Optimizing Modular Inverse Computation

David J. Walling
nuBridges, Inc.,
1000 Abernathy Road, Suite 250
Atlanta, GA, 30328
dwalling@nubridges.com

Abstract

This contribution considers several optimizations of a well-known algorithm for multiplicative inverse computation modulo 2^b. The algorithm is suitable for contemporary general purpose computers where b is the size in binary digits of a processor register. The optimizations facilitate more efficient modular multiplication when the lowest-order word of the reciprocal must be found. Such is the case when using Montgomery multiplication, which applies $m' = -m^{-1} \bmod R$, where m is a radix b integer of size n words, $R = b^n$, and $gcd(m,R) = 1$. Analyses and sample implementations are provided.

1. Introduction.

The general purpose digital computer directly performs its various logical and arithmetic operations on values within its processor *registers*. Arithmetic functions of integers with values exceeding the processor register size k bits may perform arithmetic on *multiple-precision* representations such as an array of n words expressing a value W, where $b = 2^k$, $b \leq W < b^n$ and $W = (w_n w_{n-1} \ldots w_1 w_0)_b$ in radix b.

Some applications such as RSA [7] and Diffie Hellman key exchange [1] require exponentiation $x^e \bmod m$, where $1 \leq x < m$ and both x and m are sufficiently large to render classical multiplication and division infeasible. Computational efficiencies exist when multiple-precision *modular* arithmetic is performed in Z_m, the integers modulo m rather than in radix b representation[1]. Furthermore, Montgomery *reduction* [6] improves on classical modular multiplication by reducing the need for modular reduction steps.

The Montgomery algorithm computes the product $abR^{-1} \bmod m$ and involves an intermediate value $xR \bmod m$. When $gcd(b,m) = 1$, R may be b^n and reduction proceeds by iteratively applying the pre-computed value $m' = -m^{-1} \bmod b$. A few works, including [4] and [5], suggest the Extended Euclidean algorithm to compute the multiplicative inverse m^{-1} for general cases. Dussé and Kaliski contribute an efficient algorithm [2] to produce $-m_0^{-1} \bmod b$. The present work suggests optimizations to this algorithm and considers the efficient implementation of these optimizations in software.

2. Notations and Conventions.

In addition to typical arithmetic symbols, this contribution uses the following notations.

$\leftarrow$	The assignment operation. $A \leftarrow B$ assigns the value B to the variable A.
gcd	The greatest common divisor. *gcd(* 95, 20 *)* = 5 because no number greater than five can evenly divide both 95 and 20.
mod	The modulus operation. *A mod B* produces the remainder of the division of *A* by *B*.
shl	The logical "shift left" operation. *A shl B*, produces the output such that each binary digit of *A* is shifted *B* binary digits higher in magnitude, with zeros replacing the lowest order *B* binary digits.
$\vee$	The logical "or" operation. $A \vee B$ is true if either *A* or *B* is true.
$\wedge$	The logical "and" operation. $A \wedge B$ is true if both *A* and *B* are true.
$\equiv$	The congruence evaluation. $A \equiv B \bmod n$ if $A \bmod n = B \bmod n$.
$\bullet$	The arithmetic multiply operation. A $\bullet$ B is the product of A and B.
M_n	The subscripted numeral following the variable, e.g. M_n, represents the n^{th} element or word in the multi-precision integer *M*, where M_0 is the lowest-order word of *M* and M_{n-1} is the highest order word of *M*.

[1] [3] pp. 284-294, [5] pp. 67-70.

3. Mathematical Preliminaries.

3.1. Multiplicative Inverse.

The multiplicative inverse, or *reciprocal*, of x is the number which, when multiplied by x, yields 1. The multiplicative inverse is denoted x^{-1} or $1/x$. For example, the real multiplicative inverse of 237 is approximately 0.0042194092827. In modular arithmetic, the number x has a reciprocal x' *mod* m if $x < m$ and $gcd(x,m) = 1$.

The Extended Euclidean algorithm produces integers which are modular multiplicative inverses.[2] In other words, the product of the m-bit integer x and its modular inverse $x' \equiv 1 \ mod \ m$, provided that $x < m$. For example, the modular inverse of 237 (`0xED`), an 8-bit integer, is 229 (`0xE5`) because $237 \cdot 229 = 54{,}273$, which is congruent to 1, modulo 256 or 2^8.

Figure 1 illustrates the binary representation of this example. The product $a \cdot x$, 54,273, can be seen to be congruent to 1 *mod* m, where $m = 2^b$ and $x < m$.

```
                1110 1101                 237
x               1110 0101        x        229
    --------------------             --------
=   1101 0100 0000 0001        =   54,273
MOD 0000 0001 0000 0000        MOD    256
    --------------------             --------
=   0000 0000 0000 0001        =        1
```

FIG. 1. Modular Multiplicative Inverse.

3.2. Residue Systems.

Montgomery reduction[3] produces the modular *residue* of the integer x as being the remainder of $x \bullet 2^k$ divided by the k-bit integer m, where $x < m$. A *residue system* is the set of remainders for all $x \bullet 2^k$ divided by the k-bit integer m, where $x < m$.

3.3. A Sample Residue System.

Given: $2^{k-1} \leq m < 2^k$, $r = 2^k$,
$x < m$, $x' = x \bullet r \ mod \ m$

If $k = 4$, $m = 13$, then $r = 2^4 = 16$, and for each integer $x < m$, the residue x' is shown in the following table.

[2] [5] p. 85, Algorithm 2.226.
[3] [5] pp. 600-603.

x		x • r		x • r mod m
1	• 16 =	16	mod 13 =	3
2	• 16 =	32	mod 13 =	6
3	• 16 =	48	mod 13 =	9
4	• 16 =	64	mod 13 =	12
5	• 16 =	80	mod 13 =	2
6	• 16 =	96	mod 13 =	5
7	• 16 =	112	mod 13 =	8
8	• 16 =	128	mod 13 =	11
9	• 16 =	144	mod 13 =	1
10	• 16 =	160	mod 13 =	4
11	• 16 =	176	mod 13 =	7
12	• 16 =	192	mod 13 =	10

FIG. 2. Sample Residue System.

Thus it can be seen in the table above that the residue system for $k = 4$, $m = 13$ is a complete system for there is a 1:1 correspondence between each residue $1 \leq x' < m$ and it's integer x.

3.4. MONTGOMERY PRODUCT.

The Montgomery multiplication algorithm uses residues and a multiplicative inverse to efficiently produce products required during the exponentiation process. The Montgomery product is the residue $c' = a' \bullet b' \bullet r^{-1} \bmod m$, where r^{-1} is the modular multiplicative inverse of $r \bmod m$.

Given $2^{k-1} \leq m < 2^k$, $r = 2^k$
$a < m$, $a' = a \bullet r \bmod m$
$b' = b \bullet r \bmod m$
$c' = a' \bullet b' \bullet r^{-1} \bmod m$

If $k = 4$
$m = 13$
$a = 4$
$b = 7$

Then $r = 2^4 = 16$
$a' = (a \bullet r \bmod m) = (4 \bullet 16 \bmod 13) = (64 \bmod 13) = 12$
$b' = (b \bullet r \bmod m) = (7 \bullet 16 \bmod 13) = (112 \bmod 13) = 8$
$r^{-1} \bmod 1 = (16^{-1} \bmod 13) = 9$
$c' = a' \bullet b' \bullet r^{-1} \bmod m = (12 \bullet 8 \bullet 9 \bmod 13) = (864 \bmod 13) = 6$

FIG. 3. Montgomery Product.

Note first that we provide the reciprocal 9 of our r, 16. The computation of this reciprocal is the subject of our algorithm and its optimizations that we will see below. For now, we can verify this value by computing ($9 \bullet 16$ *mod* 13) = (144 *mod* 13) = 1. Note secondly that the product c', having a value of 6, is computed according to our formula, $c' = a' \bullet b' \bullet r^{-1} \bmod m$. This value can be verified by the table introduced in the section above for the residue system. For where the residue in that table is 6, the corresponding integer is 2, which is the result of $a \cdot b \bmod m = 4 \bullet 7 \bmod 13 = 28 \bmod 13 = 2$.

This example illustrates only the principle of the Montgomery product and the relationship between the residue system and the corresponding set of integers from which it is derived. Koç, Acar, and Kaliski [4] have presented algorithms that demonstrate how the use of residues and m_0', the least significant word of m', or $-m^{-1} \bmod b$, can reduce the computation required to perform modular multiplication of very large integers. Those discussions are beyond the scope of this work. We direct our attention now, then, to the optimization of the algorithm that produces m_0'.

4. Variations.

In this section we present several variations of the algorithm. Each subsequent variation introduces one or more optimizations developed from the previous variation.

4.1. First Variation.

Our starting point is a rendering of Dussé and Kaliski's algorithm [2] for computing the least significant word of m', or simply m_0'.

4.1.1. *Exposition.*

Figure 4 presents the algorithm, introducing temporary variables for clarity.

Input:	k = the size of a word in binary digits.
	b = the radix, or 2^k.
	m_0 = the least significant word of integer m, or $m \bmod b$.
Output:	$m_0' = -m^{-1} \bmod b$.
1.1	$t \leftarrow 1$
1.2	for $i \leftarrow 2$ to k do
1.3	$a \leftarrow (m_0 \bullet t) \bmod 2^i$
1.4	if $a \geq 2^{i-1}$
1.5	$t \leftarrow t + 2^{i-1}$
1.6	return $b - t$

FIG. 4. The First Variation.

4.1.2. *Principle*.

The principle of the algorithm is the accumulation of bits in the variable *t* for each iteration *i*. The accumulated bits are of increasingly greater magnitude and are added to the previous value of *t* in step 1.5 when the residue *a* of the least significant word of the multi-precision integer *m*, that is m_0, multiplied by *t* modulo 2^i, is between 2^{i-1} and 2^i - 1, inclusive.

4.1.3. *Observations.*

The accumulator *t* is initialized to 1 in step 1.1. This can be considered an intermediate result of the iteration $i = 1$, which is not executed, since any odd $m_0 \bmod 2 = 1$. The iteration ends when *i* exceeds *k*, the number of bits in m_0. The result returned is $2^k - t$, or $b - t$.

4.1.4. *Optimizations.*

In the first variation, computing $a \leftarrow (m_0 \bullet t) \bmod 2^i$ in step 1.3 could be accomplished without requiring a division by substituting a bitwise logical *and* operation, hence computing $(m_0 \bullet t) \wedge (2^i - 1)$, because computing any positive product modulo a power of two will equal a bitwise *and* operation against that same power of two, less one. Figure 5 illustrates this substitution, showing the division in the left column and the corresponding logical *and* operation in the right column.

```
     237    =    0x00ED    =      1110 1101
x      5    =    0x0005    =           0101
--------                     --------------
    1185    =    0x04A1    = 0100 1010 0001

      74
   -----
16  1185    =    0x04A1    = 0100 1010 0001
    112                   AND 0000 0000 1111 = 15 = (2^4)-1
    ----
      65
      64
      --                     --------------
       1    =    0x0001    = 0000 0000 0001
```

FIG. 5. Substituting logical *and* to compute modulus.

The iterator *i* increases by one for each iteration. Therefore, the values representing 2^i-1 and 2^{i-1} are known for each *i* and follow the progression shown in Figure 6.

i	2^i-1	2^{i-1}
2	$2^2-1 = 3 = 0x00011$	$2^1 = 2 = 0x00010$
3	$2^3-1 = 7 = 0x00111$	$2^2 = 4 = 0x00100$
4	$2^4-1 = 15 = 0x01111$	$2^3 = 8 = 0x01000$
5	$2^5-1 = 31 = 0x11111$	$2^4 = 16 = 0x10000$

FIG. 6. 2^i-1 and 2^{i-1} progression.

It can be seen from Figure 6 that the 2^i-1 and 2^{i-1} values for each i can be efficiently computed by progressive shift and logical *or* operations. Therefore, these values may be maintained in variables updated for each i. A variable j, for example, representing 2^i-1, can be updated for each i by shifting its value once to the left and setting the low-order bit, or $j_{i+1} \leftarrow (j_i \; shl \; 1) \vee 1$. Likewise, a variable r, representing 2^{i-1}, can be updated for each i by shifting the value once to the left, or $r_{i+1} \leftarrow r_i \; shl \; 1$.

Figure 7 below illustrates the use of the variables j and r in the computation of the intermediate values a and t for iteration 3 of our example.

Given: i = 3

$a \leftarrow m_0 \cdot t \bmod 2^i$		1110 1101	m_0
	x	0001	t

		1110 1101	
	and	0111	$j = 2^3-1$

		0000 0101	$a \leftarrow$
$j_{i+1} \leftarrow (j_i \; shl \; 1) \vee 1$		0111	j
	shl	1110	
	or	0001	

		1111	$j \leftarrow$
$t \leftarrow t + 2^{i-1}$		0001	t
	+	0100	$r = 2^2$

		0101	$t \leftarrow$
$r_{i+1} \leftarrow r_i \; shl \; 1$		0100	r
	shl	1000	$r \leftarrow$

FIG. 7. Maintaining Variables for 2^i-1 and 2^{i-1}.

Finally, the addition of 2^{i-1} to t in step 1.5 and illustrated above can be implemented with a logical *or* instead of an addition because 2^{i-1} will always be both a power of two, therefore a single bit, and will always be greater than t due to the fact that i increases for every iteration.

4.2 THE SECOND ALGORITHM.

4.2.1. *Exposition*.

Applying the few simple modifications described above derives the Second Algorithm.

Input:	k = the size of a word in binary digits.
	b = the radix, or 2^k.
	m_0 = the least significant word of integer m, or $m \bmod b$.
Output:	$m_0' = -m^{-1} \bmod b$.
2.1	$t \leftarrow 1$
2.2	$r \leftarrow 2$
2.3	$j \leftarrow 3$
2.4	for $i \leftarrow 2$ to k
2.5	$a \leftarrow (m_0 \bullet t) \wedge j$
2.6	if($a \geq r$)
2.7	$t \leftarrow t \vee r$
2.8	$r \leftarrow r$ *shl* 1 *mod* 2^b
2.9	$j \leftarrow (j$ *shl* 1 *mod* $2^b) \vee 1$
2.10	return $b - t$

FIG. 8. The Second Algorithm.

The principle of this algorithm remains that of the First Algorithm, accumulation of result bits in the variable t for each iteration i. The optimizations applied include the elimination of any explicit modular division or recomputation of 2^i, 2^{i-1}, or 2^i-1.

4.2.2. *Sample Implementation*.

The following source code provides an assembly language example of the Second Algorithm, using Intel™ mnemonics for the IA32 (x86) architecture. The assembly instructions depicted below further illustrate the preference for logical instructions rather than arithmetic instructions.

```
;     in:   [ebp+4] = M[0]
;     out:  eax = -M[0]^-1

      mov   edi,1             ; t := 1
      mov   esi,2             ; r := 2^(i-1) for i = 2
      mov   ebx,3             ; j := (2^i)-1 for i = 2
      mov   ecx,31            ; for i := 2 to k

_10:  mov   eax,[ebp+4]       ; a := M[0]
      mul   edi               ; a := M[0] * t
      and   eax,ebx           ; a := M[0] * t MOD 2^i
      cmp   eax,esi           ; M[0]*t MOD 2^i >= 2^(i-1)?
      jb    _20               ; no, skip ahead
      or    edi,esi           ; t := t + 2^(i-1)

_20:  stc                     ; set carry
      rcl   ebx,1             ; next j (2^i)-1
      shl   esi,1             ; next r 2^(i-1)
      loop  _10               ; next i

      neg   edi               ; t := b – t
      mov   eax,edi
```

FIG. 9. A Second Algorithm Implementation.

A closer examination of this implementation reveals a few more optimizations we can apply at this stage. Firstly, since we are using variables *r* (`esi`) and *j* (`ebx`) to hold values based on 2^i, the variable *i* need not be explicitely represented as a value between *2* and *k*, inclusive. Therefore, we use the `ecx` as a loop count register from *31* down to *zero*. Secondly, note that instead of computing $j_{n+1} = (j \text{ shl } 1) \vee 1$ using an `or` instruction after the left shift (`shl`), we instead set the carry flag and rotate it (`rcl`) into the low-order bit of *j* (`ebx`).

Three areas remain now for more analysis and improvement. First, the value of M_0 is copied into the accumulator for each iteration. Secondly, $M_0 \bullet t$ is still computed with a multiplication instruction. Thirdly, a comparison instruction is used to evaluate *a* to *r*. Subsequent optimizations below will eliminate the need for all of these. But before proceeding to them, the following section presents an iterative example of the Second Algorithm.

4.2.3. *Iterative Example*.

The following table illustrates the behavior of the Second Algorithm by reproducing the values of the variables *t*, *r*, *a* and *j* for each iteration *i* where M_0 = 237 (`0xed`) and *k* = 32. The variable *t* is first initialized to 1. Then, for each iteration *i*, the algorithm proceeds through to $a \leftarrow (M_0 \bullet t) \bmod 2^i$, when, if this value is found to be greater or equal to *j*, *j* is added to *t*. The output *t* multiplied by M_0 can be shown to be congruent to 1, modulo *b*:

i	M[0]*t		j = (2^i)-1		a = M[0]*t mod 2^i		r = 2^(i-1)		t
									00000001
2	**000000ed**	^	**00000003**	=	**00000001**	≥	**00000002**		00000001
3	000000ed	^	**00000007**	=	**00000005**	≥	**00000004**	⇒	**00000005**
4	**000004a1**	^	**0000000f**	=	**00000001**	≥	**00000008**		00000005
5	000004a1	^	**0000001f**	=	00000001	≥	**00000010**		00000005
6	000004a1	^	**0000003f**	=	**00000021**	≥	**00000020**	⇒	**00000025**
7	**00002241**	^	**0000007f**	=	**00000041**	≥	**00000040**	⇒	**00000065**
8	**00005d81**	^	**000000ff**	=	**00000081**	≥	**00000080**	⇒	**000000e5**
9	**0000d401**	^	**000001ff**	=	00000081	≥	**00000100**		000000e5
10	0000d401	^	**000003ff**	=	00000081	≥	**00000200**		000000e5
11	0000d401	^	**000007ff**	=	**00000481**	≥	**00000400**	⇒	**000004e5**
12	**00048801**	^	**00000fff**	=	**00000801**	≥	**00000800**	⇒	**00000ce5**
13	**000bf001**	^	**00001fff**	=	**00001801**	≥	**00001000**	⇒	**00001ce5**
14	**001ac001**	^	**00003fff**	=	**00000001**	≥	**00002000**		00001ce5
15	001ac001	^	**00007fff**	=	**00004001**	≥	**00004000**	⇒	**00005ce5**
16	**00560001**	^	**0000ffff**	=	**00000001**	≥	**00008000**		00005ce5
17	00560001	^	**0001ffff**	=	00000001	≥	**00010000**		00005ce5
18	00560001	^	**0003ffff**	=	**00020001**	≥	**00020000**	⇒	**00025ce5**
19	**02300001**	^	**0007ffff**	=	**00000001**	≥	**00040000**		00025ce5
20	02300001	^	**000fffff**	=	00000001	≥	**00080000**		00025ce5
21	02300001	^	**001fffff**	=	**00100001**	≥	**00100000**	⇒	**00125ce5**
22	**11000001**	^	**003fffff**	=	**00000001**	≥	**00200000**		00125ce5
23	11000001	^	**007fffff**	=	00000001	≥	**00400000**		00125ce5
24	11000001	^	**00ffffff**	=	00000001	≥	**00800000**		00125ce5
25	11000001	^	**01ffffff**	=	01000001	≥	**01000000**	⇒	**01125ce5**
26	**fe000001**	^	**03ffffff**	=	**02000001**	≥	**02000000**	⇒	**03125ce5**
27	**2d8000001**	^	**07ffffff**	=	**00000001**	≥	**04000000**		03125ce5
28	2d8000001	^	**0fffffff**	=	**08000001**	≥	**08000000**	⇒	**0b125ce5**
29	**a40000001**	^	**1fffffff**	=	**00000001**	≥	**10000000**		0b125ce5
30	a40000001	^	**3fffffff**	=	00000001	≥	**20000000**		0b125ce5
31	a40000001	^	**7fffffff**	=	**40000001**	≥	**40000000**	⇒	**4b125ce5**
32	**4580000001**	^	**ffffffff**	=	**80000001**	≥	**80000000**	⇒	**cb125ce5**

FIG. 10. Iterative Execution of Second Algorithm.

`3,406,978,277 x 237 = 807,453,851,649 (0x00BC`**`00000001`**`) ≡ 1 mod 2^32.`

4.2.4. *Observations.*

In the Second Algorithm it is seen (step 2.5) that a is computed as $M_0 \bullet t$. From the iterative example above it is seen that in each iteration i where $M_0 \bullet t$ is recomputed, the new value is equal to sum of the previous value t and M_0 *shl* $(i\text{-}2)$. For example, when $i = 7$, $M_0 \bullet t$ becomes `0x2241`, which is equal to the previous value of $M_0 \bullet t$, `0x4a1`, plus M_0 *shl* 5, `0x1da0`.

Secondly, step 2.5 completes by reducing ($M_0 \bullet t$) *mod* 2^i. The iterative example represents this modulo by applying a logical *and* ($\wedge$) operation to $M_0 \bullet t$ with *j*, which is always 2^i-1. Following this, step 2.6 compares the modulus *a* to 2^{i-1}. It can be seen from the iterative example above that the combination of the modulo and comparison ($\geq$) operations can both be replaced by a test of the significant bit in 2^{i-1}. For example, when *i* = 12, `0x48801` modulo `0xfff` produces `0x801`, which is then found to be greater than 2^{i-1}, `0x800`. The same is found by simply applying a logical *and* ($\wedge$) to $M_0 \bullet t$ and 2^{i-1}.

Finally, Step 2.4 iterates the algorithm from *i* = 2 to *k*. The sample implementation shows that the iteration *i* no longer needs to increment from 2 to *k* but can decrement from *k*-1 to zero. Moreover, by using a simple *shl* to recompute 2^{i-1} for each iteration *i*, we can see from the iterative example that 2^{i-1} *shl* 1 will produce zero (*modulo b*) when $i > k$. If we can test for this zero condition in 2^{i-1} following our shift, then we do not need to maintain an independent variable *i* for the sole purpose of loop counting.

4.3. THE THIRD ALGORITHM.

4.3.1. *Exposition*.

The observations above can now be applied to produce the Third Algorithm.

Input:	k = the size of a word in binary digits.
	b = the radix, or 2^k.
	m_0 = the least significant word of integer m, or $m \bmod b$.
Output:	$m_0' = -m^{-1} \bmod b$.
3.1	$a \leftarrow m_0$
3.2	$t \leftarrow 1$
3.3	$i \leftarrow 2$
3.4	$r \leftarrow 3$
3.5	while $i \neq 0$ begin
3.6	if($a \wedge i$) begin
3.7	$t \leftarrow t \vee i$
3.8	$a \leftarrow a + r \bmod 2^b$
3.9	end
3.10	$r \leftarrow r$ *shl* 1 *mod* 2^b
3.11	$i \leftarrow i$ *shl* 1 *mod* 2^b
3.12	end
3.13	return $b - t$

FIG. 11. The Third Algorithm

4.3.2. *Observations.*

In the Third Algorithm, the *modulo* 2^b computation of $a + r$, r and i in steps 3.8, 3.10, and 3.11 is accomplished implicitly when b is the size of the processor register or word. Therefore, no instructions need to be performed to accomplish any *modulo* 2^b operation, if the result is stored in a register of size b bits.

Secondly, the variable r may be initialized to 1 instead of 3 or, in other words, to 2^{i-2}, if the shift of r in step 3.10 is performed before the addition of r and a shown in step 3.8. This is an improvement because it eliminates the final shift of r in the last iteration when i is found to be zero because the test of i is made before r is shifted. Furthermore, assigning an initial value of 1 to both t and r may be more efficiently accomplished if a register-to-register move instruction executes faster than an immediate-to-register move instruction.

4.4. THE FOURTH ALGORITHM.

4.4.1. *Exposition.*

The observations above can now be applied to produce the Fourth Algorithm.

Input:	k = the size of a word in binary digits.
	b = the radix, or 2^k.
	m_0 = the least significant word of integer m, or $m \bmod b$.
Output:	$m_0' = -m^{-1} \bmod b$.
4.1	$a \leftarrow m_0$
4.2	$r \leftarrow m_0$
4.3	$t \leftarrow 1$
4.4	$i \leftarrow 2$
4.5	while $i \neq 0$ begin
4.6	$r \leftarrow r$ *shl* 1
4.7	if($a \wedge i$) begin
4.8	$t \leftarrow t \vee i$
4.9	$a \leftarrow a + r \bmod 2^b$
4.10	end
4.11	$i \leftarrow i$ *shl* 1 $\bmod 2^b$
4.12	end
4.13	return $b - t$

FIG. 12. The Fourth Algorithm

4.4.2. *Sample Implementation*.

We again represent our algorithm in assembly language for analysis.

```
;     in:   [ebp+4] = M[0]
;     out:  eax = -M[0]^-1

      mov   ebx,[ebp+4]       ; a := n
      mov   edx,ebx           ; r := n
      mov   ecx,2             ; i := 2
      mov   eax,1             ; t := 1
_10:  shl   edx,1             ; r := ( r SHL 1 ) mod 2^b
      test  ebx,ecx           ; a & i ?
      jz    _20               ; no, skip ahead
      or    eax,ecx           ; t := t OR i
      add   ebx,edx           ; a := ( a + r ) mod 2^b
_20:  shl   ecx,1             ; i := ( i SHL 1 ) mod 2^b
      jnc   _10               ; next i
      neg   eax               ; t := b - t
```

FIG. 13. A Fourth Algorithm Implementation

4.4.3. *Iterative Example*.

The table below represents the values assigned to the variables *r*, *a*, *t* and *i* through each iteration when *b* is 32 and the input *n* is 237 (`0xED`).

iteration	r	a	t	i
	0x000000ed	0x000000ed	0x00000001	0x00000002
1	0x000001da	0x000000ed	0x00000001	0x00000004
2	0x000003b4	0x000004a1	0x00000005	0x00000008
3	0x00000768	0x000004a1	0x00000005	0x00000010
4	0x00000ed0	0x000004a1	0x00000005	0x00000020
5	0x00001da0	0x00002241	0x00000025	0x00000040
6	0x00003b40	0x00005d81	0x00000065	0x00000080
7	0x00007680	0x0000d401	0x000000e5	0x00000100
8	0x0000ed00	0x0000d401	0x000000e5	0x00000200
9	0x0001da00	0x0000d401	0x000000e5	0x00000400
10	0x0003b400	0x00048801	0x000004e5	0x00000800
11	0x00076800	0x000bf001	0x00000ce5	0x00001000
12	0x000ed000	0x001ac001	0x00001ce5	0x00002000
13	0x001da000	0x001ac001	0x00001ce5	0x00004000
14	0x003b4000	0x00560001	0x00005ce5	0x00008000
15	0x00768000	0x00560001	0x00005ce5	0x00010000
16	0x00ed0000	0x00560001	0x00005ce5	0x00020000
17	0x01da0000	0x02300001	0x00025ce5	0x00040000
18	0x03b40000	0x02300001	0x00025ce5	0x00080000
19	0x07680000	0x02300001	0x00025ce5	0x00100000
20	0x0ed00000	0x11000001	0x00125ce5	0x00200000
21	0x1da00000	0x11000001	0x00125ce5	0x00400000
22	0x3b400000	0x11000001	0x00125ce5	0x00800000
23	0x76800000	0x11000001	0x00125ce5	0x01000000
24	0xed000000	0xfe000001	0x01125ce5	0x02000000
25	0xda000000	0xd8000001	0x03125ce5	0x04000000
26	0xb4000000	0xd8000001	0x03125ce5	0x08000000
27	0x68000000	0x40000001	0x0b125ce5	0x10000000
28	0xd0000000	0x40000001	0x0b125ce5	0x20000000
29	0xa0000000	0x40000001	0x0b125ce5	0x40000000
30	0x40000000	0x80000001	0x4b125ce5	0x80000000
31	0x80000000	0x00000001	0xcb125ce5	0x00000000

FIG. 14. Iterative Example of Fourth Algorithm.

5. Final Observations.

5.1. PIPELINING.

Processors that support instruction pipelining may execute the algorithm more efficiently as follows since some sets of instructions can derive intermediate results independently.

```
      Path A                    Path B

      mov  ebx,[ebp+4]          mov  ecx,2
      mov  edx,ebx              mov  eax,1

_10:  shl  edx,1
      test ebx,ecx
      jz   _20

      add  ebx,edx              or   eax,ecx

_20:                            shl  ecx,1
                                jnc  _10
                                neg  eax
```

FIG. 15. Pipelining.

5.2. LOOP OPTIMIZATION.

.

Unrolling the iteration *i* loop and providing constant values for *i* for each *r* may improve the progression of values in *i*, if *b* is fixed.

```
r = a = n; t = 1;
r <<= 1; if( a & 0x00000002 ) { t |= 0x00000002; a += r };
r <<= 1; if( a & 0x00000004 ) { t |= 0x00000004; a += r };

//  ...
//  repeat statement, doubling the constant
//  ...

r <<= 1; if( a & 0x80000000 ) { t |= 0x80000000; };

return( t * -1 );
```

FIG. 16. Unrolling the Iteration Loop.

Note that in the final iteration, *r* need not be added to a since reference to *a* is no longer required.

This unrolling of *r*, while it eliminates both the shift if *i* and the comparison of *i* to zero, has the disadvantage of requiring an implementation that has an invariable *b*. Whereas, the looping variety, if implemented in a sufficiently high-level language, such as C, can function with any word size.

6. Implementations.

6.1. INTEL X86-32 (IA32) (AT&T MNEMONICS).

```
//    in:   edx = ODD n
//    out:  eax = 1/n MOD 2^32

      movl  %edx,%ebx         //  a := n
      movl  $2,%ecx           //  i := 2
      movl  $1,%eax           //  t := 1

_10:  shll  $1,%edx           //  r := ( r SHL 1 ) mod 2^s
      testl %ecx,%ebx         //  a AND i ?
      jz    _20               //  no, skip ahead

      orl   %ecx,%eax         //  t := t OR i
      addl  %edx,%ebx         //  a := ( a + r ) mod 2^s

_20:  shll  $1,%ecx           //  i := ( i SHL 1 ) mod 2^s
      jnc   _10               //  next i
```

FIG. 17. Fourth Algorithm for IA32 (AT&T Mnemonics).

6.2. INTEL X86-32 (IA32) (INTEL MNEMONICS).

```
;     in:   edx = ODD n
;     out:  eax = 1/n MOD 2^32

      mov   ebx,edx           ;   a := n
      mov   ecx,2             ;   i := 2
      mov   eax,1             ;   t := 1

_10:  shl   edx,1             ;   r := ( r SHL 1 ) mod 2^s
      test  ebx,ecx           ;   a & i ?
      jz    _20               ;   no, skip ahead

      or    eax,ecx           ;   t := t OR i
      add   ebx,edx           ;   a := ( a + r ) mod 2^s

_20:  shl   ecx,1             ;   i := ( i SHL 1 ) mod 2^s
      jnc   _10               ;   next i
```

FIG. 18. Fourth Algorithm for IA32 (Intel Mnemonics).

6.3. S/390 ASSEMBLY LANGUAGE.

```
*     in:   %4 = ODD n
*     out:  %1 = 1/n MOD 2^32

      lr    2,4
      la    3,2
      la    4,1

 L10  ds    0h

      sll   4,1
      lr    5,3
      nr    5,2
      bc    8,L20

      or    1,3
      ar    2,4

 L20  ds    0h

      sll   3,1
      or    3,3
      bc    8,L10
```

FIG. 19. Fourth Algorithm for S/390 Assembly Language.

6.4. C.

```
/*    in:   n (ODD)                                       */
/*    out:  t = 1/n MOD 2^sizeof( unsigned int )          */

unsigned int      r, a, t, i;

r = a = n;
t = 1;

for( i = 2; i; i <<= 1 ) {
      r <<= 1;
      if( a & i ) {
            t |= i;
            a += r;
      }
}

return( t );
```

FIG. 20. Fourth Algorithm for C.

7. References.

[1] W. Diffie and M.E. Hellman, New directions in cryptography, *IEEE Transactions on Information Theory* 22 (1976), 644-654.

[2] Dussé, S. R., B. S. Kaliski Jr., *A Cryptographic Library for the Motorola DSP56000*, Advances in Cryptology, Eurocrypt '90, Lecture Notes in Computer Science, Vol. 473, pp. 230-244, Springer-Verlag, 1990.

[3] Knuth, D.E., *The Art of Computer Programming 3rd. ed.*, Reading, Massachusetts, Addison-Wesley, 1997.

[4] Koç, Ç. K., T. Acar, B. S. Kaliski, Jr., *Analyzing and Comparing Montgomery Multiplication Algorithms*, IEEE Micro, Vol 16, Issue 3, pp. 26-33, June 1996.

[5] Menezes, A., P. van Oorschot and S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[6] Montgomery, P. L., *Multiplication Without Trial Division*, Mathematics of Computation, Vol. 44, pp. 519-521, 1985.

[7] *PKCS #1 v2.1: RSA Cryptography Standard*, RSA Laboratories, June 14, 2002.