Brief Contributions

On Calculating Multiplicative Inverses Modulo 2^m

Ortal Arazi, *Member*, *IEEE*, and
Hairong Qi, *Senior Member*, *IEEE*

Abstract—This paper presents a procedure for calculating multiplicative inverses modulo 2^m, based on a novel mathematical approach. The procedure is suitable for software implementation on a general-purpose processor. When counting the total number of word-level processor multiplications, the computational effort involved in calculating a multiplicative inverse is 2/3 that of a single multiplication of m-bit values, in addition to a few word-level multiplications. For standard processor word sizes, the number of these additional multiplications does not exceed 12. This introduces a clear advantage of the proposed method when compared to other known methods presented in the literature.

Index Terms—Modular arithmetic, modular multiplication, modular multiplicative inverse.

1 Introduction

THE multiplicative inverse modulo 2^m of an odd value b, denoted by $b^{-1} \bmod 2^m$, is a value r, where $b \cdot r = 1 \bmod 2^m$. Without loss of generality, it can be assumed that $b < 2^m$ (i.e., b is an m-bit value) since, otherwise, b is first taken as $\bmod 2^m$ (i.e., only the least m significant bits in the binary representation of b are selected) before proceeding with the calculation of r. The binary representation of r also consists of m bits.

Calculating a multiplicative inverse modulo 2^m is of a fundamental mathematical nature. The need to perform such a calculation arises in applications like Montgomery modular multiplication [1] and in exact division [2]. In this paper, we present a procedure for calculating a multiplicative inverse modulo 2^m using a novel mathematical approach that is not an extension or modification of known approaches [3], [4], [5], [6], [7].

The following are several known or straightforward algorithms for calculating $b^{-1} \bmod 2^m$. The first is taken from Dusse and Kaliski [3].

Algorithm 1. Dusse and Kaliski's method for calculating $r = b^{-1} \bmod 2^m$.

$y_1 = 1$;
for $i = 2$ *to* m **do**
  if $b \cdot y_{i-1} < 2^{i-1}$ *mod* 2^i **then**
    $y_i = y_{i-1}$;
  else
    $y_i = y_{i-1} + 2^{i-1}$;
  end
end
The value of y_m is the desired result $r = b^{-1} \bmod 2^m$;

• *The authors are with the Department of Electrical and Engineering and Computer Science, University of Tennessee, Knoxville, TN 37996-2100. E-mail: ortal@ieee.org, hqi@utk.edu.*

Manuscript received 7 Oct. 2006; revised 23 July 2007; accepted 12 Feb. 2008 published online 27 Mar. 2008.
Recommended for acceptance by Ç.K. Koç.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-0396-1006.
Digital Object Identifier no. 10.1109/TC.2008.54.

Algorithm 2 is a straightforward method of obtaining the same result. This algorithm is always feasible to implement since b is odd, having 1 as the least significant bit (LSB). Hence, by adding selected left shifts of b to an accumulated sum that starts with b, one can always generate a value whose m LSBs are of any form, including $000\ldots01$. In other words, Algorithm 2 is executed by "sliding" b leftward across an accumulated sum such that the LSB of b generates the bits of the given product, one at a time, from right to left.

Algorithm 2. Calculating $r = b^{-1} \bmod 2^m$ by controlled additions of left shifts of b.

Since $b \cdot r = 1 \bmod 2^m$, the m least significant bits (LSBs) of the complete product $b \cdot r$ are of the form $000\ldots01$. To obtain r, one should therefore add selected left shifts of the binary representation of b such that the addition of the shifts generates the m LSBs $000\ldots01$. The coefficients of the added left shifts form the binary representation of r. (We form here a standard shift-and-add multiplication, where the multiplicand b is known, and the m LSBs of the product are given. From this, we recover the multiplier r, bit by bit, starting from the LSB.)

Calculating $b^{-1} \bmod 2^m$ can be based on initially computing $2^{-m} \bmod b$ and then recovering $b^{-1} \bmod 2^m$ by one division of a $2m$-bit value by an m-bit value, as shown by Algorithm 3a followed by Algorithm 3b below.

Algorithm 3. A two-step procedure for calculating $r = b^{-1} \bmod 2^m$.

Algorithm 3a. Calculating $2^{-m} \bmod b$ by m successive divisions of $2 \bmod b$.

$d = 1$;
for $i = 1$ *to* m **do**
  if d *is odd* **then**
    $d = d + b$;
  end
  $d = d/2$;
end
The final value of d is, of course, $2^{-m} \bmod b$;

Algorithm 3b. Recovering $r = b^{-1} \bmod 2^m$ out of $s = 2^{-m} \bmod b$.

The desired $r = b^{-1} \bmod 2^m$ is recovered out of $s = 2^{-m} \bmod b$ as follows:
$t = s \cdot 2^m$;
$u = (t - 1)/b$;
$r = 2^m - u$;

To realize why Algorithm 3b is valid, note that the relation $s = 2^{-m} \bmod b$ means that $t = s \cdot 2^m = u \cdot b + 1$ and the value u, calculated in the second step of Algorithm 3b, is therefore an integer. That is, $u \cdot b = s \cdot 2^m - 1$. Taking both sides of the latter relation $\bmod 2^m$ yields the congruence $u \cdot b = -1 \bmod 2^m$. That is, $(-u) \cdot b = 1 \bmod 2^m$ and, therefore, $-u = r = b^{-1} \bmod 2^m$. However, $-u = (2^m - u) \bmod 2^m$, which completes the validity proof for Algorithm 3b.

Another method of calculating modular multiplicative inverses is *The Extended Euclid Algorithm* [4]. It is important to note that this algorithm treats odd moduli. Therefore, when computing $r = b^{-1} \bmod 2^m$, we first calculate the multiplicative inverse of 2^m modulo the odd b and then exchange the role of the two values 2^m

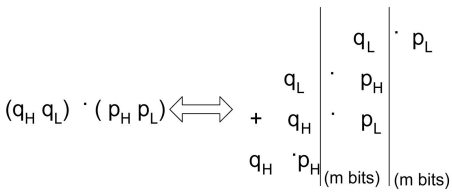

Fig. 1. The format of $(p_H \cdot q_L + p_L \cdot q_H) \cdot 2^m + p_L \cdot q_L$.

and b. Algorithm 3a is precisely the Extended Euclid Algorithm when used in calculating $2^{-m} \bmod b$, while Algorithm 3b is the procedure of exchanging 2^m and b. The methods of calculating modular multiplicative inverses further include the *Montgomery Inverse Algorithm* [5], [6]. This algorithm, which treats an odd modulus, consists of two phases, of which the second is identical to Algorithm 3a. Since the modulus is odd, calculating $r = b^{-1} \bmod 2^m$ further necessitates the procedure of Algorithm 3b. Other methods presented in the literature for calculating $b^{-1} \bmod 2^m$ include lookup table techniques such as that proposed in [7], which are inherently nonalgorithmic.

The rest of the paper is structured as follows: In Section 2, we present an efficient calculation of $b^{-1} \bmod 2^m$, followed by an example. A detailed mathematical treatment of the computational efforts entailed when calculating r is then presented, followed by considering the case in which m is not a power of 2. The performance of the proposed method for calculating $b^{-1} \bmod 2^m$ is compared to that of known algorithms in Section 3, followed by conclusions in Section 4.

2 Efficient Calculation of $r = b^{-1} \bmod 2^m$

In order to calculate $r = b^{-1} \bmod 2^m$, we first consider the calculation of $p = q^{-1} \bmod 2^{2i}$ given $q_L^{-1} \bmod 2^i$. Let k_H and k_L denote the higher half and lower half of the binary representation of a $2i$-bit value k, respectively. Each of these halves is an i-bit value. For example, in the relation $b \cdot r = 1 \bmod 2^i$, i.e., $b \cdot r = x \cdot 2^i + 1$ for some integer x, $(b \cdot r)_L = (000 \ldots 01)_2$.

Theorem 1. *Given b and r as i-bit values, where $r = b^{-1} \bmod 2^i$, and given q as a $2i$-bit value where $q_L = b$, the value $p = q^{-1} \bmod 2^{2i}$ can be efficiently obtained by calculating its lower half p_L and its higher half p_H (both are i-bit values independently). The lower half of p can be calculated as*

$$p_L = r, \tag{1}$$

while the higher half of p can be calculated as

$$p_H = -\left[\left[(r \cdot b)_H + (r \cdot q_H)_L\right] \cdot r\right] \bmod 2^i, \tag{2}$$

where p is formed by the concatenation between p_H and p_L such that $p = p_H | p_L$.

Proof. The relation $p = q^{-1} \bmod 2^{2i}$ suggests that $p \cdot q = 1 \bmod 2^{2i}$. From the multiplication $p \cdot q$, we derive that

$$(p_H p_L) \cdot (q_H q_L) = p_H \cdot q_H \cdot 2^{2i} + (p_H \cdot q_L + p_L \cdot q_H) \cdot 2^i + p_L \cdot q_L, \tag{3}$$

as depicted in Fig. 1. Therefore, the $2i$ LSBs of $p \cdot q = (p_H p_L) \cdot (q_H q_L)$ can be written as

$$\left[(p_H \cdot q_L + p_L \cdot q_H) \cdot 2^i + p_L \cdot q_L\right] \bmod 2^{2i}. \tag{4}$$

Since $p \cdot q = 1 \bmod 2^{2i}$,

$$\left[(p_H \cdot q_L + p_L \cdot q_H) \cdot 2^i + p_L \cdot q_L\right] = 1 \bmod 2^{2i}. \tag{5}$$

Calculating the lower part of p. Since the i LSBs of $(p_H \cdot q_L + p_L \cdot q_H) \cdot 2^i + p_L \cdot q_L$ are also the i LSBs of $p_L \cdot q_L$, only $p_L \cdot q_L$ dictates that the i LSBs of $(p_H \cdot q_L + p_L \cdot q_H)_L \cdot 2^i + p_L \cdot q_L$ are 1 (see Fig. 1 for details). That is,

$$p_L \cdot q_L = 1 \Longrightarrow p_L \cdot q_L = 1 \bmod 2^i \Longrightarrow p_L = q_L^{-1} \bmod 2^i. \tag{6}$$

Since $q_L = b$ and $r = b^{-1} \bmod 2^i$,

$$p_L = r.$$

Calculating the higher part of p. Given that the $2i$ LSBs of the expression $(p_H \cdot q_L + p_L \cdot q_H) \cdot 2^i + p_L \cdot q_L$ form the value 1

$$\left[(p_H \cdot q_L + p_L \cdot q_H)_L + (p_L \cdot q_L)_H\right] \bmod 2^i = 0 \tag{7}$$

(see the middle section in Fig. 1 for details). That is,

$$(p_H \cdot q_L)_L = -(p_L \cdot q_L)_H - (p_L \cdot q_H)_L \bmod 2^i. \tag{8}$$

Since the least significant component, k_L, of a number k is also $k \bmod 2^i$, we have

$$p_H = -\left[\left[(p_L \cdot q_L)_H + (p_L \cdot q_H)_L\right] \cdot q_L^{-1}\right] \bmod 2^i. \tag{9}$$

As $q_L = b$, $r = b^{-1} \bmod 2^i$, and $p_L = r$, we conclude that

$$p_H = -\left[\left[(r \cdot b)_H + (r \cdot q_H)_L\right] \cdot r\right] \bmod 2^i. \tag{10}$$

□

Example 1. To illustrate this result, we refer to the following example. For simplicity, all numbers are displayed in hexadecimal base. Our purpose is to find $p = q^{-1} \bmod 2^{32}$, given that $q = 99F8A5EF$ and $q_L = b$, where $r = b^{-1} = (A5EF)^{-1} = 290F \bmod 2^{16}$. Using the efficient calculation method described in Theorem 1, we can derive that $p_L = r = 290F$. In order to calculate p_H, we are required to follow three easy steps:

1. $(r \cdot b)_H = (290F \cdot A5EF)_H = 1A9D$,
2. $(r \cdot q_H)_L = (290F \cdot 99F8)_L = BD88$, and
3. $p_H = -[((r \cdot b)_H + (q_H \cdot r)_L) \cdot r] \bmod 2^{16} = -((1A9D + BD88) \cdot 290F) \bmod 2^{16} = 68D5$.

In order to obtain the final result, p, all that is left is the concatenation. Hence, $p = p_H | p_L = 68D5290F$. The result can be checked by validating the following equality $p \cdot q = 68D5290F \cdot 99F8A5EF = 3F0D37FD00000001 = 1 \bmod 2^{32}$.

Theorem 1 specifies the three operations: $(r \cdot b)_H$, $(r \cdot q_H)_L$, and $[(r \cdot b)_H + (r \cdot q_H)_L] \cdot r \bmod 2^i$. All values are i-bit long. The first operation involves the multiplication of i-bit values. On the other hand, when performing the latter two, only the lower half of the generated product is needed, requiring half of the full multiplication effort. This leads us to the following conclusion.

Conclusion 1. *The execution of Theorem 1 altogether involves* two *multiplications of i-bit values.*

Theorem 1 is used in reaching the final goal of calculating $b^{-1} \bmod 2^m$. The fundamental process of the theorem can be iteratively repeated $\log m$ times by doubling the number of bits when calculating the modular inverse in each iteration. At the end of such a process, we are left with $b^{-1} \bmod 2^m$. An illustrative example of this process is discussed in Section 2.1 below.

2.1 Computational Efforts

We begin with an introductory graphical illustration, depicted in Fig. 2, demonstrating the proposed procedure for calculating $b^{-1} \bmod 2^m$,, where $m = 8$-bits. Finding the multiplicative inverse of the number "$abcdefgh$" is comprised of of the following steps:

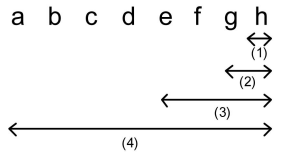

Fig. 2. Finding the multiplicative inverse of the number represented by "abcdefgh."

1. Finding the multiplicative inverse of "h" modulo 2^1. Note that "h" is 1 since the number "$abcdefgh$" is odd. For the same reason, the inverse of "h" is also 1.
2. Finding the multiplicative inverse of "gh" modulo 2^2 by exploiting the result from Step 1.
3. Finding the multiplicative inverse of "$efgh$" modulo 2^4 using the result obtained in Step 2.
4. Finding the multiplicative inverse of the entire number "$abcdefgh$" modulo 2^8 by using the result from Step 3.

It is noted that Steps 2-4 are achieved by using Theorem 1.

2.1.1 Overall Computational Efforts in Terms of Word-Level Processor Multiplications

In the case treated by this section, $r = b^{-1} \bmod 2^m$ is calculated, where m is a power of 2. Consider the case where each digit in the illustration in Fig. 2 is a processor word of k-bits. In practice, k is a power of 2. Letting $n = \frac{m}{k}$, the representation of r and b consists of n words.

The process illustrated in Fig. 2 suggests that $\log n$ consecutive doublings of $h^{-1} \bmod 2^k$ yield $r = b^{-1} \bmod 2^m$ (where h is the rightmost word of b). In the ith stage, $i = 0, 1, \cdots, \log n - 1$, a value consisting of 2^i words is doubled in size. Based on the observation in Conclusion 1, such doubling requires two multiplications of values consisting of 2^i words. One such multiplication requires $(2^i)^2 = 2^{2i}$ single-word multiplications. The entire process thus involves $2 \cdot \sum_{i=0}^{\log n-1} 2^{2i}$ single-word multiplications. It should be noted that $2 \cdot 2^{2\cdot(\log n-1)} = 2 \cdot (\frac{n}{2})^2$, i.e., during the last stage, a value consisting of $\frac{n}{2}$ words is doubled, requiring $2 \cdot (\frac{n}{2})^2$ single-word multiplications. The total number of single-word multiplications required when calculating $r = b^{-1} \bmod 2^m$, given the initial value $h^{-1} \bmod 2^k$, is therefore $2 \cdot [\sum_{i=0}^{\log n-1} 2^{2i}] = \frac{2(n^2-1)}{3}$, where n^2 is the number of single-word multiplications executed when multiplying two m-bit operands. By definition, each such operand consists of n words.

Let us now evaluate the computational effort involved in calculating $h^{-1} \bmod 2^k$. This calculation in itself requires $\log k$ executions of the process in Theorem 1, doubling 1-bit, 2-bit, $\ldots, \frac{k}{2}$-bit, values. Since this paper concerns software implementations using a general-purpose processor, these $\log k$ small values should each be considered as a complete processor word. Thus, when implementing Conclusion 1, this effort requires $2 \cdot \log k$ single-word multiplications.

As a result, when calculating $r = b^{-1} \bmod 2^m$, the overall number of single-word multiplications is

$$\frac{2(n^2-1)}{3} + 2\log k. \tag{11}$$

Standard sizes for a processor word are 8, 16, 32, 64-bits, for which $\log k = 3$, 4, 5, 6, where n^2 is the number of single-word multiplications executed when multiplying two m-bit operands. It is concluded that the computational effort involved in calculating the multiplicative inverse of an odd value b modulo 2^m is two thirds of one multiplication of m-bit values, plus a negligible number of single-word multiplications which, in practical cases, does not exceed 12 $(= 2\log k)$.

2.2 Treating the Case Where m Is Not a Power of 2

Let $a^{-1} = b \bmod p$ and q be a divisor of p. Basic observations in number theory show that $[a \bmod q]^{-1} = b \bmod q$. For clarification, letting $a = 11$, $p = 14$, and $q = 7$, it is evident that $11^{-1} = 9 \bmod 14$ and $4^{-1} = 2 \bmod 7$.

The procedure of Section 2 treats the case where m is a power of 2 (allowing for the described consecutive doublings). In cases where m is not a power of 2, the binary representation of b consists of $n = 2^{\lceil \log m \rceil}$ bits and the value $w = b^{-1} \bmod 2^n$ is calculated in $\lceil \log m \rceil$ steps, based on the procedure of Section 2. Consequently, $r = w \bmod 2^m$ is the required result $b^{-1} \bmod 2^m$. An important observation pertains to the fact that calculating $w \bmod 2^m$ can be done by taking the m LSBs in the binary representation of w.

3 Performance Comparison

The performance of the presented procedure for calculating $b^{-1} \bmod 2^m$ is compared to that of other procedures based on two criteria: 1) ability for software execution on a general-purpose processor and 2) computational effort, measured in terms of the overall number of executed word-level multiplications.

The presented procedure was shown to be suitable for software implementation on a general-purpose processor. When counting the total number of word-level multiplications, it was shown that the computational effort involved in calculating $b^{-1} \bmod 2^m$ is 2/3 of one multiplication of m-bit values, plus a few word-level multiplications, whose number in practice does not exceed 12.

Algorithm 1 [3] is executed in $m - 1$ steps, each one involving the multiplication operation $b \cdot y_{i-1}$, each multiplicand consisting of m bits. The procedure can be naturally performed in software using a general-purpose processor. In contrast, this paper presents an approach in which the overall number of single-word multiplications is equivalent to a single m-bit multiplication, also executed on a general-purpose processor.

Algorithm 2 is executed in m steps, involving shifts and adds. Altogether, this is equivalent to one multiplication of m-bit values. Shifts and decisions are being made at the bit level. Therefore, on one hand, the computational effort is equivalent to one m-bit multiplication, like the procedure presented in this paper. On the other hand, the bit-level considerations in Algorithm 2 pose implementation difficulties on a general-purpose processor, a disadvantage which the procedure presented in this paper overcomes.

Algorithm 3 is executed in m steps involving shifts and adds and one division of a $2m$-bit value by an m-bit value. Shifts and decisions in Algorithm 3a are made at the bit level. Therefore, the procedure presented in this paper outperforms Algorithm 3 as far as both discussed criteria are concerned.

It should be noted that the Montgomery modular multiplication procedure [1] yields the value $x \cdot y \cdot 2^{-m} \bmod b$, for m-bit operands. Algorithm 3a can therefore be replaced by Montgomery procedure, for $x = y = 1$ (yielding $2^{-m} \bmod b$). The replacement of Algorithm 3a by a Montgomery multiplication can possibly have the advantage of running in software on a general processor, utilizing word-level multiplications. Regardless of this effort, there is still a need to perform a division of a $2m$-bit value by an m-bit value, as required by Algorithm 3b. Therefore, the procedure presented in this paper significantly outperforms this case too.

We next compare the procedure proposed in this paper to the *Extended Euclid Algorithm* [4] and to the *Montgomery Inverse Algorithm* [5], [6]. As shown in Section 1, Algorithm 3a is precisely the Extended Euclid Algorithm when used in calculating $2^{-m} \bmod b$, while Algorithm 3b is the procedure of exchanging

the role of 2^m and b, yielding the result $b^{-1} \bmod 2^m$. Hence, Algorithms 3a plus 3b are the implementation of the Extended Euclid Algorithm for the case treated in this paper. The Montgomery Inverse Algorithm treats an odd modulus. Therefore, an implementation of this algorithm in calculating $b^{-1} \bmod 2^m$ would mean that $2^{-m} \bmod b$ is calculated first and $b^{-1} \bmod 2^m$ is then recovered by executing Algorithm 3b. Accordingly, Algorithm 3b is still needed when using the Montgomery Inverse Algorithm. Furthermore, Algorithm 3a is the second phase of the Montgomery Inverse Algorithm. Therefore, when calculating $b^{-1} \bmod 2^m$, the execution of Algorithms 3a plus 3b is more efficient than the Montgomery Inverse. It is concluded that the procedure presented in this paper for calculating $b^{-1} \bmod 2^m$, which outperforms Algorithms 3a plus 3b as shown above, outperforms the Extended Euclid Algorithm and the Montgomery Inverse Algorithm.

Besides the practical advantages (shown above) of the presented procedure, it should be noted again that the mathematical approach taken by this paper is different from those treated in all of the above referenced papers.

4 Conclusion

Various procedures have been proposed in the past for calculating $b^{-1} \bmod 2^m$. Some lend themselves to software execution on a general processor, utilizing word-level multiplications. The overall computational effort associated with such procedures is equivalent to more than one multiplication of m-bit operands. Other procedures, having the overall computational effort equivalent to one multiplication of m-bit operands, are executed on a bit level and are not suitable for software implementation using a general-purpose processor (i.e., they cannot efficiently utilize word-level multiplications).

In this paper, we presented a procedure for calculating $r = b^{-1} \bmod 2^m$, which, on one hand, is suitable for software implementation using a general-purpose processor, while, on the other hand, it utilizes processor-word multiplications. When counting the total number of such multiplications, the computational effort involved in calculating r is $2/3$ of one multiplication of m-bit values, plus a few word-level multiplications whose number in practice does not exceed 12. This combines the best individual performances of known procedures when considering minimal computational effort and ability for software execution on a general-purpose processor. To the best of the authors' knowledge, the mathematical principle on which the proposed procedure relies is novel and is not an extension or modification of known approaches.

References

[1] P. Montgomery, "Modular Multiplication without Trial Division," *Math. Computation,* vol. 44, pp. 519-521, 1985.
[2] J. Tudor, "An Algorithm for Exact Division," *J. Symbolic Computation Archive,* vol. 15, pp. 169-180, Feb. 1993.
[3] S.R. Dusse and B.S. Kaliski, "A Cryptographic Library for the Motorola DSP5600," Advances in Cryptology, *Proc. Ann. EuroCrypt Conf.,* pp. 230-244, 1990.
[4] J.L. Massey, "Cryptography: Fundamentals and Applications," *Advanced Technology Seminars,* Feb. 1993.
[5] B.S. Kaliski, "The Montgomery Inverse and Its Applications," *IEEE Trans. Computers,* vol. 44, no. 8, pp. 1064-1065, Aug. 1995.
[6] E. Savas and Ç.K. Koç, "The Montgomery Modular Inverse—Revised," *IEEE Trans. Computers,* vol. 49, no. 7, pp. 763-766, July 2000.
[7] D. Matula, A. Fit-Florea, and M. Thornton, "Table Lookup Structures for Multiplicative Inverses Modulo 2/Sup k," *Proc. 17th IEEE Symp. Computer Arithmetic,* pp. 156-163, June 2005.