New Algorithm for Classical Modular Inverse

Róbert Lórencz

Department of Computer Science and Engineering
Faculty of Electrical Engineering, Czech Technical University
Karlovo nám. 13, 121 35 Praha 2, Czech Republic
lorencz@fel.cvut.cz

Abstract. The Montgomery inverse is used in cryptography for the computation of modular inverse of b modulo a, where a is a prime. We analyse existing algorithms from the point of view of their hardware implementation. We propose a new, hardware-optimal algorithm for the calculation of the classical modular inverse. The left-shift binary algorithm is shown to naturally calculate the classical modular inverse in fewer operations than the algorithm derived from the Montgomery inverse.

1 Introduction

The basic arithmetic operations in modular arithmetic where the modulo is prime are a natural and inseparable part of cryptographic algorithms [6], [8], as well as nowadays often used elliptic curve cryptography [9], [10]. Modular inverse is especially important in computations of point operations on elliptic curves defined over a finite field $GF(p)$ [9], in acceleration of the exponentiation operation using the so-called addition-subtraction chain [11], [4], in Diffie-Hellman key exchange method [7], and in decipherment operations in RSA algorithm [6]. The modular inverse of an integer $a \in [1, p-1]$ modulo p, where p is prime, is defined as an integer $r \in [1, p-1]$ such that $a.r \equiv 1 \pmod{p}$, often written as

$$r = a^{-1} \bmod p. \tag{1}$$

This classical definition of the modular inverse and an algorithm for its calculation in a binary form is specified in [4]. Kaliski has extended the definition of the modular inverse to include the so-called Montgomery inverse [2]. The Montgomery inverse is based on the Montgomery multiplication algorithm [1]. The Montgomery inverse of an integer $a \in [1, p-1]$ is b such that

$$b = a^{-1} 2^n \bmod p, \tag{2}$$

where p is prime and $n = \lceil \log_2 p \rceil$. In this paper, we present a left-shift binary algorithm for the computation of the classical modular inverse which is more efficient than the algorithm derived from the Montgomery modular inverse algorithm [2], [3] and the ordinary inverse algorithm [4].
Our incentive for the search of effective computation of modular inverse was,

besides the above facts, to use it in a modular system for solving systems of linear equations [16], [17]. In the whole paper, we assume that least significant bit (LSB) is the rightmost position.

2 The Classical Modular Inverse in Previous Works

The two commonly used approaches for the computation of ordinary modular inverse are a binary algorithm derived from the Montgomery modular inverse [2], [3] and a binary algorithm for ordinary inverse [4]. Both of these approaches are based on the same algorithmic principle, which is the binary right-shift greatest common divisor algorithm (gcd) [4] that calculates the value for two integers using halving and subtraction. Both of the mentioned algorithms are suitable for implementation in hardware, since the halving operation is equal to a binary right shift.

2.1 The Right-Shift Algorithm for the Classical Modular Inverse

The ordinary modular inverse algorithm described in [4], attributed to M.Penk (see exercise 4.5.2.39 in [4]), calculates the modular inverse $r = a^{-1} \bmod p$ using the extended Euclidean algorithm. We have modified the Penk's algorithm with the aim to enable its easy hardware implementation. The modified Penk's algorithm, called Algorithm I, is given below:

Algorithm I

```
Input: $a \in [1, p-1]$ and $p$
Output: $r \in [1, p-1]$ and $k$, where $r = a^{-1} \bmod p$
        and $n \leq k \leq 2n$
1.  $u := p, v := a, r := 0, s := 1$
2.  $k := 0$
3.  while ($v > 0$)
4.      if ($u$ is even) then
5.          if ($r$ is even) then
6.              $u := u/2$, $r := r/2$, $k := k + 1$
7.          else
8.              $u := u/2$, $r := (r + p)/2$, $k := k + 1$
9.      else if ($v$ is even) then
10.         if ($s$ is even) then
11.             $v := v/2$, $s := s/2$, $k := k + 1$
12.         else
13.             $v := v/2$, $s := (s + p)/2$, $k := k + 1$
14.     else $x := (u - v)$
15.         if ($x > 0$) then
16.             $u := x$, $r := r - s$
17.             if ($r < 0$) then
18.                 $r := r + p$
```

```
19.         else
20.              $v := -x$, $s := s - r$
21.              if ($s < 0$) then
22.                   $s := s + p$
23. if ($r > p$) then
24.     $r := r - p$
25. if ($r < 0$) then
26.     $r := r + p$
27. return $r$ and $k$.
```

Algorithm I continuously halves (shifts to the right) both values, even and odd; if the value is odd, the modulus p which is odd (p is prime) is added to it beforehand. These operations are performed in steps 8 and 13. Any negative values of r and s that result from the subtraction are converted to positive ones in the same residue class in steps 18 and 22 by adding p so that $r, s \in [1, p-1]$. The algorithm outputs two integers, r and k, where k is the number of halvings during the calculation of $\gcd(p, a)$ and it satisfies $n \leq k \leq 2n$.

2.2 The Montgomery Algorithm for the Classical Modular Inverse

In contrast to Algorithm I, Montgomery algorithms for computing modular inverse (in integer or Montgomery domains) split the computation to two phases. In the first phase the so-called Almost Montgomery Inverse $a^{-1}2^k \bmod p$ [3] is calculated in k iterations, where k follows from input values. In case of integer domain, k is taken to be the number of deferred halvings in the second phase [3]. Hence, the modular inverse according to Equation (1) is computed by k hlavings modulo p. This algorithm, called Algorithm II, is given below:

Algorithm II

```
Phase I
Input: $a \in [1, p-1]$ and $p$
Output: $y \in [1, p-1]$ and $k$, where $y = a^{-1}2^k \pmod{p}$
        and $n \leq k \leq 2n$
1.  $u := p, v := a, r := 0, s := 1$
2.  $k := 0$
3.  while ($v > 0$)
4.      if ($u$ is even) then
5.          $u := u/2$, $s := 2s$, $k := k + 1$
6.      else (if $v$ even) then
7.          $v := v/2$, $r := 2r$, $k := k + 1$
8.      else
9.          $x := (u - v)$
10.         if ($x > 0$) then
11.              $u := x/2$, $r := r + s$, $s := 2s$, $k := k + 1$
12.         else
```

```
13.                 $v := -x/2$, $s := r + s$, $r := 2r$, $k := k + 1$
14. if ($r > p$) then
15.     $r := r - p$
16. return $y := p - r$ and $k$.
```

Phase II
Input: $y \in [1, p-1], p$ and k from Phase I
Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod{p}$, and $2k$ from Phase I

```
17. for ($i = 1$ to $k$) do
18.     if ($r$ is even) then
19.         $r := r/2$
20.     else
21.         $r := (r + p)/2$
22. return $r$ and $2k$.
```

In case of Montgomery domain, the number of deferred halvings in the second phase is $k - n$, where k is guaranteed to $n \leq k \leq 2n$ [2]. It is interesting to compare Algorithms I and II. The operation of the correction of an odd number before halving performed in steps 8 and 13 of Algorithm I is done in step 21 of Phase II of Algorithm II. Conversion of negative values of r and s is not necessary here, since no subtraction of r or s is performed during calculation. It is clear that the number of iterations in Algorithm II is $2k$, k iterations in Phase I and k iterations in Phase II.

3 New Left-Shift Algorithm for the Classical Modular Inverse

The new approach to the calculation of modular inverse, which is the subject of this paper, avoids the drawbacks of the above algorithms. In Algorithm I, these are especially: high number of tests such as '$u > v$', '$s < 0$', '$r < 0$', which essentially represent a subtraction and also an addition '$r + p$', '$s + p$' if s and r are negative so that $r, s \in [1, p-1]$. In case of the modular inverse calculation using the Montgomery modular inverse, it is necessary to perform the deferred halving in k iterations in Phase II of Algorithm II, including corrections of r if it is odd (step 21 of the algorithm). An algorithm that avoids the mentioned problems is presented below:

Algorithm III

Input: $a \in [1, p-1]$ and p
Output: $r \in [1, p-1]$, where $r = a^{-1} \pmod{p}$, c_u, c_v
        and $0 < c\_v + c\_u \leq 2n$

```
1. $u := p, v := a, r := 0, s := 1$
2. $c\_u = 0$, $c\_v = 0$
3. while($u \neq \pm 2^{c\_u}$ & $v \neq \pm 2^{c\_v}$)
4.     if ($u_n, u_{n-1} = 0$) or ($u_n, u_{n-1} = 1$ & $\mathrm{OR}(u_{n-2}, \ldots, u_0) = 1$) then
```

```
5.          if ($c\_u \geq c\_v$) then
6.              $u := 2u$, $r := 2r$, $c\_u := c\_u + 1$
7.          else
8.              $u := 2u$, $s := s/2$, $c\_u := c\_u + 1$
9.      else if ($v_n, v_{n-1} = 0$) $or$ ($v_n, v_{n-1} = 1$ & OR($v_{n-2}, \ldots, v_0$) = 1) then
10.         if ($c\_v \geq c\_u$) then
11.             $v := 2v$, $s := 2s$, $c\_v := c\_v + 1$
12.         else
13.             $v := 2v$, $r := r/2$, $c\_v := c\_v + 1$
14.     else
15.         if ($v_n = u_n$) then
16.             $oper = " - "$
17.         else
18.             $oper = " + "$
19.         if ($c\_u \leq c\_v$) then
20.             $u := u$ $oper$ $v$, $r := r$ $oper$ $s$
21.         else
22.             $v := v$ $oper$ $u$, $s := s$ $oper$ $r$
23. if ($v = \pm 2^{c\_v}$) then
24.     $r := s$, $u_n := v_n$
25. if ($u_n = 1$) then
26.     if ($r < 0$) then
27.         $r := -r$
28.     else
29.         $r := p - r$
30. if ($r < 0$) then
31.     $r := r + p$
32. return $r$, $c\_u$, and $c\_v$.
```

Algorithm III was designed to be easily implemented in hardware (Section 5). Registers Ru, Rv, Rs are $m = n + 1$ bit wide registers and contain individual values of the variables u, v, s. The value of variable r is in $m+1$ bit wide register Rr. Counters Cu and Cv are auxiliary $e = \lceil \log_2 n \rceil$ bit wide counters containing values c_u and c_v. The presented left-shifting binary algorithm computes the modular inverse of a according to Equation (1)) using the extended Euclidean algorithm and shifting the values u and v to the left, that is multiplying them by two. The multiplication is performed as long as the original value multiplied by 2^i is preserved, where i is the number of left shifts. Negative values are represented in the two's complement code. The shift is performed as long as the bits u_n, u_{n-1} or v_n, v_{n-1} are zeros for positive values or ones for negative values, while at least one of the bits u_{n-2}, u_{n-3}, $\ldots u_0$ or v_{n-2}, v_{n-3}, $\ldots v_0$ is not zero - binary 'OR' (steps 4 and 9). With each shift, counters Cu and Cv (values c_u and c_v) that track the number of shifts in Ru, Rv are incremented (steps 6, 8, 11, and 13). Registers Rr and Rs (values r and s) are also shifted to the right (steps 8 and 13) or left (steps 6 and 11) according to conditions in steps 5 and 10. In step 15, addition or subtraction, given variable $oper$, is selected according to sign bits u_n and v_n for the subsequent reduction of u, v and r, s in steps 20 and 22. Results of these operations are stored either in Ru and Rr (values u and r), if the number of shifts in Ru is less or equal to the number of shifts in Rv, or in registers Rv and Rs (values v and s) otherwise. The loop ends whenever '1' or '-1' shifted by the appropriate number of bits to the left appears in register Ru or Rv. Branch conditions used in steps 4, 9, and 15 are easily implemented in hardware. Similarly, the test in steps 5, 10, and 19 can be implemented by an e bit comparator of values *c_u* and *c_v* with two auxiliary single-bit flips-flops u/v̄ and wu (see Section 5). Table 1 shows an example of the calculation of the

Table 1. Example of the calculation

l	*operations*	*values of registers*	*tests*
0		$u^{(0)} = (13)_{10} = (01010.)_2$	$u^{(0)} \neq \pm 2^0$
		$v^{(0)} = (10)_{10} = (01010.)_2$	$v^{(0)} \neq \pm 2^0$
		$r^{(0)} = (0)_{10} = (00000.)_2$	
		$s^{(0)} = (1)_{10} = (00001.)_2$	
1	$u^{(1)} = u^{(0)} - v^{(0)}$	$u^{(1)} = (3)_{10} = (00011.)_2$	$u^{(1)} \neq \pm 2^0$
		$v^{(1)} = (10)_{10} = (01010.)_2$	$v^{(1)} \neq \pm 2^0$
	$r^{(1)} = r^{(0)} - s^{(0)}$	$r^{(1)} = (-1)_{10} = (11111.)_2$	
		$s^{(1)} = (1)_{10} = (00001.)_2$	
2	$u^{(2)} = 4u^{(1)}$	$u^{(2)} = (12)_{10} = (011.00)_2$	$u^{(2)} \neq \pm 2^2$
		$v^{(2)} = (10)_{10} = (01010.)_2$	$v^{(2)} \neq \pm 2^0$
	$r^{(2)} = 4r^{(1)}$	$r^{(2)} = (-4)_{10} = (111.00)_2$	
		$s^{(2)} = (1)_{10} = (00001.)_2$	
3		$u^{(3)} = (12)_{10} = (011.00)_2$	$u^{(3)} \neq \pm 2^2$
	$v^{(3)} = v^{(2)} - u^{(2)}$	$v^{(3)} = (-2)_{10} = (11110.)_2$	$v^{(3)} \neq \pm 2^0$
		$r^{(3)} = (-4)_{10} = (111.00)_2$	
	$s^{(3)} = s^{(2)} - r^{(2)}$	$s^{(3)} = (5)_{10} = (00101.)_2$	
4		$u^{(4)} = (12)_{10} = (011.00)_2$	$u^{(4)} \neq \pm 2^2$
	$v^{(4)} = 4v^{(3)}$	$v^{(4)} = (-8)_{10} = (110.00)_2$	$v^{(4)} \neq \pm 2^2$
	$r^{(4)} = r^{(3)}/4$	$r^{(4)} = (-1)_{10} = (11111.)_2$	
		$s^{(4)} = (5)_{10} = (00101.)_2$	
5	$u^{(5)} = u^{(4)} + v^{(4)}$	$u^{(5)} = (4)_{10} = (001.00)_2$	$u^{(5)} = 2^2$
	$r^{(5)} = r^{(4)} + s^{(4)}$	$r^{(5)} = (4)_{10} = (00100.)_2$	

modular inverse for $p = 13$ and $a = 10$. Therefore, $n = 4$ and $m = 5$. The computed result is $r = a^{-1} \bmod 13 = 4$.

Description of Table 1: l is the iteration number, column *operations* lists the performed arithmetic operations of iteration l and column *tests* lists conditions evaluated in iteration l . The notation $u^{(l)}$ means the actual value u of register Ru in the l-th iteration, etc. The dot in binary representation of values in column *values of registers* specifies the reference shift position that is equal to the current position of initial LSB. It represents the value of accumulated left-shift for u and v and left/right shift for r and s.

4 Results and Discussion

A simulation and a quantitative analysis of the number of additions or subtractions ('+/−'), shifts and tests was performed for all the algorithms presented. Simulation of modular inverse computation according to Equation (1) was performed for all integers $a \in [2, p-1]$ and all 1899 prime moduli $p < 2^{14}$ $(n \leq 14)$. A total of 14,580,841 inverses were computed by each method. Simulation results are presented in Table 2. The number of all tests, additions and subtractions are listed in column "+/− & *tests*". The tests include all "greater than" and "less than" comparisons except '$v > 0$' in the main loop, which is essentially a '$v \neq 0$' test that does not require a subtraction. The "+/−" column lists additions and subtractions without tests. The column "*total shift*" indicates the number of all shift operations during the computation. The last column lists the number of shifts minus the number of '+/−' operations, assuming the shift is performed together with storing the result of the '+/−' operation. The columns give minimum and maximum numbers of operations ($min; max$) and their average ($av.$) values.

Table 2. Results for primes less than 2^{14}

Algorithm	*+/− & tests*		*+/−*		*total shift*		*shift − (+/−)*	
	min; max	*av.*	*min; max*	*av.*	*min; max*	*av.*	*min; max*	*av.*
Algorithm III	-	-	2 - 21	9.9	2 - 26	23.3	1 - 24	13.4
Algorithm II Ph. I	3 - 28	15.7	1 - 15	10.6	3 - 27	19.1	0 - 23	8.5
Algorithm II Ph. II	2 - 25	10.5	2 - 25	10.5	3 - 27	19.1	0 - 24	8.6
Algorithm II	5 - 45	26.2	4 - 40	21.1	6 - 54	38.2	0 - 43	17.1
Algorithm I	9 - 80	40.4	6 - 53	27.1	2 - 26	18.1	0	0

Shift operations are faster in hardware than additions, subtractions, and comparison operations performed with the Ru, Rv, Rr, Rs registers. The comparison operations are about as slow as additions/subtractions, since they cannot be performed in parallel; they depend on data from previous operations. In Algorithm I, they are the conditions in steps 15, 17, and 21. If a suitable code is used to represent negative numbers, this condition can be realized as a simple sign test, avoiding the complicated testing in steps 17 and 21. The Algorithms I and Algorithm II suffer from a large number of additions and subtractions that correct odd numbers before halving in steps 8 and 13 of Algorithm I and step 21 in Algorithm II, and convert negative numbers in steps 18 and 22 of Algorithm I.

Moreover, Algorithm II needs twice the number of shifts compared to Algorithm I, required by the Phase II.

The previous analysis shows that Algorithm III removes drawbacks of Algorithm I and Algorithm II. Let us assume full hardware support for each algorithm and simultaneous execution of operations specified on the same line of the pseudocodes. Let us further assume that no test is needed in step 10 of Algorithm II. Then, we can use the values in columns "+/−" and "*total shift*" to compare the number of operations. Algorithm III needs half the number of '+/−' operations compared to Algorithm II, and 2.7 times less the number of '+/−' operations compared to Algorithm I.

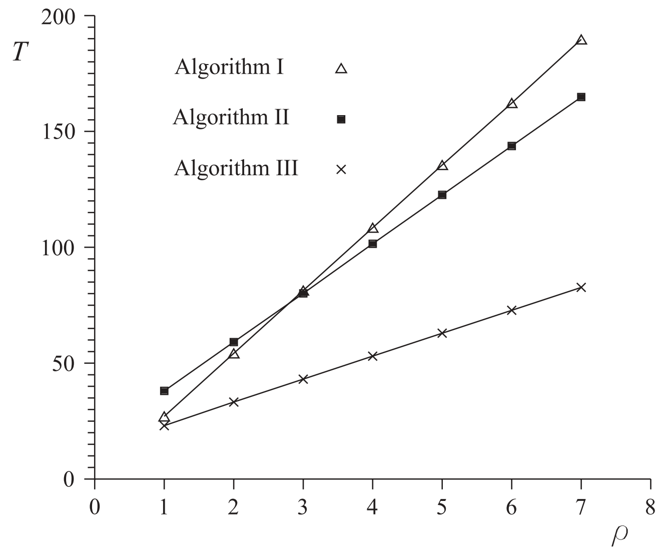

Fig. 1. Average number of execution cycles T of the three algorithms as a function of the ratio ρ

The simulation results from Table 2 for prime moduli less than 2^{14} ($n = 14$) are plotted in Figure 1. It shows the average number of cycles T needed to compute the modular inverse using Algorithms I - III as a function of the ratio ρ, where ρ is defined as the ratio of the critical path length in cycles of the shift and the critical path length in cycles of the adder/subtracter. All (Algorithm I) or a part of (Algorithms II and III) shift operations are included in '+/−' operations. Shift operations that are not performed as a part of '+/−' operations are performed individually (they are listed in the last column of Table 2).

With an increasing word length, the time complexity of shift operations remains constant. However, the complexity of additions/subtractions increases approximately $\lceil \log_2 m \rceil$ - times, m is the number of bits of a word. For long words, often used in cryptographic algorithms, the modular inverse computation for individually algorithms is strongly dependent on addition/subtraction operations. That in such cases Algorithm III is twice faster than Algorithm II and 2.7-times faster than Algorithm I.

Table 3. Results of Algorithm III for three cryptographic primes

Primes	n	$+/-$		*total shift*		*inverses*
		min; max	*av.*	*min; max*	*av.*	
$2^{192} - 2^{64} - 1$	192	64 - 182	132.9	343 - 382	380	3,929,880
$2^{224} - 2^{96} + 1$	224	81 - 213	154.8	408 - 446	441	4,782,054
$2^{521} - 1$	521	18 - 472	387.5	999 - 1040	1029	4,311,179

The statistical analysis of Algorithm III (see the previous page) for large integers of cryptographic was performed. The results of the analysis are presented in Table 3. The first column contains values of primes, the second column gives the word length and the last column gives number of inverses. Other columns have the same meaning as columns in Table 2. The average number of '$+/-$' operations grows with n approximately linearly. The multiplicative coefficient is ≈ 0.7 for all three primes. The average number of shifts is nearly equal to $2n$. Similar results hold for primes $p < 2^{14}$.

5 HW Implementation

Algorithm III is optimized in terms of reducing the number of additions and subtractions, which are critical in integer arithmetic due to carry propagation in long computer words. Other optimization criteria included making the evaluation of tests during the calculation as simple as possible and minimizing data dependencies to enable calculation in parallel calculations. Figure 2 shows the circuit implementing the computation of classical modular inverse . Only data paths for computing the classical modular inverse according to Algorithm III are shown. The system consists of three basic parts. The first two parts form the well-known "butterfly" [14], [15], typical for algorithms based on the extended Euclidean algorithm; the third part consists of the controller and support circuitry. "Master" half of the "butterfly" calculates the $\gcd(m, a)$ and consists of two m bit registers Ru, Rv, m bit adder/subtracter ADD1, multiplexer MUX1, and left-shift logic SHFT1. "Slave" half of the butterfly consists of $(m + 1)$ bit register Rr and m bit register Rs, m bit adder/subtracter ADD2, multiplexers MUX2, MUX3, MUX4, and right/left-shift logic SHFT1. The controller unit controls the operation of the entire system. The controller part also includes an m bit mask register Rm with test logic provided test in step 3, two e bit counters Cu, Cv with the comparator d and single-bit flip-flops u/$\bar{\text{v}}$ and wu.

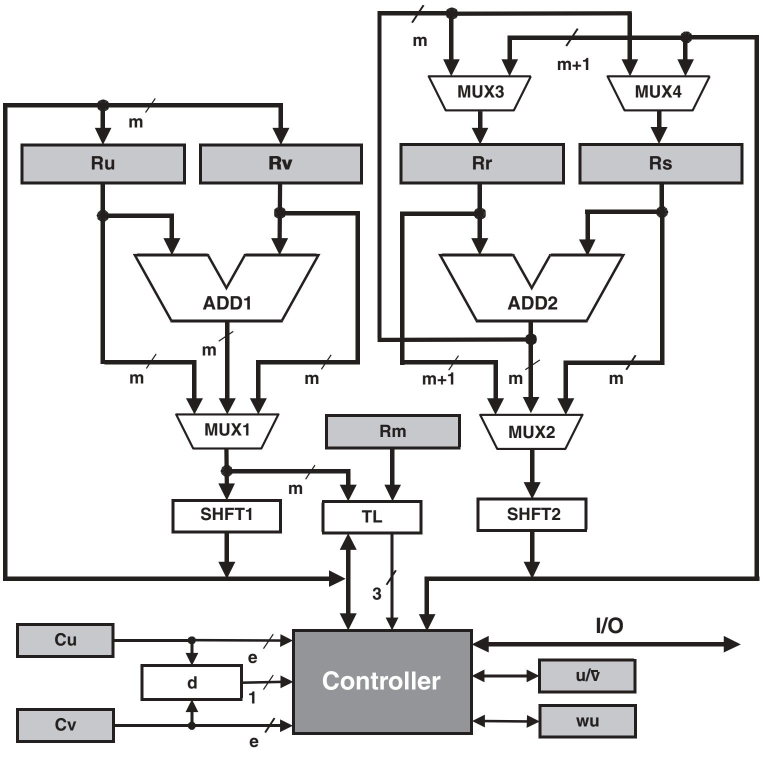

Fig. 2. The circuit implementation of Algorithm III

6 Conclusion

A new algorithm (Algorithm III) for classical modular inverse was presented and its HW implementation has been proposed. A mathematical proof (see the Appendix) that the proposed algorithm really computes classical modular inverse was performed. A statistical analysis of the new algorithm for large cryptographic integers was carried out. Computation of the modular inverse using the new algorithm is always faster and in the case of long words at least twice faster than other algorithms currently in use. The principles of the presented algorithm will also be used in a modular system for solving systems of linear equations without rounding errors [17].

References

1. P. L. Montgomery: Modular Multiplication Without Trial Division. Mathematics of Computation **44** No. 170 (1985) 519–521
2. B. S. Kaliski Jr.: The Montgomery Inverse and Its Application. IEEE Transaction on Computers **44** No. 8 (1995) 1064–1065

3. E. Savaş and Ç. K. Koç: The Montgomery Modular Inverse - Revisited. IEEE Transaction on Computers **49** No. 7 (2000)
4. D. E. Knuth: The Art of Computer Programming **2** / Seminumerical Algorithms. Addison-Wesley, Reading, Mass. Third edition (1998)
5. Ç. K. Koç: High-Radix and Bit Recoding Techniques For Modular Exponentiation. Int'l J. Computer Mathematics **40** (1991) 139–156
6. J.-J. Quisquarter and C. Couvreur: Fast Decipherment Algorithm for RSA Public-key Cryptosystem. Electronics Letters **18** No. 21 (1982) 905–907
7. W. Diffie and M. E. Hellman: New Directions in Cryptography. IEEE Transactions on Information Theory **22** (1976) 644–654.
8. Nat'l Inst. of Standards and Technology (NIST). FIPS Publication 186: Digital Signature Standard (1994)
9. N. Koblitz: Elliptic Curve Cryptosystem. Mathematics of Computation **48** No. 177 (1987) 203–209
10. A. J. Menezes: Elliptic curve Public Key Cryptosystem. Kluwer Academic Publishers, Boston, MA (1993)
11. Ö. Eğecioğlu and Ç. K. Koç: Exponentiation Using Canonical recoding. Theoretical Computer Science **129** No. 2 (1994) 407–717
12. R. T. Gregory and E. V. Krishnamurthy: Methods and Applications of Error-free Computation. Springer-Verlag, New York, Berlin, Heidelberg, Tokyo (1984)
13. K. H. Rosen: Elementary Number Theory and Its Applications. Addison-Wesley, Reading, Massachusetts (1993)
14. J. D. Dworkin, P. M. Glaser, M. J. Torla, A. Vadekar, R. J. Lambert, S. A. Vanstone: Finite Field Inverse Circuit. US Patent 6,009,450 (1999)
15. B. Bruner, A. Curiger, M. Hofstetter: On Computing Multiplicative Inverse in GF(2^m). IEEE Trans. Computer **42** (1993) 1010–1015
16. M. Morháč and R. Lórencz: A Modular System for Solving Linear Equations Exactly, I. Architecture and Numerical Algorithms. Computers and Artificial Intelligence **11** No. 4 (1992) 351–361
17. R. Lórencz and M. Morháč: Modular System for Solving Linear Equations Exactly, II. Hardware Realization. Computers and Artificial Intelligence **11** No. 5 (1992) 497–507

7 Appendix: The Mathematical Proof of Proposed Algorithm

Algorithm III has similar properties as the Euclidean Algorithm and algorithms derived from it, introduced in [4], [12], [13], where methods for their verification are also presented. By using similar proof techniques we have carried out a proof that Algorithm III computes correctly the classical modular inverse.

For computing multiplicative inverse of an integer a in the finite field GF(p), where p is a prime, the following lemma is important.

Lemma 1. *If* $\gcd(p, a) = 1$ *and if*

$$1 = px + ab,$$

then

$$a^{-1} \bmod p = b \bmod p.$$

The proof of the lemma is in [12]. By finding a pair of integers x and b that satisfy equations in Lemma 1, we prove that Algorithm III computes classical inverse in GF(p). Individual iterations of Algorithm III can be described by following system 3 of recurrent equations and guarding conditions for quotients,

$$
\begin{array}{llllll}
r_1 & = p - aq_1 & 0 < r_1 < aq_1 & q_1 & = 2^{(\langle p\rangle - \langle a\rangle)} & q_1 > 1 \\
r_2 & = |r_1| - aq_2 & 0 < r_2 < aq_2 & q_2 & = 2^{(\langle r_1\rangle - \langle a\rangle)} & q_2 > 1 \\
 & \vdots & & & & \\
r_j & = |r_{j-1}| - aq_j & 0 < |r_j| < a & q_j & = 2^{(\langle r_{j-1}\rangle - \langle a\rangle)} & q_j > 1 \\
\hline
r_{j+1} & = a - |r_j|q_{j+1} & 0 < |r_{j+1}| < |r_j|q_{j+1} & q_{j+1} & = 2^{(\langle a\rangle - \langle r_j\rangle)} & q_{j+1} > 1 \\
r_{j+2} & = |r_{j+1}| - |r_j|q_{j+2} & 0 < |r_{j+2}| < |r_j|q_{j+2} & q_{j+2} & = 2^{(\langle r_{j+1}\rangle - \langle r_j\rangle)} & q_{j+2} > 1 \\
 & \vdots & & & & \\
r_k & = |r_{k-1}| - |r_j|q_k & 0 < |r_k| < |r_j| & q_k & = 2^{(\langle r_{k-1}\rangle - \langle r_j\rangle)} & q_k > 1 \\
\hline
r_{k+1} & = |r_j| - |r_k|q_{k+1} & 0 < |r_{k+1}| < |r_k|q_{k+1} & q_{k+1} & = 2^{(\langle r_j\rangle - \langle r_k\rangle)} & q_{k+1} > 1 \\
r_{k+2} & = |r_{k+1}| - |r_k|q_{k+2} & 0 < |r_{k+2}| < |r_k|q_{k+2} & q_{k+2} & = 2^{(\langle r_{k+1}\rangle - \langle r_k\rangle)} & q_{k+2} > 1 \\
 & \vdots & & & & \\
r_l & = |r_{l-1}| - |r_k|q_l & 0 < |r_l| < |r_k| & q_l & = 2^{(\langle r_{l-1}\rangle - \langle r_k\rangle)} & q_l > 1 \\
\hline
 & \vdots & & & & \\
\hline
 & \vdots & & & & \\
r_m & = \dots & & & & \\
\hline
r_{m+1} & = \dots & & & & \\
 & \vdots & & & & \\
r_n & = |r_{n-1}| - |r_m| & 0 < |r_n| < |r_{n-1}| & q_n & = 2^{(\langle r_{n-1}\rangle - \langle r_m\rangle)} & q_n = 1 \\
\hline
r_{n+1} & = |r_{n-1}| - |r_n|q_{n+1} & 0 < |r_{n+1}| < |r_n|q_{n+1} & q_{n+1} & = 2^{(\langle r_{n-1}\rangle - \langle r_n\rangle)} & q_{n+1} > 1 \\
r_{n+2} & = |r_{n+1}| - |r_n|q_{n+2} & 0 < |r_{n+2}| < |r_n|q_{n+2} & q_{n+2} & = 2^{(\langle r_{n+1}\rangle - \langle r_n\rangle)} & q_{n+2} > 1 \\
 & \vdots & & & & \\
r_o & = |r_{o-1}| - |r_n|q_o & |r_o| = 1 & q_o & = 2^{(\langle r_{o-1}\rangle - \langle r_n\rangle)} & q_o > 1 \\
\hline
0 & = |r_n| - |r_o|q_{o+1}, & & & &
\end{array}
\tag{3}
$$

where $r_1, r_2, \dots, r_{o+1}$ are remainders, $q_1, q_2, \dots, q_{o+1}$ are quotients, $\langle r_i\rangle$ is the number of bits needed for binary representation $|r_i|$. If the recursive definition of r_i is unrolled up to p and a, each r_i can be expressed by a Diophantine equation $r_i = pf_i + aq_i$, where $f_i = f_i(q_1, q_2, \dots, q_i)$, and $g_i = g_i(q_1, q_2, \dots, q_i)$.

The last non-zero remainder equal r_o fulfils the following theorem:

Theorem 1. $\gcd(p, a) = 1$ *iff* $|r_o| = 1$.

Proof.

$$\begin{aligned}
\gcd(p,a) &= \gcd(r_1,a) = \gcd(r_2,a) = \ldots = \gcd(r_{j-1},a)\\
&= \gcd(a,|r_j|) = \gcd(|r_{j+1}|,|r_j|) = \ldots = \gcd(|r_{k-1}|,|r_j|)\\
&= \gcd(|r_j|,|r_k|) = \gcd(|r_{k+1}|,|r_k|) = \ldots = \gcd(|r_{l-1}|,|r_k|)\\
&= \gcd(|r_k|,|r_l|) = \ldots\\
&\vdots\\
&= \ldots = \gcd(|r_{n-1}|,|r_m|) = \gcd(|r_n|,|r_m|) = \gcd(|r_{n-1}|,|r_n|)\\
&= \gcd(|r_{n-1}|,|r_n|) = \gcd(|r_{n+1}|,|r_n|) = \ldots = gcd(|r_{o-1}|,|r_n|)\\
&= \gcd(|r_n|,|r_o|) = \gcd(|r_o|,0)\\
&= |r_o| = 1.
\end{aligned}$$

□

The previous statement assumed the following trivial properties of gcd:

$$\begin{aligned}
\gcd(0,d) &= |d| \text{ for } d \neq 0,\\
\gcd(c,d) &= \gcd(d,c),\\
\gcd(c,d) &= \gcd(|c|,|d|),\\
\gcd(c,d) &= \gcd(c+ed,d),
\end{aligned}$$

where c, d, and e are integers. The description of the properties are introduced in [12], [13].

The fact that the guarding conditions for quotients q_i guarantee correct values of remainders r_i follows from the Lemma 2:

Lemma 2. *Let c and d be positive integers with binary representations $c = 2^i + c_{i-1}2^{i-1} + \ldots + c_0$ and $d = 2^j + d_{j-1}2^{j-1} + \ldots + d_0$. Assume $i \geq j$. Let $q = 2^{(i-j)}$ and $e = c - qd$. Then:*

$$|e| < qd \text{ and } |e| < c.$$

Proof. Follows easily from identity

$$\sum_{k=1}^{i} \frac{1}{2^k} = 1 - \frac{1}{2^i},$$

which is proven for example in [13]. □

The computation specified in Equations (3) can be expressed in the form of Table 4, which gives the expressions for integer values f_i and g_i.

Since $r_o = pf_o + ag_o$, it follows that:

if $(r_o = 1)$ then
  $x = f_o$, $b = g_o$, and $a^{-1} \bmod p = g_o \bmod p$,
if $(r_o = -1)$ then
  $x = -f_o$, $b = -g_o$, and $a^{-1} \bmod p = (-g_o) \bmod p$.

Table 4. The computation of f_o, g_o, and r_o

i	r_i	f_i	g_i
1	r_1	1	$-q_1$
2	r_2	± 1	$\pm q_1 - q_2$
3	r_3	± 1	$\pm q_1 \pm q_2 - q_3$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
j	r_j	± 1	$\pm q_1 \pm q_2 \pm \ldots - q_j$
$j+1$	r_j	$\pm q_{j+1}$	$1 \pm q_{j+1}(\pm q_1 \pm q_2 \pm \ldots - q_j)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
k	r_k	f_k	g_k
$\vdots$	$\vdots$	$\vdots$	$\vdots$
o	r_o	f_o	g_o
$o+1$	0	f_{o+1}	g_{o+1}.

□

It is the last step of the proof that Algorithm computes classical modular inverse. Finally, we take note of the principal features of the proposed Algorithm III from the point of view of the presented proof. The algorithm employs two's complement code for additions or subtractions. Therefore, the test whether an addition or subtraction is to be performed becomes a simple sign test. If the signs of both operands are equal, we subtract one from the other and in the opposite case we add both operands. Equations in (3) respect this rule when using absolute values of operands. According to Lemma 2, successive values r_i of remainders decrease in such a way that $p > a > |r_1| > |r_2| > \ldots |r_o|$.

The selection of operands which will be rewritten with a new value of the computed remainder is based on a simple test. The write is performed into the operand which needs more bits for its binary representation. This fact is respected in (3) by the value q_i. If $q_i = 1$, the write is into one of operands without respect on their absolute values. This case is demonstrated for remainder r_n. The conditions given by inequalities of Lemma 2 for r_n are fulfilled, too. Hence it holds $p > a > |r_1| > |r_2| > \ldots > |r_{n-1}| > |r_n| > |r_{n+1}| > \ldots > |r_o|$.