

Modular Inverse Algorithms without Multiplications

Version 1.0. April 13–September 2, 2004

Laszlo Hars, Seagate Research (Laszlo@Hars.US)

Abstract the basic left-shift, right-shift and shifting Euclidean modular inverse algorithms are presented with new optimization tricks. These algorithms are based on the corresponding extended GCD algorithms, but only one multiplier, the modular inverse is computed. On many computational platforms, for operand lengths used in cryptography, the fastest modular inverse algorithms need about *twice* the modular multiplication time, or even less. This indicates that on these platforms affine coordinates are the best in elliptic curve cryptography. Some simple HW enhancements are also described, which allow acceleration of the computation for very little cost.

Keywords: *Computer Arithmetic, Cryptography, RSA cryptosystem, Elliptic curve cryptosystem, Modular multiplication, Binary Extended Greatest Common Divisor algorithm (GCD), Modular Inverse, Optimization*

Notations

- GCD: Greatest Common Divisor.
- xGCD or *extended GCD*: the algorithm calculating g and also two multipliers c and d : $[g, c, d] = \text{xCGD}(x, y)$, such that the greatest common divisor of x and y , $g = c \cdot x + d \cdot y$
- $\|M\|$ the number of bits in the integer M , its binary length
- $m = \{m_{n-1} \dots m_1, m_0\} = m_{n-1} \dots 0 = \sum_{i=0}^{n-1} 2^i m_i$, $m_i \in \{0, 1\}$ the bits of the integer m

Introduction

The inverse of long integers in modular rings corresponds to the reciprocal of real numbers. It is used extensively, among others, for RSA public key cryptography, in:

- Primality test (excluding small prime divisors). If a random number has no modular inverse in respect to the product of many small primes, then $\text{GCD}(\text{RandNum}, \prod \text{prime}_i) \neq 1$, and the random number is not prime.
- Public key generation: computing the inverse of the private key.
- Preparing for CRT (Chinese Remainder Theorem based exponentiation speedup): the pre-calculated half-size constant $C_2 = p^{-1} \bmod q$ (where the public modulus $m = p \cdot q$) helps accelerating the modular exponentiation about 4-fold. [7]
- Signed bit exponent recoding: expressing the exponent with positive and negative bits facilitates the reduction of the number of non-zero bits and many multiplications can be saved in the exponentiation chain. In the multiply-square binary exponentiation algorithm the inverse of the message $a^{-1} \bmod m$, which almost always exists, is multiplied to the partial result at negative exponent bits. [12]

Also, in prime field elliptic or hyper elliptic curve cryptosystems modular inverses are used during point addition, doubling and multiplication. [8]

There are situations where this modular inverse has to be or is better calculated without multiplications. These include

- If the multiplier hardware is slow.
- If there is no short multiplier circuit at all. For example, on computational platforms where long parallel adders perform multiplications by repeated shift-add operations. (For the design of fast adders see [10]: carry-lookahead, carry-save, carry-skip etc. architectures.)
- For RSA key generation in cryptographic processors, where the multiplier circuit is used in the background for the exponentiations of the primality (Miller-Rabin) test. [7]

- In prime field elliptic or hyper elliptic curve cryptosystems, where the inversion can be performed parallel to other calculations involving multiplications.

Of course, there are computational platforms, where multiplications are better used for modular inverse calculation. These include workstations with very fast or multiple multiplier engines (e.g. three, like in the ALU, Floating point multiplier and Multimedia Extension engine).

Below we discuss different variants of three extended GCD algorithms modified for computing modular inverses. We deal with operand lengths used in public key cryptography (128...8K bits) and describe some simple HW enhancements speeding up these algorithms in digit serial HW architectures. The considered algorithms run in quadratic time: $O(n^2)$ for n -bit input. For very long operands more complex algorithms, such as Schönhage's half-GCD algorithm [5], get faster, having $O(n \log^2 n)$ running time, but for operand lengths used in cryptography they are far too slow (see [6]).

Results

Three families of algorithms are presented, based on the left-shift binary-, the right-shift binary- and the shifting Euclidean extended GCD algorithms, with justifications of their correctness and considerations about their relative performance. The absolute speed of the algorithms has been determined experimentally. Extensive simulations were performed with several million random inputs from 16 to 1024 bit length. Linear and quadratic functions fit very well to the number of iterations and to the number of bit-operations, respectively. The leading coefficients of these approximation polynomials are tabulated, showing that the speed ratio between the slowest (plus-minus right shift algorithm RS+/-) and the fastest (modified left-shift algorithm LS3 or the optimized shifting Euclidean algorithm SE3) can be almost three fold, dependent on the computational platform. The number of operations the discussed algorithms performed has been determined in the bit level, using a large number of random inputs. These help the consideration of arbitrary cost relationships between shift operations and additions/subtractions.

The simulation code is written in C and was developed in MS Visual Studio 6. It is available at: <http://www.hars.us/Papers/ModInv.c> It uses the GNU multi precision arithmetic library, GMP [6], compiled into an MS Windows dll. It is available at <http://www.hars.us/Papers/gmp-dll.zip>

For the right shift algorithm the well-known plus-minus trick does not give a speedup, unless it is extended to handling the multipliers, too. These require minor changes in the code. Another speedup technique replaces the multiplier-halving steps by doublings, with a final correction phase. It provides 30...44% speed increase, dependent on the computational platform.

For the left-shift algorithm we present a new speedup trick: check on a few of the most significant bits which of the intermediate results $a-b$, $2a-b$ or $a-2b$ is expected to give the largest length reduction, and perform that. It speeds-up the algorithm by 4...15% for the cost of a little more complex (micro) code. This is the fastest of the presented algorithms if shifts are done parallel to other instructions.

For the shifting Euclidean algorithm a similar speedup technique proved to be useful: shift one operand to left until its difference from another internal variable is expected to be the smallest (looking at only a few most significant bits). Subtracting this optimum shifted number instead of just aligning the leftmost bits gives about 14% acceleration.

For even moduli the traditional variant of the right-shift algorithms [7] need more memory and significantly longer running time. A simple method is presented, which saves the extra memory (important e.g. for smartcards) and some of the extra computational work.

Theoretical arguments and also our computational experiments show, that the right-shift algorithms are inferior in all cases, and on most computational platforms the modular inverse can be calculated in the time of about two modular multiplications or less.

HW enhancements

In *digit serial* arithmetic engines there is usually a digit-by-digit multiplier circuit (for 8...128 bit operands), which can be utilized for calculating modular inverses. This multiplier is often the

slowest circuit component. Other parts of the circuit can operate at much higher frequency. With appropriate HW design the faster non-multiplicative operations can make up for the speed advantage of the modular inverse algorithms, which use multiplications. This way faster and less expensive HW cores can be designed for computing modular inverses.

This kind of HW design is followed by many modern microprocessors, like the Intel Pentium processors. They have 1 clock cycle base time for a 32-bit integer add or subtract instruction (discounting operand fetch and other overhead), and they can sometimes be paired with other instructions for concurrent execution. A 32-bit multiply takes 10 cycles (a divide takes 41 cycles), and neither can be paired.

The algorithms considered in this paper process the bits or digits of their long operands sequentially, so in a single clock cycle loading more digits in parallel into fast registers allows the use of slower, cheaper RAM, or pipeline registers.

The presented modular inverse algorithms employ only add/subtract, compare and shift operations. With trivial HW enhancements the shift operations can be done “on the fly” when the operands are loaded for additions or subtractions. This kind of parallelism is customarily provided in DSP chips, and it results in a close to two-fold speedup of the shifting xGCD based modular inverse algorithms.

Number representation

For multi-digit integers signed magnitude number representation is beneficial. The binary length is maintained at each operation (without extra cost), and pointers show the position of the most and least significant bit.

- **Addition** is done from right to left (from the least- to the most significant bits), the usual way.
- **Subtraction** needs a scan of the operand bits from left to right, to find the first different pair. They tell the sign of the result. The leading equal bits need not be processed further, and the right-to-left subtract from the larger number leaves no final borrow. This way subtraction is of the same speed as addition.
- **Comparisons** can be done similarly by scanning the bits from left to right. For uniform random inputs the expected number of bit operations is constant: $1 \cdot \frac{1}{2} + 2 \cdot \frac{1}{4} + 3 \cdot \frac{1}{8} \dots = 2$.
- **Comparisons to 0, 1 or 2^k** take constant time also in the worst case, if the head and tail pointers have been maintained properly

HW support for fast shift

Shift operations (multiplication or division by a power of 2) could be implemented with manipulating the length of the number and a pointer to the bits (digits). At a subsequent addition/subtraction the hardware can provide the digits with the corresponding offset, so arbitrary long shifts take only a constant number of operations with this offset-load HW support. (See [13].)

Even in traditional computers manipulating pointers save time, allowing multiple shift operations to be combined into one longer shift. Some DSP's perform the necessary word-shifts parallel to other operations, what also makes the shifts effectively 0-time.

The right-shift binary algorithm RS

At the modular inverse algorithm based on the ***right-shift binary extended GCD*** (variants of the algorithm of M. Penk, see in [1], exercise 4.5.2.39), the modulus m must be odd. The trailing 0-bits from two internal variables U and V (initialized to the input a, m) are removed by shifting them to the right. Then, their difference replaces the larger of them. It is even, so shifting it right removes the new trailing 0-bits.

Repeat these until $V = 0$, when $U = \text{GCD}(m, a)$. If $U > 1$, there is no inverse, so we return 0, which is not an inverse of anything.

In the course of the algorithm two auxiliary variables, R and S are kept updated. At termination R is the modular inverse.

```

U ← m; V ← a; R ← 0; S ← 1;
while ( V > 0 ) {
  if (U0 = 0) {
    U ← U/2;
    if (R0 = 0) R ← R/2;
    else R ← (R+m)/2;
  }
  else if (V0 = 0) {
    V ← V/2;
    if (S0 = 0) S ← S/2;
    else S ← (S+m)/2;
  }
  else // U, V odd
    if (U > V) {
      U ← U - V; R ← R - S;
      /**/ if (R < 0) R ← R + m;}
    else {
      V ← V - U; S ← S - R;
      /**/ if (S < 0) S ← S + m;}
  }
  if (U > 1) return 0;
  if (R > m) R ← R - m;
  if (R < 0) R ← R + m;
  return R; // a-1 mod m

```

Modification RS1

The 2 instructions marked with “/**/” keep R and S nonnegative and so assure that they don't grow too large (the subsequent subtraction steps decrease the larger absolute value). These instructions are slow and not necessary, if we ensure otherwise, that the intermediate values of R and S do not get too large.

Handling negative values and fixing the final result is easy, so it is advantageous if instead of the marked instructions, we only check at the add-halving steps ($R \leftarrow (R+m)/2$ and $S \leftarrow (S+m)/2$), whether R or S was already larger (or longer) than m , and **add or subtract m** such that the result becomes smaller (shorter). These steps cost no additional work beyond choosing ‘+’ or ‘-’ and, if $|R| \leq 2m$ was beforehand, we get $|R| \leq m$, the same as at the simple halving of $R \leftarrow R/2$ and $S \leftarrow S/2$. If $|R| \leq m$ and $|S| \leq m$, $|R-S| \leq 2m$ (the length could increase by one bit) but these instructions are always followed by halving steps, which prevent R and S to grow larger than $2m$ during the calculations. (See code details at the plus-minus algorithm below.)

Even modulus

There is the difficulty that m must be odd (to ensure that either R or $R \pm m$ is even for the subsequent halving step) so this algorithm cannot be used for RSA key generation. We can go around the problem: Let us swap the role of m and a (a must be odd, if m is even, otherwise there is no inverse). The algorithm returns $m^{-1} \bmod a$, such that $m \cdot m^{-1} + k' \cdot a = 1$, for some (negative) integer k' . This $k' \equiv a^{-1} \bmod m$, easily seen if we take both sides of the equation mod m . It is simple to compute the smallest positive $k \equiv k' \bmod m$:

$$k = a^{-1} \bmod m = m + (1 - m \cdot m^{-1})/a.$$

Here the division with a is exact (no remainder), so a simple and fast Montgomery-like division can be performed. It is even faster calculating only the MS half of $m \cdot m^{-1}$, plus a couple of guard digits, enough to get an approximate quotient, which is then rounded to the nearest integer. This way there is no need for longer than $\|m\|$ -bit arithmetic (except two extra digits for the proper rounding of the intermediate results).

Calculating $a^{-1} \bmod m$ was done by treating the modular inverse as a black box algorithm. Unfortunately there is no trivial modification of the algorithm to handle even moduli internally, because at halving we are only allowed to add an integer multiple of the modulus, so it has to be odd to turn odd intermediate values to even. However, the only time we need to handle even moduli in cryptography is at RSA key generation, which is so slow anyway (requiring thousands

of modular multiplications for the primality tests), that this black box workaround does not cause a noticeable difference in processing time.

An alternative was to perform the full extended GCD algorithm, calculating both multipliers c and d : $[g,c,d] = \text{xCGD}(m,a)$, such that the greatest common divisor $g = c \cdot m + d \cdot a$ [7]. The extended GCD algorithm needs storage room for 2 more long internal variables, which are constantly updated during the course of the algorithm. It is slower and uses more memory than applying the above fix after the modular inverse algorithm with swapped parameters.

xGCD from Modular Inverse

The same technique proves to be useful in general, when both multipliers of the extended GCD algorithm are needed: $[g,c,d] = \text{xCGD}(x,y)$, such that the greatest common divisor $g = c \cdot x + d \cdot y$. For example, the shifting Euclidean modular inverse algorithms we consider below can calculate one of the multipliers, say c , even if there is no inverse ($g \neq 1$). Using the method above we compute the other multiplier d .

If a right-shift binary modular inverse algorithm is used at least one of x and y must be odd, what can be assured by dividing the parameters by 2 the sufficient number of times (k), until (at least) one of them becomes odd, giving x' and y' . At the end we get $g' = c \cdot x' + d \cdot y'$, which is then multiplied by 2^k : $g = 2^k g' = c \cdot 2^k x' + d \cdot 2^k y' = c \cdot x + d \cdot y$.

This method saves memory for 2 long integers and it is faster than the full extended GCD algorithm, which was modified to provide the applied modular inverse algorithm.

Justification of RS1

The algorithm starts with $U = m$, $V = a$, $R = 0$, $S = 1$. In the course of the algorithm U and V are decreased, keeping $\text{GCD}(U,V) = \text{GCD}(m,a)$ true. The algorithm reduces U and V until $V = 0$ and $U = \text{GCD}(m,a)$: If one of U or V is even, it can be replaced by its half, since $\text{GCD}(m,a)$ is odd. If both are odd, the larger one can be replaced by the even $U-V$, which then can be decreased by halving, leading eventually to 0. The binary length of the larger of U and V is reduced by at least one bit, guarantying that the procedure terminates in at most $\|a\| + \|m\|$ iterations.

At termination of the algorithm $V = 0$ otherwise a length reduction was still possible. $U = \text{GCD}(U,0) = \text{GCD}(m,a)$. Furthermore, the calculations maintain the following two congruencies:

$$U \equiv Ra \pmod{m}, \quad V \equiv Sa \pmod{m}. \quad (1)$$

Having an odd modulus m , at the step halving U we have two cases. When R is even: $U/2 \equiv (R/2) \cdot a \pmod{m}$, and when R is odd: $U/2 \equiv ((R+m)/2) \cdot a \pmod{m}$. The algorithm assigns exactly these new values to U and R . Similarly for V and S , with their new values (1) remains true.

The difference of the two congruencies in (1) gives $U-V \equiv (R-S) \cdot a \pmod{m}$, which ensures that at the subtraction steps (1) remains true after updating the corresponding variables: U or $V \leftarrow U-V$, R or $S \leftarrow R-S$. Choosing $+m$ or $-m$, as discussed above, guarantees that R and S does not grow larger than $2m$, so at the end we can just add or subtract m to make $0 < R < m$. If $U = 1 = \text{GCD}(m,a)$ we get $1 \equiv Ra \pmod{m}$, and R is of the right magnitude, so $R = a^{-1} \pmod{m}$. \square

Plus-minus right shift algorithm RS+-

There is a very simple modification often used for the right-shift algorithm: check, if $U+V$ has 2 trailing zero bits, otherwise we know that $U-V$ does. In the former case, if $U+V$ is of the same length as the larger of them, the shift operation reduces the length by 2 bits from this larger length, otherwise by only one bit (as before with the rigid subtraction steps). It means that the length reduction is often improved, so the number of iterations decreases.

Unfortunately, this reduction is not large, only 15% (half of the time the reduction was by at least 2 bits, anyway, and the longer shifts are not affected), but it comes almost for free. Furthermore, R and S needs more halving steps, and these get a little more expensive (at least one of the halving steps need an addition of m), so the RS+- algorithm is not faster than RS1.

Right shift algorithm RS2+–

The plus-minus reduction can be applied also to R and S. In the course of the algorithm the temporary variables R and S are halved, too. If they happened to be odd, m is added or subtracted to make them even before the halving. The plus-minus trick on them ensures that the result has at least 2 trailing 0-bits. It provides a speedup, because most of the time we had exactly two divisions by 2 (shift right by two), and no more than one addition/subtraction of m is now necessary.

```
U ← m; V ← a; R ← 0; S ← 1;
Q = m mod 4;
while (V0 = 0) { V ← V/2;
  if(S0 = 0) S ← S/2;
  elseif(S > m) S ← (S-m)/2;
  else S ← (S+m)/2;
}
Loop { // U, V odd
  if (U > V) {
    if (U1 = V1)
      U ← U + V; R ← R + S;
    else
      U ← U - V; R ← R - S;
    U ← U/4; T ← R mod 4;
    if( T = 0) R ← R/4;
    if( T = 2) R ← (R+2m)/4;
    if( T = Q) R ← (R-m)/4;
    else R ← (R+m)/4;
    while (U0 = 0) { U ← U/2;
      if(R0 = 0) R ← R/2;
      elseif(R > m) R ← (R-m)/2;
      else R ← (R+m)/2;}
  }
  else {
    if (U1 = V1)
      V ← V + U; S ← S + R;
    else
      V ← V - U; S ← S - R;
      if( V = 0 ) break;
    V ← V/4; T ← S mod 4;
    if( T = 0) S ← S/4;
    if( T = 2) S ← (S+2m)/4;
    if( T = Q) S ← (S-m)/4;
    else S ← (S+m)/4;
    while (V0 = 0) { V ← V/2;
      if(S0 = 0) S ← S/2;
      elseif(S > m) S ← (S-m)/2;
      else S ← (S+m)/2;}
  }
}
if (U > 1) return 0; // no inverse
if (R ≥ m) R ← R - m;
if (R < 0) R ← R + m;
return R; // a-1 mod m
```

Delayed Halving right shift algorithm RSDH

The variables R and S get almost immediately of the same length as m , because, when they are odd, m is added to them to allow halving without remainder. We can delay these add-halving steps, by doubling the other variable instead. When R should be halved we double S, and vice versa. Of course, a power-of-2 spurious factor is introduced to the computed GCD, but keeping track of the exponent a final correction step will fix R by the appropriate number of halving- or add-halving steps. (This technique speeds up the Montgomery inverse computation even more.) There is an acceleration of the algorithm by 24...38% over RS1, due to the following:

- R and S now increases gradually, so their average length is only half as it was in RS1.
- The final halving steps are performed only with R. The variable S need not be fixed, being only an internal temporary variable.
- At the final halving steps more shifts can be combined into longer shifts, because they are not confined by the amount of shifts performed on U and V in the iterations of the algorithm.

We have to note, that in the course of the algorithm R and S are almost always of different lengths, so their difference is usually not longer than the longer of R and S. Consequently, their lengths don't increase faster than what the shifts cause.

Another note: it does not pay to check, if R or S is even, in the hope that some halving steps could be performed until the involved R or S becomes odd, and so speeding up the final correction, because they are already odd in the beginning (easily proved by induction).

Combined speedups right shift algorithm RSDH+-

The second variant of the plus-minus trick and the delayed halving trick can be combined, giving the fastest of the presented right shift modular algorithms. It is 43...60% faster than the original RS1, but still slower on most computational platforms than the left shift and shifting Euclidean algorithms, discussed below.

Working on the ends

With delayed update of the variables (using their MS and/or LS digits only, as long as we can determine the necessary reduction steps, and fix the rest of the numbers only when more precision is needed) on some computational platforms a little speed increase can be achieved, due to the reduced number of data fetch operations. Unfortunately, the resulting algorithms are much more complex, less suitable for direct HW implementations, and in our computational model data load-store are free, so we don't consider them here. These variants would benefit the most from multiplications.

The left-shift binary modular inverse algorithm LS1

The left-shift binary modular inverse algorithm (similar to the variant of R. Lórencz [9]) is described in the figure below. It keeps the temporary variables U and V aligned to the left, such that a subtraction clears the leading bit(s). Shifting the result left until the most significant bit is again in the proper position restores the alignment. The number of known trailing 0-bits increases, until a single 1-bit remains, or the result is 0 (indicating that there is no inverse). As before, keeping 2 internal variables R and S updated, the modular inverse is calculated.

```

U ← m; V ← a; R ← 0; S ← 1; u ← 0; v ← 0;
while((|U| ≠ 2u) && (|V| ≠ 2v)) {
  if(|U| < 2n-1) {
    U ← 2U; u ← u+1;
    if (u > v) R ← 2R; else S ← S/2;
  }
  else if(|V| < 2n-1) {
    V ← 2V; v ← v+1;
    if (v > u) S ← 2S; else R ← R/2;
  }
  else // |U|, |V| ≥ 2n-1
    if(sign(U) = sign(V))
      if(u ≤ v) {U ← U - V; R ← R - S;}
      else {V ← V - U; S ← S - R;}
    else // sign(U) ≠ sign(V)
      if(u ≤ v) {U ← U + V; R ← R + S;}
      else {V ← V + U; S ← S + R;}
    if (U = 0 || V = 0) return 0;
  }
if (|V| = 2v) { R ← S; U ← V; }
if (U < 0)
  if (R < 0) R ← -R;
  else R ← m - R;
if (R < 0) R ← m + R;
return R; // a-1 mod m

```

Here *u* and *v* are single-word variables, counting how many times U and V were shifted left, respectively. They tell at least how many trailing zeros the corresponding U and V long integers have, because we always add/subtract to the one, which has fewer known zeros and then shift left,

increasing the number of trailing zeros. 16 bits words for u and v allow us working with any operand length less than 64K bit, enough for all cryptographic applications in the foreseeable future. Knowing the values of u and v also helps speeding up the calculations, because we need not process the known least significant zeros. (Care must be taken, since the long integer temporary variables U and V can grow to $n+1$ bit long.)

Justification

The reduction of the temporary variables is now done by shifting left the intermediate results U and V , until they have their MS bits in the designated position (n -th bit) determined by the MS bit position of the larger of the original operands. Performing a subtraction clears this bit, and so reduces the binary length. The left-shifts introduce spurious factors of 2^k for the GCD, but tracking the number of trailing 0-bits (u and v) allows the determination of the true GCD. (For a rigorous proof see [9].)

We start with $U = m$, $V = a$, $R = 0$, $S = 1$, $u = v = 0$. During the course of the algorithm there will be at least u and v trailing 0-bits in U and V , respectively. In the beginning

$$\text{GCD}(U/2^{\min(u,v)}, V/2^{\min(u,v)}) = \text{GCD}(m, a) \quad (2)$$

If U or V is replaced by $U-V$, this relation remains true. If both U and V had their most significant (n -th) bit = 1, the above subtraction clears it. We chose the one from U and V to be updated, which had the smaller number of trailing 0-bits, say it was U . U then gets doubled until its most significant bit gets to the n -th bit position again, and u , the number of trailing 0's, is incremented in each step.

If $u \geq v$ was before the doubling, $\min(u,v)$ does not change, but U doubles. Since $\text{GCD}(m, a)$ is odd (there is no inverse if it is not 1), $\text{GCD}(2 \cdot U/2^{\min(u,v)}, V/2^{\min(u,v)}) = \text{GCD}(m, a)$ remains true. If $u < v$ was before the doubling, $\min(u,v)$ increases, leaving $U/2^{\min(u,v)}$ unchanged. The other parameter $V/2^{\min(u,v)}$ was even, and becomes halved. It does not change the GCD, either.

In each subtraction-doubling iteration either u or v (the number of trailing known 0's) is increased. U and V are never longer than n -bits, so u and $v \leq n$, and eventually a single 1-bit remains in U or V (or one of them becomes 0, showing that $\text{GCD}(m, a) > 1$). It guaranties, that the procedure stops in at most $\|a\| + \|m\|$ iterations, with U or $V = 2^{n-1}$ or 0.

During the course of the algorithm:

$$U/2^{\min(u,v)} \equiv Ra \pmod{m}, \quad V/2^{\min(u,v)} \equiv Sa \pmod{m}. \quad (3)$$

At subtraction steps $(U-V)/2^{\min(u,v)} \equiv (R-S) \cdot a \pmod{m}$, so (3) remains true after updating the corresponding variables: U or $V \leftarrow U-V$, R or $S \leftarrow R-S$.

At doubling U and incrementing u , if $u < v$ was before the doubling, $\min(u,v)$ increases, so $U/2^{\min(u,v)}$ and R remains unchanged. $V/2^{\min(u,v)}$ got halved, so it is congruent to $(S/2) \cdot a \pmod{m}$, therefore S has to be halved to keep (3) true. This halving is possible (V is even), because S has at least $v-u$ trailing 0's (can be proved by induction).

At doubling U and incrementing u , if $u \geq v$ was before the doubling, $\min(u,v)$ does not change. To keep (3) true R has to be doubled, too (which also proves that it has at least $v-u$ trailing 0's).

Similar reasoning shows the correctness of handling R and S when V is doubled.

At the end we get either $U = 2^u$ or $V = 2^v$, so one of $U/2^{\min(u,v)}$ or $V/2^{\min(u,v)}$ is 1, and $\text{GCD}(m, a)$ is the other one. If the inverse exists, $\text{GCD}(m, a) = 1$ and we get from (3) that either $1 \equiv Ra \pmod{m}$ or $1 \equiv Sa \pmod{m}$. After making R or S of the right magnitude, it is the modular inverse $a^{-1} \pmod{m}$.

Another induction argument shows that R and S does not become larger than $2m$ in the course of the algorithm, otherwise the final reduction phase of the result to the interval $[1, m-1]$ could take a lot of calculation. \square

Left-shift algorithm with best shift LS3

Unfortunately, the plus-minus trick does not work with the left-shift algorithm: addition never clears the MS bit. If U and V are close, a subtraction might clear more than one MS bits, otherwise one could try $2U-V$ and $2V-U$ for the cases when $2U$ and V or $2V$ and U are close. (With the n -th bit = 1 other two's power linear combinations, which can be calculated with only shifts, don't help.) Looking at only a few MS bits one can determine, which one of the 3 tested reductions is expected to give the largest length decrease (testing 3 reduction candidates is the reason to call the algorithm LS3). We could often clear extra MS bits this way. In general microprocessors the gain is not much, because computing $2x-y$ could take 2 instructions instead of one for $x-y$, but memory load and store steps can still be saved. With hardware for shifted operand fetch the doubling comes for free, giving a larger speedup.

Shifting Euclidean modular inverse algorithms SE

The original Euclidean GCD algorithm replaces the larger of the two parameters by subtracting the largest possible number of times the smaller parameter: $x \leftarrow x - [x/y] \cdot y$. For this we need to calculate the quotient $[x/y]$ and multiply it with y . We don't deal with algorithms in this paper, which perform division or multiplication. However, the Euclidean algorithm works with smaller coefficients $q \leq [x/y]$, too: $x \leftarrow x - q \cdot y$. In particular, we can choose q to be the largest power of 2, such that $q = 2^k \leq [x/y]$. The reductions can be performed with shifts and subtractions only, and they still clear the most significant bit of x , so the resulting algorithm will terminate in a reasonable number of iterations. It is well known (see [1]), that for random input, most of the time $[x/y] = 1$ or 2, so the shifting Euclidean algorithm is not much slower than the original one but avoids multiplications.

Repeat these until $V = 0$ or ± 1 , when $U = \text{GCD}(m, a)$. If $V = 0$, there is no inverse, so we return 0, which is not an inverse of anything. (The pathological cases, like $m = a = 1$ need special handling, but these don't occur in cryptography.)

In the course of the algorithm two auxiliary variables, R and S are kept updated. At termination S is the modular inverse, or the negative of it, within $\pm m$.

```
if (a < m) {U ← m; V ← a; R ← 0; S ← 1;}
else      {V ← m; U ← a; R ← 0; S ← 1;}
while (||V|| > 1) {
  f ← ||U|| - ||V||
  if (sgn(U)=sgn(V)) {U ← U-(V<<f); R ← R-(S<<f);}
  else                {U ← U+(V<<f); R ← R+(S<<f);}
  if (||U|| < ||V||) {U ↔ V; R ↔ S;}
}
if (V = 0) return 0;
if (V < 0) S ← -S;
if (S > m) return S - m;
if (S < 0) return S + m;
return S; // a-1 mod m
```

Justification of SE

The algorithm starts with $U = m$, $V = a$, $R = 0$, $S = 1$. If $a > m$, swap (U, V) and (R, S) . U always denotes the longer of the just updated U and V . During the course of the algorithm U is decreased, keeping $\text{GCD}(U, V) = \text{GCD}(m, a)$ true. The algorithm reduces U , swaps with V when $U < V$, until $V = \pm 1$ or 0: U is replaced by $U - 2^k V$, with such a k , that reduces the length of U , leading eventually to 0 or ± 1 , when the iteration can stop. The binary length $\|U\|$ is reduced by at least one bit in each iteration, guarantying that the procedure terminates in at most $\|a\| + \|m\|$ iterations.

At termination of the algorithm either $V = 0$ (indicating that $U = 2^k V$ was beforehand, and so there is no inverse) or $V = \pm 1$, otherwise a length reduction was still possible. In the later case $1 = \text{GCD}(|U|, |V|) = \text{GCD}(m, a)$. Furthermore, the calculations maintain the following two congruencies:

$$U \equiv Ra \pmod{m}, \quad V \equiv Sa \pmod{m}. \quad (4)$$

The weighted difference of the two congruencies in (4) gives $U-2^kV \equiv (R-2^kS)\cdot a \pmod{m}$, which ensures that at the reduction steps (4) remains true after updating the corresponding variables: $U \leftarrow U-2^kV$, $R \leftarrow R-2^kS$. As in the proof of correctness of the original extended Euclidean algorithm, we can see that $|R|$ and $|S|$ remain less than $2m$, so at the end we fix the sign of S to correspond to V , and add or subtract m to make $0 < S < m$. Now $1 \equiv Sa \pmod{m}$, and S is of the right magnitude, so $S = a^{-1} \pmod{m}$. \square

Shift Euclidean algorithm with best shift SE3

We can do a similar trick to speed up the shifting Euclidean algorithm as at the left-shift algorithm LS3. If U and 2^kV are close, the shift-subtraction might clear more than one MS bits, otherwise one could try $U-2^{k-1}V$ and $U-2^{k+1}V$. (With the choice of k being the length difference, other two's power linear combinations don't clear more MS bits.) Looking at only a few MS bits one can determine, which one of the 3 tested reductions is expected to give the largest (length) decrease. (Testing 3 reduction candidates is the reason to call the algorithm SE3). We could often clear extra MS bits this way. This technique gives about 14% reduction in the number of iterations, and a similar speedup on most computational platforms, because the shift operation takes the same time, regardless of the amount of shift (except, when it is 0).

We have a choice, how to rank the expected reductions. In the SE3 code we picked the largest expected length reduction, because it is the simplest in HW. Another possibility was to choose the shift amount, which leaves the smallest absolute value result. It is a little more complex, but gives about 0.2% speed increase.

Speed comparisons

The right-shift algorithms are the slowest, because they halve auxiliary variables (R , S) and if they happen to be odd, m is added or subtracted first, to make them even for the halving. This addition steps are not needed in the left-shift- or in the shifting Euclidean algorithms. Furthermore, in all three groups of algorithms the length of U and V decreases bit-by-bit in each iteration, and in the left-shift- and shifting Euclidean algorithms the length of R and S increases steadily from 1. In the right shift case they get very soon as long as m , except in the delayed halving variant. In the average, the changing lengths roughly halve the work on those variables. Also, the necessary additions of m in the original right-shift algorithms prevent aggregation of the shift operations of R and S . On the other hand, in the other algorithms (including the delayed halving right sift algorithm) we can first determine by how many bits we have to shift all together in that phase, and in the left-shift algorithms, dependent on the relative magnitude of u and v we need only one or two shifts by multiple bits, in the shifting Euclidean algorithm only one. These are not relevant at the most common shift lengths of 1 or 2, but they do save some work.

On the other hand, the left-shift- and shifting Euclidean algorithms perform 'lucky' multi-bit reductions somewhat less frequently. The subtraction clears some MS bits dependent on all of the other bits, too. Larger reductions happen less frequently, than at the right-shift algorithm, where only the least significant few bits play a role. Still, the large saving in additions offsets the little larger number of iterations.

Test results

We executed 1 million calls of each of the many variants of the modular inverse algorithms with random inputs of length 16...1024 bit, so the experimental complexity results are expected to be accurate within 0.1...0.3%. The table at the end of the paper contains the binary costs of the additions and shifts the corresponding modular inverse algorithms performed, and the number of iterations and shifts with some given lengths. (Multiple shifts of the same variable are assumed to be combined into longer ones.)

Remarks:

- The total number of the different UV shifts together is the number of iterations, since there is one combined shift in each iteration.
- In the left-shift algorithms the sum of RS shifts is larger than the number of iterations, because some shifts may cause the relationship between u and v to change, and in this case there are 2 shifts in one iteration.
- In [11] there are evidences cited that the binary right shift GCD algorithm performs $A \cdot \log 2^{\|m\|}$ iterations, with $A = 1.0185\dots$. The RS1 algorithm performs the same number of iterations as the binary right shift GCD algorithm. Our experiments gave a very similar (only 0.2% smaller) result: $A' = 0.7045/\log 2 = 1.0164\dots$

In the table above we listed the dominant terms of the speed of the algorithms for 3 typical computational models:

1. Shifts execute in a constant number of clock cycles

Algorithm LS3 is the fastest ($0.6662 n^2$), followed by SE3 ($0.6750 n^2$), with only a 1.3% lag. The best right-shift algorithm is RSDH+-, which is 1.66 times slower ($1.1086 n^2$).

2. Shifts are 4 times faster than add/subtracts

Algorithm SE3 is the fastest ($0.8114 n^2$), followed by LS3 ($0.9043 n^2$), within 14%. The best right-shift algorithm (RSDH+-) is 1.71 times slower ($1.3858 n^2$).

3. Shifts and add/subtracts take the same time

Again, algorithm SE3 is the fastest ($1.2176 n^2$), followed by SE ($1.3904 n^2$), within 14%. The best right-shift algorithm (RSDH+-) is 2.37 times slower ($2.8804 n^2$).

Interestingly the Plus-Minus algorithm RS+-, which only assures that U or V are reduced by at least 2 bits, performs fewer iterations, but the overall running time is not improved. When R and S are also treated the same way, the running time improves. It shows that speeding up the R,S halving steps is more important than speeding up the U,V reduction steps, because the later reduction steps operate on diminishing length numbers, while the R,S halving works mostly on more costly, full length numbers.

Performance relative to digit-serial modular multiplication

Of course, the speed ratio of the modular inverse algorithms relative to the speed of the modular multiplications depends on the computational platform and the employed multiplication algorithm. We consider quadratic time modular multiplications, like Barrett, Montgomery or school multiplication with division based modular reduction. With operand lengths in cryptography sub-quadratic time modular multiplications (like Karatsuba) are only slightly faster, more often they are even slower than the simpler quadratic time algorithms (see [6]).

If there is a HW multiplier, which computes products of d -bit digits in c clock cycles a modular multiplication takes $T = 2c \cdot (n/d)^2 + O(n)$ time, alone for the digit products [13]. With DSP-like architecture (load, shift and add instructions performed parallel to multiplications) the time complexity is $2c \cdot (n/d)^2$. Typical values are $d = 16$, $c = 4$: $T = n^2/32 \approx 0.031n^2$; or $d = 32$, $c = 12$: $T \approx 0.023n^2$.

The fastest of the presented modular inverse algorithm on **parallel shift-add** architecture takes $0.666 n^2$ bit operations, which is reduced by the digit size (processing d bits together in one addition). For the above two cases we get $0.042n^2$ and $0.021n^2$ running times, respectively. These values are very close to the running time of **one** modular multiplication.

The situation is less favorable if there are **no parallel** instructions. The time a multiplication takes is dominated by computing the digit products, additions and register manipulations are faster. On these platforms computing the modular inverse takes almost twice as much time than with parallel add and shift instructions. Consequently, without parallel instructions computing the

modular inverse takes about **twice** as much time as a modular multiplication. Still, in case of elliptic curve cryptography the most straightforward (affine) point representation and implementation of the point addition is the best (the projective, Jacobian and Chudnovsky-Jacobian coordinates are slower, see [8]).

Performance relative to bit-parallel adder based modular multiplication

When very long adders are implemented in HW, repeated shift-add steps can perform multiplications in linear time. To prevent the partial results from growing too large, interleaved modular reduction is performed. Scanning the bits of the second multiplicand from left to right, when a 1-bit is found, the first multiplicand is added in the appropriate position to the partial result r . Its length is reduced if it gets too long, $\|r\| > \|m\|$, by subtracting the modulus m . These add-subtract steps are usually done in the same clock cycle, resulting in that an n -bit modular multiplication is performed in n clock cycles.

In these kinds of HW architectures the speed of the modular inverse algorithms become very close, because there is no advantage of having additions on diminishing length operands. A typical iteration reduces the number of bits of the longer operand by about 1.4, in the average, so the left- and right shift algorithms don't differ much, in how many shift steps can be combined into one longer shift. The plus-minus right shift algorithm has the smallest number of iterations, its delayed halving variant can combine the largest number of shifts, so its running time becomes very close to that of the shifting Euclidean algorithm.

In each iteration the RSDH+- modular inverse algorithm needs to shift one of U or V, and double R or S the same many times, which give about the same amount of work as the modular multiplication performs, maybe even less. At the end we need to add-halve R, which makes the modular inverse slightly slower than **one** modular multiplication, but still faster than two.

Testing relative primness

We can simplify all of our shifting modular inverse algorithms if we only want to know, whether the two arguments a, b are relative primes: leaving out all the calculations with R and S. In this case all the plus-minus right shift algorithms become the same, so the simplest RS+- is the best, with $0.3065n^2$ cost of bit-shifts and the same for subtractions, all together $0.6130n^2$. SE3 is still slightly better, with a running time of $0.6088n^2$. The modified left-shift algorithm LS3 takes $0.3967n^2$ clock cycles for the shift operations, and $0.3331n^2$ clock cycles for the subtract operations, which is only 9...19% more. When not only relative primness has to be tested in an application, but modular inverses have to be calculated as well, this little speed advantage might not justify the implementation of 2 different algorithms, so LS3 or SE3 should be used for both purposes (without computing R and S if not needed).

Further optimizations

There are countless possibilities to speed up the presented algorithms a little further. For example, when U and V become small (short), a table lookup could immediately finish the calculations. If only one of them becomes small, we could switch to an algorithm, which is best tuned to this case on the particular computing platform.

Hybrid algorithms

The right shift- and the shifting Euclidean algorithms operate on the opposite ends of the numbers. At least when the modulus is odd, the reduction steps could be intermixed: check on a few bits on the appropriate end of the intermediate numbers which reduction is expected to reduce the lengths the most, and perform that step. However, the right shift algorithms are so much slower, that the corresponding reduction is almost never more promising than the shifting Euclidean reduction, so we could not achieve any significant speedup, but the algorithms become very complicated and convoluted.

References

- [1] Donald E. Knuth, "The Art of Computer Programming", volume 2, "Seminumerical Algorithms", 3rd edition, Addison-Wesley, 1998.
<http://www-cs-faculty.stanford.edu/~knuth/taocp.html>
- [2] Tudor Jebelean, "A Generalization of the Binary GCD Algorithm", ISSAC 93, pp. 111-116. Technical report version available
<ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1993/93-01.ps.gz>
- [3] Tudor Jebelean, "A Double-Digit Lehmer-Euclid Algorithm for Finding the GCD of Long Integers", Journal of Symbolic Computation, volume 19, 1995, pp. 145-157. Technical report version also available <ftp://ftp.risc.uni-linz.ac.at/pub/techreports/1992/92-69.ps.gz>
- [4] Kenneth Weber, "The accelerated integer GCD algorithm", ACM Transactions on Mathematical Software, volume 21, number 1, March 1995, pp. 111-122.
- [5] Arnold Schönhage and Volker Strassen, "Schnelle Multiplikation großer Zahlen", Computing 7, 1971, pp. 281-292.
- [6] GNU Multiple Precision Arithmetic Library manual
<http://www.swox.com/gmp/gmp-man-4.1.2.pdf>
- [7] A. Menezes, P. van Oorschot, and S. Vanstone, "Handbook of Applied Cryptography", CRC Press, 1996.
- [8] H. Cohen, A. Miyaji, and T. Ono, "Efficient elliptic curve exponentiation using mixed coordinates". In K. Ohta and D. Pei, editors, Advances in Cryptology - ASIACRYPT 98, Lecture Notes in Computer Science, No. 1514, pages 51-65. Springer, Berlin, Germany, 1998. <http://citeseer.ist.psu.edu/article/cohen98efficient.html>
- [9] R. Lórencz, "New Algorithm for Classical Modular Inverse", CHES 2002, LNCS 2523, pp 57-70. Springer-Verlag, 2003.
- [10] M. D. Ergovac, T. Lang, "Digital Arithmetic" Chapter 2. Morgan Kaufmann Publishers, 2004.
- [11] B. Vallée, "Complete Analysis of the Binary GCD" 1998.
<http://citeseer.ist.psu.edu/79809.html>
- [12] J. Jedwab and C. J. Mitchell. "Minimum weight modified signed-digit representations and fast exponentiation". Electronics Letters, 25(17):1171-1172, 17. August 1989.
- [13] L. Hars, "Long Modular Multiplication for Cryptographic Applications".
<http://eprint.iacr.org/2004/198/> CHES 2004, (Damaged version LNCS 3156, pp 44-61. Springer-Verlag, 2004.)

Algorithm Steps/bit	Right-Shift					Left-Shift		Shift-Euclidean	
	RS1	RS+-	RS2+-	RSDH	RSDH+-	LS1	LS3	SE	SE3
Iterations	0.7045 n	0.6115 n	0.6115 n	0.7045 n	0.6115 n	0.7650 n	0.6646 n	0.7684 n	0.6744 n
UV shift cost	0.3531 n^2 -1.2200 n	0.3065 n^2 -1.1891 n	0.3065 n^2 -1.1891 n	0.3531 n^2 -1.2200 n	0.3065 n^2 -1.1891 n	0.3834 n^2 -0.8836 n	0.3967 n^2 -0.8435 n	0.3101 n^2 -1.0646 n	0.2708 n^2 -0.8742 n
RS shift cost	1.0592 n^2 -4.9984 n	1.2259 n^2 -5.2592 n	0.9808 n^2 -5.1720 n	0.9241 n^2 -3.3945 n	0.8021 n^2 -3.3794 n	0.5300 n^2 -4.9665 n	0.5558 n^2 -5.1855 n	0.3101 n^2 -2.9784 n	0.2708 n^2 -2.5787 n
Total shift cost	1.4123 n^2 -6.2184 n	1.5324 n^2 -6.4483 n	1.2873 n^2 -6.3611 n	1.2772 n^2 -4.6145 n	1.1086 n^2 -4.5685 n	0.9134 n^2 -5.8501 n	0.9525 n^2 -6.0290 n	0.6202 n^2 -4.0430 n	0.5416 n^2 -3.4529 n
UV subtract cost	0.3531 n^2 +0.2658 n	0.3065 n^2 +0.2967 n	0.3065 n^2 +0.2967 n	0.3531 n^2 +0.2658 n	0.3065 n^2 +0.2967 n	0.3835 n^2 +0.4377 n	0.3331 n^2 +0.5942 n	0.3851 n^2 +0.4276 n	0.3380 n^2 +0.4958 n
RS subtract cost	1.4123 n^2 -4.8065 n	1.5325 n^2 -4.8844 n	1.2873 n^2 -4.5004 n	0.9241 n^2 -1.4559 n	0.8021 n^2 -0.7786 n	0.3834 n^2 -1.0101 n	0.3331 n^2 -0.9160 n	0.3851 n^2 -1.0331 n	0.3380 n^2 -0.7125 n
Total subtract cost	1.7654 n^2 -4.5407 n	1.8390 n^2 -4.5877 n	1.5938 n^2 -4.2037 n	1.2772 n^2 -1.1901 n	1.1086 n^2 -0.4819 n	0.7669 n^2 -0.5724 n	0.6662 n^2 -0.3218 n	0.7702 n^2 -0.6055 n	0.6760 n^2 -0.2167 n
Complexity @ 0 cost shift	1.7654 n^2	1.8390 n^2	1.5938 n^2	1.2772 n^2	1.1086 n^2	0.7669 n^2	0.6662 n^2	0.7702 n^2	0.6750 n^2
Complexity @ ¼ add cost shift	2.1185 n^2	2.2221 n^2	1.9156 n^2	1.5965 n^2	1.3858 n^2	0.9953 n^2	0.9043 n^2	0.9253 n^2	0.8114 n^2
Complexity @ 1 add cost shift	3.1777 n^2	3.3714 n^2	2.8811 n^2	2.5544 n^2	2.2172 n^2	1.6803 n^2	1.6187 n^2	1.3904 n^2	1.2176 n^2
UV shifts by 1	0.3522 n	-	-	0.3522 n	-	0.1983 n	0.1977 n	0.2576 n	0.2143 n
UV shifts by 2	0.1761 n	0.3058 n	0.3058 n	0.1761 n	0.3058 n	0.2463 n	0.2388 n	0.1705 n	0.1573 n
UV shifts by 3	0.0881 n	0.1529 n	0.1529 n	0.0881 n	0.1529 n	0.1516 n	0.1778 n	0.0927 n	0.0831 n
Longer UV shifts	0.0881 n	0.1529 n	0.1529 n	0.0881 n	0.1529 n	0.1689 n	0.1772 n	0.0980 n	0.0857 n
RS shifts by 1	0.7925 n	0.7644 n	0.3364 n	0.6375 n	-	0.5202 n	0.5395 n	0.2576 n	0.2143 n
RS shifts by 2	0.1982 n	0.3440 n	0.4816 n	0.3188 n	0.5534 n	0.3142 n	0.3313 n	0.1705 n	0.1573 n
RS shifts by 3	0.0495 n	0.0860 n	0.1204 n	0.1594 n	0.2767 n	0.1280 n	0.1413 n	0.0927 n	0.0831 n
Longer RS shifts	0.0165 n	0.0287 n	0.0401 n	0.1594 n	0.2767 n	0.0952 n	0.0968 n	0.0980 n	0.0857 n