An Improved Integer Multiplicative Inverse (modulo 2^w)

Jeffrey Hurchalla
jeffrey@jeffhurchalla.com

May 31, 2021

Abstract

An algorithm is presented for the integer multiplicative inverse which requires the fewest cycles known on modern microprocessors for native bit widths. 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. Formulas and proofs for integer inverse sequence starting values are also presented, some of which had been part of informal lore previously but unproven.

1 Introduction

The "integer inverse"[1] of an odd integer d is the integer x that satisfies

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

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

This problem always has a unique solution for odd integers[2], and it has no solution for even integers.

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

The integer inverse could 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[3]. 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 More precisely, the "integer multiplicative inverse (modulo 2^w)"

2 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).

3 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.

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. Academic papers [7][11] [12] and blog entries [8][13][14] further detail the inverse by Newton's method. Newton's method has two significant advantages over the better known extended Euclidean algorithm for the inverse. First, it has an order of complexity of roughly[4] $O(\log \log n)$ as opposed to $O(\log n)$ for the extended Euclidean algorithm. Second, it requires only two multiplications and one addition per iteration, as opposed to the extended Euclidean algorithm which requires a division, two multiplications, and two additions per iteration. Dumas' method retains the same order of complexity advantage as Newton's method, and has essentially the same operation count. However, the operations in its loop are split into two separate computational dependency chains, which almost doubles performance compared to Newton's method on modern CPUs [8] so long as the parameter w is equal to or less than the CPU's native arithmetic instruction bit width. For use cases larger than the CPU native instruction width (e.g. multi-precision arithmetic), it is most efficient to use this method to obtain only the first b bits of the inverse up to the instruction bit width, and then use a second method to determine all remaining bits past b [7].

2 Computing the Integer Inverse

We will present a generalized adaptation of Dumas' algorithm, improving 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

$$dx_0 \equiv 1 \pmod{2^m} \tag{2}$$

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

$$y \equiv 1 - dx_0 \pmod{2^w} \tag{3}$$

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).

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

Define the discrete recurrence relation

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

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

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

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

```
1.  uint32_t integer_inverse(uint32_t d)
2.  {
3.     assert(d%2 == 1);
4.     uint32_t x0 = (3*d)^2;      // See section 4 for this x0 formula
5.     uint32_t y = 1 - d*x0;
6.     uint32_t x1 = x0*(1 + y);
7.     y *= y;
8.     uint32_t x2 = x1*(1 + y);
9.     y *= y;
10.    uint32_t x3 = x2*(1 + y);
11.    return x3;
12.  }
```

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),

$$dx_0 \equiv 1 - y \pmod{2^w} \tag{7}$$

Since 2^0 equals 1,

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

For the purpose of induction, assume

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

Simply restating (4)

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

And multiplying both sides by d

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

And using (9)

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

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

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

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

Substituting the congruence for y given by (3),

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

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

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

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

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

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

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

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

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

$$dx_p \equiv 1 \pmod{2^w} \tag{23}$$

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)

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

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

Substituting in (14)

$$x_{n+1} \equiv x_n \left(2 - dx_n\right) \pmod{2^w} \tag{26}$$

(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

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

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

$$dx_0 \equiv 1 \pmod{2^m} \tag{28}$$

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 $d * 1 \equiv 1 \pmod{2^1}$, and by substitution, $dx_0 \equiv 1 \pmod{2^m}$.

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

$$\begin{aligned} d &= 4q + r \\ d &\equiv r \pmod{4} \\ d * d &\equiv r * r \pmod{4} \end{aligned} \tag{29}$$

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

$$r * r \equiv 1 \pmod{4} \tag{30}$$

And thus by (29)

$$d * d \equiv r * r \equiv 1 \pmod{4} \tag{31}$$

Therefore, $d*d \equiv 1 \pmod{2^2}$, and by substitution, $dx_0 \equiv 1 \pmod{2^m}$.

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((3d) % 2^w, 2). This will always satisfy (28). The formula is attributed to Peter Montgomery [2].
Proof:
Let the integer q=d/16, and the integer r=d%16. Thus

$$\begin{aligned} d &= 16q + r \\ d &\equiv r \pmod{16} \end{aligned} \tag{32}$$

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

$$r * XOR(3r, 2) \equiv 1 \pmod{16} \tag{33}$$

And using (32) with (33),

$$d * XOR(3r, 2) \equiv 1 \pmod{16} \tag{34}$$

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

Observe that ((3*16q) % 2^w) has bit values of 0 for its first two bits, and thus ((3*16q + 3r) % 2^w) must have the same value in its second bit as 3r [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 3r is 0.
Then ((3*16q + 3r) % 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 3r has a zero in its second bit, adding 2 to 3r is equivalent to XORing 3r with 2. Using these facts, and the requirement that w>=8,

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

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

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

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

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

Substituting this in (34),

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

By substitution we have $d * x_0 \equiv 1 (mod\,2^m)$, 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 Montgomery Arithmetic

One of the more common applications for the inverse is Montgomery arithmetic, which traditionally requires the negative inverse. We can use a better choice than the original, traditional Montgomery method though. The alternate Montgomery REDC [15] is inherently more efficient than the traditional Montgomery REDC, and it uses the plain inverse produced by this algorithm. If we still need the negative inverse for any reason, we can obtain it via a modular negation of the inverse, or by modifying the inverse algorithm here to directly produce the negative inverse in the same number of steps as the plain inverse (left as an exercise for the reader).

7 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 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] Ernst W. Mayer. Efficient long division via Montgomery multiply. arXiv:1303.0328v6, 21 August 2016

[3] Henry Warren. *Hacker's Delight 2nd Ed*. [Section 10.16], Addison-Wesley, 2013.

[4] Henry Warren. *Hacker's Delight 2nd Ed*. [Section 10.17], Addison-Wesley, 2013.

[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] 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