From Euclid’s GCD to Montgomery Multiplication to the Great Divide

Sheueling Chang Shantz

From Euclid's GCD to Montgomery Multiplication to the Great Divide

Sheueling Chang Shantz

SMLI TR-2001-95 June 2001

Abstract:

Euclid's method for finding the *greatest common divisor* (GCD) of two integers was first described around the year 300 B.C. This simple iterative method is often regarded as the grandfather of all algorithms in Number Theory today. Many advances have been made since then—for example, Berlekamp's algorithm for *multiplicative inverse* and Montgomery's technique for *modular multiplication*. These binary add-and-shift algorithms for efficient finite field arithmetic operations have played important roles in today's public-key cryptographic systems.

Yet, two thousand three hundred years after Euclid's GCD, one algorithm remained missing—division. For many decades we did not tackle modular division problems directly. Instead, we relied on the Extended Euclidean algorithm for calculating inversion and we computed division in a two-step process—inversion followed by multiplication. This practice is so deeply rooted in our teachings and doings today that we have neglected to ask whether the idea underlying the binary Extended Euclidean algorithm can also be applied to finding a general solution for field division. This paper describes such a solution: a binary add-and-shift algorithm for *modular division* in a residue class. This technique for fast computation of divisions in $GF(2^m)$ is the key to a highly efficient implementation of elliptic curve cryptosystems.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
sheueling.chang@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>. The entire technical report collection is available online at http://research.sun.com.

From Euclid's GCD to Montgomery Multiplication to the Great Divide

Sheueling Chang Shantz

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, California 94303

1. Introduction

Arithmetic operations in the Galois field $GF(2^m)$ have several applications in coding theory, computer algebra, and cryptography. Efficient modular arithmetic algorithms play an important role in today's cryptographic systems. Most practical public−key systems exploit the properties of arithmetic in large finite groups. For methods such as Diffie−Hellman and Elliptic Curve cryptosystems, security depends on the contrast in difficulty between performing two group operations: exponentiation vs. discrete logarithm. The discrete log problem is believed to be hard compared to the exponentiation problem; and the elliptic curve discrete logarithm problem is even harder. This is because of its different algebraic structure, its somewhat complex arithmetic rules to "add" two points on an elliptic curve, and the lack of an index calculus method for the elliptic curve domain. The core of elliptic curve computation hinges on a key arithmetic operation—modular division. For decades we did not tackle modular division problems directly. Instead, we relied on a two−step approach to this problem: an inversion followed by a multiplication.

In this paper, we present an efficient technique for calculating modular division in $GF(2^m)$ modulo an irreducible binary polynomial, and in the prime integer field $F(p)$. Before we do so, we will first give an overview of several known algorithms for efficient finite field arithmetic operations. Through this survey, the readers can see how various algorithms have evolved over time, and appreciate what complicated arithmetic results simple repetitive binary adds−and−shifts can produce.

Over 2300 years ago, Euclid described, in Book 7 Propositions 1 and 2 of his *Elements* (c. 300 B.C.), a simple algorithm for finding the greatest common divisor of two integers. This algorithm was chosen by Euclid to be the very first step in his development of the theory of numbers. The greatest common divisor of two integers m and n, gcd(m, n), is the largest integer that evenly divides both m and n. The original form of the **Euclidean algorithm** employs recursive integer division and is based on the fact that gcd($m-kn$, n) = gcd(m, n) for any k. In other words, a common divisor of m and n is also a divisor of $m-kn$ and n. Conversely, a common divisor of $m-kn$ and n must also divide both m and n.

A binary greatest common divisor algorithm was devised by Stein in 1961 [14]. This algorithm is based on the following three simple facts: 1) If m and n are even, gcd(m, n) = 2 gcd($m/2$, $n/2$); 2) If m is even and n is odd, then gcd(m, n) = gcd($m/2$, n); and 3) If m and n are both odd, then $m-n$ is even and gcd(m, n) = gcd($m-n$, n). This algorithm relies solely on the subtraction, parity testing, and right shifting of even numbers, and requires no division. It is, thus, more suited for binary arithmetic.

An algorithm for calculating the modular inverse of an integer can be traced back to the Aryabhatta (A.D. 499) of northern India. The multiplicative inverse of an integer $m \ mod \ n$ exists if and only if m and n are relatively prime, i.e., gcd(m, n) = 1. We know that the greatest common divisor of two integers can be expressed as a linear combination of the two numbers. Therefore, we look for two integers r and s that satisfy the equation $m\,r + n\,s = 1$. Further, this linear equation says that $m\,r \equiv 1 \ mod \ n$; therefore, r is the inverse of $m \ mod \ n$. By reversing the steps in the Euclidean algorithm, one can derive r and s while calculating *gcd(m, n)*, see[5,9]. This reversed procedure to derive r and s is known as the **Extended Euclidean algorithm**.

The Extended Euclidean algorithm was later adapted for computing the **multiplicative inverse** of a binary polynomial over $GF(2^m)$ by Berlekamp in 1968 [1]. Given an element $p(t)$ in the field of residue classes modulo an irreducible polynomial $M(t)$ of degree m, Berlekamp's algorithm finds a polynomial $r(t)$ of degree $< m$ that satisfies $r(t)p(t) \equiv 1 \bmod M(t)$; thus $r(t)$ is the multiplicative inverse of $p(t) \ mod \ M(t)$. The idea underlying this algorithm is similar to that of the Euclidean algorithm. By expressing the polynomials in a linear relationship $r(t)\,p(t) + M(t)\,s(t) = 1$, one can derive $r(t)$ through an iterative process similar to the Extended Euclidean algorithm. Berlekamp further improved this technique by combining the iterations of the division and Euclid's algorithm into a unified procedure such that the algorithm requires no divisions, but only shift and addition operations. Variants of this algorithm have been used in the implementation of today's Elliptic Curve cryptosystems.

In 1985, Montgomery introduced an efficient technique for multiplying two integers modulo N. This method is commonly known as the **Montgomery multiplication** [11]. Like the Berlekamp method, this algorithm computes $\boldsymbol{xy\,2^{-n}} \ mod \ N$ through an iterative process of additions and shifts. Unlike the Euclidean and the Extended Euclidean algorithms, however, Montgomery multiplication does so without involving any division by N. By setting $S_0=0$, one can iteratively calculate $S_{i+1} = \dfrac{(S_i + x_i y)}{2}$ or $\dfrac{(S_i + x_i y + N)}{2}$, whichever is an integer; for i= 0,1,... $n-1$.

Here x_i is the binary bits of x and $x_i = 0$ or 1. The result of $\boldsymbol{xy\,2^{-n}} \ mod \ N$ is either S_n or $S_n - N$. This algorithm provides an efficient technique for computing modular exponentiations and is commonly used in RSA hardware accelerator designs.

Koc et al. [7] subsequently described the finite field analogue of the Montgomery Multiplication for modular multiplication in $GF(2^m)$. Kaliski [4] also presented a binary algorithm for computing the modular inverse of an integer $\boldsymbol{x}$ in Montgomery representation, i.e., $\boldsymbol{x^{-1}2^n} \bmod N$.

The above algorithms and many others not mentioned here have contributed greatly to the advancement of today's public−key cryptosystems. And, yet, one algorithm remained missing today—field division. For many years we have relied on the Extended Euclidean algorithm for calculating division: we computed division using an inversion followed by a multiplication.

This paper is organized as follows. In Section 2 we introduce a binary add−and−shift algorithm for calculating modular division in a polynomial field $GF(2^m)$ and in a prime integer field $F(p)$. The application of this algorithm in the context of elliptic curve cryptography is described in Section 3. We then discuss the time complexity of the algorithm in Section 4 by comparing it to other algorithms.

The fundamental contributions this paper makes are to add to our understanding of the binary algorithms for efficient field arithmetic operations, and to put in place the last missing piece of the puzzle in modular arithmetic, namely, division. This algorithm opens up a new avenue for the design of a field−division based elliptic curve crypto accelerator. The new algorithm uses simple iterative binary add−and−shift operations and is suitable for hardware implementation. Using wide−registers and bit−parallel logic circuits that can perform an addition and a shift in one clock cycle, a hardware accelerator can be built to dramatically enhance the efficiency of an elliptic curve cryptosystem.

2. Computing Modular Division in Galois Field $GF(2^m)$

We present an algorithm for field division in $GF(2^m)$ in this section. This algorithm takes as input two elements of $GF(2^m)$ represented as binary polynomials of degree $< m$, an irreducible polynomial of the field of degree m, and produces the residue, which is an element in $GF(2^m)$. Let $M(t)$ be the irreducible polynomial of the field and $y(t)$, $x(t)$ be the input polynomials. We need to compute an output $r(t)$, the residue, which is defined as

$$r(t) \equiv \frac{y(t)}{x(t)} \bmod M(t), \quad y(t) \equiv r(t)\, x(t) \bmod M(t).$$

We initialize four auxiliary registers A, B, U, and V with values: A← $x(t)$, B← $M(t)$, U← $y(t)$, and V ← 0 . The routine iteratively reduces the polynomials in A and B down to 1 while adjusting the values in U and V accordingly in order to maintain the following invariant relationship between the four registers:

$$A * \mathbf{y}(t) \equiv U * \mathbf{x}(t) \bmod M(t) \quad \text{and} \quad B * \mathbf{y}(t) \equiv V * \mathbf{x}(t) \bmod M(t).$$

We now describe the algorithm in some detail. Then we prove its correctness. On a practical note, a polynomial can be represented as a binary bit−string. Using such a representation, arithmetic operations are efficient and inexpensive. Additions can be done with bit−wise xor. Dividing a polynomial by t, A/t, can be accomplished with a right−shift operation. One can compare two polynomials, A>B, by treating their bit−strings as unsigned integers.

The division algorithm below is an iterative process of additions, parity−testings, and shifts. The process iterates while steadily reducing A and B. In every iteration, either A or B is reduced by one bit, thus, the entire division routine takes no more than $2(m-1)$ iterations.

```
Modular_Division _GF(2^m){
   A← x(t), B← M(t), U← y(t), V← 0;
   while ( A not equal B ) do {
         if ( A even )  then {
                  A <− A / t;
                  if  ( U even ) then U <− U / t else U <− (U+M) / t;
         } else if  ( B even ) then {
                  B <− B / t;
                  if  ( V even ) then V <− V / t else V <− (V+M) / t;
         } else if ( A > B ) then {
                  A <− (A+B) / t;  U <− U+V;
                  if  ( U even ) then U <− U / t else U <− (U+M) / t;
         } else {
                  B <− (A+B) / t; V <− U+V;
                  if  ( V even ) then V <− V / t  else V <− (V+ M) / t;
         }
   }
}
```

When the routine terminates, register U holds the result of the modular division. The degree of U and V is bounded by M throughout the entire process; therefore, U and V never exceed *m* bits. The least significant bit of U and V are zeroed prior to a reduction by adding M. This can be done because adding an irreducible polynomial of the field is algebraically equivalent to adding zero to the system. One notices that this algorithm is very similar in its style to the b*inary Euclidean algorithm* as described in [5, 9].

Throughout the entire division process, the polynomials in A, B, U, V obey the *invariant* relationship. This is true at initialization: A=**x***(t)*, U=**y***(t)*, B=M*(t)*≡ *0*, and V=*0*. Throughout the entire process, the invariant holds true because

if *U* is even then $\mathbf{y}(t)\,(A(t)/t) \equiv \mathbf{x}(t)\,(U(t)/t) \bmod M(t)$,

else $\mathbf{y}(t)\,(A(t)/t) \equiv \mathbf{x}(t)\,((U(t)+M(t))/t) \bmod M(t)$;

if *V* is even then $\mathbf{y}(t)\,(B(t)/t) \equiv \mathbf{x}(t)\,(V(t)/t) \bmod M(t)$,

else $\mathbf{y}(t)\,(B(t)/t) \equiv \mathbf{x}(t)\,((V(t)+M(t))/t) \bmod M(t)$;

$(A(t)+B(t))\ \mathbf{y}(t) \equiv (U(t)+V(t))\ \mathbf{x}(t) \bmod M(t)$.

At the end A=1, the result in the U register satisfies $\mathbf{y}(t) \equiv U(t)\ \mathbf{x}(t) \bmod M(t)$; thus *U(t)* is the residue of the modular division. This algorithm works when *x(t)* and *M(t)* are relatively prime, and the routine terminates with A=B=1. This is always the case for elliptic curve cryptographic applications with well−formed curves. When *x(t)* and *M(t)* are not relatively prime, the routine terminates with the greatest common divisor of the two polynomials, A= B= gcd(*x(t)*, *M(t)*).

This routine can also perform an inversion of **x***(t)* by initializing the U−register with "1". Inversion is a special case of division, and in many cases, is cheaper to compute than a full division. This is not the case here. With this algorithm, the costs of inversion and division are the same because the running time is determined by the degrees of **x***(t)* and M(t) polynomials in A and B. Different initial values of U do not affect the time complexity of this routine.

This algorithm can be adapted for division in integer fields. Integer modular divisions are required in the computation of Elliptic Curve digital signature generation and verification. We

further optimize the algorithm and improve its efficiency such that a division in integer fields can be completed within $2(m-1)$ steps:

```
Modular_Division_F(p) {
   A← x, B← M, U← y, V← 0;
   while ( A not equal B ) do {
        if ( A even ) then {
                A <− A/2;
                if  ( U even ) then U <− U/2; else U <− (U+M)/2;
        } else if  ( B even ) then {
                B <− B/2;
                if  ( V even ) then V <− V/2; else V <− (V+M)/2;
        } else if ( A > B ) then {
                A <− (A − B)/2;
                U <− U − V;   if  ( U<0 ) then U <− (U+M);
                if  ( U even ) then U <− U/2; else U <− (U+M)/2;
        } else {
                B <− (B − A)/2;
                V <− V−U;  if  ( V<0 ) then V <− (V+M);
                if  ( V even ) then V <− V/2; else V <− (V+ M)/2;
        }
   }
}
```

3. Implementing Elliptic Curve Systems

Elliptic curve cryptosystem was first proposed in 1985 independently by Koblitz [6] and Miller [10]. The security of such a scheme relies upon the difficulty of the Elliptic Curve Discrete Logarithm problem. The core of elliptic curve arithmetic is an operation that computes $\boldsymbol{kP}$ by adding $\boldsymbol{k}$ copies of an elliptic curve point $\mathbf{P}$. For the fields $GF(2^m)$, this *point multiplication* operation can be performed in affine space through a combination of point doublings and point additions using the algebraic formulae below. A simple and straight forward double−and−add approach based on the binary expansion of the multiplier $\boldsymbol{k}$ will require $(m-1)$ point doublings and an average of $(m-1)/2$ point additions, where m is the size of the field.

Point addition: R = P + Q	Point doubling: R = 2 P
slope $s = (y_P - y_Q) / (x_P + x_Q)$	***Slope*** $s = x_P + y_P / x_P$
$x_R = s^2 + s + a + x_P + x_Q$ $y_R = s * (x_P + x_R) + x_R + y_P$	$x_R = s^2 + s + a$ $y_R = x_P^2 + (s + 1) * x_R$
If Q = −P, R = P + (−P) = **O**, infinity	If $x_P = 0$, then R = **O**, infinity

The rules for doubling an elliptic curve point and for adding two elliptic curve points involve a field division, either y_P/x_P or $(y_P-y_Q)/(x_P+x_Q)$. Traditionally, this field division is computed using an inversion $(1/x_P)$ and the multiplication of $1/x_P$ and y_P. Alternatively, some have avoided field inversions by performing elliptic curve computation in projective space [2, 12]. Using the projective coordinates version of Montgomery scalar multiplication as described in [8], for example, elliptic curve point computation can be done using six multiplications. This technique defers field inversions to the end of a point multiplication computation. One of the problems of projective space implementations is that the efficient routines are unable to take advantage of table look−up pre−computation.

An efficient inversion routine was described by Schroeppel et al. [13]. They optimized the algorithm for various computer architectures using either 32−bit or 64−bit word sizes, and the running time reported was roughly one field inversion to three multiplications.

```
Schroeppel, Orman, and O'Malley Inversion Routine {
    A← x(t), B← M(t), U←1, V← 0,  k←0;
    repeat {
        while ( A even ) do
                A= A/ t, V= V*t, k=k+1;
        if ( A=1 ) then return (U, k);
        if ( deg(A) < deg(B) ) then exchange A,B  and exchange U,V;
        A <−  A+B,  U<− U+V;
    }
}
```

Our division algorithm is very similar to the algorithm by Schroeppel et al. The new algorithm also allows direct use of the simple formulae for computation in affine space. We restructure the division algorithm so that a reader can easily compare the complexity of the two algorithms:

```
Modular_Division _GF(2^m){
    A← x(t), B← M(t), U← y(t), V← 0;
    repeat {
        while ( A even ) do
                { A = A/ t; if  ( U even ) then U  = U/ t  else U = (U+M) / t; }
        if ( A=1 ) then return U;
        if ( A < B ) then exchange A, B and exchange U,V;
        A <− A+B, U <− U+V;
    }
}
```

The algorithm by Schroeppel et al. does only a partial inversion, and requires an additional step at the end to divide out the t^k factor. Such a step is not needed in the division routine.

4. Complexity Analysis

There are several ways to describe the running time complexity of an algorithm: by a standard complexity analysis that does step counting, by a hardware model that counts the number of clock cycles needed to execute an algorithm in a chip, by a software model that counts the number of 32−bit or 64−bit arithmetic operations, or by actual performance timing. From the view point of a standard complexity analysis, the two algorithms have the same time complexity because both routines require $2(m-1)$ steps to complete a field operation. Comparing the inner loop complexity, the inversion routine requires one integer addition to perform $k=k+1$, whereas the division routine requires a polynomial addition to perform $U=(U+M)/t$. This polynomial addition is executed only 50% of the time on average, and can be accomplished using two 32−bit or 64−bit xor's. This is because an irreducible polynomial for practical elliptic curve applications is either a trinomial or a low−weight pentanomial. Thus, the two algorithms have roughly equivalent inner loop complexity.

Several programming optimization tricks were used by Schroeppel et al. to improve the performance of their algorithm, including inline coding within the inversion routine, using separate variables instead of arrays to hold the polynomials, duplicating the code to avoid swapping the polynomial variables, and, special coding to minimize unnecessary computation. The same optimization techniques can be applied to the new algorithm.

The new algorithm has two clear advantages in performance. First, it computes a division directly whereas an inversion routine needs an additional multiplication to compute y_P/x_P. Thus, this new algorithm is one−multiplication−time more efficient in the context of elliptic curve computation. Second, the new algorithm does not need to divide out a factor t^k at the end. Such a step can be costly for fields with pentanomial irreducible polynomials. In addition, the simplicity of this new algorithm allows for compact implementation and makes the algorithm more suited for 8−bit or 16−bit computing environments where memory space is limited. In a hardware environment, the algorithm presents a significant improvement. Using wide registers and bit−parallel logic circuits that can perform an addition and a shift in a single clock cycle, the cost of a field division can be brought down further, close to two multiplication times.

5. Conclusion

This paper presents an efficient algorithm for computing field divisions in $GF(2^m)$ and $F(p)$. Field division operations dominate the overall cost of elliptic curve computations. This operation is required in every point addition and point doubling calculation. Improving the efficiency of division can significantly increase the overall performance of elliptic curve cryptosystems.

This new algorithm enables us to compute y/x in one single operation, instead of computing the residue as the product of $1/x$ and y. Furthermore, the running time for computing a field

division ***y/x*** using this new algorithm is roughly the same as the time for computing an inversion **1/*x***. Therefore, this new algorithm is one−multiplication−time more efficient in the context of elliptic curve computations. An efficient division algorithm such as this has not appeared thus far in the literature [3].

The algorithm opens up a new avenue for the design of field−division based hardware elliptic curve crypto accelerators. It also makes the implementation of low−cost field−dividers viable in restricted computing environments. ASIC processors based on this field division algorithm can now be built to accelerate elliptic curve computations in low−power consumption devices such as smart cards and wireless phones.

This new algorithm also presents advantages for software implementation in small 8−bit or 16−bit computing environments where memory space is constrained, such as the Palm platform. First, the algorithm allows direct use of the simple point addition and point doubling formulae in affine space. The algebraic formulae involved for computation in projective space is significantly more complicated. Second, the simplicity of this new algorithm also lends itself to a small and efficient implementation of the division routine. Therefore, the overall point multiplication implementation using this division algorithm is more compact compared to using projective techniques such as Montgomery scalar multiplication.

Elliptic curve cryptosystems today have the smallest key−size requirement of all practical public−key systems, and are considered the preferred method for small restricted computing environments. As computers get faster, longer key−lengths will be required to keep these cryptosystems secure. Elliptic curve systems will play an increasingly important role in such environments and this division algorithm will be the key to a highly efficient implementation of elliptic curve cryptosystems.

References:

1. E. Berlekamp, Algebraic Coding Theory, Aegean Park Press, 1984.
2. I. Blake, G. Seroussi, and N. Smart, Elliptic Curves in Cryptography, London Math. Society Lecture Note 265, Cambridge University Press, 1999.
3. R. Harley, http://cristal.inria.fr/~harley/, personal communication.
4. B. Kaliski, "The Montgomery Inverse and its Applications," IEEE Trans. Comp., 44, 1995.
5. D. Knuth, The Art of Programming, Vol. 2, Addison−Wesley, 3rd edition, 1998.
6. N. Koblitz, "Elliptic curve cryptosystems," Mathematics of Computation, 48, 1987.
7. C. Koc and T. Acar, "Montgomery multiplication in GF(2^k)," Designs, Codes and Cryptography, 14, 1998. Also presented at the 3rd Annual Workshop on Selected Areas in Cryptography, Queen's University, Canada, August 1996.
8. J. Lopez and R. Dahab, "Fast Multiplication on elliptic curves over G$F(2^m)$ without precomputation," CHES '99 Proceedings, Springer−Verlag, August 1999.
9. A. Menezes, P. van Oorschot, and S. Vanstone, Handbook of Applied Cryptography, CRC Press, 1997.
10. V. Miller, "Use of elliptic curves in cryptography," Advances in Cryptology, CRYPTO 85, Springer−Verlag, 1986.
11. P. Montgomery, "Modular multiplication without trial division," Mathematics of Computation, 44, 1985.
12. P1398, Standard Specifications for Public−key Cryptography, IEEE, October 1998.
13. R. Schroeppel, H. Orman, and S. O'Malley, "Fast Key Exchange with Elliptic Curve Systems," Tech. Report 95−03, Department of Computer Science, The University of Arizona, Tucson.
14. J. Stein, "Computational problems associated with Racah algebra," Journal of Computational Physics, 1, 1967.

About the Author

Sheueling Chang Shantz is a Distinguished Engineer at Sun Microsystems Laboratories. She joined Sun Microsystems in 1984 with an M.S. degree and a Ph.D. Degree in Computer Science from the California Institute of Technology and a B.S. Degree in Electric Engineering from the National Taiwan University. She also received a Master in Business Management from Stanford Business School in 1998. Her background includes 2D and 3D graphics rendering algorithms, high resolution printing and halftoning, electronic commerce, Internet banking and payment systems, networking and security, and RSA and Elliptic Curve public key cryptography.

Her current areas of interest include developing secure communication technology for small handheld wireless devices, developing efficient algorithms for Elliptic Curve cryptographic systems, and designing special−purpose hardware accelerators for enhancing cryptographic functionalities and performance.