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 extension 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^k with k = 1,2,3,4, or 5, possibly providing the first proof of the preferred formula with k=4 or 5.

1 Introduction

The integer multiplicative inverse (modulo 2^w) of an odd integer $\boldsymbol{a}$ is the integer $\boldsymbol{x}$ that satisfies

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

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

We refer to this problem as the "integer inverse" . It always has a unique solution (mod 2^w) for odd integers, and it has no solution for even integers.

Some of the applications for the integer inverse include Montgomery multiplication [1][16], 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 by convention we call "Newton's method"[1] [10]; it is inspired by Newton's method for approximating a reciprocal [6]. The third method is a variant on Newton's method discovered by Dumas [7], and expanded upon in this paper.

As background, Newton's method is almost always more efficient than the extended Euclidean algorithm. For computing the inverse modulo 2^w with a CPU's native bit width w,

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

Newton's method has order of complexity $O(\log w)$, as compared to $O(w)$ for the extended Euclidean; also Newton's method has a better constant factor in its expected running time since it needs no divisions. Granlund and Montgomery [9] first mentioned Newton's method for the integer inverse, 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.

For a native bit width w, Dumas' algorithm has nearly the same number and type of operations as Newton's method. In practice though, Dumas' algorithm is more efficient on a modern CPU because the CPU can apply instruction level parallelism when executing it [8], which is not possible with Newton's method. Section 6 explores the details. The algorithm in this paper provides a version of Dumas' method with further advantages.

2 Computing the Integer Inverse

We present a generalized adaptation of Dumas' algorithm that increases its flexibility and efficiency.

Let $\boldsymbol{a}$ be an odd integer for which we wish to find the integer inverse.
Let $\boldsymbol{w}$ 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 $\boldsymbol{k}$ be a positive integer such that k divides w, and such that (w/k) is a power of 2. Common choices for k would be 1, 2, or 4 (k=4 usually is the most efficient choice). We can note in advance that doubling the size of k decreases the number of iterations required by the recurrence (4) by 1.

Set $\boldsymbol{x_0}$ to any integer that satisfies

$$ax_0 \equiv 1 \pmod{2^k} \tag{2}$$

For example, if we are using k=1, then since $\boldsymbol{a}$ is odd, x_0=1 would satisfy (2). See section 5 for formulas for x_0 using other values of k.

Set $\boldsymbol{y}$ to any integer that satisfies

$$y \equiv 1 - ax_0 \pmod{2^w} \tag{3}$$

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

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

Set $\boldsymbol{p}$ to the integer

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

Perform p iterations of recurrence relation (4).
The resulting $\boldsymbol{x_p}$ is the desired inverse of $\boldsymbol{a}$, satisfying

$$ax_p \equiv 1 \pmod{2^w} \tag{6}$$

The following C function implements the algorithm, using w=64, k=4, and thus p=4:

```
1.  uint64_t integer_inverse(uint64_t a)
2.  {
3.     assert(a%2 == 1);
4.     uint64_t x0 = (3*a)^2;      // See section 5, formula 3.
5.     uint64_t y = 1 - a*x0;
6.     uint64_t x1 = x0*(1 + y);
7.     y *= y;
8.     uint64_t x2 = x1*(1 + y);
9.     y *= y;
10.    uint64_t x3 = x2*(1 + y);
11.    y *= y;
12.    uint64_t x4 = x3*(1 + y);
13.    return x4;
14.  }
```

Figure 1: C code for the 64 bit integer inverse.

Note that any unsigned computer arithmetic instruction that has bit width w automatically gets performed modulo 2^w. 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

Rearranging (3),

$$ax_0 \equiv 1 - y \pmod{2^w} \tag{7}$$

Since 2^0 equals 1,

$$ax_0 \equiv 1 - y^{pow(2,0)} \pmod{2^w} \tag{8}$$

Temporarily assume

$$ax_n \equiv 1 - y^{pow(2,n)} \pmod{2^w} \tag{9}$$

Multiply both sides of (4) by a,

$$ax_{n+1} \equiv ax_n\left(1 + y^{pow(2,n)}\right) \pmod{2^w} \tag{10}$$

Substitute (9) for ax_n,

$$ax_{n+1} \equiv \left(1 - y^{pow(2,n)}\right)\left(1 + y^{pow(2,n)}\right) \pmod{2^w} \tag{11}$$

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

(8), (9) and (12) prove by induction that for all n >= 0,

$$ax_n \equiv 1 - y^{pow(2,n)} \pmod{2^w} \tag{13}$$

Substitute (3) for *y*,

$$ax_n \equiv 1 - (1 - ax_0)^{pow(2,n)} \pmod{2^w} \tag{14}$$

By definition of congruence, there exists an integer *q* such that we can restate (2) to be

$$ax_0 = 1 - q2^k \tag{15}$$

Substitute (15) into (14),

$$ax_n \equiv 1 - (1 - (1 - q2^k))^{pow(2,n)} \pmod{2^w} \tag{16}$$

$$ax_n \equiv 1 - q^{pow(2,n)} 2^{k\ pow(2,n)} \pmod{2^w} \tag{17}$$

Since (5) sets $p = \log_2(w/k)$,

$$ax_p \equiv 1 - q^{pow(2,p)} 2^{k\ pow(2,\log_2(w/k))} \pmod{2^w} \tag{18}$$

$$ax_p \equiv 1 - q^{pow(2,p)} 2^w \pmod{2^w} \tag{19}$$

$$ax_p \equiv 1 \pmod{2^w} \tag{20}$$

Thus $\boldsymbol{x_p}$ is the integer inverse of $\boldsymbol{a}$ (mod $2^{\mathbf{w}}$).

4 Relationship to Newton's Method

We can show that the algorithm of Section 2 is closely related to Newton's method. This is only a matter of curiosity; we will see in Section 6 that our algorithm is more efficient in practice than Newton's method.

The recurrence for the integer inverse via Newton's method [10] is given as

$$x_{n+1} = x_n(2 - ax_n) \tag{21}$$

Taking this modulo 2^w,

$$x_{n+1} \equiv x_n(2 - ax_n) \pmod{2^w} \tag{22}$$

Using the definition of *y* from (3), and (22), it can be proven by induction (left as an exercise for the reader) that for all n >= 0,

$$ax_n \equiv 1 - y^{pow(2,n)} \pmod{2^w} \tag{23}$$

Substituting (23) into (22),

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

(24) is recurrence (4) from Section 2. Thus for all *n*, the recurrence of Newton's method satisfies Section 2's congruence recurrence (4).

5 Formulas to Set x_0, and Proofs

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

$$ax_0 \equiv 1 \pmod{2^k} \tag{25}$$

Formula 1 (*k*=1)
For *k*=1, set x_0=1. This will always satisfy (25).
Proof:
Section 2 specifies that ***a*** is an odd integer. Therefore $a*1 \equiv 1 \pmod{2^1}$, and by substitution, $ax_0 \equiv 1 \pmod{2^k}$.

Formula 2 (*k*=2)
For *k*=2, set x_0=*a*.
Proof:
Let the integer *q*=*a*/4, and the integer *r*=*a*%4. Thus

$$\begin{aligned} a &= 4q + r \\ a &\equiv r \pmod 4 \\ a*a &\equiv r*r \pmod 4 \end{aligned} \tag{26}$$

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

$$r*r \equiv 1 \pmod 4 \tag{27}$$

And thus by (26)

$$a*a \equiv r*r \equiv 1 \pmod 4 \tag{28}$$

Thus $a*a \equiv 1 \pmod{2^2}$, and by substitution, $ax_0 \equiv 1 \pmod{2^k}$.

This same approach also works to show that for *k*=3, x_0=*a* always satisfies (25). We focus on *k*=2 because *k*=3 is rarely needed.

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

$$\begin{aligned} a &= 16q + r \\ a &\equiv r \pmod{16} \end{aligned} \tag{29}$$

Since *a* 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. Testing every element of the set, we see that

$$r*XOR(3r, 2) \equiv 1 \pmod{16} \tag{30}$$

And using (29) with (30),

$$a * XOR(3r, 2) \equiv 1 \pmod{16} \tag{31}$$

We could stop here and use x_0=XOR(3*r*, 2), which would satisfy congruence (25). However, we will prove we can use x_0=XOR((3*a*) % 2^w, 2), which needs one less computer instruction.

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,

$$\begin{aligned} XOR((3a)\%2^w, 2) &= XOR((3*(16q+r))\%2^w, 2) = XOR((3*16q+3r)\%2^w, 2) \\ &= (3*16q+3r)\%2^w+2 \\ XOR((3a)\%2^w, 2) &\equiv (3*16q+3r)\%2^w+2 \pmod{16} \\ &\equiv 3*16q+3r+2 \pmod{16} \\ &\equiv 3r+2 \pmod{16} \\ &\equiv XOR(3r, 2) \pmod{16} \end{aligned} \tag{32}$$

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,

$$\begin{aligned} XOR(3a\%2^w, 2) &= XOR(3*(16q+r)\%2^w, 2) = XOR((3*16q+3r)\%2^w, 2) \\ &= (3*16q+3r)\%2^w-2 \\ XOR(3a\%2^w, 2) &\equiv (3*16q+3r)\%2^w-2 \pmod{16} \\ &\equiv 3*16q+3r-2 \pmod{16} \\ &\equiv 3r-2 \pmod{16} \\ &\equiv XOR(3r, 2) \pmod{16} \end{aligned} \tag{33}$$

(32) and (33) show that in both cases,

$$XOR(3r, 2) \equiv XOR((3a)\%2^w, 2) \pmod{16} \tag{34}$$

Substituting this in (31),

$$d * XOR((3a)\%2^w, 2) \equiv 1 \pmod{16} \tag{35}$$

By substitution we have $a * x_0 \equiv 1 \pmod{2^k}$, which is congruence (25).
This same approach also works to show that for *k*=5, x_0=XOR((3*a)* % 2^w, 2) always satisfies (25). We focus on *k*=4 because *k*=5 is rarely needed.

6 Efficiency

We compare in this section the expected number of CPU cycles to complete a 64 bit inverse (i.e. modulo 2^w with $w = 64$), based on a performance model. For this performance model, we specify 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 issue and execute in parallel with a multiplication. This model also specifies that the operation 3**a* can be completed in one cycle (via an add with shift instruction, or the x86 LEA instruction), and that an XOR operation requires one cycle. This model roughly corresponds to x86 CPUs from the last 12 years (see Agner Fog's instruction tables for Intel Nehalem to Icelake [15]), and it reflects common capabilities of modern CPUs.

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

```
15.   uint64_t newtons_inverse(uint64_t a)
16.   {
17.      assert(a%2 == 1);
18.      uint64_t x0 = (3*a)^2;      // See section 5, formula 3.
19.      uint64_t x1 = x0*(2 - a*x0);

20.      uint64_t x2 = x1*(2 - a*x1);
21.      uint64_t x3 = x2*(2 - a*x2);
22.      uint64_t x4 = x3*(2 - a*x3);
23.      return x4;
24.  }
```

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**a*. 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. Since every instruction in Figure 2 depends upon its preceding instruction, there is one long dependency chain throughout the Newton's method algorithm:

```
Cycle number      Operation(s)
0.                tmp = 3*a       (normally a single cycle op)
1.                x0 = tmp ^ 2    (^ is xor)
2.                tmp = a*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 = a*x1
10.
11.
12.               tmp = 2 – tmp
13.               x2 = x1 * tmp
```

```
14.
15.
16.               tmp = a*x2
17.
18.
19.               tmp = 2 – tmp
20.               x3 = x2 * tmp
21.
22.
23.               tmp = a*x3
24.
25.
26.               tmp = 2 – tmp
27.               x4 = x5 * tmp
28.
29.
30.               return x4
```

The following C function implements Dumas' algorithm 3 without looping, for the 64 bit inverse:

```
25.   // unlooped Dumas' algorithm 3 from [7]
26.   uint64_t original_dumas_inverse(uint64_t a)
27.   {
28.      assert(a%2 == 1);
29.      uint64_t y = a - 1;
30.      uint64_t u0 = 2 - a;
31.      y *= y;
32.      uint64_t u1 = u0*(1 + y);
33.      y *= y;
34.      uint64_t u2 = u1*(1 + y);
35.      y *= y;
36.      uint64_t u3 = u2*(1 + y);
37.      y *= y;
38.      uint64_t u4 = u3*(1 + y);
39.      y *= y;
40.      uint64_t u5 = u4*(1 + y);
41.      return u5;
42.  }
```

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

Let us consider the cycle timings for Figure 3, starting on cycle 0 with its subtraction a - 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 = a – 1
1.                u0 = 2 – a,          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 for Dumas, as compared to 30 cycles for Newton.

Finally, let us consider the cycle timings for the earlier Figure 1, which implements the 64 bit inverse using the algorithm presented in this paper. This algorithm is a more flexible form of Dumas' algorithm:

```
Cycle number      Operation(s)
0.                tmp = 3*a         (normally a single cycle op)
1.                x0 = tmp ^ 2      (^ is xor)
2.                tmp = a*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
```

Compared to Figure 3's timings, Figure 1 completes one cycle faster, mostly due to its ability to use the efficient formula 3 from section 5. Under the performance model, Figure 1 improves performance by ~5% over Dumas' original algorithm from Figure 3 (with 19 cycles vs 20 cycles).

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 k=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 native bit width of a CPU's arithmetic instructions. This method can be useful even when w is larger than the native bit width, since it also provides the most efficient way to calculate the initial bits of a larger desired inverse [7].

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 Fog. 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