Improving the Integer Multiplicative Inverse (modulo 2^w)

Jeffrey Hurchalla
jeffrey@jeffhurchalla.com

May 31, 2021

Abstract

An algorithm is presented for the integer multiplicative inverse (mod 2^w) which completes in the fewest cycles known for modern microprocessors when using a native bit width w for the modulus 2^w. The algorithm is an adaptation of a method by Dumas, increasing generality and slightly improving efficiency. It is proven to be correct and shown to be closely related to the better known Newton's method algorithm for the inverse. Very simple direct formulas are also reviewed and proven (needed by Newton's method and this algorithm) for the integer inverse modulo 2^m with m = 1,2,3,4, or 5; this is likely the first proof of the preferred formula using m=4 or 5.

1 Introduction

The "integer inverse" of an odd integer d is the integer x that satisfies

$$dx \equiv 1 \left(mod\, 2^w\right) \tag{1}$$

where w is typically the bit width of arithmetic operations on a computer.

More precisely, this problem is the integer multiplicative inverse (modulo 2^w). It always has a unique solution for odd integers[1], and it has no solution for even integers.

There are a number of applications for the integer inverse: Montgomery multiplication [1], computing exact division by an integer constant [2], testing for a zero remainder after dividing by a constant [3], and integer division [4].

The integer inverse can be calculated in at least three different ways. The first and most well known method is the extended Euclidean algorithm [5]. The second method is inspired by Newton's method for approximating a reciprocal [6][10], and we will follow a convention of calling it Newton's method[2]. The third method is a variant on Newton's method discovered by Dumas [7]. We will extend Dumas' algorithm for generality and better efficiency, and provide a proof and show that it is a special case of Newton's method. Dumas' method is the most efficient known way to compute the inverse using native arithmetic instruction widths

1 The algorithm proof in this paper trivially shows by construction that a solution exists for any odd integer. As for uniqueness, the algorithm produces an integer $\boldsymbol{x}$ such that $\boldsymbol{dx} \equiv 1$ (mod 2^w), so assume a second integer $\boldsymbol{a}$ exists such that $\boldsymbol{da} \equiv 1$ (mod 2^w). Multiplying both sides by $\boldsymbol{x}$, we get $(\boldsymbol{dx})\boldsymbol{a} \equiv \boldsymbol{x}$ (mod 2^w) and thus $\boldsymbol{a} \equiv \boldsymbol{x}$ (mod 2^w).

2 More accurately it is Newton's method over p-adic numbers, or an application of Hensel lifting.

on current (i.e. pipelined and/or superscalar) computers [8]. This paper increases its advantages.

Literature on the integer inverse has usually focused on Newton's method. Granlund and Montgomery [9] first mentioned it, Warren [10] provided an easy to follow algorithm and proof for it, and academic papers [7][11][12] and blog entries [8][13][14] further explored it. The extended Euclidean algorithm is more well known than Newton's method for the integer inverse, but not as efficient. The extended Euclidean has order of complexity of $O(\log n)$ as compared to Newton's $O(\log\log n)$[3] , and the constant factor in the extended Euclidean's expected running time is larger due to the fact that it needs division operations.

We focus in this paper on the common case that the parameter w is equal to (or less than) the CPU native arithmetic bit width. We can note that this case is useful even when w is larger than the native bit width, since it allows us to calculate the lower bits of the final desired result [7]. For our case, Dumas' method requires essentially the same total number of operations (just additions and multiplications) as Newton's method. Dumas' method nevertheless has an important advantage that we detail in Section 6: it has two loop-carried instruction dependency chains, whereas Newton's method has only one. This allows a CPU in principle to use instruction level parallelism (via pipelining or superscalar execution) to execute both dependency chains in parallel, thus executing potentially twice as many operations per cycle as Newton's method.

2 Computing the Integer Inverse

We will present a generalized adaptation of Dumas' algorithm which improves its flexibility and efficiency.

Let d be an odd integer for which we want to find the integer inverse.
Let w be a positive integer. In practice we set w to the bit width of the arithmetic we will use on a computer; most commonly w is 32 or 64, for a 32bit or 64bit computer architecture.
Let m be a positive integer such that m divides w, and such that (w/m) is a power of 2. Common choices for m would be 1, 2, or 4 (m=4 usually is the most efficient choice). We can note in advance that doubling the size of m decreases the number of iterations required by the recurrence (4) by 1.

Let x_0 be an integer such that

$$dx_0 \equiv 1 \pmod{2^m} \qquad (2)$$

For example, if we are using m=1, then since d is odd, a value of x_0=1 would satisfy (2). See section 5 for formulas to find x_0 for different values of m.

Set the value

$$y \equiv 1 - dx_0 \pmod{2^w} \qquad (3)$$

3 Technically, its complexity is log log of the arithmetic instruction bit width.

For the sake of readability, we will use *pow(2, n)* to represent an integer 2^n. For example, $y^{pow(2,n)}$ would represent *y* raised to the power of (2^n).

Define the discrete recurrence relation

$$x_{n+1} \equiv x_n\left(1+y^{pow(2,n)}\right)\ \left(mod\, 2^w\right) \tag{4}$$

Set the integer *p* using

$$p=\log_2(w/m) \tag{5}$$

Perform *p* iterations of recurrence relation (4). *p* is the last required iteration of the recurrence, and x_p is the desired inverse of *d*, satisfying

$$dx_p \equiv 1\ \left(mod\, 2^w\right) \tag{6}$$

The following C function implements the algorithm, using *w*=32, *m*=4, and thus *p*=3:

```
uint32_t integer_inverse(uint32_t d)
{
   assert(d%2 == 1);
   uint32_t x0 = (3*d)^2;      // See section 5, formula 3.
   uint32_t y = 1 - d*x0;
   uint32_t x1 = x0*(1 + y);
   y *= y;
   uint32_t x2 = x1*(1 + y);
   y *= y;
   uint32_t x3 = x2*(1 + y);
   return x3;
}
```

Figure 1: C code for the 32 bit integer inverse.

Note that any unsigned computer arithmetic instruction with bit width >= *w* automatically gets performed modulo 2^w (including arbitrary precision arithmetic). For example, in C and C++, the data types uint32_t and uint64_t respectively ensure arithmetic modulo 2^{32} and 2^{64}.

3 Proof of Correctness

By rearranging (3),

$$dx_0 \equiv 1-y\ \left(mod\, 2^w\right) \tag{7}$$

Since 2^0 equals 1,

$$dx_0 \equiv 1-y^{pow(2,0)}\ \left(mod\, 2^w\right) \tag{8}$$

For the purpose of induction, assume

$$dx_n \equiv 1-y^{pow(2,n)}\ \left(mod\, 2^w\right) \tag{9}$$

Simply restating (4)

$$x_{n+1} \equiv x_n\left(1+y^{pow(2,n)}\right)\ \left(mod\, 2^w\right) \tag{10}$$

And multiplying both sides by d

$$dx_{n+1} \equiv dx_n\left(1+y^{pow(2,n)}\right)\ \left(mod\,2^w\right) \tag{11}$$

And using (9)

$$dx_{n+1} \equiv \left(1-y^{pow(2,n)}\right)\left(1+y^{pow(2,n)}\right)\ \left(mod\,2^w\right) \tag{12}$$

$$dx_{n+1} \equiv \left(1-y^{pow(2,n+1)}\right)\ \left(mod\,2^w\right) \tag{13}$$

Using (8), and assumption (9) and its consequence (13), we have proof by induction that for all n >= 0,

$$dx_n \equiv 1-y^{pow(2,n)}\ \left(mod\,2^w\right) \tag{14}$$

Substituting the congruence for y given by (3),

$$dx_n \equiv 1-\left(1-dx_0\right)^{pow(2,n)}\ \left(mod\,2^w\right) \tag{15}$$

By the definition of congruence, there exists some integer k such that (2) can be restated as

$$dx_0 = 1-k\,2^m \tag{16}$$

Substituting (16) into (15) we get

$$dx_n \equiv 1-\left(1-\left(1-k\,2^m\right)\right)^{pow(2,n)}\ \left(mod\,2^w\right) \tag{17}$$

$$dx_n \equiv 1-\left(k\,2^m\right)^{pow(2,n)}\ \left(mod\,2^w\right) \tag{18}$$

$$dx_n \equiv 1-k^{pow(2,n)}2^{m\ pow(2,n)}\ \left(mod\,2^w\right) \tag{19}$$

Since by (5) we know $p=\log_2(w/m)$,

$$dx_p \equiv 1-k^{pow(2,p)}2^{m\ pow(2,\log_2(w/m))}\ \left(mod\,2^w\right) \tag{20}$$

$$dx_p \equiv 1-k^{pow(2,p)}2^{m\,w/m}\ \left(mod\,2^w\right) \tag{21}$$

$$dx_p \equiv 1-k^{pow(2,p)}2^{w}\ \left(mod\,2^w\right) \tag{22}$$

$$dx_p \equiv 1\ \left(mod\,2^w\right) \tag{23}$$

This proves that x_p is the integer inverse of *d* (mod 2^w).

4 Relationship to the Integer Inverse via Newton's Method

We can show that the algorithm of Section 2 can be viewed as a special case of the better known integer inverse via Newton's Method [8].

First, we state again (4)

$$x_{n+1} \equiv x_n \left(1 + y^{pow(2,n)}\right) \ (mod\, 2^w) \tag{24}$$

$$x_{n+1} \equiv x_n \left(2 - \left(1 - y^{pow(2,n)}\right)\right) \ (mod\, 2^w) \tag{25}$$

Substituting in (14)

$$x_{n+1} \equiv x_n (2 - dx_n) \ (mod\, 2^w) \tag{26}$$

(26) is very similar to the integer inverse via Newton’s method, which is usually given as a recurrence with a simple assignment instead of congruence

$$x_{n+1} = x_n (2 - dx_n) \tag{27}$$

In practice, the arithmetic for Newton’s method is commonly carried out using computer instructions of bit width *w*, which in effect makes all arithmetic into a congruence mod 2^w. In such a case, (26) and (27) are equivalent. This algorithm provides an alternative, and more efficient, way to compute Newton’s method for that case.

5 Formulas to Set x_0, and Proofs

Simply restating (2), we wish to find an integer value x_0 such that

$$dx_0 \equiv 1 \ (mod\, 2^m) \tag{28}$$

Formula 1 (*m*=1)
For *m*=1, set $x_0=1$. This will always satisfy (28).
Proof:
Section 2 specifies that *d* is an odd integer. Therefore $d*1 \equiv 1 \ (mod\, 2^1)$, and by substitution, $dx_0 \equiv 1 (mod\, 2^m)$.

Formula 2 (*m*=2)
For *m*=2, set $x_0=d$. This will always satisfy (28).
Proof:
Let the integer $q=d/4$, and the integer $r=d\%4$. Thus

$$\begin{gathered} d = 4q + r \\ d \equiv r \ (mod\, 4) \\ d*d \equiv r*r \ (mod\, 4) \end{gathered} \tag{29}$$

Since *d* is odd, *r* belongs to the set {1, 3}. By testing every element of the set we see that

$$r*r \equiv 1 (mod\, 4) \tag{30}$$

And thus by (29)

$$d*d \equiv r*r \equiv 1 \ (mod\, 4) \tag{31}$$

Therefore, $d*d\equiv 1(mod\,2^2)$, and by substitution, $dx_0\equiv 1(mod\,2^m)$.

This same approach also works to show that for *m*=3, x_0=*d* always satisfies (28). We focus on *m*=2 because *m*=3 is rarely needed.

Formula 3 (*m*=4 and *w*>=8)
For *m*=4, set x_0=XOR((3*d*) % 2^w, 2). This will always satisfy (28). The formula is attributed to Peter Montgomery [4].
Proof:
Let the integer *q*=*d*/16, and the integer *r*=*d*%16. Thus

$$\begin{aligned} d&=16q+r \\ d&\equiv r\ \ (mod\,16) \end{aligned} \qquad (32)$$

Since *d* is odd, *r* belongs to the set {1, 3, 5, 7, 9, 11, 13, 15}. Since *w*>=8, we know that 3*r* <= 2^w, and thus 3*r* will not overflow when implemented. Testing every element of the set, we see that

$$r*XOR(3r,2)\equiv 1(mod\,16) \qquad (33)$$

And using (32) with (33),

$$d*XOR(3r,2)\equiv 1(mod\,16) \qquad (34)$$

We could stop here and use x_0=XOR(3*r*, 2), which would satisfy congruence (28). However, we will prove that we can use x_0=XOR((3*d*) % 2^w, 2), which requires one less instruction on a computer.

Observe that ((3*16*q*) % 2^w) has bit values of 0 for its first two bits, and thus ((3*16*q* + 3*r*) % 2^w) must have the same value in its second bit as 3*r* [Keep in mind that taking the remainder by 2^w does not change the second bit]. We'll need to explore two cases.

Case 1: assume the second bit of 3*r* is 0.
Then ((3*16*q* + 3*r*) % 2^w) also has a 0 for its second bit. And thus XORing it with 2 is equivalent to adding 2 to it. Likewise, since 3*r* has a zero in its second bit, adding 2 to 3*r* is equivalent to XORing 3*r* with 2. Using these facts, and the requirement that *w*>=8,

$$\begin{aligned} XOR((3d)\%2^w,\,2)&=XOR((3*(16q+r))\%2^w,\,2)=XOR((3*16q+3r)\%2^w,\,2) \\ &=(3*16q+3r)\%2^w+2 \\ XOR((3d)\%2^w,\,2)&\equiv(3*16q+3r)\%2^w+2\ \ (mod\,16) \\ &\equiv 3*16q+3r+2\ \ (mod\,16) \\ &\equiv 3r+2\ \ (mod\,16) \\ &\equiv XOR(3r,\,2)\ \ (mod\,16) \end{aligned} \qquad (35)$$

Case 2: assume the second bit of 3*r* is 1.
Then ((3*16*q* + 3*r*) % 2^w) also has a 1 for its second bit; and thus XORing it with 2 is equivalent to subtracting 2 from it. Likewise, since 3*r* has a 1 in the second bit, subtracting 2 from 3*r* is equivalent to XORing 3*r* with 2. Using these facts, and the requirement that *w*>=8,

$$\begin{aligned} XOR(3d \% 2^w, 2) &= XOR(3*(16q+r)\% 2^w, 2) = XOR((3*16q+3r)\% 2^w, 2) \\ &= (3*16q+3r)\% 2^w - 2 \\ XOR(3d \% 2^w, 2) &\equiv (3*16q+3r)\% 2^w - 2 \pmod{16} \\ &\equiv 3*16q+3r-2 \pmod{16} \\ &\equiv 3r-2 \pmod{16} \\ &\equiv XOR(3r, 2) \pmod{16} \end{aligned} \tag{36}$$

(35) and (36) show that in both cases,

$$XOR(3r, 2) \equiv XOR((3d)\% 2^w, 2) \pmod{16} \tag{37}$$

Substituting this in (34),

$$d * XOR((3d)\% 2^w, 2) \equiv 1 \pmod{16} \tag{38}$$

By substitution we have $d * x_0 \equiv 1 \pmod{2^m}$, which is congruence (28).

This same approach also works to show that for m=5, x_0=XOR(3d % 2^w, 2) always satisfies (28). We focus on m=4 because m=5 is rarely needed.

6 Efficiency

We will see that Dumas' algorithm (and thus this paper) possesses an instruction level parallelism advantage over Newton's method, due to two dependency chains that can execute mostly in parallel, as opposed to Newton's method's single dependency chain. We will also see that this paper slightly improves the efficiency of Dumas' algorithm.

We compare in this section the expected number of cycles to complete a 64 bit inverse (i.e. modulo 2^w with w = 64). We will use a performance model that assumes that an addition (or subtraction) requires 1 cycle to complete and that a multiplication requires 3 cycles to complete, and that multiplications are pipelined and that an addition can execute and issue in parallel with a multiplication. We also assume that the operation 3*d can be completed in one cycle (via add with shift, or LEA), and that an XOR operation requires one cycle. This model roughly corresponds to x86 CPUs from the last 12 years (see Agner Fogg's instruction tables for Intel Nehalem to Icelake [15]), and should also be an approximate model for most other modern CPUs.

The following C function implements Newton's method, for the 64 bit inverse:

```
uint64_t newtons_inverse(uint64_t d)
{
   assert(d%2 == 1);
   uint64_t x0 = (3*d)^2;      // See section 5, formula 3.
   uint64_t x1 = x0*(2 - d*x0);

   uint64_t x2 = x1*(2 - d*x1);
   uint64_t x3 = x2*(2 - d*x2);
```

```
    uint64_t x4 = x3*(2 - d*x3);
    return x4;
}
```

Figure 2: C code for the 64 bit integer inverse using Newton's method.

Following the example of Reynolds [8], let us consider the cycle timings of Figure 2 under the performance model we described. We will start on cycle 0 with 3*d. Operations are listed in order of the cycle on which they can issue. If more than one operation could issue on the same cycle, they would be listed on the same row, though we will see this never happens with Figure 2. Every instruction in Figure 2 depends upon its preceding instruction; there is only one single dependency chain throughout the Newton's method algorithm.

Cycle number	Operation(s)	
0.	tmp = 3*d	(normally a single cycle op)
1.	x0 = tmp ^ 2	(^ is xor)
2.	tmp = d*x0	
3.		(no instruction possible, waiting on tmp)
4.		(no instruction possible, waiting on tmp)
5.	tmp = 2 - tmp	
6.	x1 = x0 * tmp	
7.		
8.		
9.	tmp = d*x1	
10.		
11.		
12.	tmp = 2 – tmp	
13.	x2 = x1 * tmp	
14.		
15.		
16.	tmp = d*x2	
17.		
18.		
19.	tmp = 2 – tmp	
20.	x3 = x2 * tmp	
21.		
22.		
23.	tmp = d*x3	
24.		
25.		
26.	tmp = 2 – tmp	
27.	x4 = x5 * tmp	
28.		
29.		
30.	return x4	

The following C function implements Dumas' original algorithm 3 in a reworked form, for the 64 bit inverse:

```
// reworked version of Dumas' algorithm 3 from [7]
uint64_t original_dumas_inverse(uint64_t d)
{
    assert(d%2 == 1);
```

```c
    uint64_t y = d – 1;
    uint64_t u0 = 2 – d;
    y *= y;
    uint64_t u1 = u0*(1 + y);
    y *= y;
    uint64_t u2 = u1*(1 + y);
    y *= y;
    uint64_t u3 = u2*(1 + y);
    y *= y;
    uint64_t u4 = u3*(1 + y);
    y *= y;
    uint64_t u5 = u4*(1 + y);
    return u5;
}
```

Figure 3: C code for the 64 bit integer inverse using Dumas' algorithm 3.

Let us consider the cycle timings for the implementation of Figure 3, starting on cycle 0 with its subtraction d – 1. Operations are listed in order of the cycle on which they can issue, and when more than one operation can issue on the same cycle, they are listed on the same row:

Cycle number	Operation(s)	
0.	y = d - 1	
1.	u0 = 2 - d,	y *= y
2.	(no instruction possible, waiting on y)	
3.	(no instruction possible, waiting on y)	
4.	tmp = (1 + y),	y *= y
5.	u1 = u0 * tmp,	
6.		
7.	tmp = (1 + y),	y *= y
8.	u2 = u1 * tmp	
9.		
10.	tmp = (1 + y),	y *= y
11.	u3 = u2 * tmp	
12.		
13.	tmp = (1 + y),	y *= y
14.	u4 = u3 * tmp	
15.		
16.	tmp = (1 + y)	
17.	u5 = u4 * tmp	
18.		
19.		
20.	return u5	

We can see that there are two dependency chains (one for *y* and one for *u*) that can execute mostly in parallel. The result is 20 cycles to completion as opposed to Newton's 30 cycles.

The following C function implements the algorithm presented in this paper, for the 64 bit inverse (using *w*=64, *m*=4, and thus *p*=4):

```c
uint64_t generalized_dumas_inverse(uint64_t d)
{
   assert(d%2 == 1);
   uint64_t x0 = (3*d)^2;      // See section 5, formula 3.
```

```
    uint64_t y = 1 - d*x0;
    uint64_t x1 = x0*(1 + y);
    y *= y;
    uint64_t x2 = x1*(1 + y);
    y *= y;
    uint64_t x3 = x2*(1 + y);
    y *= y;
    uint64_t x4 = x3*(1 + y);
    return x4;
}
```
Figure 4: C code for the 64 bit integer inverse using this paper

Let us consider the cycle timings for the implementation of Figure 4:

```
Cycle number      Operation(s)
0.                tmp = 3*d        (normally a single cycle op)
1.                x0 = tmp ^ 2     (^ is xor)
2.                tmp = d*x0
3.
4.
5.                y = 1 - tmp
6.                tmp = (1 + y),       y *= y
7.                x1 = x0 * tmp
8.
9.                tmp = (1 + y),       y *= y
10.               x2 = x1 * tmp
11.
12.               tmp = (1 + y),       y *= y
13.               x3 = x2 * tmp
14.
15.               tmp = (1 + y)
16.               x4 = x3 * tmp
17.
18.
19.               return x4
```

We can see that the generalized form of Dumas' algorithm in Figure 4 allows us to reduce the total number of cycles to completion by 1, due to its ability to use an efficient formula from section 5 (with $m = 4$). Given the modeled change from 20 to 19 cycles, we might expect roughly a 5% improvement in measured performance of the 64 bit inverse on modern CPUs, compared to the original algorithm. Comparing also the cycle count to Figure 2, we might expect roughly a 60% improvement in measured performance over Newton's method.

7 Montgomery Arithmetic

The integer inverse has one of its most common applications in Montgomery arithmetic. The Montgomery REDC [1] traditionally requires the negative inverse rather than the positive inverse that we calculate here. It is simple to obtain the negative inverse if needed, by performing a negation of the inverse modulo 2^w. Or this algorithm could be modified to directly produce the negative inverse in the same number of steps as the plain inverse (left as an exercise for the reader). However, if it is practical to do so, a better recommendation is to use the Alternate Montgomery REDC [16] instead of the traditional REDC. The alternate version uses the plain inverse produced by this algorithm, and it is also more efficient than the traditional Montgomery REDC.

8 Conclusion

Dumas' algorithm for the inverse has a strong advantage over Newton's method due to its ability to exploit instruction level parallelism (from pipelining and/or superscalar execution units) available in modern microprocessors. The algorithm in this paper is an adaptation of Dumas' work that retains this advantage while allowing for a more efficient starting value for its recurrence sequence, typically using the formula from Section 5 to set x_0 using m=4. The result is the most efficient method known for determining the inverse of a number modulo 2^w, when w is less than or equal to the bit width of a CPU's arithmetic instructions.

References

[1] Wikipedia contributors. Montgomery modular multiplication. *Wikipedia, The Free Encyclopedia*. Revised December 17, 2019. https://en.wikipedia.org/w/index.php?title=Montgomery_modular_multiplication&oldid=931195404
[2] Henry Warren. *Hacker's Delight 2nd Ed*. [Section 10.16], Addison-Wesley, 2013.
[3] Henry Warren. *Hacker's Delight 2nd Ed*. [Section 10.17], Addison-Wesley, 2013.
[4] Ernst W. Mayer. Efficient long division via Montgomery multiply. arXiv:1303.0328v6, 21 August 2016
[5] Wikipedia contributors. Extended Euclidean Algorithm. *Wikipedia, The Free Encyclopedia*. Revised January 29, 2021. https://en.wikipedia.org/w/index.php?title=Extended_Euclidean_algorithm&oldid=1003613686
[6] Wikipedia contributors. Newton's Method. *Wikipedia, The Free Encyclopedia*. Revised January 25, 2020. https://en.wikipedia.org/w/index.php?title=Newton%27s_method&oldid=937542756#Multiplicative_inverses_of_numbers_and_power_series
[7] Jean-Guillaume Dumas. On Newton-Raphson iteration for multiplicative inverses modulo prime powers. *IEEE Transactions on Computers*, August 2014. *arXiv:1209.6626v5, 22 April 2019*
[8] Marc Reynolds. Integer multiplicative inverse via Newton's method. 18 September 2017, https://marc-b-reynolds.github.io/math/2017/09/18/ModInverse.html
[9] Torbjörn Granlund and Peter Montgomery. Division by Invariant Integers using Multiplication. *ACM SIGPLAN Notices*, Vol. 29, No. 6, June 1994.
[10] Henry Warren. *Hacker's Delight (1st Ed*.), [Section 10.15], Addison-Wesley, 2002.
[11] Ortal Arazi and Hairong Qi. On Calculating Multiplicative Inverses Modulo 2^m. *IEEE Transactions on Computers*, Vol. 57, Issue 10, October 2008.

[12] Çetin Kaya Koç. A New Algorithm for Inversion mod p^k. *IACR Cryptology ePrint Archive*, 28 June 2017

[13] Daniel Lemire. Computing the inverse of odd integers. *Daniel Lemire's Blog*, 18 September 2017, https://lemire.me/blog/2017/09/18/computing-the-inverse-of-odd-integers/

[14] Quentin Carbonneaux. "2019-01-15." *Notes* 15 January 2019, https://c9x.me/notes/2019-01-15.html

[15] Agner Fogg. Instruction Tables *Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD, and VIA CPUs*. Revised 2021-03-22. https://www.agner.org/optimize/instruction_tables.pdf

[16] Jeffrey Hurchalla. Montgomery REDC using the positive inverse (mod R). 28 December 2020, https://github.com/hurchalla/modular_arithmetic/blob/master/montgomery_arithmetic/include/hurchalla/montgomery_arithmetic/low_level_api/detail/platform_specific/README_REDC.md