniquely identify the power of 2 used to produce the shift (as any $k$-length substring only enters the de Bruijn sequence once). The only thing then left is to bring these bits into the least significant places (i.e., shift right by $2^k - k$ bits, which for 32-bit numbers yields a right-shift of $32 - 5 = 27$ bits). The table is compiled beforehand by considering all possible shifts of the de Bruijn sequence and the outputs thereof. Consider an illustrative example, where the de Bruijn sequence is taken to be 00011101, and a number to index is 00010000. Multiplication of the two numbers recovers the value 0000000111010000, which is brought back to 8 bits, reading 11010000. Retaining its 3 most significant bits (i.e., shifting right by $8 - 3 = 5$), we obtain 110, which is the binary representation of the index of 4 (as the original number read $2^4$). In other words, the sixth entry of the 3-bit table would read 4.
The astute reader, however, has by now pointed out that we do not seek to index a power of two; rather, we seek to index a value one less than a power of two! It turns out that some, but not all, de Bruijn sequences possess the useful property that senior bit uniqueness also holds for their multiplication by $2^n - 1$! 0x07C4ACDDU is one such sequence, which allows us to save a number of operations (i.e., we round to 1 less than a power of 2 instead of rounding to the next greater power of 2). This forms the basis for a cheap and efficient (floor of) log2 computation, which can be important in on-chain applications.

# 3 How a de Bruijn sequence shall be generated

The usefulness of the special de Bruijn sequences (which we shall call the de Bruijn-Dickinson sequences, after Mark Dickinson, who introduced the optimisation based on this observation into the original code due to Eric Cole) that preserve identifiability of both $2^n$ and $2^n - 1$ is undercut by the inconvenient generation thereof. In principle, one can just go over all de Bruijn sequences and test each in turn until a de Bruijn-Dickinson one is found. We, however, only had to test a small subset until a sequence for 7-bit logarithms revealed itself.

A primitive polynomial is a monic polynomial over a Galois field possessing a root that generates this field (except for the zero value). In other words, if a polynomial $F$ of degree $k$ over $GF(p^k)$ is primitive if it is monic and has a root $r$ such that $\{0, 1, r, r^2, \ldots, r^{p^k-1}\}$ is the entire field. Any such polynomial corresponds to a linear feedback shift register: a polynomial $c_0+c_1x+c_2x^2+\cdots+c_nx^n$ corresponds to a LFSR $c_na_1 \oplus c_{n-1}a_2 \oplus \cdots \oplus c_1a_n$ [cite Golomb]. Initialising $a_1a_2a_3 \ldots a_n$ to be $000 \ldots 01$, we apply the LFSR rule, outputting $a_1$ before each application, and prepend 0 to the resultant $2^k - 1$-length sequence to obtain a de Bruijn sequence. Generating and testing all order-7 primitive polynomia, we have discovered that the sequence 0x1FD533BA58DED6C91C2F95CD13C50C1 possesses the de Bruijn-Dickinson properties.

Unfortunately, none of the order-8 sequences obtained in such a way satisfy the requirement.

# 4 128-bit fast binary logarithm in Solidity

```
pragma solidity ^0.8.0;

contract Log2DeBruijn {
    uint128 constant deBruijnMagic = 0x1FD533BA58DED6C91C2F95CD13C50C1;
    uint8[128] deBruijnTable = [0,
 121,
 1,
 122,
 73,
 115,
 2,
 123,
 99,
 109,
 74,
 116,
 40,
 67,
 3,
 124,
 64,
 61,
 100,
 110,
 84,
 34,
 75,
 117,
 19,
 93,
```

```
41,
68,
23,
103,
4,
125,
113,
97,
65,
62,
32,
17,
101,
111,
15,
13,
85,
35,
53,
87,
76,
118,
37,
58,
20,
94,
50,
55,
42,
69,
89,
28,
24,
104,
45,
78,
5,
126,
120,
72,
114,
98,
108,
39,
66,
63,
60,
83,
33,
18,
92,
22,
102,
112,
96,
31,
16,
```

```
14,
12,
52,
86,
36,
57,
49,
54,
88,
27,
44,
77,
119,
71,
107,
38,
59,
82,
91,
21,
95,
30,
11,
51,
56,
48,
26,
43,
70,
106,
81,
90,
29,
10,
47,
25,
105,
80,
9,
46,
79,
8,
7,
6,
127];

   function log2(uint128 v) public view returns (uint128) {
       v |= v >> 1;
       v |= v >> 2;
       v |= v >> 4;
       v |= v >> 8;
       v |= v >> 16;
       v |= v >> 32;
       v |= v >> 64;
       uint256 r;
       unchecked {r = (deBruijnMagic * v)>> 121;}
       return deBruijnTable[r];
```

```
    }
}
```

Listing 2: 128-bit floor(log2)

# 5 Conclusion

We have considered the Cole-Dickinson efficient algorithm of taking the binary logarithm; implementation for 32-bit numbers is known, and we have found one for 64-bit numbers, but not for 128-bit numbers, which may well arise in, e.g., keeper stake accounting. We have considered LFSR-begotten de Bruijn sequences and found a de Bruijn-Dickinson sequence of length $2^7$, thereby obtaining the ability to implement an efficient binary logarithm for 128-bit numbers.