Efficient Computation of Multiplicative Inverses for Cryptographic Applications

M. A. Hasan
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario, N2L 3G1, Canada
Email: ahasan@ece.uwaterloo.ca

Abstract

Among the basic arithmetic operations over finite fields, the computation of a multiplicative inverse is the most time consuming operation. In this article, a number of methods are presented to efficiently compute the inverse using the extended Euclidean algorithm. The proposed methods can significantly reduce the computation time over large fields where the field elements are represented using a multi-precision format. A hardware structure for the inverter is also presented. The structure is area efficient and is suitable for resource constrained systems. Additionally, an application of the proposed inversion algorithm is given in the context of elliptic curve cryptography.

1. Introduction

A finite field is a set of a finite number of elements where one can perform the basic arithmetic operations, namely addition, subtraction, multiplication and inversion (of nonzero elements) without leaving the set [1], [2]. The finite field GF(2^n), where n is a nonzero positive integer, has 2^n elements and is an extension of the ground field GF(2) which has only two elements. The important applications of the extension fields include cryptography [3, 4] and error control coding [5]. In cryptographic applications, the number of field elements can be as high as 2^{1024} or more. Both IEEE and ANSI have included the extension field in their recent standard specifications for cryptographic applications.

Since there are exactly 2^n binary polynomials of degree less than n, all the 2^n elements of GF(2^n) can be uniquely represented in a polynomial form which is known as the *polynomial* basis representation. For such representation, the four basic arithmetic operations can be informally described as follows: The addition/subtraction of two field elements is simply mod 2 addition of the coefficients of the polynomials corresponding to the elements. To multiply two elements, one can first multiply their polynomial representations to obtain an intermediate polynomial of degree up to $2n-2$. Since the product of the two elements is also an element in the field, the polynomial of the product should have a degree less than n. This can be achieved by taking the intermediate polynomial modulo an irreducible binary polynomial of degree n. Finally, in order to find the inverse of a nonzero element, one needs to find an(other) element such that the product of their polynomial representations modulo the irreducible polynomial is the multiplicative identity which is simply 1 in the polynomial basis representation.

Among the four basis arithmetic operations, inversion is the most time consuming operation. In the past, several algorithms for computing inverses over GF(2^n) were proposed (for example, see [6, 7, 8, 9, 10, 11]). The underlying computations on which these algorithms rely on can be broadly divided into the following categories:

- Repeated squaring-and-multiplications in GF(2^n),
- Use of the extended Euclidean algorithm over GF(2),
- Solution of a system of linear equations over GF(2).

In this article, we consider the computation of inverses using the extended Euclidean algorithm (EEA) which is believed to be the most widely used scheme for inverses in large fields. Although, the EEA can be applied to a number of bases that one can use to represent the field elements [11], in this article we restrict our discuss on the polynomial basis only. Recent articles on the computation of inverses using the EEA include [8] and [12]. These articles are primarily for hardware realizations and require dedicated resources. Here, we develop methods to reduce the computation time for inverses, especially for large fields where multi-precision is used to deal with the large number of bits needed to represent the field elements. Our proposed methods include updating of the EEA polynomials up to the exact precision, reducing the delay in degree determination, and two dimensional representation of the degrees of the polynomials involved.

The outline of the remainder of this article is as follows. In the next section we present the conventional approach to

0-7695-1150-3/01 $10.00 © 2001 IEEE

use the EEA to compute inverses. Then in section 3, we develop the proposed methods to efficiently compute the inverse. An architecture of hardware realization of the inverter is presented in section 4. Then, in section 5 an application of the proposed inverse scheme is discussed and the expected improvement is quantified for elliptic curve based cryptographic systems. Finally, a few concluding remarks are given in section 6.

2. Conventional Approach

Let $F(x)$ be an irreducible polynomial of degree n over GF(2) and let this polynomial define the representation of the elements of the field GF(2^n). Let A be a non-zero element of GF(2^n) and $A(x) \triangleq \sum_{i=0}^{n-1} a_i x^i$ be its polynomial basis representation where all a_i's belong to GF(2) and deg $A(x) \leq n-1$. Since $F(x)$ and $A(x)$ are relatively prime, their greatest common divisor

$$\gcd(F(x),\ A(x)) = 1.$$

Thus, there exists a polynomial $B(x)$ of degree less than n, which for some $C(x)$ satisfies the following [2]:

$$A(x)B(x) + F(x)C(x) = 1,$$

$$i.e.,\ \ A(x)B(x) = 1 \mod F(x).$$

Thus $B(x)$ is the polynomial basis representation of the multiplicative inverse of A and it can be recursively computed using the extended Euclidean algorithm (EEA) as follows:

Algorithm 1 (Conventional Inversion)
Input: $F(x)$ and $A(x) \neq 0$
Output: $B(x)$ such that $A(x)B(x) = 1 \mod F(x)$

Step 1. $R^{(-1)}(x) := F(x), R^{(0)}(x) := A(x)$
$U^{(-1)}(x) := 0, U^{(0)}(x) := 1$
$i := 0$

Step 2. do {
$i := i + 1$
$Q^{(i)}(x) := \lfloor R^{(i-2)}(x)/R^{(i-1)}(x) \rfloor$
$R^{(i)}(x) := R^{(i-2)}(x) + Q^{(i)}(x)R^{(i-1)}(x)$
$U^{(i)}(x) := U^{(i-2)}(x) + Q^{(i)}(x)U^{(i-1)}(x)$
} while ($R^{(i)}(x) \neq 0$)

Step 3. $B(x) := U^{(i-1)}(x)$

In Step 2, $Q^{(i)}(x) := \lfloor R^{(i-2)}(x)/R^{(i-1)}(x) \rfloor$ is the quotient polynomial resulted from dividing $R^{(i-2)}(x)$ by $R^{(i-1)}(x)$. Although it is not explicitly shown above, the computation of $Q^{(i)}(x)$ (and hence $R^{(i)}(x)$) requires locating the leading coefficient of each intermediate remainder polynomial that results during the division operation. As a result, in a straight-forward realization, the computation of $Q^{(i)}(x)$ and $R^{(i)}(x)$ is an iterative process and may take a considerable amount of the loop time in Step 2.

3. Efficient Inversion over Large Fields

In cryptographic applications, the value of n can be as large as 1024 or more. When cryptographic functions are implemented using a general purpose processor, each n-bit element of GF(2^n) is usually presented as an $s = \lceil n/r \rceil$ word operand where r corresponds to the width of the processor's datapath. For such multi-precision representation, each polynomial addition in Algorithm 1 corresponds to s XOR instructions (assuming that the underlying processor has a word level XOR instruction). Thus, each iteration of the loop in the above algorithm requires two polynomial additions which in turn requires $2s$ XOR instructions.

In Algorithm 1, as the degrees of the $R^{(\cdot)}(x)$ polynomials decrease, the degrees of $U^{(\cdot)}(x)$ polynomials increase. In the following, we discuss a systematic way to use this property of the polynomials to reduce the number of XOR instructions needed to compute an inverse.

3.1. Updating Polynomials up to the Exact Precision

In Algorithm 1, $R^{(i)}(x)$ and $Q^{(i)}(x)$ are the remainder and quotient polynomials, respectively, obtained by dividing $R^{(i-2)}(x)$ with $R^{(i-1)}(x)$. As a result,

$$\deg R^{(i)}(x) \ < \ \deg R^{(i-1)}(x), \qquad (1)$$

and

$$\begin{aligned} \deg R^{(i-1)}(x) &= \deg R^{(i-2)}(x) - \deg Q^{(i)}(x) \\ &= \deg R^{(i-3)}(x) - \deg Q^{(i-1)}(x) \\ &\quad -\deg Q^{(i)}(x) \\ &\ \ \vdots \\ &= \deg R^{(-1)}(x) - \sum_{j=1}^{i} \deg Q^{(j)}(x). \quad (2) \end{aligned}$$

Also, since $\deg U^{(i-1)}(x) > \deg U^{(i-2)}(x)$,

$$\begin{aligned} \deg U^{(i)}(x) &= \deg U^{(i-1)}(x) + \deg Q^{(i)}(x) \\ &= \deg U^{(i-2)}(x) + \deg Q^{(i-1)}(x) \\ &\quad \deg Q^{(i)}(x) \\ &= \deg U^{(0)}(x) + \sum_{j=1}^{i} \deg Q^{(j)}(x). \quad (3) \end{aligned}$$

Using (2) and one of the initial conditions $\deg U^{(0)}(x) = 0$ of Algorithm 1, we can write (3) as follows:

$$\deg U^{(i)}(x) = \deg R^{(-1)}(x) - \deg R^{(i-1)}(x). \quad (4)$$

Since $R^{(-1)}(x) = F(x)$ and deg $F(x) = n$, from (1) and (4), we can write

$$\deg U^{(i)}(x) + \deg R^{(i-1)}(x) = n, \quad (5)$$

$$\deg U^{(i)}(x) + \deg R^{(i)}(x) < n. \quad (6)$$

In Algorithm 1, both $U^{(i)}(x)$ and $R^{(i)}(x)$ are calculated in the same iteration. As a result, in each iteration a maximum of $n+1$ coefficients including the constant terms of $U^{(i)}(x)$ and $R^{(i)}(x)$ are to be determined, and towards this effort the maximum number of XOR instructions required is $s+1$, i.e.,

$$\left\lceil \frac{\deg [U^{(i)}(x)] + 1}{r} \right\rceil + \left\lceil \frac{\deg [R^{(i)}(x)] + 1}{r} \right\rceil \leq s+1 \quad (7)$$

where $s = \lceil \frac{n}{r} \rceil$.

In order to limit the total number of XOR instructions to $s+1$ per iteration as shown in (7), one needs to know how these XOR instructions are split between $U^{(i-1)}(x)$ and $R^{(i-1)}(x)$. In this effect, note that in the ith iteration, $R^{(i-1)}(x)$ is already known; hence deg $U^{(i)}(x)$ can be found in advance and one can use (5) to determine deg $U^{(i)}(x)$ before computing the coefficients of the polynomial $U^{(i)}(x)$ itself. This enables us to determine the split of the $s+1$ XOR instructions between $U^{(i)}(x)$ and $R^{(i)}(x)$, and hence a way to reduce the complexity of computing an inverse.

3.2. Delay Reduction in Degree Determination

Since deg $R^{(i-2)}(x) >$ deg $R^{(i-1)}(x)$, hence deg $Q^{(i)}(x) > 0$. Assuming that $Q^{(i)}(x)$ has m_i nonzero coefficients, one can write

$$Q^{(i)}(x) = x^{d^{(i,0)}} + x^{d^{(i,1)}} + \cdots + x^{d^{(i,m_i-1)}}$$

where

$$d^{(i,0)} > d^{(i,1)} > \cdots > d^{(i,m_i-1)} \geq 0$$

and $d^{(i,0)} =$ deg $Q^{(i)}(x) > 0$. Since $R^{(i)}(x) = R^{(i-2)}(x) + Q^{(i)}(x)R^{(i-1)}(x)$, in the process of generating $R^{(i)}(x)$ by dividing $R^{(i-2)}(x)$ with $R^{(i-1)}(x)$ one obtains $m_j + 1$ intermediate reminders. Denoting these remainders as $R^{(i-2,j)}(x)$, for $0 \leq j \leq m_j$, we have the following:

$$R^{(i-2,0)}(x) = R^{(i-2)}(x) \text{ and}$$

$$R^{(i-2,j+1)}(x) = R^{(i-2,j)}(x) + x^{d^{(i,j)}} R^{(i-1)}(x), \quad 0 \leq j \leq m_j - 1.$$

Notice that $R^{(i-2,m_i)}(x) = R^{(i)}(x)$ and

$$d^{(i,j)} = \deg R^{(i-2,j)}(x) - \deg R^{(i-1)}(x) = \deg R^{(i-2,j-1)}(x) - \delta^{(i,j)} - \deg R^{(i-1)}(x) \quad (8)$$

where $\delta^{(i,j)} =$ deg $R^{(i-2,j-1)}(x) -$ deg $R^{(i-2,j)}(x)$. The determination of $\delta^{(i,j)}$ is usually a sequential process and can constitute a significant portion of the iteration time. Since, deg $R^{(i-2,j)}(x) <$ deg $R^{(i-2,j-1)}(x)$, one can start searching for the (non-zero) leading coefficient of $R^{(i-2,j)}(x)$ from the e-th coefficient where $e =$ deg $R^{(i-2,j-1)}(x) - 1$. In this process, the number of bits one tests is $\delta^{(i,j)}$.

In order to speed up the process of determining $\delta^{(i,j)}$, we can use a look-up table (LUT). The inputs to the LUT are the least significant $e+1$ bits of $R^{(i-2,j)}(x)$ and the output is $\delta^{(i,j)}$. Since both e and $\delta^{(i,j)}$ can be up to n, the size of the LUT is $2^n \log_2 n$ bits. This table is however too large to be used in most practical systems even for a moderate value of n.

In an effort to reduce the size of the table, note that $\delta^{(i,j)}$ corresponds to the number of zero bits at the significant (leading) end of the e-bit representation of $R^{(i-2,j)}(x)$. Assuming that 0 and 1 are equally likely to occur, the probability of having one zero is greater than the probability of having two zeros at the significant end, and in general, we have

$$\Pr\{\delta^{(i,j)} = l\} = 2\Pr\{\delta^{(i,j)} = l+1\}, \quad \text{for } l \geq 1. \quad (9)$$

As a result, we can use only a few (say, $g < n$) bits from the significant end of the e bits of $R^{(i-2,j)}(x)$ to correctly determine $\delta^{(i,j)}$ with only one LUT read operation in most cases. More specifically, when not all the g bits are zeros, which happens with a probability of $1 - 1/2^g$, the value of $\delta^{(i,j)}$ is determined simply by reading the LUT. However, when all the g bits are zeros (detection of the all-zero condition is simple and fast), $\delta_{i,j}$ is set to g and the next g bits are checked. If the new g bits are all zeros, the value of $\delta_{i,j}$ is increased by g. This process is repeated until there is at least one nonzero bit in the g bit group, at which point the table is accessed to obtain a value in the range $[0,\ g-1]$ which is then added to $\delta_{i,j}$.

Assuming that the g bits are directly used to address the LUT, its i-th location has

$$\text{LUT}[i] = g - 1 - \lfloor \log_2 i \rfloor, \quad 1 \leq i \leq 2^g - 1.$$

The size of the above LUT is $2^g \log_2 g$ bits. For the the sake of easy implementation, we should chose g such that $g|r$. For a processor with a 32-bit wide datapath, a practical value of g appears to be four, for which the contents of the LUT are shown below. For hardware realization with memory constraints, the table size can be easily reduced at the expense of increased time complexity.

Based on the above discussions, now we have an inversion scheme as described in Algorithm 2 below. Note that although the following algorithm appears to be longer than Algorithm 1, there is no implicit task of determining the degree other than what is already shown in the description of

Location	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Contents	-	3	2	2	1	1	1	1	0	0	0	0	0	0	0	0

the algorithm. Also, the polynomials are updated only up to the exact precision, which reduces the number of XOR instructions as well as the number of storage registers needed.

Algorithm 2 (Efficient Inversion)
Input: $F(x)$ and $A(x) \neq 0$
Output: $B(x)$ such that $A(x)B(x) = 1 \bmod F(x)$

Step 1. $R^{(-1)}(x) := F(x), R^{(0)}(x) := A(x)$
$U^{(-1)}(x) := 0, U^{(0)}(x) := 1$
deg $R^{(-1,0)}(x) :=$ deg $R^{(-1)}(x) = n$,
find deg $R^{(0)}(x)$
$d^{(1,0)} = n -$ deg $R^{(0)}(x), i = 0$

Step 2. do {
$i := i + 1$
$j := 0$
$R^{(i-2,0)}(x) := R^{(i-2)}(x)$
$U^{(i-2,0)}(x) := U^{(i-2)}(x)$
while $(d^{(i,j)} \geq 0)$ do {
$R^{(i-2,j+1)}(x) := R^{(i-2,j)}(x) - x^{d^{(i,j)}} R^{(i-1)}(x)$
/* update $R^{(i-2,j+1)}(x)$ only up to coeff of $x^{\deg R^{(i-2,j)}(x)}$ */
$U^{(i-2,j+1)}(x) := U^{(i-2,j)}(x) - x^{d^{(i,j)}} U^{(i-1)}(x)$
/* update $U^{(i-2,j+1)}(x)$ only up to coeff $x^{n-\deg R^{(i-1)}(x)}$ */
$j := j + 1$
Find $\delta^{(i,j)}$ from LUT
deg $R^{(i-2,j)}(x) :=$ deg $R^{(i-2,j-1)}(x) - \delta^{(i,j)}$
$d^{(i,j)} :=$ deg $R^{(i-2,j)}(x) -$ deg $R^{(i-1)}(x)$
}

$R^{(i)}(x) := R^{(i-2,j)}(x)$ /* $j = m_i$ */
$U^{(i)}(x) := U^{(i-2,j)}(x)$
deg $R^{(i)}(x) :=$ deg $R^{(i-2,j)}(x)$
} while $(R^{(i)}(x) \neq 0)$

Step 3. $B(x) := U^{(i-1)}(x)$

3.3. Two Dimensional Representation of Degree

In order to represent the polynomial $A(x) = \sum_{i=0}^{n-1} a_i x^i$ in a processor with a datapath width of r bits ($r < n$), one needs $\lceil \frac{n}{r} \rceil$ words or registers of the processor. Assume that the least significant r coefficients of $A(x)$ are stored in word 0 and the next r significant coefficients are in word 1 and so on. Then the determination of deg$A(x)$ requires the search of the leading coefficient which may start in word i and end in word j where $j \leq i$. When $i - j > 1$, the realization of such a search is greatly simplified by a single instruction of the processor which checks the contents of an entire word or register against zero. To take advantage of this feature, the degree of the polynomial needs to be converted to the corresponding word number and bit number. Such representation can also speed-up the task of polynomial multiplications. As a result, it is convenient to represent the degree d of a polynomial as a pair of coordinates (d_1, d_2) such that

$$d = rd_1 + d_2$$

where d_1 is the word number in which the leading coefficient is stored and d_2 is the corresponding bit position in word d_1.

In most processors, the value of r is a power of 2. As a result, one can easily determine d_1 and d_2 by simple SHIFT and AND instructions. The reconstruction of d from d_1 and d_2 can also be easily performed with SHIFT and OR instructions.

3.4. Comments

- We have implemented Algorithms 1, 2 and the *almost-inverse* algorithm of [13] using the C programming language. A truly fair comparison of the computation time of these algorithms is difficult to do since it depends on the details of the implementations. Nevertheless, in our implementations we have observed that Algorithm 2 can provide a significant speed-up compared to the others. For example, while considering the field GF(2^{191}) we have found that Algorithm 2 is twice as fast as the almost-inverse algorithm.

- For a very high speed implementation on general purpose processors, one can use the well known optimization techniques, such as loop unrolling, multiple copies of the variables to avoid swapping, etc. More on this can be found in [13].

  Also, one can combine Algorithm 2 with the almost-inverse algorithm. To this end, our efforts however show that only a marginal improvement is obtained over the sole use of Algorithm 2. This is perhaps mainly because of the extra operation of x^{-k} mod $F(x)$, $k \in [0, 2n-1]$, needed for the almost-inverse algorithm. An efficient realization of this operation requires the second leading coefficient of the field defining polynomial $F(x)$ to have a degree greater than or equal to r.

- Unlike the algorithm of [14] and [15] where n is assumed to be a composite number and the EEA is applied over a sub-field of GF(2^n), Algorithm 2 of this article can be used for any values of n including the primes. In order to reduce the risk of possible attacks, recently there has been a trend to use the field GF(2^n) with n being a prime. As a result, the proposed inverse scheme appears to be a better candidate for the realization of cryptographic functions for practical applications.

4. Hardware Architecture

The algorithm presented in section 3 can also be used for an efficient hardware realization of the finite field inversion operation. A two-bus architecture for such realization, where the dimension n of the field GF(2^n) is larger than the datapath width r, is shown in Figure 1. Only the important building blocks of the architecture are included in the figure and an informal description of the architecture is given below.

There are two sets of r-bit shift registers labeled as $\text{REG}_{L,0}$, $\text{REG}_{L,1}$, ..., $\text{REG}_{L,s}$ on the left hand side, and $\text{REG}_{R,0}$, $\text{REG}_{R,1}$, ..., $\text{REG}_{R,s}$ on the right hand side of the buses. The left hand side registers are initialized with $R^{(-1)}(x)$ and $U^{(-1)}(x)$ whereas the right hand side registers are initialized with $R^{(0)}(x)$ and $U^{(0)}(x)$. The higher order coefficients of the polynomials are stored at the upper end of the registers, e.g., $U^{(-1)}(x)$ and $U^{(0)}(x)$ corresponds to a 1 and a 0 in the bottom cell of $\text{REG}_{L,s}$ and $\text{REG}_{R,s}$, respectively. The significant g bits from each of $\text{REG}_{L,0}$ and $\text{REG}_{R,0}$ are connected to the LUT and the zero-detect block through a MUX which is operated by the control unit. The latter also provides shift-, in- and out-signals to the registers so that the matching coefficients from the polynomials stored in the two sets of registers can be added in the ALU. The number of shifts depends on the value of $\delta^{(i,j)}$ which is determined by the control unit based on reading the LUT and the outcome of the zero-detect block as discussed in section 3.

The area complexity of the architecture, as it can be seen in the figure, is $O(n)$. On the other hand, since the inner loop of Algorithm 2 is executed a maximum of $2n$ times, the time complexity of the architecture is $O(ns)$.

5. Applications in Elliptic Curve Cryptography

Inverses over GF(2^n), where n is large, have usages in cryptography. In applications, where the bandwidth for communications or the space for storage is limited, the choice of elliptic curve based cryptography can be advantageous. For the finite field GF(2^n) of characteristic two, the standard equation for an elliptic curve is

$$Y^2 + XY = X^3 + \alpha X^2 + \beta \tag{10}$$

where $\alpha, \beta \in$ GF(2^n) and $\beta \neq 0$. The points on the curve are of the form $P = (X, Y)$, referred to as affine coordinates, and X and Y are elements of GF(2^n). The solutions (X, Y) to equation (10), along with a special point $\mathcal{O}$ called the point at *infinity*, form a commutative finite group under the following addition operation [17], [18], [19], [20].

Let $P = (X, Y)$ be a point on (10). The inverse of P is defined as $-P = (X, X + Y)$. The point $\mathcal{O}$ is the group identity, i.e., $P \uplus \mathcal{O} = \mathcal{O} \uplus P = P$, were $\uplus$ denotes the elliptic curve group operation (i.e., addition). If $P_0 = (X_0, Y_0) \neq \mathcal{O}$ and $P_1 = (X_1, Y_1) \neq \mathcal{O}$ are two points on E and $P_0 \neq -P_1$, then the result of the addition $P_0 \uplus P_1 = P_2 = (X_2, Y_2)$ is given as follows.

$$X_2 = \begin{cases} \left(\frac{Y_0+Y_1}{X_0+X_1}\right)^2 + \frac{Y_0+Y_1}{X_0+X_1} + X_0 + X_1 + \alpha, & P_0 \neq P_1, \\ X_0^2 + \frac{\beta}{X_0^2}, & P_0 = P_1, \end{cases}$$

$$Y_2 = \begin{cases} \left(\frac{Y_0+Y_1}{X_0+X_1}\right)(X_0 + X_2) + X_2 + Y_0, & P_0 \neq P_1, \\ X_0^2 + \left(X_0 + \frac{Y_0}{X_0}\right) X_2 + X_2, & P_0 = P_1. \end{cases}$$

In the above formulas, the cases of $P_0 \neq P_1$ and $P_0 = P_1$ correspond to elliptic curve point addition and doubling, respectively. They are used for computing the scalar multiplication kP, i.e. add k copies of P, where k is a large integer (of about n bits) and P is a point on the curve (10). The scaler multiplication is fundamental to elliptic curve based cryptographic functions, such as, secret key generation, key exchange, signing and verification.

The above formulas for point addition are based on the affine coordinates of the elliptic curve points and require inverses over GF(2^n). Other coordinate systems have been proposed, mainly to avoid the costly inverse in finite fields (see for example [20]). This is however achieved at the expense of extra multiplications over GF(2^n) and causes a significant increase in the number of storage registers needed to temporarily hold the intermediate results. As a result, in resource constrained systems, for which the elliptic curve cryptography has an edge over its counterparts, affine coordinates appear to be a good candidate for practical implementation, especially if the inverse is not too slow compared to the multiplication. Moreover, to reduce processing time the elliptic curve in (10) is often simplified by setting $\beta = 1$ and limiting α to be either 1 or 0. Such curves, known as Koblitz curves, can eliminate the need for elliptic curve point doubling operations in computing the scalar multiplication kP [20]. Thus, it is the point addition operation that dominates the computational complexity of the scalar multiplication on the Koblitz curve.

From the point addition formulas given above, we see that the elliptic curve point addition mainly requires one inverse and two multiplications over GF(2^n). The computation times for squaring and additions over GF(2^n) are small compared to those of the inverse and the multiplication. Thus, the time for an elliptic curve point addition is essentially $I + 2M$, where I and M correspond to the computation times for the field inverse and multiplication, respectively. Depending on the implementation, the factor $\mu \triangleq I/M$ can vary considerably. Using the extended Euclidean algorithm for the inverse, the authors of [21] reported μ to be roughly in the range of [2, 12] for the binary fields recommended by NIST [22] and implemented on a Pentium II 400 MHz workstation. For such values of μ, if the proposed scheme (i.e., Algorithm 2) is used, which can speed up the inverse operation by a factor of two or more, one is expected to see an improvement of $30 - 70\%$ in the computation of elliptic curve point addition, and consequently in the corresponding scalar multiplication.

6 Concluding Remarks

In this article, the computation of inverses over very large binary extension fields has been considered. The conventional extended Euclidean algorithm for computing the inverse has been systematically optimized. To this end, a number of schemes have been proposed which significantly increase the performance of the inverter implemented using general purpose processors, such as Intel's Pentium processors. The proposed scheme of *polynomial updating up to the exact precision* alone can double the speed of the the inverse operation. The look-up table based scheme provides additional improvement by quickly determining the degree of the polynomials used in the inverse operation.

A hardware structure for the inverter has also been presented. This structure can be very useful for resource constrained systems, such as smart cards. The proposed structure requires only about one half of the total number of storage registers that the conventional extended Euclidean algorithm based inverter needs.

Various cryptographic functions can benefit from the proposed inversion algorithm. In this article, we have discussed the case of elliptic curve cryptography. It is expected that the use of the proposed inversion algorithm would result in a speed-up of $30-70\%$ for the addition of two elliptic curve points using the affine coordinates.

Also, the methods presented here are not restricted to the computations of inverses in finite fields. These methods can be used for applications which rely on the Euclidean or similar algorithms, for example, the algorithm for solving discrete-time Wiener-Hopf equations [16].

7. Acknowledgment

Most of the work presented here was done during the author's sabbatical leave with the Motorola Labs., Schaumburg, IL, USA. The author wishes to thank Larry Puhl for his encouragement to pursue this work. The author is grateful to Ezzy Dabbish and Tom Messerges for their useful comments on the draft of the manuscript. Thanks are also due to Dean Vogler for his help with the various computing resources of the labs.

References

[1] R. Lidl and H. Niederreiter, *Introduction to Finite Fields and Their Applications.* Cambridge: Cambridge University Press, 1986.

[2] R. J. McEliece, *Finite Fields for Computer Scientists and Engineers.* Boston, MA: Kluwer Academic, 1987.

[3] D. R. Stinson, *Cryptography Theory and Practice.* Roca Baton, Florida: CRC Press, 1995.

[4] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *Handbook of Applied Cryptography.* Roca Baton, Florida: CRC Press, 1996.

[5] E. R. Berlekamp, *Algebraic Coding Theory.* New York: McGraw-Hill, 1968.

[6] T. Itoh, "A fast algorithm for computing multiplicative inverses in GF(2^m)," *Inform. and Comp.*, vol. 78, pp. 171–177, 1988.

[7] M. A. Hasan and V. K. Bhargava, "Bit-Serial Systolic Divider and Multiplier for GF(2^m)," *IEEE Trans. Comput.*, vol. 41, pp. 972–980, Aug. 1992.

[8] H. Brunner, A. Curiger, and M. Hofstetter, "On Computing Multiplicative Inverses in GF(2^m)," *IEEE Trans. Comput.*, vol. 42, pp. 1010–1015, Aug. 1993.

[9] C. C. Wang, T. K. Truong, H. M. Shao, L. J. Deutsch, J. K. Omura, and I. S. Reed, "VLSI Architecture for Computing Multiplications and Inverses in GF(2^m)," *IEEE Trans. Comput.*, vol. C-34, pp. 709–717, Aug. 1985.

[10] M. Morii and M. Kasahara, "Efficient construction of gate circuit for computing multiplicative inverses over GF(2^m)," *Trans. IEICE*, vol. E 72, pp. 37–42, 1989.

[11] M. A. Hasan, "Double-Basis Inversion over GF(2^m)," *IEEE Trans. Comput.*, vol. 47, pp. 960–970, Sept. 1998.

[12] K. Araki, I. Fujita, and M. Morisue, "Fast Inverter over Finite Field Based on Euclid's Algorithm," *Trans. IEICE*, vol. E 72, pp. 1230–1234, Nov. 1989.

[13] R. Schroeppel, S. O'Malley, H. Orman, and O. Spatscheck, "A Fast Software Implementation for Arithmetic Operations in GF(2^n)," in *Advances in Cryptology- CRYPTO '95, Lecture Notes in Computer Science*, pp. 43–56, Springer, 1995.

[14] G. Harper, A. Menezes, and S. Vanstone, "Public-Key Cryptosystems with Very Small Key Lengths," in *Advances in Cryptology- EUROCRYPT '92, Lecture Notes in Computer Science*, pp. 163–173, Springer-Verlag, 1992.

[15] E. Win, A. Bosselaers, S. Vandenberghe, P. D. Gersem, and J. Vandewalle, "A Fast Software Implementation for Arithmetic Operations in GF(2^n)," in *Advances in Cryptology- ASIACRYPT '96, Lecture Notes in Computer Science*, pp. 65–76, Springer, 1996.

[16] Y. Sugiyama, "An Algorithm for Solving Discrete-Time Wiener-Hopf Equations Based on Euclid's Algorithm," *IEEE Trans. Inform. Theory*, vol. IT-32, pp. 394–409, 1986.

[17] V. S. Miller, "Use of Elliptic Curves in Cryptography," in *Advances in Cryptology- CRYPTO '85*, pp. 417–426, Springer, 1986.

[18] N. Koblitz, "Elliptic Curve Cryptosystems," *Math. Comp.*, vol. 48, pp. 203–209, 1993.

[19] A. J. Menezes, *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[20] I. F. Blake, G. Seroussi, and N. P. Smart, *Elliptic Curves in Cryptography*. Cambridge Univ Press, 1999.

[21] D. Hankerson, J. L. Hernandez, and A. Menezes, "Software Implementation of Elliptic Curve Cryptography Over Binary Fields," in *Proceedings of Workshop on Cryptographic Hardware and Embedded Systems*, pp. 1–24, LNCS, Springer-Verlag, 2000.

[22] U.S. Department of Commerce/NIST, "Digital Signature Standards (DSS)," http://csrc.nist.gov/cryptval: Federal Information Processing Standards Publications, Jan. 2000.

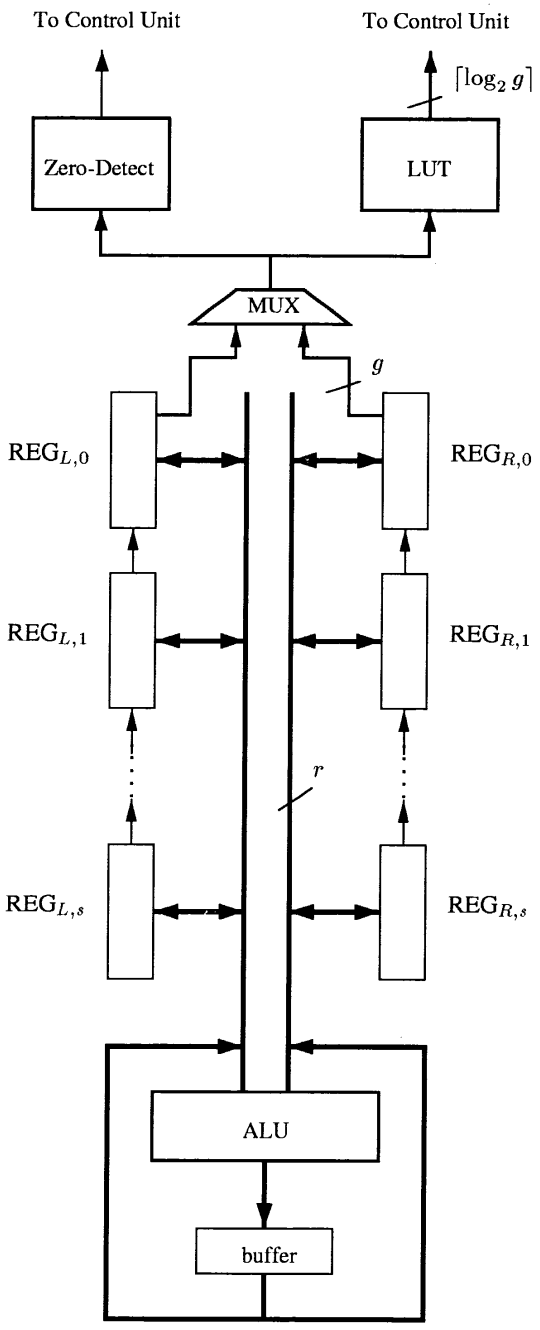

Figure 1. An r-bit wide datapath for hardware realization for an inverter in GF(2^n) where $s = \lceil \frac{n}{r} \rceil$.