Efficient Methods for Composite Field Arithmetic

E. Savaş and Ç. K. Koç
Electrical & Computer Engineering
Oregon State University
Corvallis, Oregon 97331

Technical Report, December 1999

Abstract

We propose new and efficient algorithms for basic arithmetic (squaring, multiplication, and inversion) operations in the Galois fields $GF(2^k)$ where k is a composite integer as $k = nm$. These algorithms are suitable for obtaining fast software implementations of the field operations on microprocessors and signal processors, and they are particularly useful for applications in public-key cryptography where $k \in [160, 512]$.

Key Words: Finite field arithmetic, polynomial basis, optimal normal basis, multiplication, inversion.

1 Introduction

Several algorithms for basic arithmetic operations in finite fields, suitable for hardware and software implementations have been recently developed [11, 15, 1, 9, 13, 16, 17]. The applications of these algorithms are found in error-correcting codes and public-key cryptography. In this paper, we examine the existing methods and introduce new methods for software implementations of the arithmetic operations in the Galois field $GF(2^k)$. The proposed algorithms are suitable for obtaining high-speed implementations of the field operations on signal processors and microprocessors.

In this paper, we consider a subset of the Galois fields $GF(2^k)$, the so-called *composite fields* where the exponent is a composite integer $k = nm$. It has been reported that efficient hardware and software implementations can be obtained for such fields [15, 1, 9, 8]. Two particular implementation methods are presented in [15, 1], where the field elements are represented as polynomials of length m with coefficients in the ground field $GF(2^n)$. The method in [15] carries out the field multiplication by first multiplying the input polynomials and reducing the resulting polynomial by a degree-m irreducible trinomial. On the other hand, the Karatsuba-Ofman algorithm is suggested for performing the multiplication operations in [1]. In both implementations, the logarithmic table lookup method is used for the ground field operations. The ground field is selected as $GF(2^{16})$ so that the coefficients of the elements in the composite representation would fit in a single computer word, which also makes the size of the logarithmic tables reasonable for general purpose computers. In order to decrease the complexity of the reduction operation after the polynomial multiplication, m is selected so that the condition $\gcd(16, m) = 1$ holds. Unfortunately, this selection limits the possible number of fields (i.e., the values of k), where we can take the advantage of the composite representation. Furthermore, while the size of the lookup tables is still reasonable for $GF(2^{16})$, it would be more efficient to use smaller tables in order to take advantage of the first level cache in computers.

In this paper, we improve the methods presented in [15, 1], and explore the implementation issues for more general cases. We have five main contributions:

1. We introduce new efficient table lookup methods using the values of n which are not an integer multiple of 8. For example, we can take n as 13, 14, and 15, which provide more combinations of n and m, and thus, more composite field implementations with varying efficiency values. This also necessitates a closer examination of the possible implementations in order to select the most efficient one.

2. We propose the use of the optimal normal bases (the ONB1 and ONB2) in addition to the polynomial basis for the composite fields. Although the multiplication operation is faster for the polynomial basis representation the squaring operation in the ONB1 and ONB2 is significantly faster than in the polynomial basis.

3. We propose the use of the specialized algorithms [5, 14] for multiplication in ONB1 and ONB2, rather than the general-purpose Massey-Omura multiplication algorithm [7]. The resulting methods are very efficient and provide comparable performance to the polynomial basis.

4. We give a new and efficient method for the inversion operation in the composite fields using the optimal normal basis. The new method is based on the extended Euclidean algorithm, and it is much faster than the Itoh-Tsujii algorithm [4].

5. We provide extensive timing results of our implementations for the composite fields in order to determine which n and m combinations would give better performance. This provides alternative implementations of the fields which have the same size or the similar size.

2 Composite Fields

In this section, we summarize relevant properties of the composite fields. Let $GF(2^k)$ denote the binary extention field defined over the prime field $GF(2)$. If the elements of the set

$$B_1 \quad = \quad \{1, \alpha, \alpha^2, \ldots, \alpha^{k-1}\} \tag{1}$$

are linearly independent, then B_1 forms a polynomial basis for $GF(2^k)$. Thus, given an element $A \in GF(2^k)$, we can write

$$A \quad = \sum_{i=0}^{k-1} a_i \; \alpha^i \; , \tag{2}$$

where $a_0, a_1, \ldots a_{k-1} \in GF(2)$ are the coefficients. Once the basis is chosen, the rules for the field operations (addition, multiplication, and inversion) can be derived.

There are various ways to represent the elements of $GF(2^k)$ depending on the choice of the basis or on the construction method of $GF(2^k)$. If k is the product of two integers as $k = nm$, then it is possible to derive a different representation method by defining $GF(2^k)$ over $GF(2^n)$. An extension field which is not defined over the prime field but one of its subfields is known as a composite field. It is denoted as $GF((2^n)^m)$ where $GF(2^n)$ is known as the ground field over which the composite field is defined. There is only one finite field of characteristic 2 for a given degree, and both the binary and composite fields refer to the same field although their representation methods are different. In order to represent the elements in the composite field $GF((2^n)^m)$, we can use the basis

$$B_2 \quad = \quad \{1, \beta, \beta^2, \ldots, \beta^{m-1}\} \ , \tag{3}$$

where β is the root of a degree m irreducible polynomial whose coefficients are in the base field $GF(2^n)$. Thus, an element $A \in GF((2^n)^m)$ can be written as

$$A \quad = \sum_{i=0}^{m-1} a'_i \cdot \beta^i \ , \tag{4}$$

where $a'_0, a'_1, \ldots, a'_{m-1} \in GF(2^n)$. Since the coefficients in the composite field representation are no longer in the prime field, we need to know how to calculate in the ground field $GF(2^n)$. The ground field operations are carried out using pre-calculated logarithmic lookup tables in composite field applications, and thus, the selection of the basis for the ground field is not important. However, in order to construct the logarithmic tables, we need to find a primitive element in $GF(2^n)$.

3 Arithmetic in the Ground Field

The logaritmic table lookup method for performing arithmetic in $GF(2^n)$ for small values of n is a well-known method [2, 15, 1]. A primitive element $g \in GF(2^n)$ is selected to serve as the generator of the field $GF(2^n)$, so that an element A in this field can be written as a power of g as $A = g^i$, where $0 \leq i \leq 2^n - 1$. Then, we compute the powers of the primitive element as g^i for $i = 0, 1, \ldots 2^n - 1$, and obtain 2^n pairs of the form (A, i).

We construct two tables sorting these pairs in two different ways: **the log table** sorted with respect to A and **the alog table** sorted with respect to i. For example, for $i = 5$ and $A = g^5$, we have $\log[A] = 5$ and $\text{alog}[5] = A$. These tables are then used for performing the field multiplication, the squaring, and the inversion operations. The tables are particularly very useful in software implementations. Given two elements $A, B \in GF(2^n))$, we perform the multiplication $C = AB$ as follows:

1. $i := \log[A]$
2. $j := \log[B]$
3. $k := i + j \pmod{2^n - 1}$
4. $C := \text{alog}[k]$

This is due to the fact that $C = AB = g^i g^j = g^{i+j \bmod 2^n-1}$. The ground field multiplication requires three memory access, a single modular addition operation with modulus $2^n - 1$. The squaring of an element A is slightly easier: only two memory access operations are required for computing $C = A^2$, as illustrated below:

1. $i := \log[A]$
2. $k := 2i \pmod{2^n - 1}$

3. $C :=$ alog$[k]$

Similarly, the inversion of an element A is computed using the property $C = A^{-1} = g^{-i} = g^{2^n-1-i}$, which requires two memory access operations:

1. $i :=$ log$[A]$
2. $k := 2^n - 1 - i$
3. $C :=$ alog$[k]$

In order to speed the ground field operations, particularly the multiplication and the addition operations, we propose two improvements:

- The use of **the extended alog table**.

  The extended alog table eliminates the modular addition operation in the multiplication (Step 3) and the squaring (Step 2) operations. The extended alog table is of length $2^{n+1} - 1$ which is about the twice the length of the standard alog table. It contains the values (k, g^k) sorted with respect to the index k, where $k = 0, 1, 2, \ldots, 2^{n+1} - 2$. Since the values of i and j in Step 1 and 2 of the multiplications are in the range $[0, 2^n - 1]$, the range of $k = i + j$ is $[0, 2^{n+1} - 2]$. Therefore, there is no need for computing the modular addition operation, and the ground field multiplication operation is simplified as follows:

  1. $i :=$ log$[A]$
  2. $j :=$ log$[B]$
  3. $k := i + j$
  4. $C :=$ extended-alog$[k]$

  Similarly, the squaring operation is given as

  1. $i :=$ log$[A]$
  2. $k := 2i$
  3. $C :=$ extended-alog$[k]$

  The penalty paid for gaining the improved performance is the size of the extended alog table. It is twice the size of the standard alog table.

  There is no particular reason to use the extended table for the inversion operation since $k = 2^n - 1 - i$ is still in the range $[0, 2^n - 1]$.

- The use of n values other than 8 or 16.

  The previous methods sugguest that we take n as 8 or 16 [2, 15]. We propose to use other values of n, particularly, $n = 13, 14, 15$ which are more useful for general purpose computer implementations. Since it is recommended to have relatively prime n and m, we obtain more composite fields with these choices of n. Furthermore, when we select n as 13, 14, or 15, we can also limit the size of the extended alog table to still fewer than 2^{16} words. Since the size is given as $2^{n+1} - 1$, the largest table becomes of $2^{15+1} - 1 = 2^{16} - 1$ words. We do not recommend the use of the extended alog table for $n = 16$ since in this case the length of the extended alog table becomes $2^{17} - 1 = 131,071$ words or $262,142$ bytes which may be considered excessive (may not fit most caches).

4 Polynomial Basis Representation of Composite Fields

The elements of $GF((2^n)^m)$ are treated as m-dimensional vectors over $GF(2^n)$ in the composite field representation. Since the coefficients in this representation are n-bit words, it is more advantageous for implementation on microprocessors, particularly when n is selected properly. In our implementation, we use the ground fields of degrees 13, 14, 15, and 16. Therefore, 16-bit computer words are sufficient to store the coefficients, however, we do not utilize a few bits in the most significant positions of the computer word. This is not a significant loss, but it can cause the number of words to represent composite field elements to increase, especially when $n = 13$. Since the arithmetic operations have to handle more words this can slow down the implementation. Using smaller lookup tables, on the other hand, will have better performance due to the localization of the memory access.

In the polynomial basis implementation, an irreducible polynomial of degree m with coefficients from $GF(2^n)$ is chosen to perform arithmetic operations in $GF((2^n)^m)$. An m-th degree polynomial which is irreducible over $GF(2)$ is also irreducible over $GF(2^n)$ if $\gcd(n, m) = 1$. Since we use both even and odd numbers between 13 and 16 for n such selections of n and m do not limit m to odd numbers as in [15]. We tabulate all possible composite field degrees between 160 and 512 for $n = 13, 14, 15, 16$ in Table 1. The rule for obtaining Table 1 is as follows:

- $\gcd(n, m) = 1$ and $n \in [13, 16]$ and $nm \in [160, 512]$.

These are the composite fields for which we can produce efficient implementations by selecting the aforementioned values of n. This gives much more composite fields than we can obtain by using $n = 16$ only.

The use of the composite field representation substantially speeds up the reduction operation following the polynomial multiplication operation. The reduction operation can be accelerated even further if an irreducible trinomial or pentanomial is used. It is established that for each integer $m \in [2, 10000]$ there exists either an irreducible trinomial or pentanomial [12]. In our implementation, we performed arithmetic in the composite fields using the polynomial basis similar to [15] whenever the field polynomial is an irreducible trinomial. However, when there does not exist an irreducible trinomial for a particular degree of m, we used an irreducible pentanomial. It is observed that the performance is still good for pentanomials.

We give the timing results for the field operations in Table 2 for a subset of the composite fields enumerated in Table 1. As can be observed from Table 2, the advantage of using the values of n other than 16 is apparent. For example, while the multiplication for $(n, m) = (16, 15)$ takes 14.1 microseconds, it takes only 10.8 microseconds for $(n, m) = (15, 16)$. Since lookup tables for $n = 15$ are smaller than those for $n = 16$, the memory access times are shorter, hence the multiplication is faster.

5 Optimal Normal Basis Representation of Composite Fields

A normal basis for the binary field $GF(2^k)$ is given as

$$B = \{\beta, \beta^2, \beta^{2^2}, \ldots, \beta^{2^{k-1}}\} \tag{5}$$

with the property that the elements of B are linearly independent. There exists at least one normal basis for $GF(2^k)$ for every positive integer k. The normal basis representations have the computational advantage that the squaring of an element can be performend by a circular shift. On the other hand, the multiplication of two distinct elements requires a more complicated circuit, whose complexity is reduced only for a subset of normal bases, called the optimal normal bases [6].

There exists two types of the optimal normal bases which are historically named as the optimal normal basis of type 1 (ONB1) and the optimal normal basis of type 2 (ONB2). There are 117 and 319 m values in the range $m \in [2, 2001]$, such that the field $GF(2^m)$ has an optimal normal basis of type 1 and type 2, respectively [6]. In other words, the ONB2 is more abundant, and thus, this representation is much more useful.

The composite field elements can be represented using optimal normal bases when there exists an ONB1 or ONB2 for $GF(2^m)$ in the setting $GF((2^n)^m)$. As long as $\gcd(n, m) = 1$, a linearly independent set which forms an optimal normal basis for a binary field $GF(2^m)$ also forms an optimal normal basis for the composite field $GF((2^n)^m)$. The element A in the composite field can be written as

$$A = \sum_{i=0}^{m-1} A_i \ \beta^i \ , \tag{6}$$

where $A_i \in GF(2^n)$. Since the degree-m normal polynomial is also a normal polynomial in $GF((2^n)^m)$, every algorithm for performing arithmetic in the binary field for the normal bases also works in the composite field without any modification.

In Table 3, we enumerate all possible composite fields expressed using the ONB1 and ONB2 for $160 < nm < 512$, where the ground field is taken as $GF(2^n)$ for $n = 13, 14, 15, 16$. As expected, we have more composite fields expressed in ONB2 than in ONB1 in this range. The rule for obtaining Table 3 is as follows:

- ONB1 for $GF((2^n)^m)$:

  An ONB1 exists for $GF(2^m)$ and $\gcd(n, m) = 1$ and $n \in [13, 16]$ and $nm \in [160, 512]$.

- ONB2 for $GF((2^n)^m)$:

  An ONB2 exists for $GF(2^m)$ and $\gcd(n, m) = 1$ and $n \in [13, 16]$ and $nm \in [160, 512]$.

In the following, we present the squaring, multiplication, and inversion algorithms for the optimal normal bases, which are used in obtaining the composite field implementations. The Massey-Omura [7] algorithm can be used for multiplication of elements represented using the ONB1 and ONB2, however, there are also specialized algorithms [5, 14]. We promote the use of these specialized algorithms for the multiplication operation in the composite fields.

5.1 Squaring in ONB1 and ONB2

The squaring in a normal basis is simply a bitwise circular shift of the binary vector. For the composite fields, the squaring is performed using a circular word shift after each word is squared in the ground field $GF(2^n)$ using the lookup tables. This squaring operation is significantly more efficient than the one in the polynomial basis because it does not require a modular reduction. Let $A \in GF((2^n)^m)$ be represented using an m-dimensional vector as

$$A = (A_0, A_1, \ldots, A_{m-2}, A_{m-1}) \ , \tag{7}$$

where $A_i \in GF(2^n)$ for $i = 0, 1, \ldots, m-1$. Using the property $\beta^{2^m} = \beta$, we obtain A^2 as follows:

$$A^2 \ = \ (\sum_{i=0}^{m-1} A_i \beta^{2^i})^2 \ = \ \sum_{i=0}^{m-1} A_i^2 \beta^{2^{i+1}}$$

$$= \quad (A_{m-1}^2, A_0^2, A_1^2, ..., A_{m-3}^2, A_{m-2}^2) \ .$$

5.2 Multiplication in ONB1

We use the algorithm proposed in [5] in our implementation. Here we give a brief description of this multiplication algorithm. An ONB1 is generated by an element $\beta \in GF((2^m)^n)$ of order $p = m+1$. Since 2 is primitive modulo p, the set which is the basis for the ONB1 representation

$$N = (\beta, \beta^2, \beta^{2^2}, ..., \beta^{2^{m-1}}) \tag{8}$$

is equivalent to the set

$$M = (\beta, \beta^2, \beta^3, ..., \beta^{m-1}) \ . \tag{9}$$

The set M is called the shifted polynomial basis, and its field polynomial is an irreducible all-one-polynomial [5]. Furthermore, M is obtained from N by a permutation. Let the field element $A \in GF((2^n)^m)$ be expressed in ONB1 as

$$A = \sum_{i=0}^{m-1} A_i \beta^{2^i} . \tag{10}$$

We can also express A in the shifted polynomial basis as

$$\bar{A} = \sum_{i=0}^{m-1} \bar{A}_i \beta^{i+1} \ . \tag{11}$$

The conversion between them is established using the following permutation P:

$$\bar{A}_{(2^i - 1) \pmod{m+1}} = A_i \ \text{ for } \ i = 0, 1, \ldots, m-1 \ . \tag{12}$$

The multiplication in the ONB1 reduces to the polynomial multiplication taking advantage of the arithmetic with an irreducible all-one-polynomial [5].

1. Obtain the shifted polynomial representation of A and B using permutation P.
2. Perform the polynomial multiplication and obtain $\bar{C}$.
3. Apply inverse permutation P^{-1} to $\bar{C}$ and obtain the result in the ONB1.

Since the permutation gives shifted polynomial representation of A and B, we need to perform an extra correction in Step 2 of the algorithm. Let $A, B \in GF((2^n)^m)$ be represented in the shifted polynomial basis as $A = (\bar{A}_0, \bar{A}_1, \ldots, \bar{A}_{m-1})$ and $B = (\bar{B}_0, \bar{B}_1, \ldots, \bar{B}_{m-1})$. After the multiplication operation, the result is obtained as $F = \bar{A}\bar{B}/\beta^2$, and represented in polynomial base as follows:

$$F = F_0 + F_1\beta + F_2\beta^2 + \ldots + F_{m-1}\beta^{m-1} \ .$$

In order to obtain the correct result, we first multiply F by β^2 (i.e., we shift F two words to the left), and obtain

$$E = F_0\beta^2 + F_1\beta^3 + \ldots + F_{m-1}\beta^{m+1} \ .$$

However, the result is still not in the shifted polynomial basis since the weight of the term F_{m-1} is β^{m+1}, which needs to be reduced using the relation

$$\beta^{m+1} = \beta + \beta^2 + \ldots + \beta^m \ .$$

Therefore, after the correction operation, we obtain the shifted polynomial representation of the result as

$$C = F_{m-1}\beta + (F_{m-1} + F_0)\beta + \ldots + (F_{m-1} + F_{m-2})\beta^m \ .$$

We then need to apply the inverse permutation P^{-1} to C, and obtain the final result expressed in ONB1.

5.3 Multiplication in ONB2

We used the multiplication algorithm proposed in [14] in this case. Since this algorithm is not published, we will provide a description. It is somewhat similar to the algorithm described in the previous section. It also requires a basis conversion from the ONB2 to a new type of basis. However, the new basis is not a polynomial basis, and the multiplication in this new basis is more complicated. An ONB2 for the field $GF((2^n)^m)$ is constructed using the normal element $\beta = \gamma + \gamma^{-1}$ where γ is a primitive $(2m+1)$th root of unity, i.e. $\gamma^{2m+1} = 1$ and $\gamma^i \neq 1$ for any $1 \leq i \leq 2m+1$. It turns out that an ONB2 can be constructed if $p = 2m+1$ is prime and also if either of the following two conditions holds:

- 2 is primitive modulo p
- $p \equiv 3 \pmod 4$ and 2 generates the quadratic residues modulo p [6].

An element $A \in GF((2^n)^m)$ is represented using the ONB2

$$N = \{\beta, \beta^2, \ldots, \beta^{2^{m-1}}\} = \{\gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^{2^2} + \gamma^{-2^2}, \ldots \gamma^{2^{m-1}} + \gamma^{-2^{m-1}}\}$$

as follows:

$$A = A_0\beta + A_1\beta^2 + \ldots + A_{m-1}\beta^{2^{m-1}} \ .$$

A basis element can be written as $\beta^{2^i} = \gamma^j + \gamma^{-j}$ for $j \in [1, 2m]$ following the fact that 2 is primitive modulo p. The set

$$M = \{\gamma + \gamma^{-1}, \gamma^2 + \gamma^{-2}, \gamma^3 + \gamma^3, \ldots \gamma^m + \gamma^m\}$$

is a permutation of the ONB2 basis N, hence forms another basis for $GF((2^n)^m)$. Then $A \in GF((2^n)^m)$ is expressed in the new basis M as

$$A = \bar{A}_0\beta_1 + \bar{A}_1\beta_2 + \ldots + \bar{A}_{m-1}\beta_m \ .$$

where $\beta_i = \gamma^i + \gamma^{-i}$. The conversion from one representation to the other involves only a permutation, which can be given in terms of coefficients $\bar{A}_j = A_i$ as

$$j = \begin{cases} k & \text{if } k \in [1, m] \\ (2m+1) - k & \text{if } k \in [m+1, 2m] \end{cases} ,$$

where $k = 2^{i-1} \pmod p$ for $i = 1, 2, \ldots, m$. The multiplication in basis M is performed using the formulae which were derived in [14]. Let $A, B \in GF((2^n)^m)$ are represented in M as

$$A = \sum_{i=1}^{m} \bar{A}_i\beta_i \ \text{ and } \ B = \sum_{i=1}^{m} \bar{B}_i\beta_i \ .$$

Then, the product $C = AB$ can be calculated using $C = C_1 + D_1 + D_2$, where

$$\begin{aligned} C_1 &= \sum_{\substack{1 \le i,j \le m \\ i \ne j}} \bar{A}_i \bar{B}_j \beta_{i-j} \\ D_1 &= \sum_{i=1}^{m} \sum_{j=1}^{m-i} \bar{A}_i \bar{B}_j \beta_{i+j} \\ D_2 &= \sum_{i=1}^{m} \sum_{j=m-i+1}^{m} \bar{A}_i \bar{B}_j \beta_{i+j} \end{aligned}$$

Then, the ONB2 representation of the result is obtained using the inverse permutation.

5.4 Inversion in ONB1 and ONB2

The Itoh-Tsujii algorithm [4] for inversion in the binary fields using the normal bases is also suggested for the composite fields using the polynomial basis in [1]. The algorithm reduces the inversion problem in the composite field to the inversion in the ground field $GF(2^n)$. However, it requires several field multiplications, and the number of these multiplications increases as m gets larger. On the other hand, the inversion algorithm for the polynomial basis described in [15] is very efficient, and it is possible to apply the same algorithm to calculate the inversion of the composite field elements expressed in a normal basis. A method for inversion in the binary extension fields expressed in the optimal normal basis was described in [10]. We generalize the methodology to the composite fields expressed using the optimal normal basis.

In order to invert an element given in the optimal normal basis, we need to transform it to the polynomial basis using the field polynomial of the normal basis. Although the field polynomial of the normal basis may have many non-zero terms, this is not a disadvantage since using an arbitrary field polynomial does not affect the performance of the inversion operation. The inversion operation is performed on the transformed element, and finally the result is mapped back into the normal basis. Although the conversion operations seem to complicate the calculation, our experimental results show that we can obtain better results with the new algorithm.

The inversion algorithm in polynomial basis is based on the extended Euclidean algorithm, which can be given as follows:

- **Input:** $A \in GF((2^n)^m)$ and P (the irreducible field polynomial)
- **Output:** $B \in GF((2^n)^m)$ such that $AB = 1 \pmod{P}$

```
1. Initialize polynomials $B := 1$, $C := 0$, $F := A$, and $G := P$
2. if $F = 1$ then return $B \cdot F^{-1}$
3. if $\deg(F) < \deg(G)$ then exchange $F$ & $G$ and exchange $B$ & $C$
4. $\delta := \deg(F) - \deg(G)$
5. $\alpha = F_{\deg(F)} \cdot G_{\deg(G)}$
6. $F := F + \alpha x^\delta G$ and $B := B + \alpha x^\delta C$
7. Go to Step 2
```

We implemented the Itoh-Tsujii and the extended Euclidean algorithms in the C language, and obtained some timing results using the Microsoft Visual C++ 5.0 on a 450-MHz Pentium II based PC running Windows NT 4.0. The timing values given in Table 4 are in microseconds. As can be observed from Table 4, the extended Euclidean algorithm is much faster when the subfield degree is large since the length of F decreases in each iteration.

In order to apply the polynomial inversion algorithm for optimal normal bases, the elements represented in the optimal normal basis need to be converted to the polynomial basis. The polynomial basis constructed using the field polynomial of the normal basis provides considerable advantage for this conversion because only XOR and assignment operations are required during the conversion. The field polynomials for the ONB1 bases are irreducible all-one-polynomials. On the other hand, the field polynomials for the ONB2 can be computed using the following recursion:

$$\begin{aligned} f_0(x) &= 1 \\ f_1(x) &= x+1 \\ f_n(x) &= xf_{n-1}(x) + f_{n-2} \ \text{ for } \ n \geq 2 \end{aligned} \tag{13}$$

More information about field polynomials of optimal normal bases can be found in [6]. The change of basis matrix which allows us to perform the conversion between optimal normal basis and polynomial basis can be calculated using the algorithms given in [3, pages 37-39].

We now give an example in order to illustrate the use of a polynomial basis inversion algorithm in an optimal normal basis. Let $A \in GF((2^n)^{11})$ be expressed in the ONB2 as

$$A = A_0\beta + A_1\beta^2 + A_2\beta^{2^2} + \cdots + A_{10}\beta^{2^{10}} \ ,$$

where $\gcd(n, 11) = 1$. We first compute the field polynomial for $GF((2^n)^{11})$ using the recursion (13), and obtain it as

$$f_{11}(x) = x^{11} + x^{10} + x^8 + x^4 + x^3 + x^2 + 1 \ .$$

Let the polynomial representation of A be

$$A = \bar{A}_0 + \bar{A}_1\alpha + \bar{A}_2\alpha^2 + \ldots + \bar{A}_{10}\alpha^{10} \ ,$$

where α is a root of $f_{11}(x)$. We then use the field polynomial and the algorithm in [3] in order the obtain the change of basis matrices between the polynomial basis and the ONB2. We summarize the final change of basis formulae below. The conversion from the ONB2 representation to the polynomial is performed using:

$$\begin{aligned} \bar{A}_0 &= A_0 \\ \bar{A}_1 &= A_0 + A_4 + A_5 + A_6 + A_8 + A_{10} \\ \bar{A}_2 &= A_1 + A_7 + A_9 + A_{10} \\ \bar{A}_3 &= A_6 + A_8 \\ \bar{A}_4 &= A_2 + A_{10} \\ \bar{A}_5 &= A_4 + A_5 + A_{10} \\ \bar{A}_6 &= A_7 + A_9 \\ \bar{A}_7 &= A_4 + A_5 \\ \bar{A}_8 &= A_3 + A_{10} \\ \bar{A}_9 &= A_5 + A_{10} \\ \bar{A}_{10} &= A_7 + A_{10} \ . \end{aligned}$$

The conversion from the polynomial representation to the ONB2 representation is performed using:

$$\begin{aligned}
A_0 &= \bar{A}_1 + \bar{A}_3 + \bar{A}_7 + \bar{A}_{10} \\
A_1 &= \bar{A}_2 + \bar{A}_6 + \bar{A}_0 \\
A_2 &= \bar{A}_4 + \bar{A}_0 \\
A_3 &= \bar{A}_8 + \bar{A}_0 \\
A_4 &= \bar{A}_7 + \bar{A}_9 + \bar{A}_0 \\
A_5 &= \bar{A}_9 + \bar{A}_0 \\
A_6 &= \bar{A}_5 + \bar{A}_7 + \bar{A}_0 \\
A_7 &= \bar{A}_{10} + \bar{A}_0 \\
A_8 &= \bar{A}_3 + \bar{A}_5 + \bar{A}_7 + \bar{A}_0 \\
A_9 &= \bar{A}_6 + \bar{A}_{10} + \bar{A}_0 \\
A_{10} &= \bar{A}_0 \ .
\end{aligned}$$

6 Implementation Results and Conclusions

In order to test the proposed methods, we wrote test routines in C, and obtained the timing results using the Microsoft Visual C++ 5.0 on a 450-MHz Pentium II based PC running Windows NT 4.0. The timing results are summarized in Tables 2, 4, 5, 6 and Figures 1, 2, 3. These timing values are all in microseconds.

The timing results for the polynomial basis implementation of the composite fields are given in Table 2. The squaring, multiplication, and the inversion methods are the same as in [15] except that we allow irreducible trinomials and pentanomials while the methods in [15] cover only irreducible trinomials. Table 2 clearly illustrates the advantage of using the subfields other than $GF(2^{16})$. For example, the multiplication operation in $GF((2^{16})^{13})$ takes 11.0 microseconds, while it takes only 9.6 microseconds for $GF((2^{13})^{16})$. Since the lookup tables for $n = 13$ are smaller than those for $n = 16$, the memory access times are shorter, and thus, the multiplication is faster.

The squaring algorithm for the ONB1 and ONB2 was described §5.1. Its timing values are tabulated in Tables 4 and 5. The multiplication algorithms for the ONB1 and ONB2 were described in §5.2 and §5.3, respectively, which are based on the previously developed methods reported in [5, 14]. Both algorithms use a permutation to convert the elements expressed in the optimal normal basis to the polynomial basis (actually, to a basis similar to the polynomial basis), and perform the multiplication operation in the polynomial basis using these specialized algorithms, and the convert the result back to the optimal normal basis.

The inversion method we proposed in §5.4 for the composite fields is an alternative to the well-known Itoh-Tsujii [4] algorithm. Our method performs the inversion of an element expressed in ONB1 or ONB2 by first converting it to the polynomial basis using the field polynomial of the optimal normal basis. It then uses the extended Euclidean algorithm to obtain the inverse of the given element in the polynomial basis. The result is converted back to the optimal normal basis. The field polynomial is an all-one-polynomial for the ONB1 and a random polynomial for the ONB2. The type of the field polynomial makes a difference only in the conversions between the optimal normal basis and the polynomial basis: there will be more additions (XORs) for the ONB2 case in general. The performance of the extended Euclidean algorithm is the same for any field polynomial. We compare the timings of the Itoh-Tsujii algorithm and the extended Euclidean algorithm in Table 4, which shows that even though the overhead of the conversions slows down the inversion operation, the extended Euclidean algorithm as proposed for the composite fields is still faster than Itoh-Tsujii algorithm; in some cases it is more than twice faster. Thus, we used the extended Euclidean in the rest of our implementation. The inversion timings given in Tables 5 and 6 are obtained using the extended Euclidean algorithm.

In Figures 1, 2, and 3, we illustrate the squaring, multiplication, and inversion timings together in order to compare the performance of these three bases. We clearly see from these figures that the squaring operation in the ONB1 and ONB2 is much faster than in the polynomial basis since the reduction is avoided. On the other hand, the multiplication operation is only slightly slower for the optimal normal bases than for the polynomial basis. Furthermore, comparing the timing results for the inversion operation in these three bases, we notice that their timings are very close to one another. Between the ONB1 and ONB2, we also see that the ONB2 is more advantageous since it provides more composite fields in the specified range $[160, 512]$ as it can be seen in Table 3.

References

[1] J. Guajardo and C. Paar. Efficient algorithms for elliptic curve cryptosystems. In B. S. Kaliski Jr., editor, *Advances in Cryptology — CRYPTO 97*, Lecture Notes in Computer Science, No. 1294, pages 342–356. Springer-Verlag, Berlin, Germany, 1997.

[2] G. Harper, A. Menezes, and S. Vanstone. Public-key cryptosystems with very small key lengths. In R. A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT 92*, Lecture Notes in Computer Science, No. 658, pages 163–173. Springer-Verlag, Berlin, Germany, 1992.

[3] IEEE P1363. Standard specifications for public-key cryptography. Draft version 7, September 1998.

[4] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78(3):171–177, September 1988.

[5] Ç. K. Koç and B. Sunar. Low-complexity bit-parallel canonical and normal basis multipliers for a class of finite fields. *IEEE Transactions on Computers*, 47(3):353–356, March 1998.

[6] A. J. Menezes, I. F. Blake, X. Gao, R. C. Mullen, S. A. Vanstone, and T. Yaghoobian. *Applications of Finite Fields.* Kluwer Academic Publishers, Boston, MA, 1993.

[7] J. Omura and J. Massey. Computational method and apparatus for finite field arithmetic. U.S. Patent Number 4,587,627, May 1986.

[8] C. Paar, P. Fleischmann, and P. Soria-Rodriguez. Fast arithmetic for public-key algorithms in Galois fields with composite exponents. *IEEE Transactions on Computers*, 48(10):1025–1034, October 1999.

[9] C. Paar and P. Soria-Rodriguez. Fast arithmetic architectures for public-key algorithms over Galois fields $GF((2^n)^m)$. In W. Fumy, editor, *Advances in Cryptology — EUROCRYPT 97*, Lecture Notes in Computer Science, No. 1233, pages 363–378. Springer-Verlag, Berlin, Germany, 1997.

[10] M. Rosing. *Implementing Elliptic Curve Cryptography.* Manning Publications, Greenwich, CT, 1999.

[11] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. In D. Coppersmith, editor, *Advances in Cryptology — CRYPTO 95*, Lecture Notes in Computer Science, No. 973, pages 43–56. Springer-Verlag, Berlin, Germany, 1995.

[12] G. Seroussi. Table of low-weight binary irreducible polynomials. Hewlett-Packard, HPL-98-135, August 1998.

[13] J. H. Silverman. Fast multiplication in finite field $GF(2^N)$. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 122–134. Springer-Verlag, Berlin, Germany, 1999.

[14] B. Sunar and Ç. K. Koç. An efficient optimal normal basis type II multiplier, January 1999. Submitted for publication.

[15] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in GF(2^n). In K. Kim and T. Matsumoto, editors, *Advances in Cryptology — ASIACRYPT 96*, Lecture Notes in Computer Science, No. 1163, pages 65–76. Springer-Verlag, Berlin, Germany, 1996.

[16] H. Wu. Low complexity bit-parallel finite field arithmetic using polynomial basis. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 280–291. Springer-Verlag, Berlin, Germany, 1999.

[17] H. Wu, M. A. Hasan, and I. F. Blake. Highly regular architectures for finite field computation using redundant basis. In Ç. K. Koç and C. Paar, editors, *Cryptographic Hardware and Embedded Systems*, Lecture Notes in Computer Science, No. 1717, pages 269–279. Springer-Verlag, Berlin, Germany, 1999.

Table 1: Composite field degrees ($165 < k < 512$) using the polynomial basis.

n	m	nm	n	m	nm	n	m	nm	n	m	nm
13	14	182	14	13	182	15	11	165	16	11	176
	15	195		15	210		14	210		13	208
	16	208		17	238		16	240		15	240
	17	221		19	266		17	255		17	272
	18	234		23	322		19	285		19	304
	19	247		25	350		22	330		21	336
	20	260		27	378		23	345		23	368
	21	273		29	406		26	390		25	400
	22	286		31	434		28	420		27	432
	23	299		33	462		29	435		29	464
	24	312					31	465		31	496
	25	325					32	480			
	27	351					34	510			
	28	364									
	29	377									
	30	390									
	31	403									
	32	416									
	33	429									
	34	442									
	35	455									
	36	468									
	37	481									
	38	494									

Table 2: The timings in microseconds for the polynomial basis.

n	m	nm	Squaring	Multiplication	Inversion
13	14	182	1.29	7.6	29
	15	195	1.28	8.6	32
	16	208	1.70	9.6	36
	18	234	1.56	12.1	43
	23	299	1.84	18.6	66
	28	364	2.23	24.4	93
	29	377	2.30	25.9	99
	30	390	2.38	27.6	104
	33	429	2.40	32.9	125
	35	455	2.49	36.4	137
	36	468	2.58	38.3	148
	38	494	2.97	43.1	164
14	13	182	1.53	7.4	28
	15	210	1.37	9.1	35
	19	266	2.02	14.2	53
	23	322	1.97	19.4	72
	27	378	2.73	25.7	96
	29	406	2.46	28.2	109
	33	462	2.66	35.6	137
15	11	165	1.20	5.8	23
	13	195	1.59	7.6	29
	14	210	1.44	8.4	33
	16	240	1.88	10.8	42
	19	285	2.08	14.6	55
	23	345	2.05	19.9	75
	26	390	2.75	24.4	100
	28	420	2.49	27.2	107
	29	435	2.57	29.0	113
	31	465	2.34	33.2	127
	34	510	2.63	38.7	151
16	11	176	1.44	8.9	30
	13	208	1.79	11.0	40
	15	240	1.79	14.1	50
	23	368	2.60	31.0	97
	27	432	3.25	42.1	127
	29	464	2.90	46.5	144
	31	496	3.30	54.9	164

Table 3: Composite field degrees ($160 < k < 512$) using the ONB1 and ONB2.

ONB1			ONB2		
n	m	nm	n	m	nm
13	18	234	13	14	182
	28	364		18	234
	36	468		23	299
15	28	420		29	377
				30	390
				33	429
				35	455
			14	23	322
				29	406
				33	462
			15	11	165
				14	210
				23	345
				26	390
				29	435
			16	11	176
				23	368
				29	464

Table 4: The inversion timings in microseconds.

n	m	nm	Itoh-Tsujii	Extended Euclid
13	14	182	62	29
	18	234	108	45
15	11	165	33	22
	14	210	64	34
16	11	176	44	27

Table 5: The timings in microseconds for the ONB1.

n	m	nm	Squaring	Multiplication	Inversion
13	18	234	0.96	13.21	46
	28	364	1.33	28.50	98
	36	468	1.60	44.80	149
15	28	420	1.65	32.26	114

Table 6: The timings in microseconds for the ONB2.

n	m	nm	Squaring	Multiplication	Inversion
13	14	182	0.73	12.37	29
	18	234	0.86	21.93	45
	23	299	1.04	26.81	70
	29	377	1.23	43.00	106
	30	390	1.27	43.18	113
	33	429	1.38	55.21	135
	35	455	1.45	61.82	153
14	23	322	1.18	28.45	73
	29	406	1.42	46.34	118
	33	462	1.57	59.22	148
15	11	165	0.78	8.31	22
	14	210	0.89	13.70	34
	23	345	1.28	35.27	76
	26	390	1.48	36.75	94
	29	435	1.51	47.55	124
16	11	176	1.01	10.38	27
	23	368	1.73	38.16	100
	29	464	2.12	59.71	154

Figure 1: Squaring timings in microseconds.

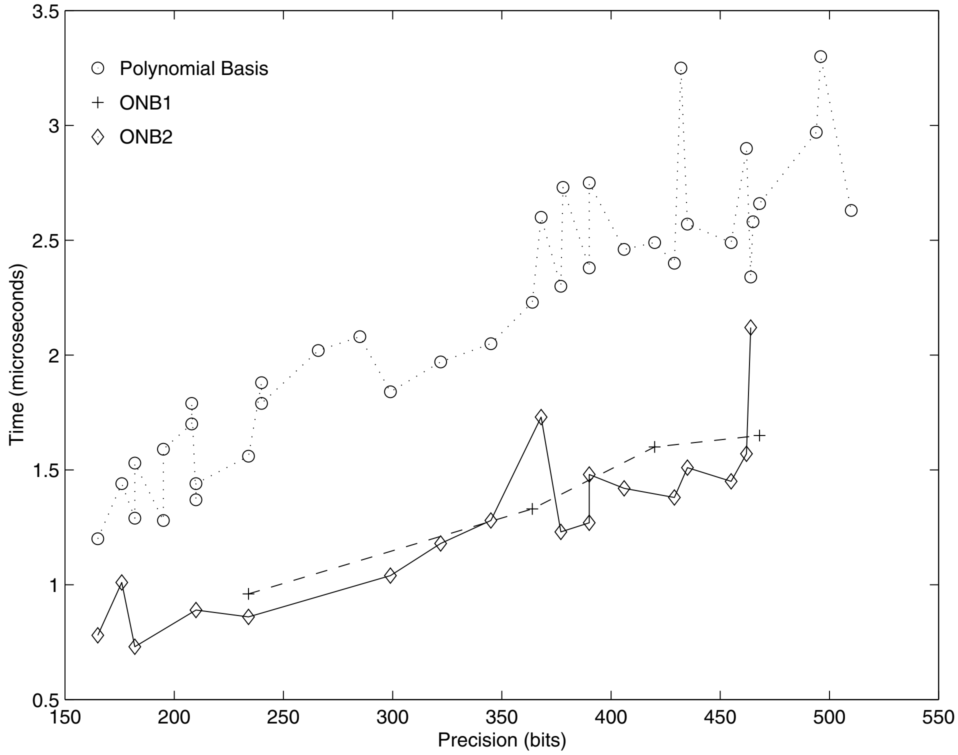


Figure 2: Multiplication timings in microseconds.

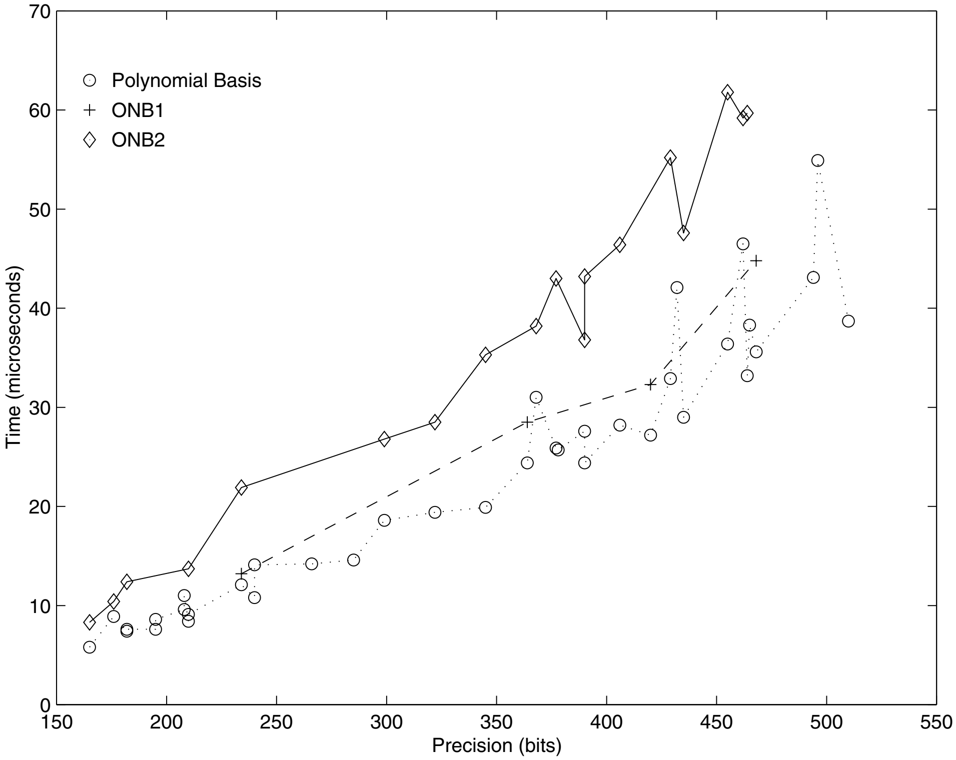
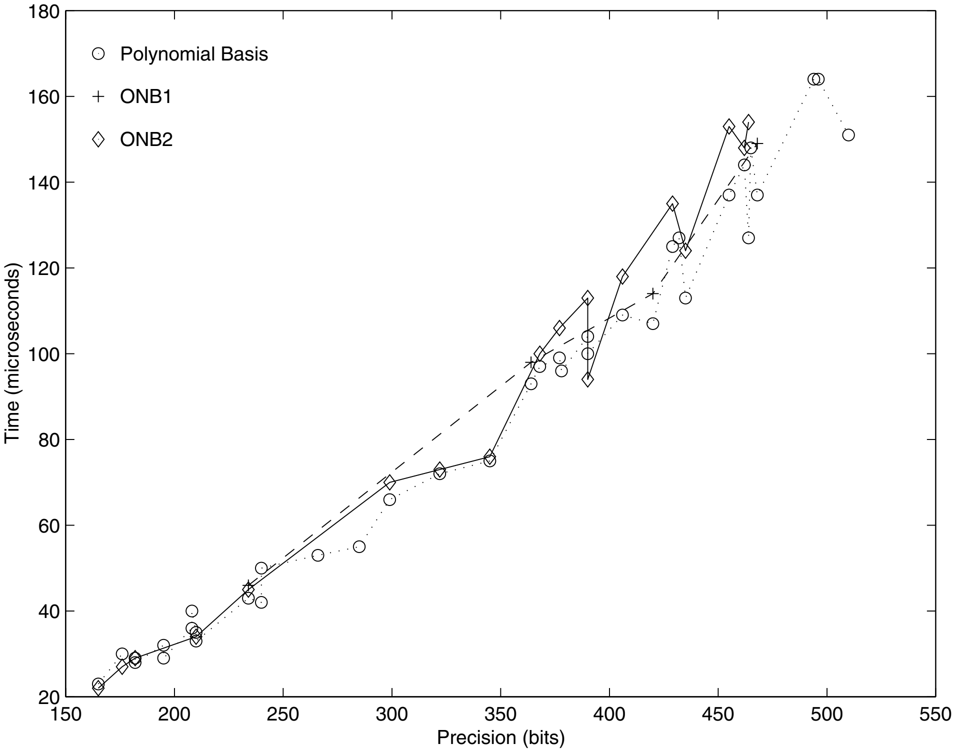


Figure 3: Inversion timings in microseconds.


Polynomial Basis
ONB1
ONB2
Time (microseconds)
Precision (bits)